CONSULTORES EDITORIALES:

SEBASTIÁN DORMIDO BENCOMO
Departamento de Informática y Automática
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

LUIS JOYANES AGUILAR
Departamento de Lenguajes, Sistemas Informáticos e Ingeniería del Software
UNIVERSIDAD PONTIFICIA DE SALAMANCA. Campus Madrid

SISTEMAS DISTRIBUIDOS
CONCEPTOS Y DISEÑO

Tercera edición

GEORGE COULOURIS
Queen Mary and Westfield College
University of London, and Cambridge University

JEAN DOLLIMORE
Queen Mary and Westfield College
University of London

TIM KINDBERG
Hewlett-Packard Laboratoires, Palo Alto

Traducción:

José Belarmino Pulido Junquera
Benjamín Sahelices Fernández
Jesús María Vegas Hernández
Universidad de Valladolid

Coordinadores de la traducción:

Pablo de la Fuente Redondo
Cesar Llamas Bello
Universidad de Valladolid

Revisión técnica:

Sebastián Dormido Bencomo
Universidad Nacional de Educación a Distancia

Addison Wesley

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo • San Juan
San José • Santiago • São Paulo • Reading, Massachusetts • Harlow, England

Datos de catalogación bibliográfica

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.
SISTEMAS DISTRIBUIDOS. Conceptos y diseño.
Tercera edición

PEARSON EDUCACIÓN, S. A., Madrid, 2001

ISBN: 84-7829-049-4
Materia: Informática: 681.3

Formato 195 × 250 Páginas: 744

No está permitida la reproducción total o parcial de esta obra
ni su tratamiento o transmisión por cualquier medio o método,
sin autorización escrita de la Editorial.

DERECHOS RESERVADOS
© 2001 respecto a la primera edición en español por:
PEARSON EDUCACIÓN, S. A.
Ribera del Loira, 28
28042 Madrid (España)

GEORGE COULOURIS, JEAN DOLLIMORE, TIM KINDBERG
SISTEMAS DISTRIBUIDOS. Conceptos y diseño. Tercera edición

ISBN: 84-7829-049-4
Depósito legal: M. 45.496-2003

ADDISON WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN, S. A.

Traducido de:
Distributed Systems Concepts and Design, Third Edition
© Addison Wesley Publishers Limited 1990, 1994
© Pearson Education Limited 2001
ISBN 0-201-61918-0

Edición en español:
Equipo editorial:
Editor: Andrés Otero
Asistente editorial: Ana Isabel García
Equipo de producción:
Director: José Antonio Clares
Técnico: Diego Marín
Diseño de cubierta: Mario Guindel y Yann Boix
Composición: COPIBOOK, S. L.
Impreso en Imprenta Fareso, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Addison Wesley

Este libro ha sido impreso con papel y tintas ecológicos

CONTENIDO

PRÓLOGO XIII
Objetivos y audiencia XIII
Organización del libro XIV
Referencias XIV
Cambios en esta edición XV
Reconocimientos XV
Sitio web XVI

1. CARACTERIZACIÓN DE LOS SISTEMAS DISTRIBUIDOS 1
1.1. Introducción 2
1.2. Ejemplos de sistemas distribuidos 3
1.2.1. Internet 3
1.2.2. Intranets 4
1.2.3. Computación móvil y ubicua 5
1.3. Recursos compartidos y Web 7
1.3.1. El World Wide Web 8
1.4. Desafíos 15
1.4.1. Heterogeneidad 15
1.4.2. Extensibilidad 17
1.4.3. Seguridad 18
1.4.4. Escalabilidad 19
1.4.5. Tratamiento de fallos 20
1.4.6. Concurrencia 21
1.4.7. Transparencia 22
1.5. Resumen 24
Ejercicios 25

2. MODELOS DE SISTEMA 27
2.1. Introducción 28
2.2. Modelos arquitectónicos 28
2.2.1. Capas de software 29
2.2.2. Arquitecturas de sistema 31
2.2.3. Variaciones en el modelo de cliente-servidor 34

2.2.4. Interfaces y objetos ... 40
2.2.5. Requisitos de diseño para arquitecturas distribuidas ... 41
2.3. Modelos fundamentales ... 44
2.3.1. Modelo de interacción ... 45
2.3.2. Modelo de fallo ... 50
2.3.3. Modelo de seguridad ... 54
2.4. Resumen ... 58
Ejercicios ... 59

3. REDES E INTERCONEXIÓN DE REDES ... 61

3.1. Introducción ... 62
3.1.1. Las redes y los sistemas distribuidos ... 62
3.2. Tipos de redes ... 65
3.3. Fundamentos de redes ... 69
3.3.1. Transmisión de paquetes ... 69
3.3.2. Flujos de datos ... 69
3.3.3. Esquemas de conmutación ... 70
3.3.4. Protocolos ... 71
3.3.5. Encaminamiento ... 77
3.3.6. Control de la congestión ... 80
3.3.7. Interconexión de redes ... 81
3.4. Protocolos Internet ... 84
3.4.1. Direccionamiento IP ... 86
3.4.2. El protocolo IP ... 89
3.4.3. Encaminamiento IP ... 90
3.4.4. IP versión 6 ... 93
3.4.5. IP móvil ... 96
3.4.6. TCP y UDP ... 97
3.4.7. Nombres de dominio ... 100
3.4.8. Cortafuegos ... 100
3.5. Casos de estudio: Ethernet, LAN inalámbrica y ATM ... 104
3.5.1. Ethernet ... 105
3.5.2. LAN inalámbrica IEEE 802.11 ... 109
3.5.3. Redes de modo de transferencia asíncrono ... 112
3.6. Resumen ... 114
Ejercicios ... 115

4. COMUNICACIÓN ENTRE PROCESOS ... 117

4.1. Introducción ... 118
4.2. API para los protocolos de Internet ... 119
4.2.1. Las características de la comunicación entre procesos ... 119
4.2.2. Sockets ... 121
4.2.3. Comunicación de datagramas UDP ... 122
4.2.4. Comunicación de streams TCP ... 125
4.3. Representación externa de datos y empaquetado ... 128
4.3.1. Representación común de datos de CORBA (CDR) ... 130
4.3.2. Serialización de objetos en Java ... 132
4.3.3. Referencias objetos remotos ... 134
4.4. Comunicación cliente-servidor ... 135
4.5. Comunicación en grupo ... 143
4.5.1. Multidifusión IP. Una implementación de la comunicación en grupo ... 144
4.5.2. Fiabilidad y orden en multidifusión ... 146
4.6. Caso de estudio: comunicación entre procesos en UNIX ... 147
4.6.1. Comunicación de datagramas ... 148

4.6.2. Comunicación con streams ... 149
4.7. Resumen ... 150
Ejercicios ... 151

5. OBJETOS DISTRIBUIDOS E INVOCACIÓN REMOTA ... 155

5.1. Introducción ... 156
5.1.1. Interfaces ... 157
5.2. Comunicación entre objetos distribuidos ... 159
5.2.1. El modelo de objetos ... 159
5.2.2. Objetos distribuidos ... 161
5.2.3. El modelo de objetos distribuido ... 161
5.2.4. Cuestiones de diseño para RMI ... 163
5.2.5. Implementación de RMI ... 166
5.2.6. Compactación automática de memoria ... 171
5.3. Llamada a un procedimiento remoto ... 172
5.3.1. Caso de estudio Sun RPC ... 173
5.4. Eventos y notificaciones ... 175
5.4.1. Los participantes en una notificación de eventos distribuida ... 178
5.4.2. Especificación de eventos distribuidos de Jini ... 180
5.5. El caso de estudio Java RMI ... 182
5.5.1. Construcción de programas clientes y servidores ... 184
5.5.2. Diseño e implentación de Java RMI ... 188
5.6. Resumen ... 189
Ejercicios ... 190

6. SOPORTE DEL SISTEMA OPERATIVO ... 193

6.1. Introducción ... 194
6.2. El nivel de sistema operativo ... 195
6.3. Protección ... 198
6.4. Procesos e hilos ... 199
6.4.1. Espacios de direcciones ... 200
6.4.2. Creación de un proceso nuevo ... 202
6.4.3. Hilos ... 205
6.5. Comunicación e invocación ... 216
6.5.1. Prestaciones de la invocación ... 217
6.5.2. Operación asíncrona ... 224
6.6. Arquitectura del sistema operativo ... 226
6.7. Resumen ... 230
Ejercicios ... 231

7. SEGURIDAD ... 235

7.1. Introducción ... 236
7.1.1. Amenazas y ataques ... 238
7.1.2. Seguridad de las transacciones electrónicas ... 240
7.1.3. Diseño de sistemas seguros ... 242
7.2. Visión general de las técnicas de seguridad ... 244
7.2.1. Criptografía ... 244
7.2.2. Usos de la criptografía ... 245
7.2.3. Certificados ... 248
7.2.4. Control de acceso ... 250
7.2.5. Credenciales ... 252
7.2.6. Cortafuegos ... 254
7.3. Algoritmos criptográficos ... 254

7.3.1. Algoritmos de clave secreta (simétricos) ... 258
7.3.2. Algoritmos de clave pública (asimétricos) ... 261
7.3.3. Protocolos criptográficos híbridos ... 264
7.4. Firmas digitales ... 264
7.4.1. Firmas digitales con claves públicas ... 265
7.4.2. Firmas digitales con claves secretas, MAC ... 266
7.4.3. Funciones de resumen seguro ... 268
7.4.4. Estándares de certificación y autoridades de certificación ... 269
7.5. Práctica de la criptografía ... 270
7.5.1. Prestaciones de los algoritmos cripotográficos ... 271
7.5.2. Aplicaciones de la criptografía y obstáculos políticos ... 271
7.6. Casos de estudio: Needham-Schroeder, Kerberos, SSL y Millicent ... 273
7.6.1. El protocolo de autenticación de Needham y Schroeder ... 273
7.6.2. Kerberos ... 275
7.6.3. Seguridad de transacciones electrónicas con Sockets seguros ... 280
7.6.4. Transacciones electrónicas de pequeño importe: el protocolo Millicent ... 284
7.7. Resumen ... 287
Ejercicios ... 288

8. SISTEMAS DE ARCHIVOS DISTRIBUIDOS ... 291

8.1. Introducción ... 292
8.1.1. Características de los sistemas de archivos ... 294
8.1.2. Requisitos del sistema de archivos distribuidos ... 296
8.1.3. Casos de estudio ... 298
8.2. Arquitectura del servicio de archivos ... 300
8.3. Sistema de archivos en red de Sun (NFS) ... 305
8.4. Sistema de archivos Andrew ... 316
8.4.1. Implementación ... 318
8.4.2. Consistencia de la caché ... 321
8.4.3. Otros aspectos ... 324
8.5. Avances recientes ... 325
8.6. Resumen ... 331
Ejercicios ... 332

9. SERVICIOS DE NOMBRES ... 335

9.1. Introducción ... 336
9.1.1. Nombres, direcciones y otros atributos ... 336
9.2. Servicios de nombres y el sistema de nombres de dominio ... 339
9.2.1. Espacios de nombres ... 340
9.2.2. Resolución de nombres ... 343
9.2.3. El sistema de nombres de dominio ... 346
9.3. Servicios de directorio y descubrimiento ... 352
9.4. Estudio del caso del servicio de nombres global ... 356
9.5. Estudio del caso del servicio de directorio X.500 ... 359
9.6. Resumen ... 363
Ejercicios ... 364

10. TIEMPO Y ESTADOS GLOBALES ... 367

10.1. Introducción ... 368
10.2. Relojes eventos y estados de proceso ... 369
10.3. Sincronización de relojes físicos ... 371
10.3.1. Sincronización en un sistema síncrono ... 372
10.3.2. Método de Cristian para sincronizar relojes ... 373

10.3.3. El algoritmo de Berkeley ... 374
10.3.4. El protocolo del tiempo de red ... 375
10.4. Tiempo lógico y relojes lógicos ... 377
10.5. Estados globales ... 381
10.5.1. Estados globales y cortes consistentes ... 382
10.5.2. Predicados de estado global, estabilidad, seguridad y vitalidad ... 384
10.5.3. El algoritmo de *instantánea* de Chandy y Lamport ... 385
10.6. Depuración distribuida ... 389
10.6.1. Observación de estados globales consistentes ... 390
10.6.2. Evaluando *posiblemente* ϕ ... 392
10.6.3. Evaluando sin duda alguna ϕ ... 393
10.6.4. Evaluando *definitivamente* ϕ y *sin duda alguna* ϕ en sistemas síncronos ... 394
10.7. Resumen ... 395
Ejercicios ... 395

11. COORDINACIÓN Y ACUERDO ... 399

11.1. Introducción ... 400
11.1.1. Suposiciones sobre fallos y detectores de fallos ... 401
11.2. Exclusión mutua distribuida ... 403
11.2.1. Algoritmos para la exclusión mutua ... 404
11.3. Elecciones ... 411
11.4. Comunicación por multidifusión ... 415
11.4.1. Multidifusión básica ... 417
11.4.2. Multidifusión fiable ... 417
11.4.3. Multidifusión ordenada ... 421
11.5. Consenso y sus problemas relacionados ... 428
11.5.1. Definición del modelo del sistema y del problema ... 429
11.5.2. Consenso en un sistema síncrono ... 432
11.5.3. El problema de los generales bizantinos en un sistema síncrono ... 433
11.5.4. Imposibilidad en sistemas asíncronos ... 436
11.6. Resumen ... 439
Ejercicios ... 439

12. TRANSACCIONES Y CONTROL DE CONCURRENCIA ... 443

12.1. Introducción ... 444
12.1.1. Sincronización sencilla (sin transacciones) ... 444
12.1.2. Modelo de fallos para transacciones ... 446
12.2. Transacciones ... 447
12.2.1. Control de concurrencia ... 451
12.2.2. Recuperabilidad de transacciones abortadas ... 455
12.3. Transacciones anidadas ... 457
12.4. Bloqueos ... 459
12.4.1. Bloqueos indefinidos ... 466
12.4.2. Incrementando la concurrencia en esquemas de bloqueo ... 469
12.5. Control optimista de la concurrencia ... 472
12.6. Ordenación por marcas de tiempo ... 476
12.7. Comparación de métodos para el control de concurrencia ... 483
12.8. Resumen ... 484
Ejercicios ... 485

13. TRANSACCIONES DISTRIBUIDAS ... 491

13.1. Introducción ... 492
13.2. Transacciones distribuidas planas y anidadas ... 492

13.2.1. El coordinador de una transacción distribuida ... 493
13.3. Protocolos de consumación atómica ... 495
13.3.1. El protocolo de consumación en dos fases ... 496
13.3.2. Protocolo de consumación en dos fases para transacciones anidadas ... 499
13.4. Control de concurrencia en transacciones distribuidas ... 503
13.4.1. Bloqueo ... 503
13.4.2. Control de concurrencia con ordenación de marcas temporales ... 504
13.4.3. Control de concurrencia optimista ... 505
13.5. Interbloqueos distribuidos ... 506
13.6. Recuperación de transacciones ... 513
13.6.1. Registro histórico ... 515
13.6.2. Versiones sombra ... 518
13.6.3. La necesidad de entradas del estado de la transacción y lista de intenciones en un archivo de recuperación ... 519
13.6.4. Recuperación del protocolo de consumación en dos fases ... 520
13.7. Resumen ... 523
Ejercicios ... 524

14. REPLICACIÓN ... 527

14.1. Introducción ... 528
14.2. Modelo de sistema y comunicación en grupo ... 530
14.2.1. Modelo del sistema ... 530
14.2.2. Comunicación en grupo ... 533
14.3. Servicios tolerantes a fallos ... 538
14.3.1. Replicación pasiva (primario-respaldo) ... 541
14.3.2. Replicación activa ... 543
14.4. Servicios con alta disponibilidad ... 545
14.4.1. La arquitectura cotilla ... 545
14.4.2. El sistema Bayou y la aproximación de la transformación operacional ... 554
14.4.3. El sistema de archivos Coda ... 556
14.5. Transacciones con datos replicados ... 563
14.5.1. Arquitecturas para transacciones replicadas ... 564
14.5.2. Replicación de copias disponibles ... 566
14.5.3. Particiones en la red ... 568
14.5.4. Copias disponibles con validación ... 569
14.5.5. Métodos de consenso con quórum ... 570
14.5.6. El algoritmo de la partición virtual ... 572
14.6. Resumen ... 575
Ejercicios ... 576

15. SISTEMAS MULTIMEDIA DISTRIBUIDOS ... 579

15.1. Introducción ... 580
15.2. Características de los datos multimedia ... 583
15.3. Gestión de la calidad de servicio ... 585
15.3.1. Negociación de la calidad de servicio ... 587
15.3.2. Control de admisión ... 593
15.4. Gestión de recursos ... 594
15.4.1. Planificación de recursos ... 594
15.5. Adaptación de caudales ... 596
15.5.1. Escalado ... 596
15.5.2. Filtrado ... 597
15.6. Caso de estudio: el servidor de vídeo Tiger ... 598
15.7. Resumen ... 602
Ejercicios ... 602

16. MEMORIA COMPARTIDA DISTRIBUIDA ... 605

16.1. Introducción ... 606
16.1.1. DSM frente a paso de mensajes ... 607
16.1.2. Aproximaciones a la implementación de DSM ... 608
16.2. Cuestiones de diseño e implementación ... 610
16.2.1. Estructura ... 610
16.2.2. Modelo de sincronización ... 612
16.2.3. Modelo de consistencia ... 612
16.2.4. Opciones de actualización ... 616
16.2.5. Granularidad ... 617
16.2.6. *Thrashing* (fustigamiento) ... 619
16.3. Consistencia secuencial e Ivy ... 619
16.3.1. El modelo de sistema ... 619
16.3.2. Invalidación de escritura ... 621
16.3.3. Protocolos de invalidación ... 622
16.3.4. Un algoritmo de gestión distribuida dinámico ... 624
16.3.5. *Thrashing* ... 626
16.4. Liberación de consistencia y Munin ... 626
16.4.1. Accesos a memoria ... 627
16.4.2. Consistencia relajada ... 629
16.4.3. Munin ... 630
16.5. Otros modelos de consistencia ... 632
16.6. Resumen ... 633
Ejercicios ... 634

17. EL CASO DE ESTUDIO CORBA ... 637

17.1. Introducción ... 638
17.2. CORBA RMI ... 638
17.2.1. Ejemplo de cliente y servidor CORBA ... 641
17.2.2. La arquitectura de CORBA ... 644
17.2.3. Lenguaje de definición de interfaz de CORBA ... 647
17.2.4. Referencias a objetos remotos en CORBA ... 651
17.2.5. Correspondencia con el lenguaje en CORBA ... 652
17.3. Servicios de CORBA ... 653
17.3.1. El servicio de nombres de CORBA ... 654
17.3.2. Servicio de eventos de CORBA ... 657
17.3.3. Servicio de notificación de CORBA ... 658
17.3.4. Servicio de seguridad de CORBA ... 660
17.4. Resumen ... 661
Ejercicios ... 661

18. ESTUDIO DEL CASO: MACH ... 665

18.1. Introducción ... 666
18.1.1. Objetivos y principales características del diseño ... 667
18.1.2. Repaso de las principales abstracciones de Mach ... 668
18.2. Puertos, nombres y protección ... 669
18.3. Tareas e hilos ... 671
18.4. Modelo de comunicación ... 672
18.4.1. Mensajes ... 672
18.4.2. Puertos ... 674
18.4.3. Mach_msg ... 675
18.5. Implementación de la comunicación ... 676
18.5.1. Gestión transparente de mensajes ... 676

18.5.2. Extensibilidad: protocolos y gestores de dispositivos 678
18.6. Gestión de memoria 678
18.6.1. Estructura del espacio de direcciones 678
18.6.2. Compartición de memoria: herencia y paso de mensajes 679
18.6.3. Evaluación de la copia-en-escritura 680
18.6.4. Paginadores externos 681
18.6.5. Gestión de accesos a un objeto de memoria 682
18.7. Resumen 684
Ejercicios 685

REFERENCIAS 687

ÍNDICE ALFABÉTICO 711

PRÓLOGO

Esta tercera edición de nuestro libro de texto llega en un momento en el que los sistemas distribuidos, particularmente el Web y otras aplicaciones y servicios basados en Internet, aparecen con un vigor sin precedentes por su interés e importancia. El libro pretende proporcionar comprensión, y conocimiento, sobre los principios y la práctica que subyace en el diseño de los sistemas distribuidos, tanto basados en Internet como de cualquier otro tipo. Se proporciona información con suficiente profundidad como para permitir que los lectores evalúen sistemas existentes o diseñen otros nuevos. Los casos de estudios que se detallan esclarecen los conceptos de cada tema importante.

Las técnicas de sistemas distribuidos desarrolladas durante las últimas dos o tres décadas, tales como la comunicación entre procesos y la invocación remota, nomenclatura distribuida, seguridad criptográfica, sistemas de archivos distribuidos, replicación de datos y mecanismos de transacciones distribuidas, proporcionan la infraestructura de ejecución sobre la que se apoyan las aplicaciones de redes de computadoras de hoy en día.

El desarrollo de sistemas distribuidos confía cada vez más en el soporte del *middleware* mediante el uso de marcos de software que proporcionan abstracciones como objetos compartidos y servicios distribuidos, incluyendo comunicación segura, autenticación y control de acceso, código móvil, transacciones y mecanismos de almacenamiento persistente.

En un futuro cercano, las aplicaciones distribuidas posibilitarán una cooperación más inmediata entre los usuarios por medio de datos replicados y flujos de datos multimedia, dando soporte a la movilidad del usuario y de los dispositivos empleando redes inalámbricas y un funcionamiento en red espontáneo.

Los programadores de sistemas y aplicaciones distribuidas se benefician actualmente de un amplio abanico de lenguajes, herramientas y entornos útiles. Lo cual permite que tanto estudiantes como desarrolladores profesionales construyan aplicaciones distribuidas que funcionen.

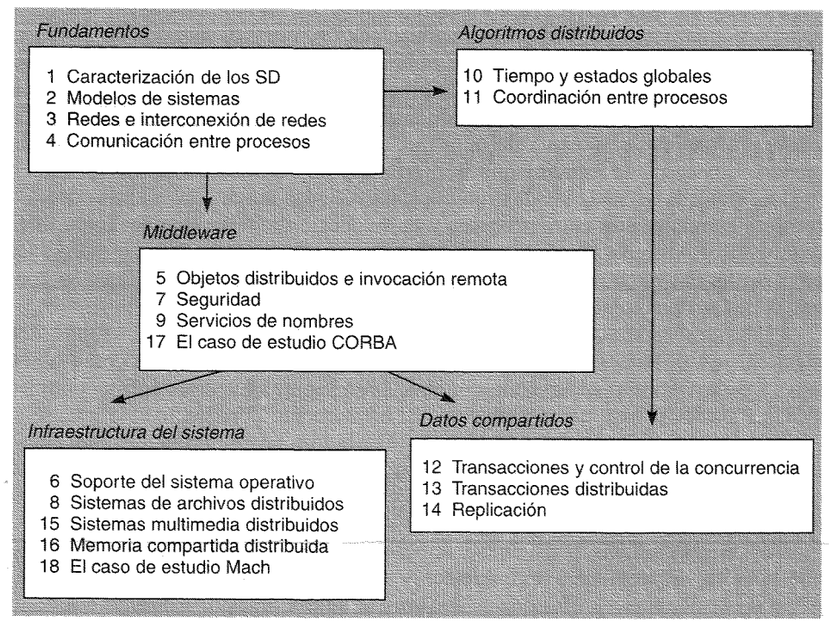
OBJETIVOS Y AUDIENCIA

Se pretende que el libro sea útil en cursos de graduación y de postgraduado de carácter introductorio. También puede utilizarse como texto de autoestudio. En él tomamos una aproximación descendente, mostrando los temas que habrá que resolver en el diseño de los sistemas distribuidos y describiendo aproximaciones acertadas en forma de modelos abstractos, algoritmos y casos de estudio detallados sobre sistemas ampliamente usados. El campo se cubre con suficiente profundidad y amplitud como para permitir a los lectores continuar el estudio de la mayoría de los trabajos de investigación que se publican sobre sistemas distribuidos.

Intentamos que la materia sea suficientemente accesible a los estudiantes que tienen ya un conocimiento básico sobre programación orientada a objetos, sistemas operativos y arquitectura elemental de computadores. El libro incluye aquellos aspectos de redes de computadores que son relevantes a los sistemas distribuidos, incluyendo las tecnologías subyacentes a Internet, y redes de área extendida, área local e inalámbricas. Los algoritmos e interfaces que se presentan a lo largo del libro se encuentran escritos en Java o, en algún caso, ANSI C. Para brevedad y claridad de la presentación, también se utiliza pseudo-código derivado de Java y C.

ORGANIZACIÓN DEL LIBRO

El siguiente diagrama muestra los capítulos del libro bajo las cinco áreas de interés principales. Se pretende que el diagrama proporcione una guía de la estructura del libro e indique también de recorridos alternativos de lectura recomendables para aquellos instructores, y lectores, que deseen lograr una comprensión de los diversos campos del diseño de los sistemas distribuidos:



REFERENCIAS

La existencia del World Wide Web ha cambiado la forma en que cualquier libro como éste debe hacer mención de las fuentes documentales, incluyendo los artículos sobre investigación, especificaciones técnicas y estándares. Muchas de ellas están ahora disponibles en el Web; incluso algunas sólo están disponibles ahí. Por razones de brevedad y legibilidad, emplearemos una forma de referencia especial al material web y que se asemeja lejanamente a un URL: las referencias como [www.omg.org] y [www.rsasecurity.com I] se refieren a documentación disponible en el Web.

Pueden buscarse en la lista de referencias al final de libro, aunque el URL completo sólo se puede obtener en la versión en línea de la lista de referencias en el lugar web del libro: www.cdk3.net/refs, donde están en forma de enlaces vivos. Ambas versiones de la lista de referencias incluyen una explicación más detallada de este esquema.

Capítulos extendidos y reelaborados:	1	Caracterización de los SD
	3	Redes e Interconexión de Redes
	4	Comunicación entre Procesos
	5	Objetos Distribuidos e Invocación Remota
	7	Seguridad
	10	Tiempo y estados globales
	14	Replicación
Capítulos completamente nuevos:	2	Modelos de Sistemas
	11	Coordinación y Acuerdo
	15	Sistemas Multimedia Distribuidos
	17	El Caso de Estudio CORBA
Capítulos actualizados:	6	Soporte de Sistemas Operativos
	8	Sistemas de Archivos Distribuidos
	9	Servicios de Nombres
	12	Transacciones y Control de la Concurrencia
	13	Transacciones Distribuidas
	16	Memoria Compartida Distribuida
	18	El Caso de Estudio Mach

CAMBIOS EN ESTA EDICIÓN

Esta tercera edición aparece aproximadamente seis años después de la segunda. En la tabla superior se resume el trabajo realizado. Los capítulos introductorios y algunos otros han sido modificados profundamente y se han incluido nuevos capítulos para reflejar perspectivas y direcciones novedosas. En otros capítulos se aprecia una profunda reorganización, que afecta a las cuestiones tratadas, la profundidad con que se cubren y la ubicación del material. Hemos condensado el material antiguo para introducir temas nuevos. Algunas materias han sido desplazadas a un plano más prominente para reflejar su nuevo nivel de importancia. Los casos de estudio eliminados de la segunda edición pueden encontrarse en el lugar web del libro como ya se ha comentado.

RECONOCIMIENTOS

Expresamos nuestro agradecimiento a los muchos cursos de estudiantes del College Queen Mary and Westfield College cuyas contribuciones en los seminarios y disertaciones nos han ayudado a producir esta nueva edición, así como las útiles sugerencias de muchos profesores de otros lugares. Kohei Honda probó el material preliminar en sus clases en el QMW y de ello resultaron algunos comentarios particularmente valiosos.

Los que se citan a continuación cedieron su tiempo desinteresadamente para revisar una parte sustancial del borrador final, de lo que se obtuvieron mejoras importantes: Ángel Álvarez, Dave Bakken, John Barton, Simon Boggis, Brent Callaghan, Keith Clarke, Kurt Jensen y Roger Prowse.

Queremos agradecer también a quienes dieron su permiso para emplear su material o sugirieron ciertos temas para su inclusión en este libro: Tom Berson, Antoon Bosselaers, Ralph Herrtwitch,

Frederik Hirsch, Bob Hopgood, Ajay Kshemkalyan, Roger Needham, Mikael Petterson, Rick Schantz y David Wheeler.

Damos las gracias al Departamento de Ciencias de la Computación del Queen Mary and Westfield College, por alojar el sitio web que acompaña al libro y a Keith Clarke y Tom King por su apoyo en llevarlo a cabo.

Finalmente, agradecemos a Keith Mansfield, Birdget Allen, Julie Knight y Kristin Erickson de Pearson Education/Addison-Wesley por su apoyo esencial a lo largo del arduo proceso de publicar este libro.

SITIO WEB

Mantenemos un sitio web con un amplio catálogo de material diseñado para asistir a los profesores y a los lectores. Puede accederse a él mediante cualquiera de las URL:

www.cdk3.net	www.booksites.net/cdkbook

El sitio incluye:

Lista de referencias: La lista de referencias que puede encontrarse al final de los libros se encuentra replicada en el sitio web. La versión web de la lista de referencia incluye enlaces activos del material que está disponible en línea.

Fe de erratas: Una lista de los errores descubiertos en el libro, con su correspondiente corrección.

Material de apoyo: Planeamos mantener un conjunto de material suplementario para cada capítulo. Inicialmente, éste consta de código fuente para los programas del libro y material relevante de lectura que estando presente en la edición previa del libro ha sido eliminado de éste por razones de espacio. Las referencias a este material suplementario aparecen en el libro con enlaces tales como www.cdk3.net/ipc.

Contribuciones al material de enseñanza: Esperamos extender el material suplementario para cubrir nuevos temas según vayan emergiendo durante la vida útil de esta edición. Para lograrlo, invitamos a los profesores a remitirnos material de enseñanza, incluyendo apuntes y proyectos de laboratorio. En el sitio web se describe un procedimiento para remitir el material de apoyo. Las propuestas serán revisadas por un equipo que incluirá varios profesores que emplean este libro como material docente.

Enlaces a sitios web para cursos que emplean el libro: Rogamos que los profesores que lo empleen nos lo notifiquen, con su URL, para incluirlos en la lista.

Guía del instructor: Conteniendo:

- Todos los gráficos y tablas del libro en forma adecuada para su uso como transparencias guía.
- Soluciones a los ejercicios (protegidas por clave y accesibles sólo para los profesores).
- Sugerencias de enseñanza capítulo a capítulo.
- Sugerencias para proyectos de laboratorio.

George Colouris
Jean Dollimore
Tim Kindberg
Londres & Palo Alto, junio 2000
<***authors@cdk3.net***>

1

CARACTERIZACIÓN DE LOS SISTEMAS DISTRIBUIDOS

1.1. Introducción
1.2. Ejemplos de sistemas distribuidos
1.3. Recursos compartidos y Web
1.4. Desafíos
1.5. Resumen

Un sistema distribuido es aquel en el que los componentes localizados en computadores, conectados en red, comunican y coordinan sus acciones únicamente mediante el paso de mensajes. Esta definición lleva a las siguientes características de los sistemas distribuidos: concurrencia de los componentes, carencia de un reloj global y fallos independientes de los componentes.

Proporcionamos tres ejemplos de sistemas distribuidos:

- Internet.
- Una intranet, que es una porción de Internet gestionada por una organización.
- La computación móvil y ubicua.

Compartir recursos es uno de los motivos principales para construir sistemas distribuidos. Los recursos pueden ser administrados por servidores y accedidos por clientes o pueden ser encapsulados como objetos y accedidos por otros objetos clientes. Se analiza el Web como un ejemplo de recursos compartidos y se introducen sus principales características.

Los desafíos que surgen en la construcción de sistemas distribuidos son la heterogeneidad de sus componentes, su carácter abierto, que permite que se puedan añadir o reemplazar componentes, la seguridad y la escalabilidad, que es la capacidad para funcionar bien cuando se incrementa el número de usuarios, el tratamiento de los fallos, la concurrencia de sus componentes y la transparencia.

1.1. INTRODUCCIÓN

Existen redes de computadores en cualquier parte. Una de ellas es Internet, como lo son las muchas redes de las que se compone. Las redes de teléfonos móviles, las redes corporativas, las de las empresas, los campus, las casas, redes dentro del coche, todas, tanto separadas como combinadas, comparten las características esenciales que las hacen elementos importantes para su estudio bajo el título de *sistemas distribuidos*. En este libro se pretenden explicar las características de los computadores en red que deben considerar los diseñadores e implementadores de sistemas y presentar los conceptos y técnicas fundamentales que han sido desarrolladas para ayudar en las tareas de diseño e implementación de sistemas que se basan en dichas características.

Definimos un sistema distribuido como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante red, comunican y coordinan sus acciones sólo mediante paso de mensajes. Esta definición sencilla cubre el rango completo de sistemas en los que se utilizan normalmente computadores en red.

Los computadores que están conectados mediante una red pueden estar separados espacialmente por cualquier distancia. Pueden estar en continentes distintos, en el mismo edificio o en la misma habitación. Nuestra definición de sistemas distribuidos tiene las siguientes consecuencias significativas:

Concurrencia: en una red de computadores, la ejecución de programas concurrentes es la norma. Yo puedo realizar mi trabajo en mi computador, mientras tú realizas tu trabajo en la tuya, compartiendo recursos como páginas web o ficheros, cuando es necesario. La capacidad del sistema para manejar recursos compartidos se puede incrementar añadiendo más recursos (por ejemplo, computadores) a la red. Describiremos formas en las que esta capacidad extra puede ser usada de forma útil, en muchos puntos de este libro. La coordinación de programas que comparten recursos y se ejecutan de forma concurrente es también un tema importante y recurrente.

Inexistencia de reloj global: cuando los programas necesitan cooperar coordinan sus acciones mediante el intercambio de mensajes. La coordinación estrecha depende a menudo de una idea compartida del instante en el que ocurren las acciones de los programas. Pero resulta que hay límites a la precisión con lo que los computadores en una red pueden sincronizar sus relojes, no hay una única noción global del tiempo correcto. Esto es una consecuencia directa del hecho que la única comunicación se realiza enviando mensajes a través de la red. En el Capítulo 10 se describen ejemplos de estos problemas de temporización y soluciones a los mismos.

Fallos independientes: todos los sistemas informáticos pueden fallar y los diseñadores de sistemas tienen la responsabilidad de planificar las consecuencias de posibles fallos. Los sistemas distribuidos pueden fallar de nuevas formas. Los fallos en la red producen el aislamiento de los computadores conectados a él, pero eso no significa que detengan su ejecución. De hecho, los programas que se ejecutan en ellos pueden no ser capaces de detectar cuando la red ha fallado o está excesivamente lenta. De forma similar, la parada de un computador o la terminación inesperada de un programa en alguna parte del sistema (*crash*) no se da a conocer inmediatamente a lo demás componentes con los que se comunica. Cada componente del sistema puede fallar independientemente, permitiendo que los demás continúen su ejecución. Las consecuencias de esta característica de los sistemas distribuidos serán un tema recurrente a lo largo de este libro.

La motivación para construir y utilizar sistemas distribuidos tiene su origen en un deseo de compartir recursos. El término *recurso* es un poco abstracto, pero caracteriza bien el rango de cosas que pueden ser compartidas de forma útil en un sistema de computadores conectados en red. Éste se extiende desde los componentes hardware como los discos y las impresoras hasta las entidades

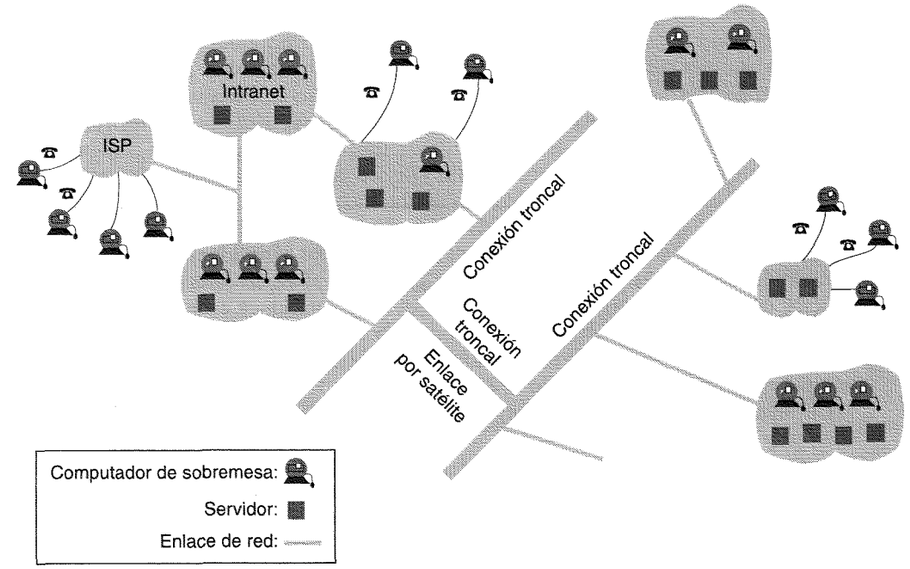


Figura 1.1. Una porción típica de Internet.

de software definidas como ficheros, bases de datos y objetos de datos de todos los tipos. Incluye la secuencia de imágenes que sale de una cámara de vídeo digital y la conexión de audio que representa una llamada de teléfono móvil.

El propósito de este capítulo es transmitir una visión clara de la naturaleza de los sistemas distribuidos y de los retos que deben ser considerados para asegurar que se alcanzan con éxito. La Sección 1.2 presenta algunos ejemplos fundamentales de sistemas distribuidos, los componentes de los que están formados y sus objetivos. La Sección 1.3 explora el diseño de sistemas de recursos compartidos en el contexto del World Wide Web. La Sección 1.4 describe los desafíos fundamentales a los que deben enfrentarse los diseñadores de sistemas distribuidos: heterogeneidad, carácter abierto, seguridad, escalabilidad, gestión de fallos, concurrencia y la necesidad de transparencia.

1.2. EJEMPLOS DE SISTEMAS DISTRIBUIDOS

Nuestros ejemplos están basados en redes de computadores conocidos y utilizados ampliamente: Internet, intranets y la tecnología emergente basada en dispositivos móviles. Se han elegido para proporcionar ejemplos del amplio rango de servicios y aplicaciones que son soportados por redes de computadores y para comenzar la discusión de las cuestiones técnicas que entraña su implementación.

1.2.1. INTERNET

Internet es una vasta colección de redes de computadores de diferentes tipos interconectados. La Figura 1.1 muestra una porción típica de Internet. Programas ejecutándose en los computadores conectados a ella interactúan mediante paso de mensajes, empleando un medio común de comuni-

cación. El diseño y la construcción de los mecanismos de comunicación Internet (los protocolos Internet) es una realización técnica fundamental, que permite que un programa que se está ejecutando en cualquier parte dirija mensajes a programas en cualquier otra parte.

Internet es también un sistema distribuido muy grande. Permite a los usuarios, donde quiera que estén, hacer uso de servicios como el World Wide Web, el correo electrónico, y la transferencia de ficheros (de hecho, a veces se confunde incorrectamente el Web con Internet). El conjunto de servicios es abierto, puede ser extendido por la adición de servidores y nuevos tipos de servicios. La figura nos muestra una colección de intranets, subredes gestionadas por compañías y otras organizaciones. Los proveedores de servicios de Internet (ISP[1]) son empresas que proporcionan enlaces de módem y otros tipos de conexión a usuarios individuales y pequeñas organizaciones, permitiéndolas el acceso a servicios desde cualquier parte de Internet, así como proporcionando servicios como correo electrónico y páginas web. Las intranets están enlazadas conjuntamente por conexiones troncales (*backbones*). Una conexión o red troncal es un enlace de red con una gran capacidad de transmisión, que puede emplear conexiones de satélite, cables de fibra óptica y otros circuitos de gran ancho de banda.

En Internet hay disponibles servicios multimedia, que permiten a los usuarios el acceso a datos de audio y vídeo, incluyendo música, radio y canales de televisión y mantener videoconferencias. La capacidad de Internet para mantener los requisitos especiales de comunicación de los datos multimedia es actualmente bastante limitada porque no proporciona la infraestructura necesaria para reservar capacidad de la red para flujos individuales de datos. En el Capítulo 15 se discute la necesidad de sistemas distribuidos multimedia.

La implementación de Internet y los servicios que mantiene ha implicado el desarrollo de soluciones prácticas para muchas cuestiones de sistemas distribuidos (incluyendo la mayoría de las definidas en la Sección 1.4). Nosotros destacaremos esas soluciones a lo largo del libro, señalando su ámbito y sus limitaciones cuando sea apropiado.

1.2.2. INTRANETS

Una intranet es una porción de Internet que es, administrada separadamente y que tiene un límite que puede ser configurado para hacer cumplir políticas de seguridad local. La Figura 1.2 muestra una intranet típica. Está compuesta de varias redes de área local (LANs) enlazadas por conexiones backbone. La configuración de red de una intranet particular es responsabilidad de la organización que la administra y puede variar ampliamente, desde una LAN en un único sitio a un conjunto de LANs conectadas perteneciendo a ramas de la empresa u otra organización en diferentes países.

Una intranet está conectada a Internet por medio de un encaminador (*router*), lo que permite a los usuarios hacer uso de servicios de otro sitio como el Web o el correo electrónico. Permite también acceder a los servicios que ella proporciona a los usuarios de otras intranets. Muchas organizaciones necesitan proteger sus propios servicios frente al uso no autorizado por parte de usuarios maliciosos de cualquier lugar. Por ejemplo, una empresa no querrá que la información segura esté accesible para los usuarios de organizaciones competidoras, y un hospital no querrá que los datos sensibles de los pacientes sean revelados. Las empresas también quieren protegerse a sí mismas de que programas nocivos, como los virus, entren y ataquen los computadores de la intranet y posiblemente destrocen datos valiosos.

El papel del *cortafuegos* es proteger una intranet impidiendo que entren o salgan mensajes no autorizados. Un cortafuegos se implementa filtrando los mensajes que entran o salen, por ejemplo de acuerdo con su origen o destino. Un cortafuegos podría permitir, por ejemplo, sólo aquellos mensajes relacionados con el correo electrónico o el acceso web para entrar o salir de la intranet que protege.

[1] ISP es el acrónimo de Internet Service Providers.

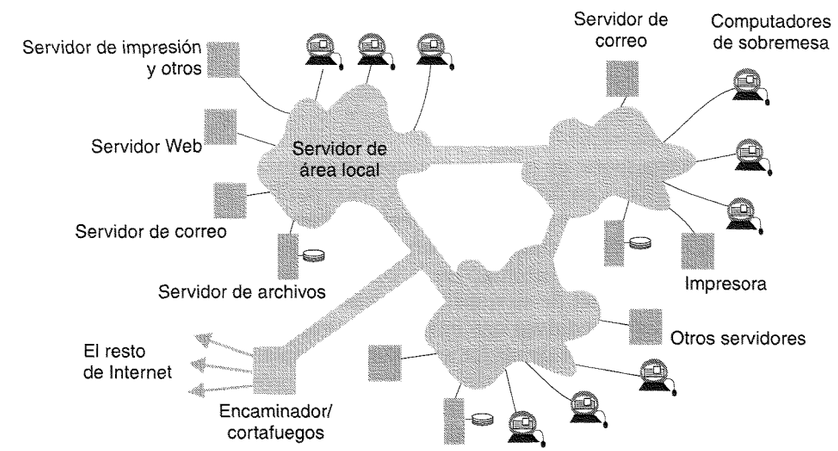


Figura 1.2. Una intranet típica.

Algunas organizaciones no desean conectar sus redes internas a Internet. Por ejemplo, la policía y otras agencias para la seguridad y la vigilancia de la ley prefieren disponer de algunas redes internas que están aisladas del mundo exterior, y el Servicio Nacional de Salud del Reino Unido ha tomado la opción de que los datos médicos sensibles relacionados con los pacientes sólo pueden ser protegidos adecuadamente manteniéndolos en una red interna separada físicamente. Algunas organizaciones militares desconectan sus redes internas de Internet en tiempos de guerra. Pero incluso dichas organizaciones desearán beneficiarse del amplio rango de aplicaciones y software de sistemas que emplean los protocolos de Internet. La solución que se adopta en tales organizaciones es realizar una intranet como se ha indicado, pero sin conexiones a Internet. Tal intranet puede prescindir de cortafuegos, o, dicho de otro modo, dispone del cortafuegos más efectivo posible, la ausencia de cualquier conexión física a Internet.

Los principales temas relacionados con el diseño de componentes para su uso en intranets son:

- Los servicios de ficheros son necesarios para permitir a los usuarios compartir datos, el diseño de éstos se discutirá en el Capítulo 8.
- Los cortafuegos tienden a impedir el acceso legítimo a servicios, cuando se precisa compartir recursos entre usuarios externos e internos, los cortafuegos deben ser complementados con el uso de mecanismos de seguridad más refinados, que se verán en el Capítulo 7.
- El coste de instalación y mantenimiento del software es una cuestión importante. Estos costes pueden ser reducidos utilizando arquitecturas de sistema como redes de computadores y clientes ligeros, descritos en el Capítulo 2.

1.2.3. COMPUTACIÓN MÓVIL Y UBICUA

Los avances tecnológicos en la miniaturización de dispositivos y en redes inalámbricas han llevado cada vez más a la integración de dispositivos de computación pequeños y portátiles en sistemas distribuidos. Estos dispositivos incluyen:

- Computadores portátiles.
- Dispositivos de mano (*handheld*), entre los que se incluyen asistentes digitales personales (PDA), teléfonos móviles, buscapersonas y videocámaras o cámaras digitales.

- Dispositivos que se pueden llevar puestos, como relojes inteligentes con funcionalidad semejante a la de los PDAs.
- Dispositivos insertados en aparatos, como lavadoras, sistemas de alta fidelidad, coches y frigoríficos.

La facilidad de transporte de muchos de estos dispositivos, junto con su capacidad para conectarse adecuadamente a redes en diferentes lugares, hace posible la *computación móvil*. Se llama computación móvil (también llamada *computación nómada* [Kleinrock 1997, www.cooltown.hp.com]) a la realización de tareas de cómputo mientras el usuario está en movimiento o visitando otros lugares distintos de su entorno habitual. En la computación móvil, los usuarios que están fuera de su *hogar* intranet (la intranet de su trabajo o su casa) disponen de la posibilidad de acceder a los recursos mediante los dispositivos que llevan con ellos. Pueden continuar accediendo a Internet; pueden acceder a los recursos de su intranet; y se está incrementando la posibilidad de que utilicen recursos, como impresoras, que están suficientemente próximos a donde se encuentren. Esto último se conoce como *computación independiente de posición*.

Computación ubicua [Weiser 1993] es la utilización concertada de muchos dispositivos de computación pequeños y baratos que están presentes en los entornos físicos de los usuarios, incluyendo la casa, la oficina y otros. El término *ubicuo* está pensado para sugerir que los pequeños dispositivos llegarán a estar tan extendidos en los objetos de cada día que apenas nos daremos cuenta de ellos. O sea, su comportamiento computacional estará ligado con su función física de forma íntima y transparente.

La presencia de computadores en cualquier parte sólo será útil cuando se puedan comunicar entre sí. Por ejemplo, podría ser conveniente para los usuarios controlar su lavadora y su equipo de alta fidelidad desde un dispositivo de *control remoto universal* en su casa. Del mismo modo, la lavadora podría avisar al usuario a través de una tarjeta inteligente o reloj cuando el lavado hubiera finalizado.

La computación ubicua y móvil se solapan, puesto que un usuario moviéndose puede beneficiarse, en principio, de los computadores que están en cualquier parte. Pero, en general, son distintas. La computación ubicua podrá beneficiar a los usuarios mientras permanecen en un entorno sencillo como su casa o un hospital. De forma similar, la computación móvil tiene ventajas si sólo se consideran computadores convencionales y dispositivos como computadores portátiles e impresoras.

La Figura 1.3 muestra a un usuario que está visitando una organización. En la figura se aprecia la intranet de la casa del usuario y la de la organización anfitriona en el lugar que está visitando dicho usuario. Ambas intranets están conectadas al resto de Internet.

El usuario tiene acceso a tres formas de conexión inalámbrica. Su computador portátil tiene un medio para conectarse a la red de área local (LAN) inalámbrica de su anfitrión. Esta red proporciona una cobertura de unos pocos cientos de metros (por ejemplo, una planta de un edificio). Se conecta con el resto de la intranet del anfitrión mediante una pasarela. El usuario dispone además de un teléfono móvil (celular), que está conectado a Internet utilizando el protocolo de acceso inalámbrico (WAP) a través de una pasarela (véase el Capítulo 3). El teléfono da acceso a páginas de información sencilla que se presenta en la pantalla del mismo. Por último, el usuario lleva una cámara digital, que puede comunicar a través de un enlace de infrarrojos cuando apunta al correspondiente dispositivo como una impresora.

Con una infraestructura adecuada del sistema, el usuario puede realizar algunas tareas sencillas en el lugar del anfitrión utilizando los dispositivos que lleva. Mientras viaja hacia el lugar, el usuario puede buscar las últimas cotizaciones en un servidor web utilizando el teléfono móvil. Durante las reuniones con sus anfitriones, el usuario puede mostrarles una fotografía reciente enviándola directamente desde la cámara digital a una impresora adecuada en la sala de la reunión. Esto sólo precisa el enlace de infrarrojos entre la cámara y la impresora. Y pueden enviar un documento

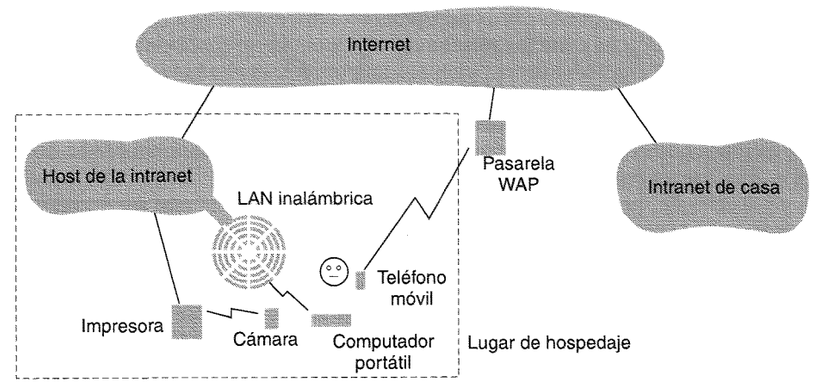


Figura 1.3. Dispositivos portátiles y de mano en un sistema distribuido.

desde un computador portátil a la misma impresora utilizando la red inalámbrica y enlaces cableados de Internet a la impresora.

La computación móvil y ubicua plantea temas significativos sobre el sistema [Milojicic y otros 1999, p. 266, Weiser 1993]. La Sección 2.2.3 presenta una arquitectura para computación móvil y esboza los temas que se plantean desde ella, incluyendo cómo realizar el descubrimiento de recursos en un entorno anfitrión, eliminando la necesidad de que los usuarios reconfiguren sus dispositivos móviles a medida que se mueven, ayudándoles a arreglárselas con la conectividad limitada cuando viajan, y proporcionándoles garantías de privacidad y seguridad a ellos y a los entornos que visitan.

1.3. RECURSOS COMPARTIDOS Y WEB

Los usuarios están tan acostumbrados a los beneficios de compartir recursos que pueden pasar por alto su significado. Normalmente compartimos recursos hardware como impresoras, recursos de datos como ficheros, y recursos con una funcionalidad más específica como máquinas de búsqueda.

Considerado desde el punto de vista de la provisión de hardware, se comparten equipos como impresoras y discos para reducir costes. Pero es mucho más significativo para los usuarios compartir recursos de alto nivel que forman parte de sus aplicaciones y su trabajo habitual y sus actividades sociales. Por ejemplo, los usuarios están preocupados con compartir datos en forma de una base de datos compartida o un conjunto de páginas web, no los discos o los procesadores sobre los que están implementados. Igualmente, los usuarios piensan en términos de recursos como una máquina de búsqueda o un conversor de monedas, sin considerar el servidor o servidores que los proporcionan.

En la práctica, los patrones de compartir recursos varían mucho en su alcance y cuán estrechamente trabajan juntos los usuarios. En un extremo, una máquina de búsqueda en el Web proporciona una función para los usuarios en todo el mundo, los usuarios no necesitan establecer contacto con los demás directamente. En el otro extremo, en un sistema de trabajo *cooperativo mantenido por computador* (CSCW, *computer-supported cooperative working*), un grupo de usuarios que colaboran directamente entre ellos comparte recursos como documentos en un grupo pequeño y cerrado. El patrón de compartir y la distribución geográfica de los usuarios particulares determina qué mecanismos debe proporcionar el sistema para coordinar sus acciones.

Utilizamos el término *servicio* para una parte diferente de un sistema de computadores que gestiona una colección de recursos relacionados y presenta su funcionalidad a los usuarios y aplicaciones. Por ejemplo, accedemos a ficheros compartidos mediante el servicio de ficheros, enviamos documentos a las impresoras a través del servicio de impresión, compramos regalos a través de un servicio de pago electrónico. El único acceso que tenemos al servicio es mediante un conjunto de operaciones que él ofrece. Por ejemplo, un servicio de ficheros proporciona las operaciones de *lectura*, *escritura* y *borrado* en los ficheros.

El hecho de que los servicios limiten el acceso a los recursos a un conjunto bien definido de operaciones es una práctica habitual en la ingeniería de software. Pero también refleja la organización física de los sistemas distribuidos. Los recursos en un sistema distribuido están encapsulados físicamente con los computadores y sólo pueden ser accedidos desde otros computadores a través de comunicación. Para que se compartan de forma efectiva, cada recurso debe ser gestionado por un programa que ofrece una interfaz de comunicación permitiendo que se acceda y actualice el recurso de forma fiable y consistente.

El término *servidor* es probablemente familiar para la mayoría de los lectores. Se refiere a un programa en ejecución (un *proceso*) en un computador en red que acepta peticiones de programas que se están ejecutando en otros computadores para realizar un servicio y responder adecuadamente. Los procesos solicitantes son llamados *clientes*. Las peticiones se envían a través de mensajes desde los clientes al servidor y las contestaciones se envían mediante mensajes desde el servidor a los clientes. Cuando un cliente envía una petición para que se realice una operación, decimos que el cliente *invoca una operación* del servidor. Se llama *invocación remota* a una interacción completa entre un cliente y un servidor, desde el instante en el que el cliente envía su petición hasta que recibe la respuesta del servidor.

El mismo proceso puede ser tanto un cliente como un servidor, puesto que los servidores a veces invocan operaciones en otros servidores. Los términos *cliente* y *servidor* se aplican a los roles desempeñados en una única solicitud. Ambos son distintos, en muchos aspectos, los clientes son activos y los servidores pasivos, los servidores se están ejecutando continuamente, mientras que los clientes sólo lo hacen el tiempo que duran las aplicaciones de las que forman parte.

Hay que señalar que por defecto los términos *cliente* y *servidor* se refieren a *procesos* no a los computadores en las que se ejecutan, aunque en lenguaje coloquial dichos términos se refieren también a los propios computadores. Otra distinción que se discutirá en el Capítulo 5, es que en un sistema distribuido escrito en un lenguaje orientado a objetos, los recursos pueden ser encapsulados como objetos y accedidos por objeto clientes, en cuyo caso hablaremos de un *objeto cliente* que invoca un método en un *objeto servidor*.

Muchos sistemas distribuidos, aunque no todos, pueden ser construidos completamente en forma de clientes y servidores que interaccionan. El World Wide Web, el correo electrónico y las impresoras en red concuerdan con esta modelo. En el Capítulo 2 se verán alternativas a los sistemas cliente-servidor.

Un navegador (*browser*) es un ejemplo de cliente. El navegador se comunica con el servidor web para solicitarle páginas. A continuación se examina el Web con más detalle.

1.3.1. EL WORLD WIDE WEB

El World Wide Web [www.w3.org I, Berners-Lee 1991] es un sistema en evolución para publicar y acceder a recursos y servicios a través de Internet. Utilizando el software de un navegador web, fácilmente disponible como Netscape o Internet Explorer, los usuarios utilizan el Web para recuperar y ver documentos de muchas clases, para escuchar secuencias de audio y ver secuencias de vídeo, y para interaccionar con un conjunto ilimitado de servicios.

El Web comenzó su vida en el centro europeo para la investigación nuclear (CERN), Suiza, en 1989 como un vehículo para el intercambio de documentos entre una comunidad de físicos, conectados a Internet [Berners-Lee 1999]. Una característica fundamental del Web es que proporciona una estructura *hipertexto* entre los documentos que almacena, reflejando los requisitos de los usuarios para organizar su conocimiento. Esto significa que los documentos tienen *enlaces*, referencias a otros documentos y recursos, también almacenados en la red.

Es fundamental para la experiencia del usuario en el Web que cuando encuentra una imagen determinada o una parte de texto en un documento, esto estará acompañado de enlaces a documentos relacionados y otros recursos. La estructura de los enlaces puede ser arbitrariamente compleja y el conjunto de recursos que puede ser añadido es ilimitado, la «telaraña» (web) de enlaces está por consiguiente repartida por todo el mundo (world-wide). Bush [1945] concibió las estructuras hipertextuales hace 50 años; el desarrollo de Internet fue lo que propició la manifestación de esta idea en una escala mundial.

El Web es un sistema *abierto*: puede ser ampliado e implementado en nuevas formas sin modificar su funcionalidad existente (véase la Sección 1.4.2). Primero, su operación está basada en estándares de comunicación y en documentos estándar que están publicados libremente e implementados ampliamente. Por ejemplo, existen muchos tipos de navegador, cada uno de ellos implementados en muchos casos sobre diferentes plataformas; y existen muchas implementaciones de servidores web. Cualquier navegador conforme puede recuperar recursos de cualquier servidor conforme. Por lo tanto los usuarios pueden tener acceso a los navegadores en la mayoría de los dispositivos que utilizan, desde un PDA a computadores portátiles.

Segundo, el Web es abierto respecto a los tipos de recursos que pueden ser publicados y compartidos en él. En su forma más simple, un recurso es una página web o algún otro tipo de *contenido* que puede ser almacenado en un fichero y presentado al usuario, como ficheros de programa, de imágenes, de sonido y documentos en formato PostScript o PDF. Si alguien inventa, por ejemplo, un nuevo formato de almacenamiento de imágenes, las imágenes en dicho formato pueden ser publicadas inmediatamente en el Web. Los usuarios necesitan un medio de ver imágenes en este nuevo formato, pero los navegadores están diseñados para acomodar la nueva funcionalidad de presentación en forma de aplicaciones *colaboradoras* y conectores (*plug-ins*).

El Web se ha desarrollado mas allá de estos recursos de datos sencillos para abarcar servicios como la compra electrónica de regalos. Ha evolucionado sin cambiar su arquitectura básica. El Web está basado en tres componentes tecnológicos de carácter estándar básicos:

- El lenguaje de etiquetado de hipertexto (HTML, *Hypertext Markup Language*) es un lenguaje para especificar el contenido y el diseño de las páginas que son mostradas por los navegadores.
- Localizadores Uniformes de Recursos (URL, *Uniform Resource Locator*) que identifican documentos y otros recursos almacenados como parte del Web. El Capítulo 9 examina otros identificadores web.
- Una arquitectura de sistema cliente-servidor, con reglas estándar para interacción (el protocolo de transferencia hipertexto-HTTP, *HyperText Transfer Protocol*) mediante la cual los navegadores y otros clientes obtienen documentos y otros recursos de los servidores web. La Figura 1.4 muestra algunos servidores web, y navegadores que les hacen peticiones. Una característica importante es que los usuarios puedan localizar y gestionar sus propios servidores web en cualquier parte de Internet.

A continuación se detallan estos componentes y se explica la operación de los navegadores y servidores web cuando un usuario localiza páginas web y selecciona los enlaces que contiene.

◇ **HTML**. El lenguaje de etiquetado de hipertexto [www.w3.org II] se utiliza para especificar el texto e imágenes que forman el contenido de una página web, y para especificar cómo serán formateados para la presentación al usuario. Una página web contiene elementos estructurados como cabeceras, párrafos, tablas e imágenes. HTML se utiliza también para especificar enlaces y qué recursos están asociados con ellos.

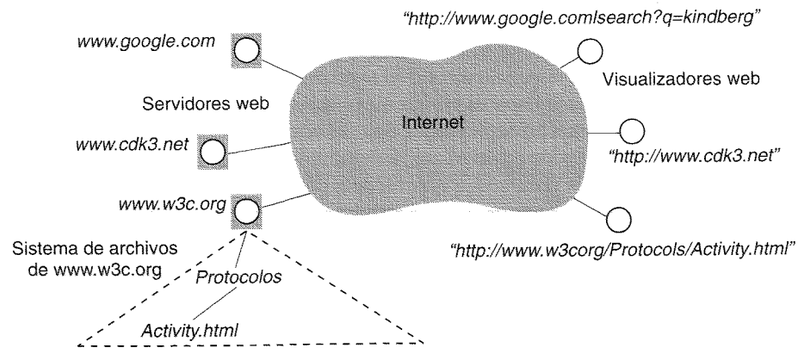


Figura 1.4. Servidores web y visualizadores web.

Los usuarios producen HTML a mano, utilizando un editor de texto estándar, o pueden utilizar a un editor *wysiwyg*[2] que genera HTML a partir de un diseño que pueden crear gráficamente. Un ejemplo típico de texto HTML puede ser:

<IMG SRC = «http://www.cdk3.net/WebExample/Images/earth.jpg»> 1
<P> 2
¡Bienvenido a la Tierra! Los visitantes pueden estar interesados también en echar un vistazo a la 3
<A HREF = «http://www.cdk3.net/WebExample/moon.html»>Luna</A> 4
<P> 5
(etcétera) 6

Este texto HTML se almacena en un fichero al que puede acceder un servidor web, supongamos que es el fichero *earth.html*. Un navegador recupera el contenido de este fichero desde un servidor web, en este caso un servidor en un computador llamado *www.cdk3.net*. El navegador lee el contenido devuelto por el servidor y lo organiza en texto formateado y en imágenes presentadas en una página web en la forma habitual. Sólo el navegador, no el servidor, interpreta el texto HTML. Pero el servidor debe informar al navegador sobre el tipo de contenido que devuelve, para distinguirlo de, por ejemplo, un documento en PostScript. El servidor puede deducir el tipo de contenido de la extensión del fichero *.html*.

Hay que señalar que las directivas HTML, conocidas como *etiquetas*, están encerradas entre ángulos como *<P>*. La línea 1 del ejemplo identifica un fichero que contiene una imagen para la presentación. Su URL es *http://www.cdk3.net/WebExample/Images/earth.jpg*. Las líneas 2 y 5 son una directiva de comienzo de un nuevo párrafo cada una. Las líneas 3 y 6 contienen texto que será mostrado por el navegador en el formato estándar de párrafo.

La línea 4 especifica un enlace en la página. Contiene la Palabra «Luna» rodeada por dos etiquetas HTML relacionadas *<A HREF...>* y *</A>*. El texto comprendido entre dichas etiquetas es lo que aparece en el enlace tal como se presenta en la página web. La mayoría de los navegadores están configurados para mostrar el texto de los enlaces subrayado, así que lo que el usuario verá en el párrafo es:

¡Bienvenido a la Tierra! Los visitantes pueden estar interesados en echar un vistazo a la Luna.

[2] wysiwyg es el acrónimo de la frase inglesa «what you see is what you get» lo que se ve es lo que se obtiene (*N. del T.*).

El navegador registra la asociación entre el texto mostrado del enlace y el URL contenido en la etiqueta *<A HREF...>*, en este caso:

http://www.cdk3 net/WebExample/moon.html

Cuando el usuario selecciona el texto, el navegador localiza el recurso identificado por el correspondiente URL y se lo presenta. En el ejemplo, el recurso es un fichero HTML que especifica una página sobre la Luna.

◇ **URLs.** El propósito de un URL [www.w3.org III] es identificar un recurso de tal forma que permita al navegador localizarlo. Los navegadores examinan los URLs con el fin de buscar los recursos correspondientes de los servidores web. A veces el usuario teclea un URL en el navegador. Habitualmente, el navegador busca el URL correspondiente cuando el usuario hace clic en un enlace o selecciona uno de sus enlaces anotados (*bookmarks*), o cuando el navegador busca un recurso insertado en una página web, como una imagen.

Cada URL, en su forma global, tiene dos componentes:

esquema: *localización-específica-del-esquema.*

El primer componente, el *esquema*, declara qué tipo de URL es. Se precisan los URLs para especificar posiciones de una variedad de recursos y también para especificar una variedad de protocolo de comunicación para recuperarlos. Por ejemplo, *mailto:joe@anISP.net* identifica la dirección de correo de un usuario; *ftp://ftp.downloadIt.com/software/aProg.exe* identifica un fichero que será recuperado utilizando el Protocolo de Transferencia de Ficheros (FTP, *File Transfer Protocol*) en lugar de HTTP, utilizado con más frecuencia. Otros ejemplos de esquemas son *nntp* (utilizado para especificar un grupo de noticias de Usenet), y *telnet* (utilizado para hacer log in en un computador).

El Web es abierto con respecto a los tipos de recursos que pueden ser utilizados para acceder, mediante los designadores de esquema de los URLs. Si alguien inventa un nuevo tipo de recurso llamémoslo *artefacto*, quizá con su propio esquema de direccionamiento para localizar arterfactos y su propio protocolo para acceder a ellos, entonces la gente puede comenzar a utilizar URLs de la forma *artefacto*:... Como es lógico, los navegadores deben disponer de la capacidad para utilizar el nuevo protocolo *artefacto*, pero esto se puede conseguir añadiendo una aplicación colaboradora o un conector.

Los URLs de HTTP son los más utilizados para localizar recursos utilizando el protocolo estándar HTTP. Un URL HTTP tiene dos tareas importantes que hacer: identificar qué servidor Web mantiene el recurso, e identificar cuál de los recursos del servidor es el solicitado. En la Figura 1.4 se ven tres servidores haciendo peticiones por recursos gestionados por tres servidores web. El web en la parte superior está realizando una consulta a una máquina de búsqueda. El del medio solicita la página por defecto de otro sitio web. El de la parte inferior solicita una página que está especificada completamente, con la inclusión de un nombre de recorrido relativo para el servidor. Los ficheros para un servidor web determinado se localizan en uno o más subdirectorios del sistema de ficheros del servidor, y cada recurso es identificado por el nombre del recorrido del fichero (*path name*) relativo al servidor.

En general, los URLs de HTTP son de la forma:

http://nombredelservidor [:puerto] [/nombredelpathdelservidor] [?argumentos]

en el que los elementos entre corchetes son opcionales. Un URL de HTTP siempre comienza con *«http://»* seguido por un nombre del servidor, expresado como un nombre del Servicio de Nombres de Dominio (DNS, *Domain Name Service*) (véase la Sección 9.2). El nombre DNS del servidor

está seguido opcionalmente por el nombre del «puerto» en el que el servidor escucha las solicitudes (véase el Capítulo 4). Después viene un nombre de recorrido opcional del recurso del servidor. Si éste no aparece se solicita la página web por defecto del servidor. Por último, el URL finaliza opcionalmente con un conjunto de argumentos, por ejemplo, cuando un usuario envía las entradas en una forma como una página de consulta de una máquina de búsqueda.

Considérense los URLs:

http://www.cdk3.net/
http://www.w3.org/Protocols/Activity.html
http://www.google.com/search?q=kindberg

Pueden ser separados de la forma siguiente:

Nombre del servidor de DNS	*Ruta en el servidor*	*Argumentos*
www.cdk3.net	(por defecto)	(ninguno)
www.w3.org	Protocols/Activity.html	(ninguno)
www.google.com	search	q=kindberg

El primer URL indica la página por defecto proporcionada por *www.cdk3.net*. El siguiente identifica un fichero en el servidor *www.w3.org*, cuyo nombre de recorrido es *Protocols/Activity.html*. El tercer URL especifica una consulta a una máquina de búsqueda. El recorrido identifica un programa llamado *search* y la cadena de caracteres después del carácter *?* codifica los argumentos para este programa, en este caso especifica la cadena de búsqueda. Se discutirán los URL que indican programas con más detalle cuando se analicen después características más avanzadas.

Los lectores pueden haber observado también la presencia de un ancla al final de un URL, un nombre procedido por un # como *#referencias*, que indica un punto dentro de un documento. Las anclas no son parte de los URLs sino que están definidas como una parte de la especificación HTML. Únicamente los navegadores interpretan anclas, para colocar en pantalla páginas web a partir de un punto determinado. Los navegadores siempre recuperan páginas web completas de los servidores, no partes de ellas indicadas por anclas.

Publicación de un recurso: Mientras el Web tiene un modelo claro para recuperar un recurso a partir de su URL, el método para publicar un recurso en el Web todavía es difícil de manejar y normalmente precisa intervención humana. Para publicar un recurso en el Web, un usuario debe colocar, en primer lugar, el correspondiente fichero en un directorio al que pueda acceder el servidor web. Conocido el nombre de un servidor *S* y un nombre de recorrido *P* que el servidor puede reconocer, el usuario puede construir el URL de la forma *http://S/P*. El usuario coloca este URL en un enlace de un documento existente o distribuye el URL a otros usuarios, por ejemplo mediante correo electrónico.

Existen unas ciertas convenciones para nombres de recorrido que los servidores reconocen. Por ejemplo, un nombre de recorrido comenzando por *~juan* está por convención en un subdirectorio *public-html* del usuario juan. De forma similar, un nombre de recorrido que termina en un nombre de directorio en lugar de un único fichero se refiere a un fichero en ese directorio llamado *index.html*.

Huang y otros [2000] proporcionan un modelo para insertar contenido en el Web con la mínima intervención humana. Esto es particularmente relevante allí donde los usuarios necesitan extraer contenido de una variedad de dispositivos, como cámaras, para su publicación en páginas web.

◇ **HTTP.** El protocolo de transferencia hipertexto [www.w3.org IV] define las formas en las que los navegadores y otros tipos de clientes interaccionan con los servidores web. En el Capítulo 4 se verá HTTP con más detalle, pero aquí se esbozan sus principales características (restringiendo la discusión a la recuperación de recursos en ficheros):

Interacciones petición-respuesta: HTTP es un protocolo de petición-respuesta. El cliente envía un mensaje de petición al servidor que contiene el URL del recurso solicitado. (El servidor sólo precisa la parte del URL que sigue al propio nombre DNS del servidor.) El servidor localiza el nombre de recorrido y, si existe, devuelve el contenido del fichero en un mensaje de respuesta al cliente. En caso contrario, devuelve un mensaje de error.

Tipos de contenido: los navegadores no son necesariamente capaces de manejar o hacer buen uso de cualquier tipo de contenido. Cuando un navegador hace una petición, incluye una lista de los tipos de contenido que prefiere, por ejemplo, en principio puede ser capaz de sacar en pantalla imágenes en formato *GIF* pero no en *JPEG*. El servidor puede ser capaz de tener esto en cuenta cuando devuelve el contenido al navegador. El servidor incluye el tipo de contenido en el mensaje de respuesta de forma que el navegador sabrá cómo procesarlo. Las cadenas de caracteres que indican el tipo de contenido se llaman tipos MIME, y están estandarizados en el RFC 1521 [Borenstein y Freed 1993]. Por ejemplo, si el contenido es de tipo *text/html* entonces el navegador interpretará el texto como HTML y lo mostrará en pantalla; si el contenido es de tipo *image/GIF* el navegador lo tratará como una imagen en formato *GIF*, si el contenido es de tipo *aplication/zip* entonces los datos están comprimidos en formato *zip* y el navegador lanzará una aplicación externa para descomprimirlos. El conjunto de acciones que un navegador tomará para un tipo de contenido dado es configurable, y los lectores deben preocuparse de comprobar estos defectos en sus propios navegadores.

Un recurso por solicitud: en la versión 1.0 de HTTP (que es la versión más utilizada en el momento en que se escribe esto), el cliente solicita un recurso por cada petición HTTP. Si una página web contiene nueve imágenes, por ejemplo, el navegador realizará un total de diez peticiones separadas para obtener el contenido completo de la página. Los navegadores normalmente hacen varias peticiones concurrentemente, para reducir el retardo total para el usuario.

Control de acceso simple: por defecto, cualquier usuario con una conexión de red a un servidor web puede acceder a cualquiera de los recursos publicados. Si los usuarios desean restringir el acceso a un recurso, pueden configurar el servidor para plantear un desafío a cualquier usuario que lo pida. Los usuarios correspondientes deben probar entonces que tienen derecho para acceder al recurso, por ejemplo tecleando una contraseña (*password*).

◇ **Características más avanzadas, servicios y páginas dinámicas.** Hasta ahora hemos descrito cómo los usuarios pueden publicar páginas Web y otros contenidos almacenados en ficheros en el Web. El contenido puede cambiar en el tiempo, pero es lo mismo para cualquiera. Sin embargo, mucha de la experiencia de los usuarios del Web es la de los servicios con los que el usuario puede interaccionar. Por ejemplo, cuando se compra algo en una tienda electrónica, el usuario rellena con frecuencia un *formulario web* para proporcionar sus detalles personales o para especificar exactamente lo que se desea comprar. Un formulario web es una página que contiene instrucciones para el usuario y elementos para la introducción de datos como campos de texto y cajas de comprobación. Cuando un usuario envía el formulario (normalmente pulsando un botón o una tecla de *retorno* (*return*), el navegador envía una petición HTTP a un servidor web, que contiene los valores enviados por el usuario.

Puesto que el resultado de la petición depende de los datos introducidos por el usuario, el servidor debe *procesar* dichos datos. Por tanto, el URL o su componente inicial representa un *programa* en el servidor, no un archivo. Si los datos introducidos por el usuario son pocos, entonces son enviados como el componente final del URL, siguiendo a un carácter *?* (en el resto de los casos son

enviados como datos adicionales en la petición). Por ejemplo, una solicitud que contenga el URL siguiente llama a un programa llamado *search* en www.google.com y especifica una consulta para *Kindberg*: *http://www.google.com/search?q=kindberg*.

El programa «search» produce como resultado texto HTML, y el usuario verá una lista de páginas que contienen la palabra *kindberg*. (El lector puede realizar una consulta en su buscador favorito y darse cuenta del URP que el navegador saca en pantalla cuando se devuelve el resultado.) El servidor devuelve el texto HTML que genera el programa del mismo modo que si hubiera sido obtenido de un fichero. Dicho de otro modo, la diferencia entre el contenido estático localizado en un fichero y el contenido que es generado dinámicamente es transparente para el navegador.

Un programa que se ejecuta en los servidores web para generar contenido para sus clientes se suele llamar, a menudo, programa de Interfaz de Pasarela Común (CGI, *Common Gateway Interface*). Un programa CGI puede tener una funcionalidad específica de la aplicación, ya que puede analizar los argumentos que el cliente le proporciona y producir un resultado del tipo requerido (normalmente texto HTML). El programa consultará o modificará una base de datos cuando procesa la solicitud.

Código descargado: Un programa CGI se ejecuta en el servidor. A veces los diseñadores de servicios Web precisan algún código relacionado con el servicio para ejecutar en el navegador, en el computador del usuario. Por ejemplo, código escrito en Javascript [www.netscape.com] se descarga a menudo con un formulario web para proporcionar una interacción con el usuario de mejor calidad que la proporcionada por los artefactos estándar de HTML. Una página mejorada con Javascript puede dar al usuario información inmediata sobre entradas inválidas (en lugar de forzar al usuario a comprobar los valores en el servidor, lo que precisaría mucho más tiempo). Javascript puede ser utilizado también para modificar partes del contenido de una página web sin que sea preciso traer una nueva versión completa de la página y reformatearla.

Javascript tiene una funcionalidad bastante limitada. Como contraste, un *applet* es una pequeña aplicación que descarga automáticamente el navegador y se ejecuta cuando se descarga la página correspondiente. Los applets pueden acceder a la red y proporcionar interfaces de usuario específicas, utilizando las posibilidades del lenguaje, Java [java.sun.com, Flanagan 1997]. Por ejemplo, aplicaciones de tipo «chat» están implementadas, a veces, como applets que corren en los navegadores de los usuarios, junto con un programa de servidor. Los applets envían el texto al que lo distribuye a todos los applets para su presentación al usuario. Se discutirán los applets con más detalle en la Sección 2.2.3.

◇ **Discusión sobre el Web.** El extraordinario éxito del Web se basa en la facilidad con la que pueden publicarse recursos, una estructura de hipertexto apropiada para la organización de muchos tipos de información, y la extensibilidad arquitectónica del sistema. Los estándares en que se basa su arquitectura son simples y fueron difundidos ampliamente desde un principio. Esto ha posibilitado que se integraran muchos tipos nuevos de recursos y servicios.

El éxito del Web contradice algunos principios de diseño. Primero, su modelo de hipertexto es deficiente en algunos aspectos. Si se borra o mueve algún recurso, ocurre que los llamados enlaces *descolgados* a este recurso permanecen, causando cierta frustración a los usuarios. También aparece el problema habitual de los usuarios *perdidos en el hiperespacio*. Los usuarios a menudo se encuentran a sí mismos siguiendo confusamente un montón de vínculos, que apuntan a páginas de una colección dispar de enlaces, en algunos casos de dudosa fiabilidad. Los motores de búsqueda son un complemento útil para seguir enlaces con el fin de buscar información en el Web, pero son claramente imperfectos a la hora degenerar lo que quiere concretamente el usuario. Una aproximación a este problema, ejemplificada en el Marco de Descripción de Recursos (*Resource Description Framework*) [www.w3.org V], sería estandarizar el formato de metadatos acerca de los recursos web. Los metadatos describirían los atributos de los recursos web, y serían leídos por herramientas que asistirían a los usuarios que procesaran recursos web *masivamente*, tal como ocurre al buscar y recopilar listas de enlaces relacionados.

Existe una necesidad creciente de intercambio de muchos tipos de datos estructurados en el Web, pero HTML se encuentra limitado en que no es extensible a aplicaciones más allá de la «inspección» de la información. HTML consta de un conjunto estático de estructuras tales como los párrafos, que se limitan a indicar la forma en que se presentan los datos a los usuarios. Más recientemente se ha diseñado el Lenguaje de Marcado Extensible (*Extensible Markup Language*, XML) [www.w3.org VI] como medio de representar datos en formularios estándar, estructurados, y específicos para cada aplicación. Pongamos por ejemplo que XML se emplee para describir las características de ciertos dispositivos y para describir información personal almacenada acerca de los usuarios. XML es un metalenguaje de descripción de datos, lo cual hace que los datos sean intercambiables entre aplicaciones. El Lenguaje Extensible de Hojas de Estilo (*Extensible Stylesheet Language*, XSL) [www.w3.org VII] se emplea para declarar cómo serán presentados a los usuarios los datos almacenados en el formato XML. Por ejemplo, empleando dos hojas de estilo diferentes, la misma información sobre un usuario concreto podría presentarse en una página web bien gráficamente bien como una simple lista.

Como arquitectura del sistema, el Web plantea problemas de escala. Los servidores web más populares pueden experimentar muchos *accesos* por segundo, y como resultado la respuesta a los usuarios se ralentiza. El Capítulo 2 describe el empleo de memorias (caché) en los visualizadores y de servidores *proxy* para aliviar estos efectos. A pesar de ello la arquitectura cliente-servidor del Web implica que no hay medios eficientes para mantener a los usuarios al día con las últimas versiones de las páginas. Los usuarios tienen que presionar el botón de *recarga* de su navegador para asegurar que poseen la última información, y éstos se ven forzados a comunicarse con los servidores para comprobar si la copia local del recurso es válida aún.

Finalmente, una página web no siempre es una interfaz de usuario satisfactoria. Los elementos de diálogo definidos para HTML son limitados, y los diseñadores incluyen a menudo en la páginas web, pequeñas aplicaciones, o applets, o bien muchas imágenes para darles una función y apariencia más aceptable. Consecuentemente el tiempo de carga se incrementa.

1.4. DESAFÍOS

Los ejemplos de la Sección 1.2 pretenden ilustrar el alcance de los sistemas distribuidos y sugerir las cuestiones que aparecen en su diseño. Aunque se encuentran sistemas distribuidos por todas partes, su diseño es aún bastante simple y quedan todavía grandes posibilidades de desarrollar servicios y aplicaciones más ambiciosas. Muchos de los desafíos que se discuten en esta sección están ya resueltos, pero los futuros diseñadores necesitan estar al tanto y tener cuidado de considerarlas.

1.4.1. HETEROGENEIDAD

Internet permite que los usuarios accedan a servicios y ejecuten aplicaciones sobre un conjunto heterogéneo de redes y computadores. Esta heterogeneidad (es decir, variedad y diferencia) se aplica a todos los siguientes elementos:

- Redes.
- Hardware de computadores.
- Sistemas operativos.
- Lenguajes de programación.
- Implementaciones de diferentes desarrolladores.

A pesar de que Internet consta de muchos tipos de redes diferentes (como vimos en la Figura 1.1), sus diferencias se encuentran enmascaradas dado que todos los computadores conectados a éste utilizan los protocolos de Internet para comunicarse una con otra. Por ejemplo, un computador conectado a Ethernet tiene una implementación de los protocolos de Internet sobre Ethernet, así un computador en un tipo de red diferente necesitará una implementación de los protocolos de Internet para esa red. El Capítulo 3 explica cómo se implementan los protocolos de Internet sobre una variedad de redes diferentes.

Los tipos de datos, como los enteros, pueden representarse de diferente forma en diferentes clases de hardware por ejemplo, hay dos alternativas para ordenar los bytes en el caso de los enteros. Hay que tratar con estas diferencias de representación si se va a intercambiar mensajes entre programas que se ejecutan en diferente hardware.

Aunque los sistemas operativos de todas los computadores de Internet necesitan incluir una implementación de los protocolos de Internet, no todas presentan necesariamente la misma interfaz de programación para estos protocolos. Por ejemplo, las llamadas para intercambiar mensajes en UNIX son diferentes de las llamadas en Windows NT.

Lenguajes de programación diferentes emplean representaciones diferentes para caracteres y estructuras de datos como cadenas de caracteres y registros. Hay que tener en cuenta estas diferencias si queremos que los programas escritos en diferentes lenguajes de programación sean capaces de comunicarse entre ellos.

Los programas escritos por diferentes programadores no podrán comunicarse entre sí a menos que utilicen estándares comunes, por ejemplo para la comunicación en red y la representación de datos elementales y estructuras de datos en mensajes. Para que esto ocurra es necesario concertar y adoptar estándares (como así lo son los protocolos de Internet).

◇ **Middleware.** El término *middleware* se aplica al estrato software que provee una abstracción de programación, así como un enmascaramiento de la heterogeneidad subyacente de las redes, hardware, sistemas operativos y lenguajes de programación. CORBA, el cual se describe en los Capítulos 4, 5 y 17, es un ejemplo de ello. Algún middleware, como Java RMI (véase el Capítulo 5) sólo se soporta en un único lenguaje de programación. La mayoría de middleware se implementa sobre protocolos de Internet, enmascarando éstos la diversidad de redes existentes. Aun así cualquier middleware trata con las diferencias de sistema operativo y hardware. El cómo se obtiene esto es el tema principal del Capítulo 4.

Además de soslayar los problemas de heterogeneidad, el middleware proporciona un modelo computacional uniforme al alcance de los programadores de servidores y aplicaciones distribuidas. Los posibles modelos incluyen invocación sobre objetos remotos, notificación de eventos remotos, acceso remoto mediante SQL y procesamiento distribuido de transacciones. Por ejemplo, CORBA proporciona invocación sobre objetos remotos, lo que permite que un objeto en un programa en ejecución en un computador invoque un método de un objeto de un programa que se ejecuta en otro computador. La implementación oculta el hecho de que los mensajes se transmiten en red en cuanto al envío de la petición de invocación y su respuesta.

◇ **Heterogeneidad y código móvil.** El término *código móvil* se emplea para referirse al código que puede ser enviado desde un computador a otro y ejecutarse en éste, por eso los applets de Java son un ejemplo de ello. Dado que el conjunto de instrucciones de un computador depende del hardware, el código de nivel de máquina adecuado para correr en un tipo de computador no es adecuado para ejecutarse en otro tipo. Por ejemplo, los usuarios de PC envían a veces archivos ejecutables agregados a los correos electrónicos para ser ejecutados por el destinatario, pero el receptor bien pudiera no ser capaz de ejecutarlo, por ejemplo, sobre un Macintosh o un computador con Linux.

La aproximación de *máquina virtual* provee un modo de crear código ejecutable sobre cualquier hardware: el compilador de un lenguaje concreto generará código para una máquina virtual en lugar de código apropiado para un hardware particular, por ejemplo el compilador Java produce código para la máquina virtual Java, la cual sólo necesita ser implementada una vez para cada tipo de máquina con el fin de poder lanzar programas Java. Sin embargo, la solución Java no se puede aplicar de modo general a otros lenguajes.

1.4.2. EXTENSIBILIDAD

La extensibilidad de un sistema de cómputo es la característica que determina si el sistema puede ser extendido y reimplementado en diversos aspectos. La extensibilidad de los sistemas distribuidos se determina en primer lugar por el grado en el cual se pueden añadir nuevos servicios de compartición de recursos y ponerlos a disposición para el uso por una variedad de programas cliente.

No es posible obtener extensibilidad a menos que la especificación y la documentación de las interfaces software clave de los componentes de un sistema estén disponibles para los desarrolladores de software. Es decir, que las interfaces clave estén *publicadas*. Este procedimiento es similar a una estandarización de las interfaces, aunque a menudo puentea los procedimientos oficiales de estandarización, que por lo demás suelen ser lentos y complicados.

Sin embargo, la publicación de interfaces sólo es el punto de arranque de la adición y extensión de servicios en un sistema distribuido. El desafío para los diseñadores es hacer frente a la complejidad de los sistemas distribuidos que constan de muchos componentes diseñados por personas diferentes.

Los diseñadores de los protocolos de Internet presentaron una serie de documentos denominados «Solicitudes de Comentarios» (*Request For Comments*), o RFC, cada una de las cuales se conoce por un número.

Las especificaciones de los protocolos de Internet fueron publicados en esta serie a principios de los años ochenta, seguido por especificaciones de aplicaciones que corrieran sobre ellos, tales como transferencia de archivos, correo electrónico y *telnet* a mediados de los años ochenta. Esta práctica continúa y forma la base de la documentación técnica sobre Internet. Esta serie incluye discusiones así como especificaciones de protocolos. Se puede obtener copias en [www.ietf.org]. Así la publicación de los protocolos originales de comunicación de Internet ha posibilitado que se construyera una enorme variedad de sistemas y aplicaciones sobre Internet. Los documentos RFC no son el único modo de publicación. Por ejemplo, CORBA está publicado a través de una serie de documentos técnicos, incluyendo una especificación completa de las interfaces de sus servicios. Véase [www.omg.org].

Los sistemas diseñados de este modo para dar soporte a la compartición de recursos se etiquetan como *sistemas distribuidos abiertos* (*open distributed systems*) para remarcar el hecho de ser extensibles. Pueden ser extendidos en el nivel hardware mediante la inclusión de computadores a la red y en el nivel software por la introducción de nuevos servicios y la reimplementación de los antiguos, posibilitando a los programas de aplicación la compartición de recursos. Otro beneficio más, citado a menudo, de los sistemas abiertos es su independencia de proveedores concretos.

En resumen:

- Los sistemas abiertos se caracterizan porque sus interfaces están publicadas.
- Los sistemas distribuidos abiertos se basan en la providencia de un mecanismo de comunicación uniforme e interfaces públicas para acceder a recursos compartidos.
- Los sistemas distribuidos abiertos pueden construirse con hardware y software heterogéneo, posiblemente de diferentes proveedores. Sin embargo, la conformidad con el estándar publicado de cada componente debe contrastarse y verificarse cuidadosamente si se desea que el sistema trabaje correctamente.

1.4.3. SEGURIDAD

Entre los recursos de información que se ofrecen y se mantienen en los sistemas distribuidos, muchos tienen un alto valor intrínseco para sus usuarios. Por esto su seguridad es de considerable importancia. La seguridad de los recursos de información tiene tres componentes: confidencialidad (protección contra el descubrimiento por individuos no autorizados); integridad (protección contra la alteración o corrupción); y disponibilidad (protección contra interferencia con los procedimientos de acceso a los recursos).

La Sección 1.1 apuntaba que a pesar de que Internet permite a un programa de un computador comunicarse con un programa en otro computador sin mencionar su ubicación, el permitir un acceso libre a todos los recursos de una intranet lleva asociados riesgos contra la seguridad. Aunque se pueda emplear un cortafuegos para disponer una barrera alrededor de una intranet, restringiendo el tráfico que pudiera entrar y salir, no pretende asegurar el uso apropiado de los recursos por usuarios del interior de la intranet, o del uso apropiado de los recursos en Internet, no protegidos por el cortafuegos.

En un sistema distribuido, los clientes envían peticiones de acceso a datos administrados por servidores, lo que trae consigo enviar información en los mensajes por la red. Por ejemplo:

1. Un médico puede solicitar acceso a los datos hospitalarios de un paciente o enviar modificaciones sobre ellos.
2. En comercio electrónico y banca, los usuarios envían su número de tarjeta de crédito a través de Internet.

En ambos casos, el reto se encuentra en enviar información sensible en un mensaje, por la red, de forma segura. Pero la seguridad no sólo es cuestión de ocultar los contenidos de los mensajes, también consiste en conocer con certeza la identidad del usuario u otro agente en nombre del cual se envía el mensaje. En el primer ejemplo, el servidor necesita conocer que el usuario es realmente un médico y en el segundo, el usuario necesita estar seguro de la identidad de la tienda o del banco con el que está tratando. El segundo reto consiste en identificar un usuario remoto u otro agente correctamente. Ambos desafíos pueden lograrse a través de técnicas de encriptación desarrolladas al efecto. Éstas se utilizan ampliamente en Internet y se discutirán en el Capítulo 7.

Sin embargo, aún existen dos desafíos de seguridad que no han sido cumplimentados completamente:

Ataques de denegación de servicio: otro problema de seguridad ocurre cuando un usuario desea obstaculizar un servicio por alguna razón. Esto se obtiene al bombardear el servicio con un número suficiente de peticiones inútiles de modo que los usuarios serios sean incapaces de utilizarlo. A esto se le denomina ataque de *denegación de servicio*. En el momento de escribir esto (primeros meses del año 2000), han ocurrido, recientemente, varios ataques de denegación de servicio sobre conocidos servicios web. Por el momento tales ataques se contrarrestan intentando atrapar y castigar a los perpetradores con posterioridad al suceso, aunque no es una solución general al problema. Se encuentran en desarrollo contramedidas basadas en mejorar la administración de las redes, éstas se comentarán en el Capítulo 3.

Seguridad del código móvil: el código móvil necesita ser tratado con cuidado. Suponga que alguien recibe un programa ejecutable adherido a un correo electrónico: los posibles efectos al ejecutar el programa son impredecibles; por ejemplo, pudiera parecer que presentan un interesante dibujo en la pantalla cuando en realidad están interesados en el acceso a los recursos locales, o quizás pueda ser parte de un ataque de denegación de servicio. En el Capítulo 7 se bosquejarán algunas medidas para asegurar el código móvil.

Fecha	*Computadores*	*Servidores web*
Diciembre de 1979	188	0
Julio de 1989	130.000	0
Julio de 1999	56.218.000	5.560.866

Figura 1.5. Computadores en Internet.

1.4.4. ESCALABILIDAD

Los sistemas distribuidos operan efectiva y eficientemente en muchas escalas diferentes, desde pequeñas intranets a Internet. Se dice que un sistema es *escalable* si conserva su efectividad cuando ocurre un incremento significativo en el número de recursos y el número de usuarios. Internet proporciona un ejemplo de un sistema distribuido en el que el número de computadores y servicios experimenta un dramático incremento. La Figura 1.5 muestra el número de computadores y servicios en Internet durante 20 años, desde 1979 hasta 1999, y la Figura 1.6 muestra el creciente número de computadores y servidores *web* durante los 16 años de historia del Web hasta 1999, véase [info.isoc.org].

El diseño de los sistemas distribuidos escalables presenta los siguientes retos:

Control del coste de los recursos físicos: según crece la demanda de un recurso, debiera ser posible extender el sistema, a un coste razonable, para satisfacerla. Por ejemplo, la frecuencia con la que se accede a los archivos de una intranet suele crecer con el incremento del número de usuarios y computadores. Debe ser posible añadir servidores para evitar el embotellamiento que aparece cuando un solo servidor de archivos ha de manejar todas las peticiones de acceso a éstos. En general, para que un sistema con n usuarios fuera escalable, la cantidad de recursos físicos necesarios para soportarlo debiera ser como máximo $O(n)$, es decir proporcional a n. Por ejemplo, si un solo servidor de archivos pudiera soportar 20 usuarios, entonces 2 servidores del mismo tipo tendrán capacidad para 40 usuarios. Aunque parezca una meta obvia, no es tan fácil lograrlo en la práctica, según se mostrará en el Capítulo 8.

Control de las pérdidas de prestaciones: considere la administración de un conjunto de datos cuyo tamaño es proporcional al número de usuarios o recursos del sistema, sea por ejemplo la tabla con la relación de nombres de dominio de computadores y sus direcciones Internet sustentado por el Sistema de Nombres de Dominio (*Domain Name System*), que se emplea principalmente para averiguar nombres DNS tales como *www.amazon.com*. Los algoritmos que emplean estructuras jerárquicas se comportan mejor frente al crecimiento de la escala que los algoritmos que emplean estructuras lineales. Pero incluso con estructuras jerárquicas un incremento en tamaño traerá consigo pérdidas en prestaciones: el tiempo que lleva acceder a datos estructurados jerárquicamente es $O(\log n)$, donde n es el tamaño del conjunto de datos. Para que un sistema sea escalable, la máxima pérdida de prestaciones no debiera ser peor que esta medida.

Fecha	*Computadores*	*Servidores web*	*Porcentaje (%)*
Julio de 1993	1.776.000	130	0,008
Julio de 1995	6.642.000	23.500	0,4
Julio de 1997	19.540.000	1.203.096	6
Julio de 1999	56.218.000	6.598.697	12

Figura 1.6. Comparación entre Computadores y Servidores web en Internet.

Prevención de desbordamiento de recursos software: un ejemplo de pérdida de escalabilidad se muestra en el tipo de número usado para las direcciones Internet (direcciones de computadores en Internet). A finales de los años setenta, se decidió emplear para esto 32 bits, pero como se explicará en el Capítulo 3 el suministro de direcciones para Internet se desbordará probablemente al comienzo de la década del año 2000. Por esta razón, la nueva versión del protocolo empleará direcciones Internet de 128 bits. A pesar de ello, para ser justos con los primeros diseñadores de Internet, no hay una solución idónea para este problema. Es difícil predecir la demanda que tendrá que soportar un sistema con años de anticipación. Además, sobredimensionar para prever el crecimiento futuro pudiera ser peor que la adaptación a un cambio cuando se hace necesario; las direcciones Internet grandes ocupan espacio extra en los mensajes, y en la memoria de los computadores.

Evitación de cuellos de botella de prestaciones: en general, para evitar cuellos de botella de prestaciones, los algoritmos deberían ser descentralizados. Ilustramos este punto aludiendo al predecesor del Sistema de Nombres de Dominio en el cual la tabla de nombres se alojaba en un solo archivo maestro que podía descargarse a cualquier computador que lo necesitara. Esto funcionaba bien cuando sólo había unos cientos de computadores en Internet, pero pronto se convirtió en un serio cuello de botella de prestaciones y de administración. El Sistema de Nombres de Dominio eliminó este cuello de botella particionando la tabla de nombres entre servidores situados por todo Internet y siendo administrados localmente, véanse los Capítulos 3 y 9.

Algunos recursos compartidos son accedidos con mucha frecuencia; por ejemplo, puede que muchos usuarios accedan a la misma página web, causando un declive de las prestaciones. En el Capítulo 2 veremos que el empleo de *caché* y replicación puede mejorar las prestaciones de los recursos que estén siendo muy fuertemente utilizadas.

Idealmente, el software de sistema y aplicación no tiene por qué cambiar cuando la escala del sistema se incremente, pero esto es difícil de conseguir. La cuestión del escalado de un sistema es un tema dominante en el desarrollo de sistemas distribuidos. Las técnicas que han demostrado éxito se describen extensamente en este libro. Éstas incluyen el uso de datos replicados (Capítulos 8 y 14), la técnica asociada de caché (Capítulos 2 y 8) y la implementación de múltiples servidores para tratar tareas frecuentes, permitiendo varias tareas similares concurrentemente.

1.4.5. TRATAMIENTO DE FALLOS

Los sistemas computacionales a veces fallan. Cuando aparecen fallos en el hardware o el software, los programas pueden producir resultados incorrectos o pudieran parar antes de haber completado el cálculo pedido. En el Capítulo 2 discutiremos y clasificaremos un rango de tipos de fallos posibles que pueden acaecer en los procesos y redes de que consta un sistema distribuido.

Los fallos en un sistema distribuido son parciales; es decir, algunos componentes fallan mientras otros siguen funcionando. Consecuentemente, el tratamiento de fallos es particularmente difícil. A lo largo del libro se discuten las siguientes técnicas para tratar fallos:

Detección de fallos: algunos fallos son detectables. Por ejemplo, se pueden utilizar sumas de comprobación (*checksums*) para detectar datos corruptos en un mensaje o un archivo. El Capítulo 2 explica que es difícil o incluso imposible detectar algunos otros fallos como la caída de un servidor remoto en Intenet. El reto está en arreglárselas en presencia de fallos que no pueden detectarse pero que sí pueden esperarse.

Enmascaramiento de fallos: algunos fallos que han sido detectados pueden ocultarse o atenuarse. Dos ejemplos de ocultación de fallos son:

1. Los mensajes pueden retransmitirse cuando falla la recepción.

2. Los archivos con datos pueden escribirse en una pareja de discos de forma que si uno está deteriorado el otro seguramente está en buen estado.

Simplemente eliminar un mensaje corrupto es un ejemplo de atenuar un fallo (pudiera retransmitirse de nuevo). Probablemente el lector se dará cuenta de que las técnicas indicadas para ocultar los fallos no tienen garantía de funcionamiento las peores situaciones; por ejemplo, los datos en el segundo disco pudieran también estar corrompidos, o el mensaje bien pudiera no llegar a tiempo no importa cuantas veces se retransmita.

Tolerancia de fallos: la mayoría de los servicios en Internet exhiben fallos; es posible que no sea práctico para ellos pretender detectar y ocultar todos los fallos que pudieran aparecer en una red tan grande y con tantos componentes. Sus clientes pueden diseñarse para tolerar ciertos fallos, lo que implica que también los usuarios tendrán que tolerarlos generalmente. Por ejemplo, cuando un visualizador web no puede contactar con un servidor web no hará que el cliente tenga que esperar indefinidamente mientras hace sucesivos intentos; informará al usuario del problema, dándole la libertad de intentarlo más tarde.

Recuperación frente a fallos: la recuperación implica el diseño de software en el que, tras una caída del servidor, el estado de los datos pueda reponerse o retractarse (*roll back*) a una situación anterior. En general, cuando aparecen fallos los cálculos realizados por algunos programas se encontrarán incompletos y al actualizar datos permanentes (archivos e información ubicada en almacenamiento persistente) pudiera encontrarse en un estado inconsistente. La recuperación se describirá en el Capítulo 13.

Redundancia: puede lograrse que los servicios toleren fallos mediante el empleo redundante de componentes. Considere los siguientes ejemplos:

1. Siempre deberá haber al menos dos rutas diferentes entre cualesquiera dos *routers* (encaminadores) en Internet.
2. En el Sistema de Nombres de Dominio, cada tabla de nombres se encuentra replicada en dos servidores diferentes.
3. Una base de datos puede encontrarse replicada en varios servidores para asegurar que los datos siguen siendo accesibles tras el fallo de cualquier servidor concreto; los servidores pueden diseñarse para detectar fallos entre sus iguales; cuando se detecta algún error en un servidor se redirigen los clientes a los servidores restantes.

El diseño de técnicas eficaces para mantener réplicas actualizadas de datos que cambian rápidamente sin una pérdida excesiva de prestaciones es un reto; en el Capítulo 14 se discutirán varias aproximaciones a él.

Los sistemas distribuidos proporcionan un alto grado de disponibilidad frente a los fallos del hardware. La *disponibilidad* de un sistema mide la proporcion de tiempo en que está utilizable. Cuando falla algún componente del sistema distribuido sólo resulta afectado el trabajo relacionado con el componente defectuoso. Así como cuando un computador falla el usuario puede desplazarse a otro, también puede iniciarse un proceso de servicio en otra ubicación.

1.4.6. CONCURRENCIA

Tanto los servicios como las aplicaciones proporcionan recursos que pueden compartirse entre los clientes en un sistema distribuido. Existe por lo tanto una posibilidad de que varios clientes intenten acceder a un recurso compartido a la vez. Por ejemplo, una estructura de datos que almacena licitaciones de una subasta puede ser accedida muy frecuentemente cuando se aproxima el momento de cierre.

El proceso que administra un recurso compartido puede atender las peticiones de cliente una por una en cada momento, pero esta aproximación limita el ritmo de producción del sistema (*throughput*). Por esto los servicios y aplicaciones permiten, usualmente, procesar concurrentemente múltiples peticiones de los clientes. Más concretamente, suponga que cada recurso se encapsula en un objeto y que las invocaciones se ejecutan en hilos de ejecución concurrentes (*threads*). En este caso es posible que varios *threads* estuvieran ejecutando concurrentemente el contenido de un objeto, en cuyo caso las operaciones en el objeto pueden entrar en conflicto entre sí y producir resultados inconsistentes. Por ejemplo, sean dos ofertas que concurren a una subasta como «Pérez:122$» y «Rodríguez:111$» y las operaciones correspondientes se entrelazan sin control alguno, estas ofertas se pueden almacenar como «Pérez:111$» y «Rodríguez:122$».

La moraleja de esta historia es que cada objeto que represente un recurso compartido en un sistema distribuido debe responsabilizarse de garantizar que opera correctamente en un entorno concurrente. De este modo cualquier programador que recoge una implementación de un objeto que no está concebido para su aplicación en un entorno distribuido deberá realizarlas modificaciones necesarias para que sea seguro su uso en un entorno concurrente.

Para que un objeto sea seguro en un entorno concurrente, sus operaciones deben sincronizarse de forma que sus datos permanezcan consistentes. Esto puede lograrse mediante el empleo de técnicas conocidas como los semáforos, que se usan en la mayoría de los sistemas operativos. Esta cuestión y su extensión a conjuntos de objetos compartidos se discutirá en los Capítulos 6 y 12.

1.4.7. TRANSPARENCIA

Se define transparencia como la ocultación al usuario y al programador de aplicaciones de la separación de los componentes en un sistema distribuido, de forma que se perciba el sistema como un todo más que como una colección de componentes independientes. Las implicaciones de la transparencia son de gran calado en el diseño del software del sistema.

El Manual de Referencia ANSA (ANSA *Reference Manual*) [ANSA 1989] y el Modelo de Referencia para el Procesamiento Distribuido Abierto (RM-ODP: *Reference Model for Open Distributed Processing*) de la Organización Internacional de Estándares [ISO 1992] identifican ocho formas de transparencia. Hemos reproducido las definiciones originales de ANSA, remplazando su transparencia de migración por nuestra propia transparencia de movilidad, de mayor alcance:

Transparencia de acceso que permite acceder a los recursos locales y remotos empleando operaciones idénticas.

Transparencia de ubicación que permite acceder a los recursos sin conocer su localización.

Transparencia de concurrencia que permite que varios procesos operen concurrentemente sobre recursos compartidos sin interferencia mutua.

Transparencia de replicación que permite utilizar múltiples ejemplares de cada recurso para aumentar la fiabilidad y las prestaciones sin que los usuarios y los programadores de aplicaciones necesiten su conocimiento.

Transparencia frente a fallos que permite ocultar los fallos, dejando que los usuarios y programas de aplicación completen sus tareas a pesar de fallos del hardware o de los componentes software.

Transparencia de movilidad que permite la reubicación de recursos y clientes en un sistema sin afectar la operación de los usuarios y los programas.

Transparencia de prestaciones que permite reconfigurar el sistema para mejorar las prestaciones según varía su carga.

Transparencia al escalado que permite al sistema y a las aplicaciones expandirse en tamaño sin cambiar la estructura del sistema o los algoritmos de aplicación.

Las dos más importantes son la transparencia de acceso y la transparencia de ubicación; su presencia o ausencia afecta principalmente a la utilización de recursos distribuidos. A veces se les da el nombre conjunto de *transparencia de red*.

Como ilustración de la transparencia de acceso, considere una interfaz gráfica de usuario basada en carpetas, donde los contenidos de las carpetas se observan igual ya sean éstas locales o remotas. Otro ejemplo pudiera ser el de una interfaz de programación de aplicaciones [API] para archivos que emplea las mismas operaciones para acceder a éstos ya sean locales o remotos (véase el Capítulo 8). Como ejemplo de carencia de transparencia de acceso, considere un sistema distribuido que no permite acceder a los archivos de un computador remoto a menos que se emplee el programa «ftp».

Los nombres de recursos web o URLs son transparentes a la ubicación dado que la parte del URL que identifica el nombre del dominio del servidor web se refiere a un nombre de computador en un dominio, más que a una dirección en Internet. Sin embargo, un URL no es transparente a la movilidad, porque una página web dada no puede moverse a un nuevo lugar en un dominio diferente sin que todos los enlaces anteriores a esta página sigan apuntando a la página original.

En general, los identificadores como los URLs que incluyen los nombres de dominio en los computadores contravienen la transparencia de replicación. Aunque el DNS permite que un nombre de dominio se refiera a varios computadores, sólo se escoge uno de ellos cuando se utiliza un nombre. Ya que un esquema de replicación generalmente necesita ser capaz de acceder a todos los computadores del grupo, sería necesario acceder a cada entrada del DNS por nombre.

Como ilustración de la presencia de transparencia de red, considere el uso de una dirección de correo electrónico como *Pedro.Picapiedra@piedradura.com*. La dirección consta de un nombre de usuario y un nombre de dominio. Observe que a pesar de que los programas de correo aceptan nombres de usuario para usuarios locales, añaden el nombre del dominio. El envío de correo a un usuario no implica el conocimiento de su ubicación física en la red. Tampoco el procedimiento de envío de un mensaje de correo depende de la ubicación del receptor. En resumen, el correo electrónico en Internet proporciona ambas cosas: transparencia de ubicación y transparencia de acceso (en definitiva, transparencia de red).

La transparencia frente a fallos puede ilustrarse también en el contexto del correo electrónico, el cual eventualmente se envía, incluso aunque los servidores o los enlaces de comunicaciones fallen. Los fallos se enmascaran intentando retransmitir los mensajes hasta que se envían satisfactoriamente, incluso si lleva varios días. El middleware convierte generalmente los fallos de redes y procesos en excepciones del nivel de programación (para una explicación véase el Capítulo 5).

Para ilustrar la transparencia a la movilidad, considere el caso de los teléfonos móviles. Supongamos que ambos, el emisor y el receptor, viajan en tren por diferentes partes del país, moviéndose de un entorno (célula) a otro. Veamos al terminal del emisor como un cliente y al terminal del receptor como un recurso. Los dos usuarios telefónicos no perciben el desplazamiento de sus terminales (el cliente y el recurso) entre dos células.

La transparencia oculta y difumina anónimamente los recursos que no son relevantes directamente para la tarea entre manos de los usuarios y programadores de aplicaciones. Por ejemplo, en general es deseable que el uso de ciertos dispositivos físicos sea intercambiable. Por ejemplo, en un sistema multiprocesador, la identificación del procesador en que se ejecuta cada proceso es irrelevante. Aún puede que la situación sea otra: por ejemplo, un viajero que conecta un computador portátil a una red local, en cada oficina que visita hace uso de servicios locales como el correo, utilizando diferentes servidores en cada ubicación. Incluso dentro de un edificio, es normal preparar las cosas para imprimir cada documento en una impresora concreta: generalmente la más próxi-

ma. También para un programador que desarrolla programas paralelos, no todos los procesadores son anónimos. Él o ella pudiera estar interesado en qué procesadores utilizar para la tarea, o al menos cuántos y su topología de interconexión.

1.5. RESUMEN

Los sistemas distribuidos están por todas partes. Internet permite que los usuarios de todo el mundo accedan a sus servicios donde quiera que estén situados. Cada organización administra una intranet, que provee servicios locales y servicios de Internet a los usuarios locales y habitualmente proporciona servicios a otros usuarios de Internet. Es posible construir pequeños sistemas distribuidos con computadores portátiles y otros dispositivos computacionales pequeños conectados a una red inalámbrica.

La compartición de recursos es el principal factor que motiva la construcción de sistemas distribuidos. Recursos como impresoras, archivos, páginas web o registros de bases de datos se administran mediante servidores del tipo apropiado. Por ejemplo los servidores web administran páginas y otros recursos web. Los recursos son accedidos por clientes, por ejemplo, los clientes de los servidores web se llaman normalmente visualizadores o navegadores web.

La construcción de los sistemas distribuidos presenta muchos desafíos:

Heterogeneidad: debe construirse desde una variedad de diferentes redes, sistemas operativos, hardware de computador y lenguajes de programación. Los protocolos de comunicación de Internet enmascaran las diferencias entre redes y el middleware puede tratar con las diferencias restantes.

Extensibilidad: los sistemas distribuidos deberían ser extensibles, el primer paso es la publicación de las interfaces de sus componentes, pero la integración de componentes escritos por diferentes programadores es un auténtico reto.

Seguridad: se puede emplear encriptación para proporcionar una protección adecuada a los recursos compartidos y mantener secreta la información sensible cuando se transmite un mensaje a través de la red. Los ataques de denegación de servicio son aún un problema.

Escalabilidad: un sistema distribuido es escalable si el coste de añadir un usuario es una cantidad constante en términos de recursos que se deberán añadir. Los algoritmos empleados para acceder a los datos compartidos deberían evitar cuellos de botella y los datos deberían estar estructurados jerárquicamente para dar los mejores tiempos de acceso. Los datos frecuentemente accedidos pudieran estar replicados.

Tratamiento de fallos: cualquier proceso, computador o red puede fallar independientemente de los otros. En consecuencia cada componente necesita estar al tanto de las formas posibles en que pueden fallar los componentes de los que depende y estar diseñado para tratar apropiadamente con cada uno de estos fallos.

Concurrencia: la presencia de múltiples usuarios en un sistema distribuido es una fuente de peticiones concurrentes a sus recursos. Cada recurso debe estar diseñado para ser seguro en un entorno concurrente.

Transparencia: el objetivo es que ciertos aspectos de la distribución sean invisibles al programador de aplicaciones de modo que sólo necesite ocuparse del diseño de su aplicación particular. Por ejemplo, no debe ocuparse de su ubicación o los detalles sobre cómo se accede a sus operaciones por otros componentes, o si será replicado o migrado. Incluso los fallos de las redes y los procesos pueden presentarse a los programadores de aplicaciones en forma de excepciones, aunque deban de ser tratados.

EJERCICIOS

1.1. Proponga cinco tipos de recursos hardware y cinco tipos de recursos software o de datos que puedan compartirse útilmente. Proponga ejemplos de su uso compartido tal y como ocurre en la práctica en los sistemas distribuidos.

1.2. ¿Cómo podría sincronizarse los relojes de dos computadores unidos por una red local, sin hacer uso de una referencia temporal externa? ¿Qué factores limitarían la precisión del procedimiento propuesto? ¿Cómo podrían sincronizarse los relojes de un mayor número de computadores conectados a Internet? Discuta la precisión de este procedimiento.

1.3. Un usuario llega a una estación de ferrocarril que no conoce, portando un PDA capaz de conectarse a una red inalámbrica. Sugiera cómo podría proporcionársele al usuario información sobre los servicios locales y las comodidades en la estación, sin necesidad de insertar el nombre de la estación o sus características. ¿Qué dificultades técnicas hay que superar?

1.4. ¿Cuáles son las ventajas y desventajas de HTML, URL y HTTP como tecnologías de base para la consulta y visualización de información? ¿Son algunas de estas tecnologías adecuadas como plataforma de cómputo cliente-servidor en general?

1.5. Tome World Wide Web como ejemplo para ilustrar el concepto de compartición de recursos, cliente y servidor.

Los recursos en World Wide Web y otros servicios se direccionan mediante URL. ¿Qué significan la siglas URL? Proporcione ejemplos de tres tipos de recursos web a los que pueda darse un nombre URL.

1.6. Dé un ejemplo de URL.

Enumere los tres componentes principales de un URL, indicando cómo se delimitan e ilustre cada uno a partir de un ejemplo.

¿Hasta qué límite es transparente a la ubicación en URL?

1.7. Un programa servidor escrito en un lenguaje (por ejemplo C++) proporciona un objeto BURBUJA al que se pretende que accedan clientes que pudieran estar escritos en un lenguaje diferente (por ejemplo Java). Los computadores clientes y servidores pueden tener un hardware diferente, pero todas están conectadas a Internet. Describa los problemas debidos a cada uno de los cinco aspectos de la heterogeneidad que necesitan resolverse para posibilitar que un objeto cliente invoque un método sobre el objeto servidor.

1.8. Un sistema distribuido abierto permite la adición de nuevos servicios de compartición de recursos como el objeto BURBUJA del Ejercicio 1.7 y que sean accesibles por una variedad de programas cliente. Discuta en el contexto de este ejemplo, hasta dónde las necesidades de extensibilidad difieren de las de heterogeneidad.

1.9. Suponga que las operaciones del objeto BURBUJA están separadas en dos categorías: operaciones públicas disponibles para todos los usuarios y operaciones protegidas disponibles sólo para ciertos usuarios conocidos por un nombre concreto. Presente todos los problemas relacionados con la operación de garantizar que sólo los usuarios con nombre conocido puedan acceder a la operación protegida. Suponiendo que el acceso a una operación protegida da información que no debiera revelarse al resto de los usuarios. ¿Qué más problemas aparecen?

1.10. El servicio INFO admite un conjunto de recursos potencialmente muy grande, cada uno de los cuales puede ser accedido por usuarios de Internet mediante una clave (en forma de

«*string*»). Discuta una aproximación al diseño de los nombres de los recursos que logra la mínima pérdida de prestaciones según crece el número de recursos en el servicio. Sugiera cómo puede implementarse el servicio INFO para evitar cuellos de botella en las prestaciones cuando el número de usuarios se vuelve muy grande.

1.11. Enumere los tres componentes software principales que pueden fallar cuando un proceso cliente invoca un método en un objeto servidor, proporcionando un ejemplo del fallo de cada clase. Sugiera cómo pueden construirse los componentes para que toleren sus fallos mutuamente.

1.12. Un servidor mantiene un objeto de información compartida tal como el objeto BURBUJA del Ejercicio 1.7. Argumente en pro y en contra de si admitir que las peticiones de los clientes se ejecuten concurrentemente en el servidor. En este caso, dé un ejemplo de posible «interferencia» que pudiera aparecer entre las operaciones de diferentes clientes. Sugiera cómo puede prevenirse tal interferencia.

1.13. Varios servidores implementan cierto servicio. Explique el porqué pueden transferirse los recursos entre ellos. ¿Sería satisfactorio para los clientes la multidifusión de todas las peticiones al grupo de servidores como un medio de obtener la transparencia de movilidad para los clientes?

2

MODELOS DE SISTEMA

2.1. Introducción
2.2. Modelos arquitectónicos
2.3. Modelos fundamentales
2.4. Resumen

Un modelo arquitectónico de un sistema distribuido trata sobre la colocación de sus partes y las relaciones entre ellas. Algunos ejemplos pueden ser el modelo cliente-servidor y el modelo de procesos «de igual a igual» (*peer to peer*). El modelo cliente-servidor admite varias modificaciones debido a:

- La partición de datos o la replicación en servidores cooperativos.
- El uso de caché para los datos en clientes y servidores proxy.
- El uso de código y agentes móviles.
- Los requisitos para añadir o eliminar dispositivos móviles de forma conveniente.

Los modelos fundamentales están implicados en una descripción más formal de las propiedades que son comunes en todos los modelos arquitectónicos.

No hay un tiempo global en un sistema distribuido, por lo que los relojes en las diferentes computadores no tienen necesariamente el mismo tiempo en una que en otra. Toda comunicación entre procesos se realiza por medio de mensajes. La comunicación mediante mensajes sobre una red puede verse afectada por retrasos, puede sufrir variedad de fallos y es vulnerable a los ataques a la seguridad. Estas cuestiones se consideran en tres modelos:

- El modelo de interacción trata de las prestaciones y de la dificultad de poner límites temporales en un sistema distribuido, por ejemplo para la entrega de mensajes.
- El modelo de fallos intenta dar una especificación precisa de los fallos que se pueden producir en los procesos y en los canales de comunicación. Define comunicación fiable y procesos correctos.
- El modelo de seguridad discute sobre las posibles amenazas para los procesos y los canales de comunicación. Introduce el concepto de canal seguro, que lo es frente a dichas amenazas.

2.1. INTRODUCCIÓN

Los sistemas pensados para trabajar en entornos reales deben diseñarse para funcionar correctamente en el rango de circunstancias más amplio posible y considerando todas las dificultades y amenazas (en el cuadro que viene a continuación se describen algunos ejemplos). La discusión y los ejemplos del Capítulo 1 sugieren que sistemas distribuidos de tipos diferentes comparten importantes propiedades subyacentes y dan lugar a problemas de diseño comunes. En este capítulo se presentan las propiedades y temas de diseño comunes para los sistemas distribuidos bajo la forma de modelos descriptivos. Cada modelo está pensado para proporcionar una descripción abstracta, simplificada pero consistente, de cada aspecto relevante del diseño de un sistema distribuido.

Un modelo arquitectónico define la forma en que los componentes de los sistemas interaccionan uno con otro y en cómo están vinculados con la red de computadores subyacente. En la Sección 2.2, describimos la estructura de niveles del software de sistema distribuido y los principales modelos arquitectónicos que determinan las ubicaciones y las interacciones de los componentes. Discutimos las variantes del modelo cliente-servidor, incluyendo aquellas que se deben al empleo de código móvil. Tenemos en cuenta las características de un sistema distribuido en el que se puedan añadir o quitar dispositivos móviles a conveniencia. Finalmente, contemplamos los requisitos generales de diseño para los sistemas distribuidos.

En la Sección 2.3, presentamos tres modelos fundamentales que ayudan a localizar los problemas clave para los diseñadores de sistemas distribuidos. Su propósito es especificar las cuestiones, dificultades y amenazas que deben resolverse para desarrollar sistemas distribuidos que realicen correctamente, de forma fiable y segura sus tareas. Los modelos fundamentales proporcionan vistas abstractas de aquellas características de los sistemas distribuidos que afectan a sus características de confianza, corrección, fiabilidad y seguridad.

◇ **Dificultades y amenazas para los sistemas distribuidos.** Algunos de los problemas con los que se deben encarar los diseñadores de sistemas distribuidos son:

Modos de utilización muy variables: Las partes componentes de los sistemas están sujetas a grandes variaciones en la carga de trabajo; por ejemplo, algunas páginas web son accedidas varios millones de veces al día. Algunas partes de un sistema pueden estar desconectadas, o deficientemente conectadas en algún momento; por ejemplo cuando están incluidos en un sistema de computadores móviles. Algunas aplicaciones tienen requisitos especiales para comunicaciones de gran ancho de banda y baja latencia, por ejemplo las aplicaciones multimedia.

Amplio rango de entornos: Un sistema distribuido se debe acomodar a hardware, sistemas operativos y redes heterogéneas. Las redes pueden variar mucho en sus prestaciones, de hecho las redes sin cable trabajan a una fracción de la velocidad de las redes locales. Se deben soportar sistemas de escalado muy diferente, desde decenas hasta millones de computadores.

Problemas internos: Relojes no sincronizados, actualizaciones conflictivas de datos, muchas formas de fallos en hardware y software implicando a componentes individuales de un sistema.

Amenazas externas: Ataques a la integridad y el secreto de los datos, denegación de servicio.

2.2. MODELOS ARQUITECTÓNICOS

La arquitectura de un sistema es su estructura en términos de componentes especificados por separado. El objetivo global es asegurar que la estructura satisfará las demandas presentes y previsibles sobre él. El diseño arquitectónico de un edificio tiene aspectos similares y determina no sólo su apariencia, sino también su estructura general y su estilo arquitectónico (gótico, neoclásico, moderno), proporcionando un marco de referencia consistente para el diseño.

En esta sección, describiremos los principales modelos arquitectónicos empleados en los sistemas distribuidos: los estilos arquitectónicos de los mismos. Construimos nuestros modelos arquitectónicos en torno a los conceptos de proceso y objeto introducidos en el Capítulo 1. Un modelo arquitectónico de un sistema distribuido simplifica y abstrae, inicialmente, las funciones de los componentes individuales de dicho sistema y posteriormente considera:

- La ubicación de los componentes en la red de computadores, buscando definir patrones utilizables para la distribución de datos y carga de trabajo.
- Las interrelaciones entre los componentes, es decir, sus papeles funcionales y los patrones de comunicación entre ellos.

Una simplificación inicial se obtiene clasificando los procesos entre *servidores*, *clientes e iguales*, siendo estos últimos procesos que cooperan y se comunican de forma simétrica para realizar una tarea. Esta clasificación de procesos identifica las responsabilidades de cada uno y ayuda, por tanto, a valorar sus cargas de trabajo y a determinar el impacto de los fallos en cada uno de ellos. Los resultados de este análisis pueden ser utilizados para especificar la distribución de los procesos de una forma que concuerde con los objetivos de prestaciones y fiabilidad del sistema resultante.

Se pueden construir otros sistemas dinámicos como variaciones del modelo cliente servidor:

- La posibilidad de mover código de un proceso a otro permite que un proceso delegue tareas en otro; por ejemplo, los clientes pueden descargar código de los servidores y ejecutarlo localmente. Los objetos y el código al que acceden puede reubicarse para reducir los retardos de acceso y minimizar el tráfico de la comunicación.
- Algunos sistemas distribuidos se diseñan para permitir que los computadoress y otros dispositivos móviles se añadan o eliminen sin incidencias, permitiendo el descubrimiento de servicios disponibles y el ofrecer sus servicios a otros.

Existen varios patrones utilizados ampliamente para la distribución del trabajo en un sistema distribuido y que tienen un impacto importante en las prestaciones y efectividad del sistema resultante. La ubicación actual de los procesos que conforman un sistema distribuido sobre una red de computadores está también influenciada por muchas características detalladas de prestaciones, fiabilidad, seguridad y coste. Los modelos arquitectónicos descritos aquí pueden proporcionar sólo una versión simplificada de los patrones de distribución más importantes.

2.2.1. CAPAS DE SOFTWARE

El término *arquitectura de software* se refería inicialmente a la estructuración del software como capas o módulos en un único computador y más recientemente en términos de los servicios ofrecidos y solicitados entre procesos localizados en el mismo o diferentes computadores. Esta vista orientada a proceso y a servicio puede expresarse en términos de capas de servicio.

Esta visión se presenta en la Figura 2.1 y se desarrolla con detalle creciente en los Capítulos 3 al 6. Un servidor es un proceso que acepta peticiones de otros procesos. Un servicio distribuido puede proveerse desde uno o más procesos servidor, es interactuando entre cada uno de ellos, y con los procesos clientes, para mantener una visión de los recursos del servicio consistente con el sistema. Por ejemplo, un servicio de tiempo de red está implementada en Internet basado en el Protocolo de Tiempo de Red (NTP, *Network Time Protocol*) mediante procesos servidor, corriendo sobre máquinas de Internet, que proporcionan el tiempo actual a cualquier cliente que lo solicite y ajuste su versión del tiempo actual como resultado de las interacciones con los otros.

La Figura 2.1 muestra términos importantes como *plataforma* y *middleware*, que definimos a continuación:

◇ **Plataforma.** El nivel de hardware y las capas más bajas de software se denominan, a menudo, plataforma para sistemas distribuidos y aplicaciones. Estas capas más bajas proporcionan servi-

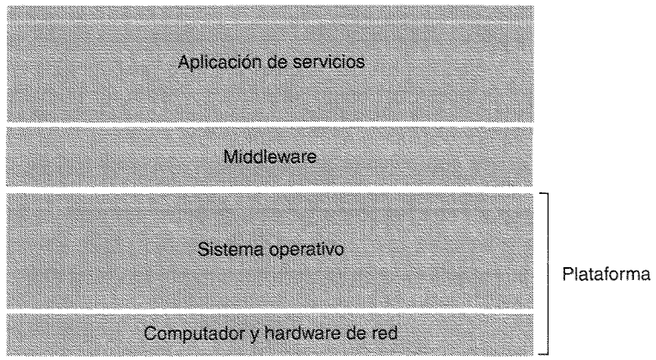


Figura 2.1. Capas de servicio software y hardware en los sistemas distribuidos.

cios a las que están por encima de ellas, y que son implementadas independientemente en cada computador, proporcionando una interfaz de programación del sistema a un nivel que facilita la comunicación y coordinación entre procesos. Windows para Intel x86, Sun OS para Sun SPARC, Solaris para Intel x86, Mac Os para Power PC, Linux para Intel x86 son los principales ejemplos.

◇ **Middleware.** Se definió en la Sección 1.4.1 como una capa de software cuyo propósito es enmascarar la heterogeneidad y proporcionar un modelo de programación conveniente para los programadores de aplicaciones. Se representa mediante procesos u objetos en un conjunto de computadores que interactúan entre sí para implementar mecanismos de comunicación y de recursos compartidos para aplicaciones distribuidas. El middleware se ocupa de proporcionar bloques útiles para la construcción de componentes software que puedan trabajar con otros en un sistema distribuido. En particular, mejora el nivel de las actividades de comunicación de los programas de aplicación soportando abstracciones como: procedimiento de invocación remota, comunicación entre un grupo de procesos, notificación de eventos, replicación de datos compartidos y transmisión de datos multimedia en tiempo real. Las comunicaciones entre grupos se considerarán en el Capítulo 4 y se tratarán con detalle en los Capítulos 11 y 14. La notificación de eventos se describirá en el Capítulo 5. La replicación de datos se discutirá en el Capítulo 14 y los sistemas multimedia en el Capítulo 15.

Los paquetes de llamadas a procedimientos remotos como Sun RPC (descrito en el Capítulo 5) y sistemas de comunicación en grupo como ISIS (Capítulo 14) fueron de los primeros, y son actualmente los ejemplos de middleware más ampliamente utilizados. Entre los productos y estándares de middleware orientados al objeto están CORBA (*Common Object Request Broker Architecture* de OMG), invocación de objetos remotos en Java (RMI, *Remote Method Invocation*), Modelo Común de Objetos Distribuidos de Microsoft (DCOM) y el Modelo de Referencia para Proceso Distribuido Abierto de la ISO/ITU-T (RM-ODP, *Reference Model for Open Distributed Processing*). CORBA y Java RMI serán descritos en los Capítulos 5 y 17, y se pueden encontrar detalles sobre DCOM y RM-ODP en Redmond [1997] y Blair y Stefani [1997].

El middleware también puede proporcionar servicios para su uso en los programas de aplicación. Existen servicios de infraestructuras ligadas fuertemente al modelo de programación distribuida que proporciona el middleware. Por ejemplo, CORBA ofrece una variedad de servicios que proporcionan a las aplicaciones funciones que incluyen la gestión de nombres, seguridad, transacciones, almacenamiento persistente y notificación de eventos. Alguno de los servicios de CORBA se analizarán en el Capítulo 17. Los servicios de la capa superior de la Figura 2.1 son servicios específicos del dominio que utiliza el middleware, sus operaciones de comunicación y sus propios servicios.

Limitaciones del middleware: Muchas aplicaciones distribuidas dependen enteramente de los servicios proporcionados por el middleware disponible, para soportar sus necesidades de comunicación y de compartir datos. Por ejemplo, una aplicación adecuada para el modelo cliente-servidor, como una base de datos de nombres y direcciones, puede basarse en middleware que proporcione sólo invocación de métodos remoto.

Se ha conseguido mucho en la simplificación de la programación de los sistemas distribuidos mediante el desarrollo del soporte de middleware, aunque algunos aspectos de la confiabilidad de los sistemas precisan soporte al nivel de aplicación.

Consideremos la transferencia de mensajes grandes de correo electrónico desde el servidor de correo del emisor hacia el del receptor. A primera vista es un uso sencillo del protocolo de transmisión de datos TCP (se verá en el Capítulo 3). Pero, contemplemos el problema de un usuario que intente transferir un fichero muy grande sobre una red potencialmente poco fiable. TCP proporciona algo de detección y corrección de errores, pero no puede resolver los problemas de interrupciones importantes de red. El servicio de transferencia de correo añade otro nivel de tolerancia a fallos, manteniendo un registro de progreso y reanudando la transmisión, utilizando una nueva conexión TCP, si la primera falla.

Un trabajo clásico de Saltzer, Reed y Clarke [Saltzer y otros, 1984] presenta un apunte similar y valioso sobre el diseño de sistemas distribuidos, que ellos llaman «el argumento final». Parafraseándolos:

> Algunas funciones relacionadas con la comunicación pueden implementarse completamente y de forma fiable sólo con el conocimiento y la ayuda de cómo es la aplicación en los puntos finales del sistema de comunicación. Por tanto, proporcionar dicha función como una característica del sistema de comunicación no siempre es sensato. (Una versión incompleta de la función proporcionada por el sistema de comunicación puede ser a veces útil como una mejora de prestaciones).

Pudiera pensarse que su argumento va en contra de la idea de que todas las actividades de comunicación pueden abstraerse de la programación de aplicaciones mediante la introducción de las capas de middleware adecuadas.

El núcleo de su argumento es que el funcionamiento correcto de los programas distribuidos depende de las comprobaciones, mecanismos de corrección de errores y medidas de seguridad en muy distintos niveles, algunos de los cuales requieren acceso a datos del espacio de direcciones de la aplicación. Cualquier intento de realizar comprobaciones únicamente con el sistema de comunicación garantizará sólo una parte de la corrección exigida. Es muy probable, entonces, que se esté duplicando trabajo en los programas de aplicación, malgastando esfuerzo en la programación, y lo que es más importante, añadiendo una complejidad innecesaria y realizando cómputos redundantes.

No hay aquí espacio suficiente para detallar más sus argumentos, y se recomienda fuertemente la lectura del trabajo citado, que está lleno de ejemplos ilustrativos. Recientemente, uno de los autores originales señala que los beneficios sustanciales que trajo este argumento al diseño de Internet están en peligro por los recientes movimientos hacia la especialización de los servicios de red para satisfacer los requisitos de las aplicaciones actuales [www.reed.com].

2.2.2. ARQUITECTURAS DE SISTEMA

La división de responsabilidades entre los componentes del sistema (aplicaciones, servidores y otros procesos) y la ubicación de los componentes en los computadores en la red, es quizá el aspecto más evidente del diseño de un sistema distribuido. Sus implicaciones fundamentales están en las prestaciones, fiabilidad y seguridad del sistema resultante. En esta sección, se esbozan los principales modelos arquitectónicos sobre los cuales se basa esta distribución de responsabilidades.

En un sistema distribuido, los procesos con responsabilidades bien definidas interactúan con los otros para realizar una actividad útil. En esta sección, nuestra atención se enfocará en la colocación

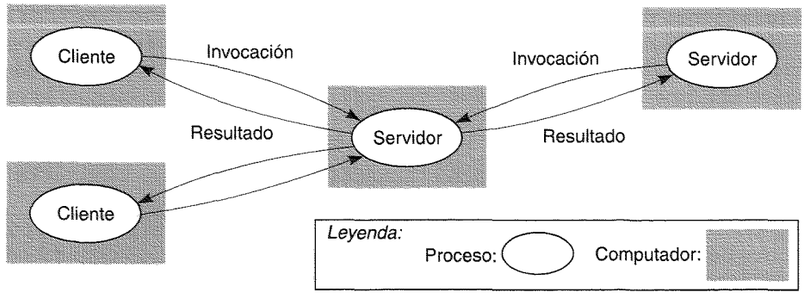


Figura 2.2. Clientes que invocan a servidores individuales.

de los procesos, ilustrada en la Figura 2.2, mostrando la disposición de los procesos (elipses) en los computadores (cajas grises). Utilizamos los términos *invocación* y *resultados* para etiquetar los mensajes, aunque se podrían haber etiquetado igualmente como *solicitud* y *respuesta*.

Los principales tipos de modelos arquitectónicos están ilustrados en las Figuras 2.2 a la Figura 2.5 y se describen a continuación.

◇ **Modelo cliente-servidor.** Es la arquitectura que se cita más a menudo cuando se discuten sistemas distribuidos. Históricamente es la más importante, y continúa siendo la más ampliamente utilizada. La Figura 2.2 muestra la sencilla estructura sobre la que interaccionan los procesos clientes con los procesos servidores individuales, en computadores separados, con el fin de acceder a los recursos compartidos que ellos gestionan.

Los servidores pueden, a su vez, ser clientes de otros servidores, como se indica en la figura. Por ejemplo, un servidor web es, a veces, un cliente de un servidor de ficheros local que gestiona los ficheros en los que están almacenadas las páginas web. Los servidores web y la mayoría del resto de los servicios de Internet son clientes del servicio DNS, que traduce Nombres de Dominio de Internet en direcciones de red. Otro ejemplo relacionado con web son los *buscadores*, que permiten a los usuarios buscar resúmenes de la información disponible en las páginas web de los sitios de Internet. Estos resúmenes están realizados por programas llamados *escaladores* (*crawlers*) *web*, que se ejecutan en segundo plano en el sitio del buscador utilizando peticiones HTTP para acceder a los servidores web a través de Internet. Por tanto, una máquina de búsqueda es tanto un cliente como un servidor: responde a las consultas del navegador de los clientes y ejecuta web crawlers que actúan como clientes de otros servidores web. En este ejemplo, las tareas del servidor (responder a las consultas de usuarios) y las tareas de crawler (hacer peticiones a otros servidores web) son totalmente independientes; hay muy poca necesidad de sincronizarlas y pueden ejecutarse concurrentemente. De hecho, un buscador típico debiera incluir mucho hilos (*threads*) de ejecución concurrente, algunos sirviendo a sus clientes y otros ejecutando crawlers web. En el Ejercicio 2.4, se invita al lector a considerar el único tema de sincronización que se presenta en un buscador concurrente del tipo descrito aquí.

◇ **Servicios proporcionados por múltiples servidores.** Los servicios pueden implementarse como distintos procesos de servidor en computadores separados interaccionando, cuando es necesario, para proporcionar un servicio a los procesos clientes (véase la Figura 2.3). Los servidores pueden dividir el conjunto de objetos en los que está basado el servicio y distribuírselos entre ellos mismos, o pueden mantener copias replicadas de ellos en varias máquinas. Estas dos opciones se ilustran en los ejemplos siguientes.

El Web proporciona un ejemplo típico de partición de datos en el que cada servidor web administra su propio conjunto de recursos. Un usuario puede emplear un navegador para acceder al recurso en cualquiera de los servidores.

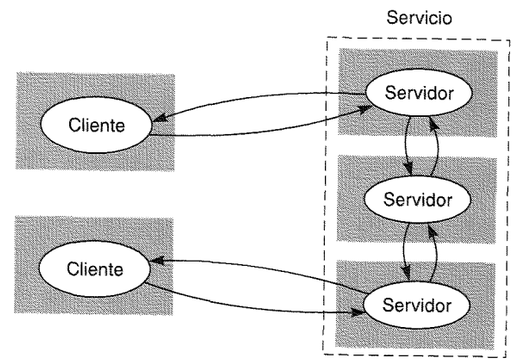


Figura 2.3. Un servicio proporcionado por múltiples servidores.

La replicación se utiliza para aumentar las prestaciones y disponibilidad y para mejorar la tolerancia a fallos. Proporciona múltiples copias consistentes de datos en proceso que se ejecutan en diferentes computadores. Por ejemplo, el servicio web proporcionado por *altavista.digital.com* se encuentra redirigido hacia varios servidores con la base de datos replicada en memoria.

Otro ejemplo de un servicio basado en datos replicados es el servicio de Información de Red de Sun (NIS, *Network Information Service*), que se utilizaría en una red de área local (LAN) cuando los usuarios entran en el sistema. Cada servidor NIS tiene su propia réplica del archivo de contraseñas que contiene una lista de los nombres de usuario y las claves de acceso criptografiadas. En el Capítulo 14 se discutirán con detalle las técnicas de replicación.

◇ **Servidores proxy y cachés.** Una caché es un almacén de objetos de datos utilizados recientemente, y que se encuentra más próximo que los objetos en sí. Al recibir un objeto nuevo en un computador se añade al almacén de la caché, reemplazando, si fuera necesario, algunos objetos existentes. Cuando se necesita un objeto en un proceso cliente, el servicio caché comprueba inicialmente la caché y le proporciona el objeto de una copia actualizada. Si no, se buscará una copia actualizada. Las cachés pueden estar ubicadas en cada cliente o en un servidor proxy que puede compartirse desde varios clientes.

Las cachés se utilizan intensivamente en la práctica. Los navegadores web mantienen una caché de las páginas visitadas recientemente, y de otros recursos web, en el sistema local de ficheros del cliente; entonces utilizan una petición especial HTTP para comprobar si dichas páginas han sido actualizadas en el servidor antes de sacarlas en pantalla. Los servidores proxy para el web (véase la Figura 2.4) proporcionan una caché compartida de recursos web a las máquinas cliente de uno o más sitios. El propósito de los servidores proxy es incrementar la disponibilidad y prestaciones del servicio, reduciendo la carga en redes de área amplia y en servidores web. Los servidores proxy pueden desempeñar otros roles, por ejemplo, pueden ser utilizados para acceder a servidores web remotos a través de un cortafuegos.

◇ **Procesos «de igual a igual».** En esta arquitectura todos los procesos desempeñan tareas semejantes, interactuando cooperativamente como iguales para realizar una actividad distribuida o cómputo sin distinción entre clientes y servidores. En este modelo, el código en los procesos parejos o «iguales» mantiene la consistencia de los recursos y sincroniza las acciones a nivel de la aplicación cuando es necesario. La Figura 2.5 muestra un ejemplo con tres instancias de esta situación; en general, n procesos parejos podrán interactuar entre ellos, dependiendo el patrón de comunicación de los requisitos de la aplicación.

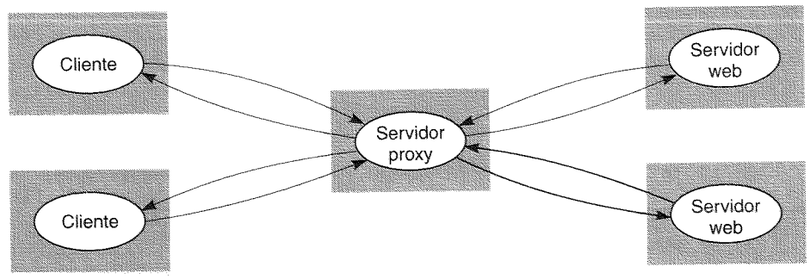


Figura 2.4. Un servidor proxy de tipo web.

La eliminación del proceso servidor reduce los retardos de comunicación entre procesos al acceder a objetos locales. Considérese una aplicación de *pizarra* distribuida que permite que usuarios en varios computadores vean y modifiquen interactivamente un dibujo que se comparte entre ellos (Floyd y otros [1997] es un ejemplo). Esto puede implementarse con un proceso de aplicación en cada sitio y que se basa en capas de middleware para realizar notificación de eventos y comunicación a grupos para indicar a todos los procesos de la aplicación de los cambios en el dibujo. Esto proporciona, a los usuarios de objetos distribuidos compartidos, una respuesta interactiva mejor de la que se puede obtener con una arquitectura basada en servidor.

2.2.3. VARIACIONES EN EL MODELO DE CLIENTE-SERVIDOR

Podemos distinguir distintas variaciones del modelo cliente-servidor, dependiendo de la consideración de los factores siguientes:

- El uso de código móvil y agentes móviles.
- Las necesidades de los usuarios de computadores de coste bajo y con recursos hardware limitados, que son muy sencillos de manejar.
- El requisito de añadir o eliminar de una forma conveniente dispositivos móviles.

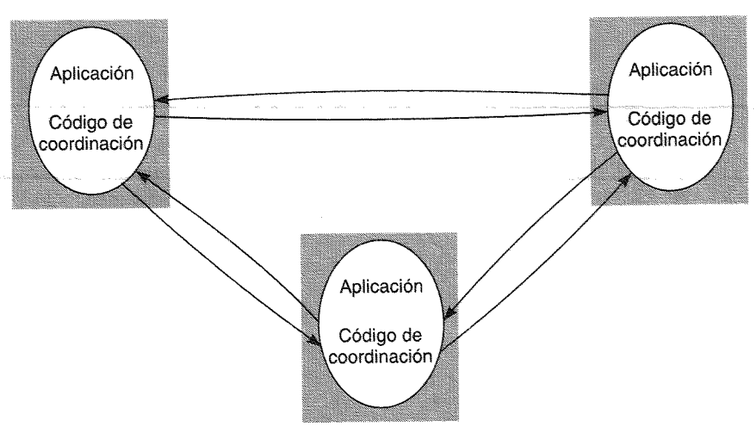


Figura 2.5. Una aplicación distribuida basada en procesos de igual a igual.

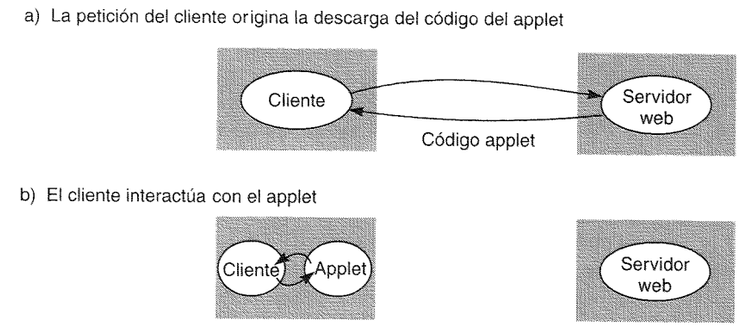


Figura 2.6. Applets de tipo web.

◇ **Código móvil.** En el Capítulo 1 se introdujo el concepto de código móvil. Los applets son el ejemplo más conocido y más ampliamente extendido de código móvil; un usuario que ejecute un navegador y que seleccione un enlace con un *applet*, cuyo código esté almacenado en un servidor web, descargará el código en el navegador y se ejecutará allí (tal y como se ve en la Figura 2.6). Una ventaja de ejecutar el código descargado localmente es que puede proporcionar una buena respuesta interactiva puesto que no sufre ni de los retardos ni de la variabilidad del ancho de banda asociados con la comunicación en red.

Acceder a los servicios significa ejecutar código que pueda invocar sus operaciones. Algunos servicios están probablemente tan estandarizados que podemos acceder a ellos mediante una aplicación existente y bien conocida, el Web es el ejemplo más común, pero incluso en este caso, algunos sitios web utilizan funcionalidades que no se encuentran en los navegadores estándar y precisan la descarga de código adicional. Este código adicional puede, por ejemplo, comunicar con el servidor. Consideremos una aplicación donde se necesite que los usuarios dispongan de información actualizada según se hagan cambios en el origen de los datos en el servidor. Esto no se puede conseguir mediante las interacciones normales con el servidor web, que siempre se inician por parte del cliente. La solución es utilizar software adicional que trabaja de una forma a la que a veces se refiere como modelo *push*, donde el servidor, en lugar del cliente, es el que inicia las interacciones.

Por ejemplo, un corredor de bolsa puede proporcionar un servicio personalizado para notificar a los usuarios los cambios en las cotizaciones; para utilizar dicho servicio, cada usuario debiera haber descargado un *applet* especial, que recibe las actualizaciones del servidor del corredor, las presenta en pantalla al usuario y quizá realiza operaciones automáticas de compra y venta, desencadenadas por las condiciones prefijadas por el usuario y almacenados localmente en el computador del mismo.

En el Capítulo 1 se ha mencionado que el código móvil es una amenaza potencial de seguridad para los recursos locales en el computador destino. Es así que, los navegadores proporcionan un acceso restringido a los recursos locales, siguiendo un esquema como el que se verá en la Sección 7.1.1.

◇ **Agentes móviles.** Un agente móvil es un programa en ejecución (lo que incluye tanto código como datos) que se traslada de un computador a otro en la red realizando una tarea para alguien; por ejemplo, recolectando información, y retornando eventualmente con los resultados. Un agente móvil puede hacer muchas solicitudes a los recursos locales de los sitios que visita, por ejemplo, accediendo a anotaciones individuales en una base de datos. Si se compara esta arquitectura con un cliente estático que realiza solicitudes de algunos recursos, transfiriendo posiblemente

grandes cantidades de datos, hay una reducción en el coste de la comunicación y en el tiempo con la sustitución de las solicitudes remotas por las locales.

Se pueden utilizar agentes móviles para instalar y mantener software en los computadores de una organización, o comparar los precios de los productos de un número de vendedores visitando el sitio de cada vendedor y realizando una serie de operaciones sobre la base de datos. Un ejemplo inicial, con una idea similar, es el llamado programa gusano (*worm*) desarrollado en Xerox PARC [Shoch and Hupp 1982], y que se diseñó para utilizar los computadores desocupados con el fin de realizar cálculos intensivos.

Los agentes móviles (como el código móvil) son una amenaza potencial de seguridad para los recursos de los computadores que visitan. El entorno que recibe el agente móvil debe decidir a cuál de los recursos locales le estará permitido tener acceso, en base a la identidad del usuario en cuyo nombre está actuando el agente; la identidad de éste debe incluirse de una forma segura en el código y los datos del agente móvil. Además, los agentes móviles también pueden ser vulnerables, y pueden no ser capaces de finalizar su tarea si se les niega el acceso a la información que necesitan. Las tareas realizadas por agentes móviles pueden realizarse por otros medios. Por ejemplo, los escaladores (*crawlers*) web que necesitan acceder a recursos en servidores web a través de Internet trabajan bastante bien realizando invocaciones remotas a los procesos del servidor. Por estas razones, la aplicabilidad de agentes móviles puede ser limitada.

◇ **Computadores de red.** En la arquitectura representada en la Figura 1.2, las aplicaciones se ejecutan en un computador de oficina local del usuario. El sistema operativo y el software de aplicación para computadores de oficina necesitan normalmente que gran parte del código y datos activos estén ubicados en un disco local. Pero la gestión de los archivos de aplicación y el mantenimiento del software de base local precisa un esfuerzo técnico considerable y de una naturaleza que la mayoría de los usuarios no están cualificados para proporcionarlo.

El computador de red es una respuesta a este problema. Descarga su sistema operativo y cualquier aplicación software que necesite el usuario desde un servidor de archivos remoto. Las aplicaciones se lanzan localmente pero los archivos se gestionan desde un servidor de archivos remoto. También pueden ejecutarse aplicaciones en red, como un navegador web. Como todos los datos y el código de las aplicaciones están almacenados en un servidor de archivos, los usuarios pueden migrar de un computador de red a otro. Las capacidades de procesador y de memoria de un computador de red pueden restringirse con el fin de reducir el coste.

Si se incluyera un disco, alojaría un software mínimo. El resto del disco se utilizaría como *almacenamiento intermedio* (*caché*) manteniendo copias de los archivos de programas y datos que hayan sido cargados recientemente desde los servidores. El mantenimiento de la caché no precisa esfuerzo manual alguno: los objetos en la caché se invalidan cuando se escribe una nueva versión del archivo en el servidor relevante.

◇ **Clientes ligeros.** El término *cliente ligero* se refiere a una capa de aplicación que soporta una interfaz de usuario basada en ventanas sobre un computador local del usuario mientras se ejecutan programas de aplicación en un computador remoto (Figura 2.7). Esta arquitectura tiene los mismos costes bajos de gestión y hardware que en el esquema de computador de red, pero en lugar de descargar el código de las aplicaciones en el computador del usuario, se ejecutan en un *servidor de cómputo*, un potente computador que tiene capacidad para ejecutar gran número de aplicaciones simultáneamente. El servidor de cómputo será normalmente un multiprocesador o un sistema de computadores fuertemente acoplado (*cluster*) (véase el Capítulo 6), y que ejecuta una versión multiprocesador de un sistema operativo como UNIX o Windows NT.

El principal inconveniente de una arquitectura de cliente ligero se encuentra en actividades gráficas fuertemente interactivas, como CAD o proceso de imagen, en las que los retrasos experimentados por los usuarios se ven aumentados por la necesidad de transferir imágenes e información

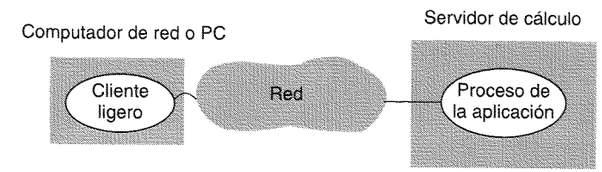


Figura 2.7. Clientes ligeros y servidores de cálculo.

vectorial entre el cliente ligero y los procesos de aplicación, padeciendo latencias tanto de red como de sistema operativo.

Implementaciones de clientes ligeros: Los sistemas de clientes ligeros son sencillos en concepto y sus implementaciones son inmediatas en algunos entornos. Por ejemplo, la mayoría de las variantes de UNIX incluyen el sistema de ventanas X-11 que se describe en el recuadro sombreado de la página siguiente.

Existen varios puntos en el flujo de las operaciones gráficas que se precisan para soportar una interfaz gráfica de usuario en los que se puede establecer la frontera entre cliente y servidor. X11 establece la frontera a nivel de las primitivas gráficas para dibujar líneas y formas y para el manejo de ventanas. El producto *WinFrame* de Citrix [www.citrix.com] es una implementación comercial del concepto de cliente ligero, utilizada ampliamente, que funciona de una forma similar. Este producto proporciona un cliente ligero que puede ejecutarse en una gran variedad de plataformas y soporta un entorno (*desktop*) que proporciona acceso interactivo a las aplicaciones corriendo en máquinas Windows NT. Otras aproximaciones incluyen los sistemas de computadores de red virtual (*Teleporting* y *Virtual Network Computer*, VNC) desarrollados en los laboratorios AT&T de Cambridge, Inglaterra [Richardson y otros, 1998]. VNC establece la frontera en el nivel de operaciones de píxeles en pantalla y mantiene el contexto gráfico para los usuarios cuando éstos se mueven entre computadores.

◇ **Dispositivos móviles y enlace espontáneo a red.** Como ya se ha explicado en el Capítulo 1, el mundo está cada vez más poblado de dispositivos de cómputo pequeños y portátiles, incluyendo computadores portátiles, dispositivos de mano como asistentes personales digitales (PDA), teléfonos móviles y cámaras digitales, computadores que se pueden llevar encima como relojes y dispositivos insertados en aparatos de hogar como en lavadoras. Muchos de estos dispositivos permiten su conexión a una red sin cable, de rango metropolitano o más grandes (GSM, CDPD), cientos de metros (Wavelan), o unos pocos metros (Blue Tooth, infrarrojos y HomeRF). Las redes de rango más pequeño tienen anchos de banda del orden de hasta 10 megabits/segundo; GSM promete anchos de banda del orden de cientos de kilobits/segundo.

Con la integración apropiada en nuestros sistemas distribuidos, estos dispositivos dan soporte para la computación móvil (véase la Sección 1.2.3), en la que los usuarios llevan sus dispositivos móviles entre los entornos de red y se benefician de los servicios locales y remotos según se mueven. La forma de distribución que integra dispositivos móviles y otros dispositivos en una red determinada se describe quizá mejor por el término *enlace a red espontáneo*. Este término se utiliza para abarcar aplicaciones que implican conexión de dispositivos tanto móviles como fijos de una forma más informal de lo que ha sido hasta ahora. Los dispositivos insertados en aparatos proporcionan servicios a los usuarios y a otros dispositivos parecidos, los dispositivos transportables como PDA proporcionan al usuario acceso a los servicios en su posición actual, así como a servicios globales como el Web.

◇ **El sistema de ventanas X-11.** El sistema de ventanas X-11 [Scheifler y Gettys 1986] es un proceso que gestiona la pantalla y los dispositivos interactivos de entrada (teclado, ratón) del computador en la que se ejecuta. X-11 proporciona una amplia biblioteca de procedimientos (el protocolo X-11) para mostrar en pantalla y modificar objetos gráficos en ventanas así como para la creación y modificación de las ventanas.

Al sistema X-11 se le referencia como un proceso *servidor de ventanas*. Los clientes del servidor son los programas de aplicación con los que el usuario interactúa normalmente. Los programas cliente comunican con el servidor invocando operaciones del protocolo X-11, éstas incluyen operaciones para dibujar texto y objetos gráficos en ventanas. Los clientes no tienen por qué ubicarse en el mismo computador que el servidor, puesto que los procedimientos del servidor se invocan siempre utilizando un mecanismo RPC. El servidor de ventanas X-11 tiene, por tanto, las propiedades fundamentales de un cliente ligero. (X-11 invierte la terminología cliente-servidor. El servidor X-11 debe su nombre a los servicios de representación gráfica que proporciona a los programas de aplicación, mientras el software del cliente ligero se llama así por referencia al uso que hace de programas de aplicación ejecutándose en un servidor de cómputo.)

En el Capítulo 1 se vio el caso de un usuario visitando una organización que actúa como anfitrión. En la Figura 2.8 se ve otro ejemplo de conexión espontánea a la red, en ella se puede observar una red inalámbrica que cubre una estancia de un hotel. Los sistemas de alta fidelidad de la habitación dan servicio de música, en el que los usuarios pueden dirigir cualquier combinación de selecciones musicales hacia los altavoces de la habitación o el baño, o a los auriculares inalámbricos. El sistema de alarma da un servicio de despertador mediante el servicio de música. El TV/PC (televisión más caja añadida) sirve tanto de televisión como de PC. Permite acceder al Web (mediante la pasarela del hotel a Internet), incluyendo una interfaz basada en el Web para todos los servicios del hotel. Proporciona un servicio de visualización para las imágenes que el usuario ha almacenado en su cámara digital o videocámara. También puede servir al usuario sus aplicaciones favoritas, bajo cierto coste de arrendamiento.

La figura muestra los dispositivos que el usuario ha llevado a la suite del hotel: un computador portátil, una cámara digital y una agenda portátil (PDA), cuyas posibilidades a través de infrarrojos le permiten actuar como un *controlador universal* (similar al control remoto de una televisión pero que se puede utilizar para controlar varios dispositivos en la habitación del hotel, incluyendo el servicio de iluminación y de música).

Aunque algunos de los dispositivos del hotel, como impresoras y máquinas de venta automáticas no suelen ser portátiles, existe un requisito para presentarlas con la mínima intervención humana. Su conexión a la red y el acceso a los servicios de red deben ser posibles sin que sea preciso notificarlo previamente o realizar cualquier gestión administrativa previa.

Las características esenciales de la conexión a red espontánea son:

Conexión fácil a la red local: los enlaces sin cable eliminan la necesidad de cableado preinstalado y eliminan el inconveniente y las cuestiones de fiabilidad que rodean a los conectores y enchufes. Sólo existen aquellos enlaces que pueden mantenerse aunque haya movimiento (en trenes, aviones, coches, bicicletas o barcos). Un dispositivo colocado en un nuevo entorno de red se reconfigura de modo transparente para conectarse allí; el usuario no tiene que introducir los nombres o direcciones de los servicios locales para obtenerlo.

Integración fácil con servicios locales: los dispositivos que se encuentran insertados (y moviéndose) entre redes existentes de dispositivos descubren automáticamente qué servicios se proporcionan, sin acciones especiales de configuración por el usuario. En el ejemplo del hotel, la cámara digital del huésped descubre servicios de visualización, como los de TV/PC, automáticamente, de modo que el huésped puede enviar las imágenes almacenadas de la cámara a la pantalla.

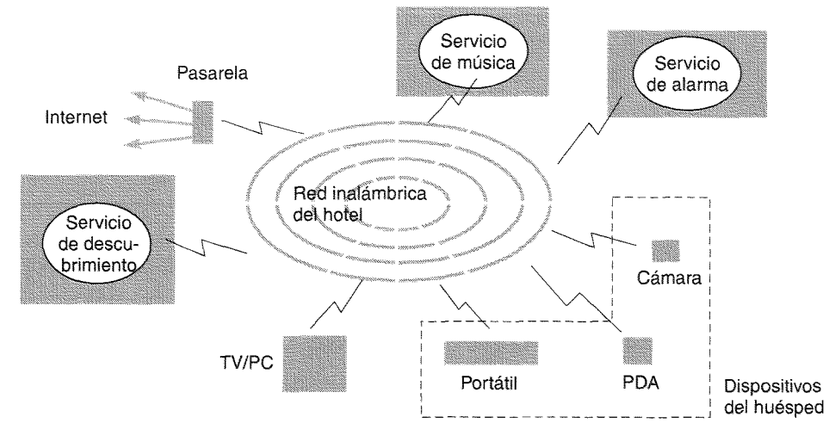


Figura 2.8. Intercomunicación espontánea en un hotel.

El enlace a red espontáneo conlleva algunas características de diseño significativas. Primero, está el reto de dar soporte a una conexión e integración adecuadas. Por ejemplo, los algoritmos de direccionamiento y rutado de Internet suponen que los computadores están ubicados en una subred particular. Si un computador se mueve a otra subred requiere otra dirección de Internet. En el Capítulo 3 se discute una aproximación a este problema.

Para usuarios móviles se presentan otras cuestiones. Éstos ven limitada su conectividad cuando viajan, y la naturaleza espontánea de su conexión incrementa los problemas de seguridad.

Conectividad limitada: los usuarios no siempre están conectados cuando se desplazan. Por ejemplo, al viajar en un tren por un túnel pueden estar desconectados eventualmente de su red inalámbrica. También pueden desconectarse durante largos períodos de tiempo en zonas sin cobertura o cuando el permanecer conectado sea demasiado costoso. El tema es aquí cómo puede el sistema dar soporte a un usuario para que siga trabajando mientras está desconectado. Este punto se discutirá en el Capítulo 14.

Seguridad y privacidad: en un escenario como el del huésped de un hotel se presentan muchas cuestiones de seguridad y privacidad personal. El hotel y sus huéspedes son vulnerables a los ataques de seguridad de los empleados y de otros huéspedes del hotel, que intenten conexiones inalámbricas de formas no supervisadas. Algunos sistemas siguen las direcciones físicas de los usuarios a medida que éstos se mueven (como los sistemas basados en «insignias activas» [Want y otros 1992]), y esto puede amenazar la privacidad de los usuarios. Por último, una instalación que permita a los usuarios acceder a su intranet mientras se mueven puede exponer los datos, que se supone permanecen detrás del cortafuegos de la intranet, o pueden abrir la intranet a ataques del exterior.

Servicios de detección: El enlace espontáneo a red necesita procesos clientes ejecutándose en los dispositivos portátiles, y otros aparatos, para acceder a los servicios de las redes a las que están conectados. Pero en un mundo de dispositivos que realizan diversas funciones, ¿cómo pueden conectarse los clientes a los servicios que necesitan para completar sus tareas habituales? Vamos a detallar esto en el contexto de un ejemplo específico: al volver al hotel, nuestro huésped desea ver

algunas fotografías que ha tomado con una cámara digital, imprimir algunas para su uso local y enviar una selección a su casa electrónicamente. Su habitación tiene una TV/PC. El hotel podría tener un servicio de impresión digital de color, o en caso contrario, probablemente dispone de un servicio de fax.

No se puede esperar que cada fabricante de impresoras implemente exactamente el mismo protocolo para que los clientes accedan a su servicio de impresión. Incluso si éste *fuera* el caso, los clientes pueden construirse adaptables, como nuestro ejemplo parece plantear: una cámara digital del cliente puede mostrar las fotografías en el aparato de TV local o enviarlas al fax del hotel.

Lo que se necesita es un medio que permita a los clientes descubrir qué servicios están disponibles en la red a la que están conectados e investigar sus propiedades. El propósito de un *servicio de detección* (o *descubrimiento*) es aceptar y almacenar detalles de los servicios que están disponibles en la red y responder a las consultas de los clientes sobre los mismos. De forma más precisa, un servicio de descubrimiento ofrece dos interfaces:

- Un *servicio de admisión* que acepta solicitudes de ingreso de los servidores y almacena sus detalles en la base de datos del servicio de descubrimiento.
- Un *servicio de búsqueda* que acepta consultas relacionadas con los servicios disponibles y busca en su base de datos aquellos servicios registrados que coinciden con las consultas. El resultado obtenido ofrecería detalles suficientes para permitir a los usuarios seleccionar entre distintos servicios semejantes, basándose en sus atributos, y hacer una conexión a uno o más de ellos.

Volviendo al ejemplo, un *servicio de descubrimiento* en el hotel incluye detalles de los servicios de impresión y visualización disponibles en el hotel, con detalles como el número de habitación en la que está situada la pantalla o la impresora, la calidad (resolución) del dispositivo, modelos de imagen que acepta el dispositivo, posibilidades de color, etc. La cámara digital del huésped consulta al servicio de descubrimientos los servicios de impresión y visualización, procesa la lista resultante por proximidad a la habitación en la que está localizada, y por compatibilidad con su modelo de imágenes, y ofrece al usuario un menú de los dispositivos adecuados. El usuario selecciona la TV/PC de su habitación o una impresora y envía algunas imágenes. El resto de imágenes se pueden enviar a la familia, o a los amigos, como anexos en un correo electrónico.

El Capítulo 9 incluye una revisión más profunda de los servicios de descubrimiento.

2.2.4. INTERFACES Y OBJETOS

Una *definición de interfaz* de un proceso (ya sea en el modelo cliente-servidor o de comunicación entre iguales) es la especificación del conjunto de funciones que se pueden invocar sobre él. En el Capítulo 5 se describirá totalmente este concepto, aunque sonará familiar a aquellos que hayan trabajado con lenguajes como Modula, C++ o Java. En la forma básica de la arquitectura cliente-servidor, se considera cada proceso servidor como una entidad aislada con una interfaz prescrita que define las funciones que ofrece.

En lenguajes orientados a objetos como C++ y Java, con un soporte adicional apropiado, los procesos distribuidos pueden ser construidos de una forma más orientada al objeto. En un proceso de servicio o entre iguales es posible encapsular multitud de objetos. Las referencias a estos objetos se pasan a otros procesos de forma que se pueda acceder a sus métodos mediante invocaciones remotas. Ésta es la aproximación adoptada por CORBA, y Java con su mecanismo de invocación remota de métodos (RMI). En este modelo, el número y los tipos de los objetos alojados por cada proceso pueden variar como lo requieran las actividades del sistema (en lenguajes que soportan código móvil como Java); incluso, en algunas implementaciones, también las ubicaciones de los objetos.

Tanto si adoptamos una arquitectura cliente-servidor estática o el modelo orientado a objetos, más dinámico, esbozado en el párrafo anterior, la distribución de responsabilidades entre procesos y entre computadores sigue siendo un aspecto importante del diseño. En el modelo arquitectónico tradicional, las responsabilidades se reservan estáticamente (un servidor de ficheros, por ejemplo, es sólo responsable de los ficheros, no de las páginas web, sus proxies u otros tipos de objetos). Pero en el modelo orientado a objetos los nuevos servicios, y en algunos casos los nuevos tipos de objeto, pueden ser instanciados e inmediatamente estar disponibles para la invocación.

2.2.5. REQUISITOS DE DISEÑO PARA ARQUITECTURAS DISTRIBUIDAS

Los factores que motivan la distribución de objetos y procesos en un sistema distribuido son numerosos y de considerable significado. La idea de compartir se logró, por vez primera, en los sistemas de tiempo compartido de los años sesenta con la utilización de archivos compartidos. Los beneficios de la compartición fueron reconocidos rápidamente y se explotaron en sistemas operativos como Oracle para permitir que los procesos compartieran recursos, dispositivos (capacidad de almacenamiento para ficheros, impresoras, secuencias de audio y de vídeo) y procesos de aplicación para compartir objetos de aplicación.

La llegada de potencia de cómputo barata, en forma de chips de microprocesador, eliminó la necesidad de compartir los procesadores centrales. La disponibilidad de redes de computadores de prestaciones medias y la continua necesidad de compartir recursos de hardware relativamente costosos como impresoras y almacenamiento en disco condujeron al desarrollo de los sistemas distribuidos de los años setenta y ochenta. Hoy, el compartir recursos se da por descontado. Pero compartir datos en gran escala de forma efectiva continúa siendo un reto mayor; con datos que cambian (y la mayoría de los datos cambian en el tiempo), aparece la posibilidad de actualizaciones concurrentes y conflictivas. El control de las actualizaciones concurrentes en datos compartidos es el tema de los Capítulos 12 y 13.

◇ **Temas de prestaciones.** Los retos que surgen de la distribución de recursos aumentan la necesidad de gestión de las actualizaciones concurrentes. Los temas de prestaciones que se presentan por las limitadas capacidades del proceso y comunicación de los computadores y las redes se consideran bajo los siguientes epígrafes:

Capacidad de respuesta (*Responsiveness*)**:** Los usuarios de aplicaciones interactivas necesitan rapidez y consistencia en las interfaces, pero, a menudo, los programas cliente necesitan acceder a recursos compartidos. Cuando está implicado un servicio remoto, la velocidad a la que se genera la respuesta está determinada no sólo por la carga y prestaciones del servidor y la red sino también por los retardos de todos los componentes software implicados, la comunicación entre los sistemas operativos del cliente y del servidor y los servicios *middleware* (soporte para invocación remota, por ejemplo) así como el código del proceso que implementa el servicio.

En resumen, la transferencia de datos entre los procesos y la conmutación del control es relativamente lenta, incluso cuando los procesos residen en el mismo computador. Para obtener buenos tiempos de respuesta, los sistemas deben estar compuestos de relativamente pocas capas de software y la cantidad de datos transferidos entre el cliente y el servidor debe ser pequeña.

Estas cuestiones se pueden ver en las prestaciones de los visualizadores web clientes, donde se obtiene una respuesta más rápida cuando se accede localmente a páginas e imágenes de la caché, por lo que se encuentran almacenadas en la aplicación cliente. El acceso a páginas de texto remotas se produce con una rapidez razonable porque son pequeñas, pero las imágenes gráficas se retrasan mucho más por el volumen de datos implicado.

Productividad (*Throughput*)**:** Una medida tradicional de prestaciones para computadores es la productividad, la rapidez a la que se realiza el trabajo computacional. La capacidad de un sistema

distribuido para realizar el trabajo para todos sus usuarios es importante. Y depende de las velocidades de proceso de los clientes y los servidores y de las tasas de transferencia de datos. Los datos que están localizados en un servidor remoto deben transferirse del proceso servidor al proceso cliente pasando a través de varias capas en ambos computadores. La productividad de las capas de software que intervienen es bastante importante, así como la de la red.

Balance de cargas computacionales: Uno de los propósitos de los sistemas distribuidos es permitir que las aplicaciones y los procesos de servicio evolucionen concurrentemente sin competir por los mismos recursos y explotando los recursos computacionales disponibles (procesador, memoria y capacidades de red). Por ejemplo, la capacidad para ejecutar applets en los computadores cliente elimina carga del servidor web, permitiendo proporcionar un mejor servicio. Un ejemplo más significativo está en el uso de varios computadores para alojar un único servicio. Esto se necesita, por ejemplo, en algunos servidores web excesivamente cargados (máquinas de búsqueda, grandes lugares comerciales). Esto explota la funcionalidad del servicio de búsqueda de nombres de dominio DNS para devolver una de las varias direcciones de host para un único nombre de dominio (se verá en la Sección 9.2.3).

En algunos casos, el balance de cargas puede implicar mover el trabajo parcialmente completado, como carga a un computador alternativo.

◇ **Calidad de servicio.** Una vez que se proporciona a los usuarios la funcionalidad que precisan de un servicio, como el servicio de archivos en un sistema distribuido, nos podemos plantear preguntas sobre la calidad del servicio proporcionado. Las principales propiedades no funcionales de los sistemas, que afectan a la calidad del servicio experimentado por los clientes y usuarios, son: fiabilidad seguridad y prestaciones. La facilidad de adaptación (*adaptability*) para adecuar configuraciones variables de sistema y disponibilidad de recursos ha sido reconocida recientemente como otro aspecto importante de la calidad del servicio. Birman ha indagado profundamente en la importancia de estos aspectos de calidad, y su libro [Birman 1996] proporciona algunas perspectivas interesantes de su impacto en el diseño del sistema.

Los aspectos de fiabilidad y seguridad son críticos en el diseño de la mayoría de los sistemas de computadores. Éstos están fuertemente relacionados con dos de nuestros modelos fundamentales: el modelo de fallos y el de seguridad que se introducirán en las Secciones 2.3.2 y 2.3.3.

Los aspectos de prestaciones de la calidad del servicio se definían, hasta hace poco, en términos de la respuesta y la productividad computacional, pero recientemente ha sido redefinido en términos de la capacidad para proporcionar las garantías oportunas, como se describe en los siguientes párrafos. Con cualquiera de estas definiciones, el aspecto de las prestaciones de los sistemas distribuidos está fuertemente relacionado con el modelo de interacción definido en la Sección 2.3.1. Los tres modelos fundamentales son puntos de referencia importantes a lo largo del libro.

Algunas aplicaciones mantienen *datos críticos en el tiempo*, flujos de datos que precisan ser procesados o transferidos de un proceso a otro a una tasa prefijada. Por ejemplo, un servicio de películas puede constar de un programa cliente que está recuperando una película de un servidor de vídeo y presentándolo en la pantalla del usuario. Para un resultado satisfactorio los sucesivos marcos de imágenes deben presentarse al usuario dentro de unos límites de tiempo especificados.

De hecho, se ha recomendado la abreviatura QoS (*Quality of Service*) para referirse a la capacidad de los sistemas para satisfacer dichos tiempos límites. Su consecución depende de la disponibilidad de los recursos necesarios de computación y de red en los instantes adecuados. Esto implica un requisito para que el sistema proporcione recursos garantizados de computación y comunicación que sean suficientes para permitir a las aplicaciones finalizar cada tarea a tiempo (por ejemplo, una tarea para colocar en pantalla un marco de vídeo).

Las redes que se utilizan hoy normalmente, por ejemplo para navegar en el Web, pueden tener buenas características de prestaciones, pero cuando están cargadas intensamente sus prestaciones

se deterioran significativamente, por lo que no se puede decir que proporcionen calidad de servicio. La calidad de servicio se aplica tanto a los sistemas operativos como a las redes. Cada recurso crítico debe reservarse para las aplicaciones que requieren QoS y deben ser los gestores de los recursos los que proporcionen las garantías. Las solicitudes de reserva que no se puedan cumplir se rechazan. Estos temas se considerarán además en el Capítulo 15.

◇ **Uso de caché y de replicación.** Los temas de prestaciones esbozados anteriormente resultan ser, a menudo, los principales obstáculos para el correcto lanzamiento de los sistemas distribuidos, aunque se han hecho nuevos progresos en el diseño de los sistemas, que los superan utilizando replicación y caché. La Sección 2.2.2 introdujo la caché y los servidores web proxy, sin discutir cómo las copias de los recursos en la caché pueden mantenerse actualizadas cuando se actualiza el recurso en el servidor. Para favorecer a diferentes aplicaciones se utiliza una variedad de protocolos de consistencia de caché; por ejemplo, en el Capítulo 8 se detallarán los protocolos utilizados por dos diseños diferentes de servidores de archivos. Aquí esbozamos el protocolo de caché de web que forma parte del protocolo HTTP, que se describirá en la Sección 4.4.

Protocolo de caché de web: Tanto las cachés de los navegadores web como de los servidores proxy responden a las solicitudes a los servidores web. Por tanto, una solicitud del cliente puede ser satisfecha por una respuesta en la caché del navegador o del servidor proxy entre él y el servidor web. El protocolo de consistencia de caché puede configurarse para proporcionar a los navegadores copias recientes (razonablemente actualizadas) de los recursos controlados por el servidor de web. Pero, para permitir prestaciones, disponibilidad y operación desconectada, puede relajarse la condición de actualización.

Un navegador o un proxy pueden *validar* una respuesta de la caché comprobando con el servidor web original para ver si estaba actualizada. Si la comprobación falla, el servidor devuelve una copia actualizada, que se guardará en la caché en lugar de la antigua. Cada navegador y cada proxy validan las respuestas de la caché cuando los clientes solicitan los recursos correspondientes. Pero, no necesitan realizar una validación si la respuesta almacenada está suficientemente actualizada. Incluso aunque los servidores web conozcan cuándo se ha actualizado un recurso, no lo notifica al navegador o proxy con caché, para poder hacer esto el usuario necesitaría mantener un registro de cada navegador y cada proxy interesado en cada uno de sus recursos. Para permitir que un navegador o un proxy determinen si sus respuestas almacenadas están obsoletas, los servidores web asignan tiempos de expiración aproximada a sus recursos, estimados, por ejemplo, desde la última vez que fueron actualizados. Un tiempo de expiración puede resultar incorrecto, dado que un recurso web puede actualizarse en cualquier instante. Cuando un servidor web responde a una solicitud, el tiempo de expiración del recurso y el tiempo actual del servidor se adjuntan a la respuesta.

Cada navegador y proxy almacena el tiempo de expiración y el del servidor junto con la respuesta que se almacena en la caché. Esto permite que un navegador o proxy que reciban futuras solicitudes calculen si la respuesta almacenada en la caché está obsoleta. Esto se calcula comparando su edad con el tiempo de expiración. La *edad* de una respuesta es la suma del tiempo en el que la respuesta fue guardada en la caché y el tiempo del servidor. Hay que señalar que este cálculo no depende de que los relojes del servidor web y del navegador o proxy estén de acuerdo entre sí.

◇ **Aspectos de fiabilidad.** La fiabilidad es un requisito en la mayoría de los dominios de aplicación. Es crucial no sólo en actividades de control y gobierno, en las que la vida puede estar en juego, sino también en muchas aplicaciones comerciales, incluyendo las de comercio en Internet, que han experimentado un notable crecimiento, en las que la seguridad y solidez financiera de los participantes depende de la fiabilidad de los sistemas sobre los que operan. En la Sección 2.1 se ha definido la fiabilidad de los sistemas informáticos como *corrección*, *seguridad* y *tolerancia a fallos*. Aquí se discute el impacto arquitectónico de las necesidades de seguridad y tolerancia a fallos, dejando la descripción de las técnicas relevantes para su consecución para más adelante en el

libro. El desarrollo de técnicas para comprobar y asegurar la corrección de los programas distribuidos y concurrentes es un tema de investigación actual e importante. A pesar de presentar resultados prometedores, sólo algunos de éstos se encuentran suficientemente maduros para su implantación en aplicaciones reales.

Tolerancia frente a fallos: Las aplicaciones estables deben continuar funcionando correctamente en presencia de fallos en el hardware, el software y las redes. La fiabilidad se consigue introduciendo redundancia; proporcionando múltiples recursos de modo que el software de sistema y aplicación pueda ser reconfigurado y seguir realizando sus tareas en presencia de fallos. La redundancia es costosa y existen límites a la amplitud en que puede ser empleada; en consecuencia también hay límites en el grado de tolerancia que puede obtenerse.

A nivel arquitectónico, la redundancia precisa el empleo de varios computadores donde lanzar cada proceso componente del sistema y múltiples caminos de comunicación sobre los que transmitir los mensajes. Los datos y los procesos pueden replicarse donde quiera que sea necesario para proporcionar el debido nivel de tolerancia a fallos. Una forma usual de redundancia es la presencia de varias réplicas de los datos en diferentes computadores; en tanto en cuanto uno cualquiera de los computadores continúe funcionando, se podrá acceder a los datos. Algunas aplicaciones críticas, tales como los sistemas de control de tráfico aéreo, precisan un alto nivel de garantías en cuanto a la tolerancia frente a fallos, lo que implica un alto coste de mantenimiento de la actualización de las múltiples réplicas existentes. En el Capítulo 14 se discute esta materia más profundamente.

Para hacer fiables los protocolos de comunicación se emplean otras técnicas. Por ejemplo, los mensajes se retransmiten hasta que se ha recibido un mensaje de reconocimiento. La fiabilidad de los protocolos subyacentes a RMI se cubren en los Capítulos 4 y 5.

Seguridad: El impacto arquitectónico de los requisitos de seguridad afecta a la necesidad de ubicar los datos y otros recursos sensibles sólo en aquellos computadores equipados de un modo eficaz contra ataques. Por ejemplo, una base de datos de un hospital alojará registros con un componente sensible sobre los pacientes y sólo ciertos empleados podrán acceder a ellos, mientras que otros componentes de los mismos registros pueden estar disponibles más ampliamente. No sería apropiado construir un sistema en el que al acceder a los datos de un paciente se descargara el registro completo de éste en el computador de sobremesa del usuario, dado que el computador típico de sobremesa no constituye un entorno seguro, y los usuarios podrían lanzar programas para acceder o actualizar cualquier parte de los datos almacenados en su computador personal. En la Sección 2.3.3 se presenta un modelo de seguridad que trata con requisitos amplios para la seguridad, el Capítulo 7 describirá las técnicas disponibles para obtenerlo.

2.3. MODELOS FUNDAMENTALES

Todos los modelos de sistemas anteriores, aunque bien diferentes, comparten algunas propiedades fundamentales. En particular, todos ellos se componen de procesos que se comunican unos con otros mediante el envío de mensajes en una red de computadores. Todos los modelos comparten los requisitos de diseño dados en la sección previa, y que conciernen principalmente a las características de prestaciones y fiabilidad de los procesos y las redes, y a la seguridad de los recursos en el sistema. En esta sección, presentamos modelos basados en las propiedades fundamentales que nos permiten ser más específicos sobre las características y los riesgos de fallos y seguridad que puedan exhibir.

En general, un modelo contiene solamente aquellos ingredientes esenciales que necesitamos para comprender y razonar sobre algunos aspectos del comportamiento del sistema. Un modelo de sistema debe tratar las siguientes cuestiones:

- ¿Cuáles son las entidades principales en el sistema?
- ¿Cómo interactúan?
- ¿Cuáles son las características que afectan su comportamiento individual y colectivo?

El objetivo de un modelo es:

- Hacer explícitas todas las premisas relevantes sobre los sistemas que estamos modelando.
- Hacer generalizaciones respecto a lo que es posible o no, dadas las premisas anteriores. Las generalizaciones pueden tomar la forma de algoritmos de propósito general o de propiedades deseables que se garantizan. Las garantías surgirán del análisis lógico y, cuando sea pertinente, de la prueba matemática.

Hay mucho que ganar si conocemos lo que hacen y de lo que dependen nuestros diseños, y de lo que no. Así podremos decidir si un diseño funcionará si intentamos implementarlo en un sistema particular: sólo necesitamos preguntarnos si nuestras premisas son válidas en ese sistema. A la vez, aclarando y explicitando nuestras premisas, podemos esperar demostrar propiedades del sistema empleando técnicas matemáticas. Estas propiedades valdrán para cualquier sistema que concuerde con nuestras premisas. Finalmente, al abstraer sólo entidades y características esenciales del sistema (y no detalles, como el hardware) podemos arrojar luz sobre nuestra comprensión de estos sistemas.

Los aspectos de los sistemas distribuidos que queremos capturar en nuestros modelos fundamentales están destinados a ayudarnos a discutir y razonar sobre:

Interacción: el cómputo ocurre en los procesos; los procesos interaccionan por paso de mensajes, lo que deviene en comunicación (esto es, flujo de información) y coordinación (sincronización y ordenamiento de actividades) entre procesos. En el análisis y diseño de los sistemas distribuidos trataremos especialmente con estas interacciones. El modelo de interacción debe reflejar hechos como que la comunicación tiene lugar con retrasos, que a menudo son de considerable duración, y la precisión con que la pueden coordinarse procesos independientes está limitada por esos retrasos y por la dificultad de mantener la misma noción de tiempo en todos los computadores de un sistema distribuido.

Fallo: la correcta operación de un sistema distribuido se ve amenazada allá donde aparezca un fallo en cualquiera de los computadores sobre el que se ejecuta (incluyendo fallos de software) o en la red que los conecta. Nuestro modelo define y clasifica los fallos. Esto nos da la base para el análisis de los efectos potenciales de los fallos y para el diseño de sistemas que tolerando fallos de cualquier tipo puedan seguir funcionando correctamente.

Seguridad: la naturaleza modular de los sistemas distribuidos y su extensibilidad los expone a ataques tanto de agentes externos como internos. Nuestro modelo de seguridad define y clasifica los modos en que pueden tener lugar tales ataques, y proporciona una base para el análisis de las amenazas a un sistema con el fin de diseñar sistemas capaces de resistirse a ellas.

Para discutir y razonar más fácilmente, los modelos presentados en este capítulo están necesariamente simplificados, omitiéndose muchos detalles de los sistemas reales. Su relación con los sistemas del mundo real y la solución que esos modelos nos ayudan a traer a colación, en ese contexto, son el tema principal de este libro.

2.3.1. MODELO DE INTERACCIÓN

La discusión de arquitecturas de sistemas de la Sección 2.2 indica que los sistemas distribuidos se componen de muchos procesos, y que interactúan de formas complejas. Por ejemplo:

- Múltiples procesos servidores pueden cooperar entre sí para proporcionar un servicio; los ejemplos mencionados anteriormente son el Servicio de Nombres de Dominio, que parte y

replica sus datos en los servidores a lo largo de Internet; y el Servicio de Información en Red de Sun, que mantiene copias replicadas de los archivos de contraseñas en varios servidores de una red de área local.

- Un conjunto de procesos similares pueden cooperar entre sí para lograr una meta común: por ejemplo, un sistema de conferencia para voz que distribuye flujos de datos de audio de forma similar, pero con estrictas restricciones de tiempo real.

La mayoría de los programadores estarán familiarizados con el concepto de *algoritmo*: secuencia de pasos que hay que seguir para realizar un cómputo deseado. Los programas simples están controlados por algoritmos en los que los pasos son estrictamente secuenciales. El comportamiento del programa y el estado de las variables del programa está determinado por aquéllos. Tal programa se ejecuta como un proceso aislado. Los sistemas distribuidos compuestos de varios procesos, tales como los delineados anteriormente, son más complejos. Su comportamiento y el estado puede describirse mediante un *algoritmo distribuido*: una definición de los pasos que hay que llevar a cabo por cada uno de los procesos de que consta el sistema, *incluyendo los mensajes de transmisión entre ellos*. Los procesos transmiten mensajes de uno a otro para transferir información y coordinar su actividad.

La tasa con que procede cada proceso y la temporización de la transmisión de los mensajes entre ellos no puede predecirse, en general. También es difícil describir todos los estados de un algoritmo distribuido, dado que éste debe tratar con los fallos de uno, o más, de los procesos involucrados, o el fallo en la transmisión de los mensajes.

En un sistema distribuido toda la acción la soportan los procesos que interactúan. Cada proceso tiene su propio estado, que consiste en el conjunto de datos a los que puede acceder y actualizar, incluyendo las variables internas al programa. El estado concerniente a cada proceso es completamente privado, esto es, no puede ser accedido o actualizado por otro proceso.

En esta sección, discutiremos dos factores significativos que afectan a los procesos interactuantes de un sistema distribuido:

- Las prestaciones de las comunicaciones son con frecuencia una característica limitadora.
- Es imposible mantener una única noción global del tiempo.

◇ **Prestaciones de los canales de comunicaciones.** Los canales de comunicación, en nuestro modelo, se implementan de muchas formas en los sistemas distribuidos; por ejemplo mediante una implementación de *streams* o por un simple paso de mensajes sobre la red de computadores. Las características de prestaciones de la comunicación sobre una red de computadores incluyen la latencia, el ancho de banda y las fluctuaciones:

- El retardo entre el envío de un mensaje por un proceso y su recepción por otro se denomina *latencia*. La latencia incluye:
  - — El tiempo que se toma en llegar a su destino el primero de una trama de bits transmitidos a través de una red. Por ejemplo, la latencia para la transmisión de un mensaje mediante un enlace por satélite es el lapso de tiempo en que la señal va y vuelve del satélite.
  - — El retardo en acceder a la red, que es grande cuando la red está muy cargada. Por ejemplo, al transmitir sobre Ethernet la estación de emisión esperará a que la red esté libre de tráfico.
  - — El tiempo empleado por los servicios de comunicación del sistema operativo tanto en el proceso que envía como en el que recibe, y que varía según la carga actual de cada sistema operativo.
- El *ancho de banda* de una red de computadores es la cantidad total de información que puede transmitirse en un intervalo de tiempo dado. Cuando una red está siendo utilizada por un número grande de canales de comunicación, habrá que compartir el ancho de banda disponible.

- La fluctuación (*jitter*) es la variación en el tiempo invertido en completar el reparto de una serie de mensajes. La fluctuación es importante para los datos multimedia. Éste es el caso que si se reproducen muestras de audio consecutivas a un ritmo variable el sonido presentará graves distorsiones.

◇ **Relojes de computadores y eventos de temporización.** Cada computador de un sistema distribuido tiene su propio reloj interno; los procesos locales obtienen de él, el valor del tiempo actual. Ésta es la forma en que dos procesos en ejecución sobre dos computadores diferentes asocian marcas (o sellos) temporales a sus eventos. Sin embargo, incluso si dos procesos leen sus relojes a la vez, sus relojes locales proporcionarán valores de tiempo diferentes. Esto ocurre porque los relojes presentan derivas con respecto al tiempo perfecto y, más importante, sus tasas de deriva difieren de una a otra. El término *tasa de deriva del reloj* alude a la proporción en que el reloj de un computador difiere del reloj de referencia perfecto. Incluso, si los relojes de todos los computadores de un sistema distribuido se ponen en hora a la vez, a partir de un tiempo sus relojes variarán en una magnitud significativa a menos que se apliquen correcciones.

Hay varias aproximaciones a la corrección de los tiempos del reloj de los computadores. Así, podría utilizarse receptores de radio para obtener lecturas de tiempo de un GPS con aproximaciones cercana al microsegundo. Pero los receptores GPS no operan dentro de los edificios, ni se justifica el coste para cada computador. En su lugar, un computador dotado de una fuente de tiempo precisa, como un GPS, sí que podría enviar mensajes de temporización a otros computadores de la red. La concordancia final entre los tiempos de los relojes locales está, por supuesto, afectada por retardos variables en los mensajes. Para una discusión más detallada de la deriva del reloj y sincronización del reloj, véase el Capítulo 11.

◇ **Dos variantes del modelo de interacción.** En un sistema distribuido es difícil establecer cotas sobre el tiempo que debe tomar la ejecución de un proceso, el reparto de un mensaje o la deriva del reloj. Tenemos dos modelos simples que parten de posiciones extremas y opuestas: el primero tiene en cuenta una fuerte restricción sobre el tiempo; en el segundo no se hace ninguna presuposición.

Sistemas distribuidos síncronos: Hadzilacos y Toueg [1994] definen un sistema distribuido síncrono como aquel en el que se establecen los siguientes límites:

- El tiempo de ejecución de cada etapa de un proceso tiene ciertos límites inferior y superior conocidos.
- Cada mensaje transmitido sobre un canal se recibe en un tiempo limitado conocido.
- Cada proceso tiene un reloj local cuya tasa de deriva sobre el tiempo real tiene un límite conocido.

Es posible sugerir valores parecidos para los límites superior e inferior del tiempo de ejecución de un proceso, el retardo de mensajes y las tasas de deriva de relojes en un sistema distribuido. Pero es difícil obtener valores realistas y además dar garantías sobre los valores elegidos. A menos que podamos garantizar los límites, cualquier diseño basado en los valores elegidos no será fiable. Sin embargo, pudiera ser útil modelar un algoritmo como un sistema síncrono para dar una idea de cómo se comportará en un sistema distribuido real. En un sistema síncrono, es posible utilizar tiempo límite (*timeouts*), por ejemplo para detectar el fallo de un proceso, tal y como se muestra en la sección que trata del modelo de fallo.

Se pueden construir sistemas distribuidos síncronos. Esto requiere que los procesos implementen tareas con requisitos de recursos conocidos, para los que se pueda garantizar ciclos suficientes de procesador y capacidad de red; y que se provea a estos procesos de relojes con tasas de deriva limitada.

Sistemas distribuidos asíncronos: muchos sistemas distribuidos, por ejemplo Internet, son de gran utilidad aun sin ser sistemas síncronos. En consecuencia necesitamos un modelo alternativo: un sistema distribuido asíncrono es aquel en que no existen limitaciones sobre:

- La velocidad de procesamiento (por ejemplo, un paso de un proceso puede requerir tan sólo un picosegundo y otro un siglo; lo único que podemos decir es que cada paso se tomará un tiempo arbitrariamente largo).
- Los retardos de transmisión de mensaje (por ejemplo, al repartir un mensaje de un proceso A a otro B puede tardar un tiempo cero o bien varios años. En otras palabras, un mensaje puede recibirse tras un tiempo arbitrariamente largo).
- Las tasas de deriva de reloj (de nuevo, la tasa de deriva de reloj es arbitraria).

El modelo asíncrono no presupone nada sobre los intervalos de tiempo involucrados en cualquier ejecución. Éste es exactamente el modelo de Internet, en él no hay límite intrínseco en la carga del servidor o la red (en consecuencia no se sabe cuánto tomará, por ejemplo, transferir un archivo usando ftp). A veces un mensaje de correo puede tomarse varios días en llegar. El recuadro de la página siguiente ilustra la dificultad de llegar a un acuerdo en un sistema distribuido asíncrono.

Aun así con estas premisas se pueden resolver algunos problemas de diseño. Por ejemplo, a pesar de que el Web no siempre puede proveer una respuesta en un límite de tiempo razonable, los navegadores se diseñan para permitir que los usuarios realicen otras tareas mientras están a la espera. Cualquier solución válida para un sistema distribuido asíncrono es válida también para un sistema síncrono.

Los sistemas distribuidos reales son frecuentemente asíncronos a causa de la necesidad de los procesos de compartir procesadores y de los canales de comunicación de compartir la red. Por ejemplo, si demasiados procesos de índole desconocida comparten un procesador, no habrá garantías sobre las prestaciones resultantes de ninguno de ellos. Pero existen muchos problemas de diseño que no pueden ser resueltos para un sistema asíncrono que pueden resolverse cuando se tienen en cuenta algunos aspectos temporales. La necesidad de que cada elemento de un flujo de datos multimedia se reparta antes de un tiempo límite es un problema de este tipo. Para problemas como éste se requiere un modelo síncrono. El recuadro de más adelante ilustra la imposibilidad de sincronizar los relojes en un sistema asíncrono.

◇ **Ordenamiento de eventos.** En muchos casos, nos interesa saber si un evento (enviar o recibir un mensaje) en un proceso ocurrió antes, después o concurrentemente con otro evento en algún otro proceso. La ejecución de un sistema puede describirse en términos de los eventos y su ordenación aun careciendo de relojes precisos.

Por ejemplo, considere el siguiente conjunto de intercambios entre un pequeño grupo de usuarios X, Y, Z y A de una lista de correo:

1. El usuario X envía un mensaje con el tema *Reunión*.
2. Los usuarios Y y Z responden con un mensaje con el tema *Re: Reunión*.

En tiempo real, se envía primero el mensaje de X, Y lo lee y responde; Z lee ambos mensajes, el de X y la respuesta de Y y entonces envía otra respuesta, que hace referencia a los dos anteriores. Pero debido a los retardos independientes en el reparto de los mensajes, los mensajes pudieran haber sido repartidos como se muestra en la Figura 2.9, y algunos usuarios podrían ver estos mensajes en el orden equivocado, por ejemplo, el usuario A podría ver:

	Bandeja de entrada:	
De	*Asunto*	*Elemento*
23	Z	Re: Reunión
24	X	Reunión
25	Y	Re: Reunión

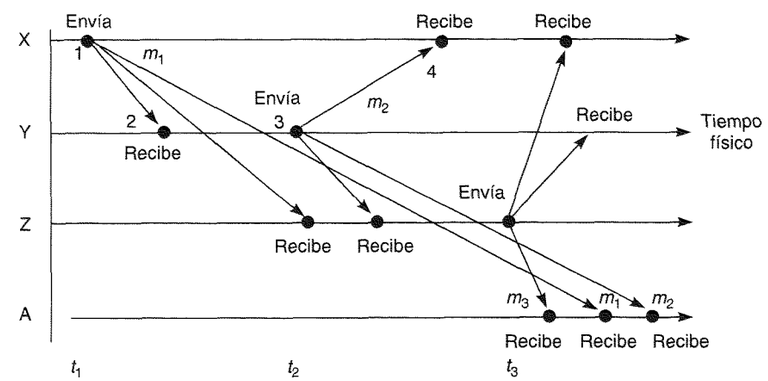


Figura 2.9. Ordenamiento real en el tiempo de los eventos.

◇ **Acuerdo en Pimientolandia.** Dos divisiones del ejército de Pimientolandia «Manzana» y «Naranja», se encuentran acampadas en lo alto de dos colinas adyacentes. Por otro lado a lo largo del valle entre ambas están los invasores Tontorrones Azules. Las divisiones de Pimientolandia se encuentran seguras tanto en cuanto permanezcan en sus campamentos, y puedan enviar mensajeros con seguridad a través del valle para comunicarse. Las divisiones de Pimientolandia necesitan ponerse de acuerdo sobre cuál de ellas encabezará la carga contra los Tontorrones Azules, y cúando tendrá lugar. Incluso en una Pimientolandia asíncrona, es posible acordar sobre quién conducirá la carga. Por ejemplo, cada división comunica el número de sus efectivos, y el que tenga más atacará primero (si están empatados, la división Manzana gana sobre la Naranja). Pero, ¿cuándo deben cargar? Desafortunadamente, en una Pimientolandia asíncrona, los mensajeros se desplazan con una velocidad muy variable. Pongamos que, Manzana envía un mensajero con el texto «¡Carguen!», Naranja podría no recibir el mensaje hasta, digamos, tres horas; o bien pudiera ser que fueran cinco minutos. En una Pimientolandia síncrona, aun existiría un problema de coordinación, pero las divisiones saben de ciertas restricciones útiles: cada mensaje toma al menos *min* minutos y a lo más *max* minutos en llegar. Si la división que conduce la carga envía un mensaje «¡Carguen!», esperaría durante *min* minutos y entonces cargaría. La otra división esperaría durante 1 minuto tras recibir el mensaje y cargaría a continuación. Está garantizado que su carga ocurrirá siempre en segundo lugar, pero no más tarde de $(max - min + 1)$ minutos.

Si los relojes de los computadores de X, Y y Z pudieran sincronizarse, podría utilizarse el tiempo asociado localmente a cada mensaje enviado. Por ejemplo, los mensajes m_1, m_2 y m_3 llevarían los tiempos t_1, t_2 y t_3 donde $t_1 < t_2 < t_3$. Los mensajes recibidos se mostrarían a los usuarios según este ordenamiento temporal. Si los relojes están sincronizados aproximadamente la temporización ocurrirá en el orden correcto.

Como los relojes de un sistema distribuido no pueden sincronizarse de manera perfecta, Lamport [1987] propuso un modelo de *tiempo lógico* que pudiera utilizarse para ordenar los eventos de procesos que se ejecutan en computadores diferentes de un sistema distribuido. El tiempo lógico permite inferir el orden en que se presentan los mensajes sin recurrir a relojes. Se presenta en detalle en el Capítulo 10, aunque aquí bosquejaremos cómo pueden aplicarse algunos aspectos del ordenamiento lógico a nuestro problema de ordenamiento de correos electrónicos.

Es obvio que los mensajes se reciben después de su envío, así podemos admitir un ordenamiento lógico para los pares de sucesos que se muestran en la Figura 2.9; por ejemplo, considerando sólo los eventos relativos a X e Y:

X envía m_1 antes de que Y reciba m_1; Y envía m_2 antes de que X reciba m_2.

También sabemos que las respuestas se reciben después de que se reciben los mensajes, de modo que tenemos el siguiente ordenamiento lógico para Y:

Y recibe m_1 antes de enviar m_2.

El tiempo lógico lleva esta idea más lejos asignando un número a cada evento, que se corresponde con su ordenamiento lógico, de modo que los últimos eventos tendrán números mayores que los primeros. Como ejemplo, en la Figura 2.9 se consignan los números 1 a 4 en los eventos en X e Y.

2.3.2. MODELO DE FALLO

En un sistema distribuido pueden fallar tanto los procesos como los canales de comunicación; esto es, podrían apartarse de lo que se considera el comportamiento correcto o deseable. El modelo de fallo define las formas en que puede ocurrir el fallo para darnos una comprensión de los efectos de los fallos. Hadzilacos y Toueg [1994] proponen una taxonomía que distingue entre los fallos de procesos y los fallos de canales de comunicación. Éstos se presentan bajo los encabezamientos: fallos por omisión, fallos arbitrarios y fallos de temporización.
Este modelo de fallo se utilizará a lo largo del libro. Éste es el caso de:

- En el Capítulo 4, donde presentamos las interfaces Java de la comunicación de tipo *datagrama* y *stream*, que proporcionan diferentes grados de fiabilidad.
- El Capítulo 4 en el que presentamos el protocolo petición-respuesta, que subyace a RMI. Sus características de fallo dependen de las características de fallo tanto de los procesos como de los canales de comunicación. El protocolo puede construirse mediante comunicación basada en datagramas o streams. La elección podrá decidirse en función de la simplicidad de la implementación, las prestaciones y la fiabilidad.
- El Capítulo 13 donde se presenta el protocolo de realización en dos fases (*two-phase commit*) para transacciones. Está diseñado para completarse en presencia de fallos bien definidos de los procesos y de los canales de comunicación.

◇ **Fallos por omisión.** Las faltas clasificadas como *fallos por omisión* se refieren a casos en los que los procesos o los canales de comunicación no consiguen realizar acciones que se suponen que pueden hacer.

Fallos por omisión de procesos: El principal fallo por omisión de un proceso es el fracaso o ruptura accidentada del procesamiento (*crash*). Cuando decimos que un proceso se rompe queremos decir que ha parado y no ejecutará ningún paso de programa más. El diseño de servicios que puedan sobrevivir en presencia de fallos se puede simplificar si asumimos que los servicios de los que depende fracasan limpiamente; es decir, los procesos de los que depende o funcionan correctamente o se detienen. Otros procesos pueden ser capaces de detectar tales fracasos por el hecho de que los procesos rotos fallan repetidamente en responder a los mensajes de invocación. Desgraciadamente, este método de detección de rupturas descansa en el uso de *timeouts*; es decir un método en el cual un proceso otorga un período fijo de tiempo para que algo ocurra. En un sistema asíncrono un timeout puede indicar tan sólo que un proceso no responde; puede que se haya roto o sea lento, o que los mensajes no hayan llegado.

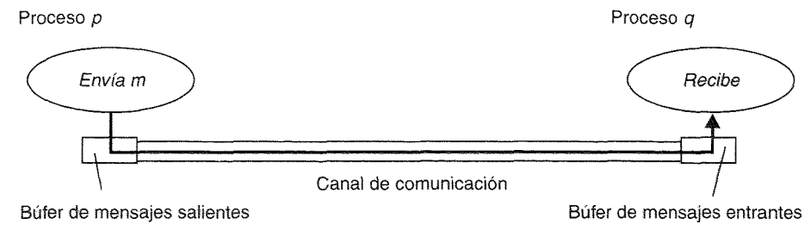


Figura 2.10. Procesos y canales.

La rotura de un proceso se denomina fallo-parada (*fail-stop*) si los otros procesos pueden detectar con certeza que el proceso ha fracasado. El comportamiento fallo-parada puede producirse en un sistema síncrono si el proceso utiliza timeouts para detectar cuando los otros procesos fallan en la respuesta y está garantizado que los mensajes llegan. Por ejemplo, si los procesos *p* y *q* están programados para que *q* responda a un mensaje de *p*, y el proceso *p* no ha recibido respuesta del proceso *q* en un tiempo máximo medido por el reloj local de *p*, entonces el proceso *p* puede concluir que el proceso *q* ha fallado. El recuadro próximo ilustra la dificultad de detectar los fallos en un sistema asíncrono o de llegar a un acuerdo en presencia de fallos.

Fallos por omisión de comunicaciones: Considere las primitivas de comunicación *envía* y *recibe*. Un proceso *p* realiza un *envío* insertando el mensaje *m* en su búfer de mensajes salientes. El canal de comunicación transporta *m* al búfer de mensajes entrantes de *q*. El proceso *q* realiza un *recibe* tomando *m* de su búfer de mensajes entrantes y repartiéndolo (véase la Figura 2.10). Los búferes entrante y saliente se proporcionan usualmente desde el sistema operativo.

El canal de comunicación produce un fallo de omisión si no transporta un mensaje desde el búfer de mensajes salientes de *p* al búfer de mensajes entrantes de *q*. A esto se denomina «perder mensajes» y su causa suele ser la falta de espacio en el búfer de recepción de alguna pasarela de entre medias, o por un error de red, detectable por una suma de chequeo de los datos del mensaje. Hadzilacos y Toueg [1994] se refieren a la pérdida de mensajes entre el proceso emisor y el búfer de mensajes de salida como *fallos por omisión de envío*; la pérdida de mensajes entre el búfer de mensajes de entrada y el proceso receptor como *fallos por omisión de recepción*; y la pérdida de mensajes entre los búferes como *fallos por omisión del canal*. En la Figura 2.11 se clasifican los fallos por omisión junto con los fallos arbitrarios.

◇ **Detección de fallos.** En el caso de las divisiones de Pimientolandia acampadas en la cumbre de las colinas (véase la página 49), suponga que los Tontorrones Azules son, después de todo, suficientemente fuertes como para atacar y vencer a cualquier división mientras está acampada (es decir, que cualquiera de ambas puede fallar). Suponga aún más, mientras se encuentran invictas, las divisiones envían mensajeros regularmente para comunicar su estado. En un sistema asíncrono, ninguna división puede distinguir si la otra ha sido derrotada o si es que el tiempo que emplean los mensajeros en cruzar el valle entre ambas es un poco largo. En una Pimientolandia síncrona, una división puede tener por cierto si la otra ha sido derrotada, por la ausencia del puntual mensajero. Sin embargo, la otra división pudiera haber sido derrotada justo después de haber enviado su último mensajero.

◇ **Imposibilidad de alcanzar un acuerdo en presencia de fallos.** Hemos estado presuponiendo que los mensajeros de Pimientolandia siempre se las arreglan para cruzar el valle de alguna forma; pero supongamos ahora que los Tontorrones Azules pueden capturar cualquier

mensajero e impedir que llegue. (Supondremos que, para los Tontorrones Azules, es imposible realizar un lavado de cerebro a los mensajeros para que den el mensaje falso; los Tontorrones no estaban al tanto de sus precursores, los traicioneros bizantinos.) ¿Podrán las divisiones Manzana y Naranja enviar mensajes de modo que puedan ponerse de acuerdo en decidir cargar contra los Tontorrones, o siquiera rendirse a la vez? Desafortunadamente, como probó el teórico pimentonés Ringo el Grande, en estas circunstancias las divisiones no pueden garantizar el decidir consistentemente qué hacer. Para convencerse de esto, suponga el contrario, que las divisiones manejan un protocolo Pimientonés que permite lograrlo. Cada uno propone «¡Carguen!» o «¡Ríndanse!» y el resultado es que ambos deciden ponerse de acuerdo en uno u otro curso de acción. Ahora considere el último mensaje enviado en cualquier secuencia del protocolo. El mensajero que lo lleva podría haber sido capturado por los Tontorrones Azules. De modo que el resultado debiera ser el mismo tanto si el mensaje llega como si no: podemos prescindir de él. Ahora apliquemos el mismo argumento al mensaje final que resta. Pero, aplicaríamos este argumento de nuevo al mensaje y ¡así hasta que no nos quedara nada de él! Esto muestra que no puede existir ningún protocolo que garantice el acuerdo entre las divisiones de Pimientolandia si los mensajeros pueden ser capturados.

Clase de fallo	*Afecta a*	*Descripción*
Fallo-parada	Proceso	El proceso para y permanece parado. Otros procesos pueden detectar este estado.
Ruptura	Proceso	El proceso para y permanece parado. Otros procesos pueden no ser capaces de detectar este estado.
Omisión	Canal	Un mensaje insertado en el búfer de mensajes salientes nunca llega al búfer de mensajes entrantes del otro extremo.
Omisión de envío	Proceso	Un proceso completa *envía*, pero el mensaje no es colocado en su búfer de mensajes salientes.
Omisión de recepción	Proceso	El mensaje es colocado en la cola de mensajes del proceso, pero el proceso no lo recibe.
Arbitrario (Bizantino)	Proceso o canal	El proceso/canal presenta un comportamiento arbitrario: puede enviar/transmitir arbitrariamente mensajes en instantes arbitrarios, cometer omisiones; un proceso puede parar o realizar un paso incorrecto.

Figura 2.11. Fallos por omisión y fallos arbitrarios.

Los fallos pueden categorizarse según su severidad. Todos los fallos que hemos descrito hasta el momento son *benignos*. La mayoría de los fallos de los sistemas distribuidos son de esta clase. Los fallos benignos incluyen fallos por omisión así como fallos de temporización y fallos de prestaciones.

◇ **Fallos arbitrarios.** El término *arbitrario*, o fallo bizantino, se emplea para describir la peor semántica de fallo posible, en la que puede ocurrir cualquier tipo de error. Por ejemplo, un proceso podría cargar valores equivocados en su área de datos, o podría responder un valor incorrecto a una petición.

Un fallo arbitrario en un proceso es aquel en el que se omiten pasos deseables para el procesamiento o se realizan pasos no intencionados de procesamiento. En consecuencia los fallos arbitrarios en los procesos no pueden detectarse observando si el proceso responde a las invocaciones, dado que podría omitir arbitrariamente la respuesta.

Los canales de comunicación sufren fallos arbitrarios; por ejemplo: los contenidos de un mensaje pueden estar corrompidos o se pueden repartir mensajes no existentes, o incluso algunos

Clase de fallo	*Afecta a*	*Descripción*
Reloj	Proceso	El reloj local del proceso excede el límite de su tasa de deriva sobre el tiempo real.
Prestaciones	Proceso	El proceso excede el límite sobre el intervalo entre dos pasos.
Prestaciones	Canal	La transmisión de un mensaje toma más tiempo que el límite permitido.

Figura 2.12. Fallo de temporización.

mensajes auténticos pueden repartirse más de una vez. Los fallos arbitrarios de los canales de comunicación son raros porque el software de comunicación es capaz de reconocerlos y rechazar los mensajes fallidos. Por ejemplo, para detectar mensajes corrompidos empleamos sumas de chequeo, y podemos emplear números de secuencia para detectar mensajes no existentes y mensajes duplicados.

◇ **Fallos de temporización.** Los fallos de temporización se aplican en los sistemas distribuidos síncronos donde se establecen límites en el tiempo de ejecución de un proceso, en el tiempo de reparto de un mensaje y en la tasa de deriva de reloj. Los fallos de temporización se muestran en la Figura 2.12. Cualquiera de estos fallos puede ocasionar que las respuestas no estuvieran disponibles para los clientes en un intervalo de tiempo dado.

En un sistema distribuido asíncrono, un servidor sobrecargado podría responder demasiado lentamente, pero no podemos decir que haya habido un fallo de temporización dado que no se ha ofrecido ninguna garantía al respecto.

Los sistemas operativos de tiempo real se diseñan con vistas a proporcionar garantías de temporización, pero son más complejos de diseñar y pueden requerir hardware redundante. La mayoría de los sistemas operativos de propósito general, como UNIX, no tienen que cumplir restricciones de tiempo real.

La temporización es particularmente relevante en los computadores multimedia con canales de audio y vídeo. La información de vídeo puede requerir la transferencia de una gran cantidad de datos. El reparto de tal cantidad de información, sin fallos de temporización, puede exigir demandas especiales tanto en el sistema operativo como en el sistema de comunicación.

◇ **Enmascaramiento de fallos.** Cada componente de un sistema distribuido se construye generalmente a partir de un conjunto de componentes ya existentes. Es posible construir servicios fiables con componentes que presenten fallos. Por ejemplo, múltiples servidores que alojen réplicas de datos pueden continuar prestando un servicio aunque uno de ellos se rompa. El conocimiento de las características de fallo de un componente puede permitirnos crear un nuevo servicio diseñado para enmascarar los fallos del primero. Un servicio *enmascara* un fallo, bien, ocultándolo completamente o convirtiéndolo en otro tipo de fallo más aceptable. Un ejemplo de este último es el cómo se utilizan las sumas de chequeo para enmascarar los mensajes corrompidos, convirtiendo un fallo arbitrario en un fallo por omisión. Veremos en los Capítulos 3 y 4 que los fallos por omisión pueden ocultarse empleando un protocolo que retransmita los mensajes que no alcanzan su destino. El Capítulo 14 presenta el enmascaramiento mediante replicación. Incluso la ruptura de procesos puede enmascararse, remplazando el proceso y restaurando su memoria de la información almacenada en el disco sobre su predecesor.

◇ **Fiabilidad y comunicación uno a uno.** Aunque un canal de comunicación básico pueda presentar fallos por omisión como los descritos anteriormente, es posible utilizarlo para construir un servicio de comunicación que enmascare algunos de esos fallos.

El término *comunicación fiable* se define en términos de validez e integridad, definidos como sigue:

Validez: Cualquier mensaje en el búfer de mensajes salientes será hecho llegar eventualmente al búfer de mensajes entrantes;

Integridad: El mensaje recibido es idéntico al enviado, y no se reparten mensajes por duplicado.

Las amenazas a la integridad provienen de dos fuentes independientes:

- Cualquier protocolo que retransmita mensajes pero no rechace un mensaje que llegue dos veces. Los protocolos pueden adjuntar números de secuencia a los mensajes para detectar aquellos que se reparten por duplicado.
- Usuarios malintencionados que insertan mensajes espurios, repiten mensajes antiguos o modifican mensajes auténticos. Se pueden tomar medidas de seguridad para mantener la propiedad de integridad en caso de tales ataques.

2.3.3. MODELO DE SEGURIDAD

En la Sección 2.2 identificamos la compartición de recursos como un factor motivador para los sistemas distribuidos y describimos su arquitectura de sistema en términos de procesos que encapsulan objetos y proporcionan acceso a ellos mediante interacciones con otros procesos. El modelo arquitectónico da la base de nuestro modelo de seguridad:

la seguridad de un sistema distribuido puede lograrse asegurando los procesos y los canales empleados para sus interacciones y protegiendo los objetos que encapsulan contra el acceso no autorizado.

La protección se describe en términos de objetos, aunque los conceptos se aplican tal cual a recursos de todos los tipos.

◇ **Protección de objetos.** La Figura 2.13 muestra un servidor que administra un conjunto de objetos en beneficio de algunos usuarios. Los usuarios pueden lanzar programas cliente que envían invocaciones al servidor para realizar operaciones sobre los objetos. El servidor realiza la operación indicada en cada invocación y envía el resultado al cliente.

Los objetos están construidos para ser usados de formas diferentes por usuarios diferentes. Por ejemplo, algunos objetos podrían incluir datos privados de un usuario, tales como un buzón de correo, y otros objetos pueden incluir datos compartidos como páginas web. Para dar soporte a esto los *derechos de acceso* especifican a quién se permite realizar ciertas operaciones de un objeto, por ejemplo, a quién se permite leer o escribir su estado.

Así, debemos incluir a los usuarios en nuestro modelo como los beneficiarios de los derechos de acceso. Esto se realiza asociando a cada invocación y a cada resultado la autoridad con que se ordena. Tal autoridad se denomina *principal*. Un principal pudiera ser un usuario o un proceso. En nuestra ilustración, la invocación proviene de un usuario y el resultado de un servidor.

El servidor es responsable de verificar la identidad del principal tras cada invocación y comprobar que dispone de los suficientes derechos de acceso como para realizar la operación requerida sobre el objeto particular invocado, rechazando aquellas que no dispongan de ellos. El cliente puede comprobar la identidad del principal tras el servidor para asegurar que el resultado proviene del servidor requerido.

◇ **Asegurar procesos y sus interacciones.** Los procesos interaccionan enviando mensajes. Los mensajes están expuestos a un ataque dado que la red y el servicio de comunicación que usan es abierto, de modo que cualquier par de procesos puedan interactuar. Los servidores y procesos parejos exponen sus interfaces, permitiendo ser el destino de los mensajes de otros procesos.

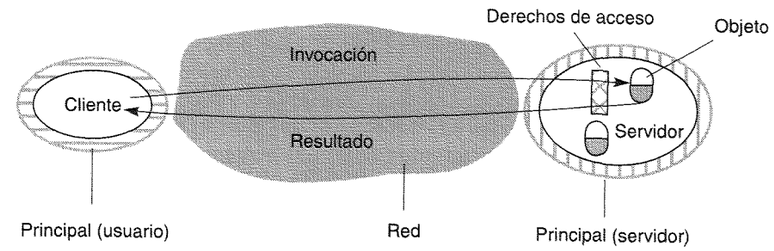


Figura 2.13. Objetos y principales.

Los sistemas distribuidos se despliegan y usan, probablemente, en tareas propensas a ser objetivo de ataques externos por usuarios hostiles. Esto es especialmente cierto en aplicaciones que tratan con transacciones financieras, confidenciales o con información clasificada o cualquier otra información cuya privacidad e integridad sea crucial. La integridad se ve amenazada por las violaciones de seguridad así como por fallos de la comunicación. Es así que sabemos que probablemente habrá ataques a los procesos de los que se componen tales aplicaciones y a los mensajes que viajan entre los procesos. Pero, ¿cómo podemos analizar estas amenazas para identificarlas y derrotarlas? La siguiente discusión presenta un modelo para el análisis de amenazas de seguridad.

◇ **El enemigo.** Para modelar amenazas de seguridad, postulamos que el enemigo (también conocido como adversario) es capaz de enviar cualquier mensaje a cualquier proceso y también de leer o copiar cualquier mensaje entre un par de procesos, como se muestra en la Figura 2.14. Tales ataques pueden cometerse simplemente utilizando un computador conectado a una red para lanzar un programa que lea los mensajes de la red dirigidos a otros computadores de la misma red, o un programa que genere mensajes que realicen peticiones falsas a servicios dando a entender que provienen de usuarios autorizados. El ataque puede provenir de un computador legítimamente conectado a la red o también de alguna conectada de forma no autorizada.

Las amenazas de un enemigo potencial se discutirán según los siguientes apartados: *amenazas a procesos*, *amenazas a los canales de comunicación* y *denegación de servicio*.

Amenazas a procesos: Cualquier proceso diseñado para admitir peticiones puede recibir un mensaje de cualquier otro proceso del sistema distribuido, y bien pudiera no ser capaz de determinar la identidad del emisor. Los protocolos de comunicación como IP incluyen la dirección del computador origen de cada mensaje, pero no es problema para un enemigo el generar un mensaje con una dirección fuente falsa. Esta carencia de conocimiento fiable del origen de un mensaje es una amenaza al correcto funcionamiento, tanto de los servidores como de los clientes, tal y como se explica a renglón seguido:

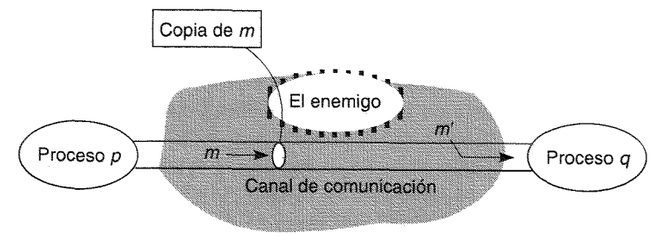


Figura 2.14. El enemigo.

Servidores: dado que un servidor puede recibir invocaciones de muchos clientes diferentes, no necesariamente puede determinar la identidad del principal que se halla tras cualquier invocación particular. Incluso si un servidor requiere la inclusión de la identidad del principal en cada invocación, un enemigo podría generar una invocación con una identidad falsa. Sin un conocimiento fiable de la identidad del emisor, un servidor no puede decir si realizar la operación o rechazarla. Por ejemplo, un servidor de correo pudiera no saber si el usuario tras una invocación que recupera un correo de un buzón particular está autorizado para ello, o si era una petición de un enemigo.

Clientes: cuando un cliente recibe el resultado de una invocación de un servidor, no necesariamente puede decir si la fuente del mensaje resultado es del servidor que se pretendía o de un enemigo, quizás uno que esté suplantando (*spoofing*) el servidor de correo. Entonces el cliente pudiera recibir un resultado no relacionado con la invocación original, tal como un correo falso (uno que no esté en el buzón de correo del usuario).

Amenazas a los canales de comunicación: Un enemigo puede copiar, alterar o insertar mensajes según viajan a través de la red y sus pasarelas intermedias. Tales ataques presentan una amenaza a la privacidad y a la integridad de la información según ésta viaja a través de la red, y a la integridad del sistema. Por ejemplo, un mensaje resultado que contenga un correo de un usuario pudiera ser visto por otro usuario o podría ser convertido en algo bastante diferente.

Otra forma de ataque es la pretensión de hacer acopio de mensajes para volver a enviarlos más tarde, haciendo posible volver a usar el mismo mensaje una y otra vez. Por ejemplo, alguien podría beneficiarse de un mensaje en el que se invoca una petición de transferencia de dinero de una cuenta bancaria a otra.

Todas estas amenazas pueden vencerse empleando *canales seguros*, como los que se describen más adelante y que se basan en la criptografía y la autenticación.

◇ **Vencer amenazas de seguridad.** Aquí presentamos las técnicas principales en que se basan los sistemas seguros. El Capítulo 7 discute el diseño y la implementación de sistemas distribuidos seguros con cierto detalle.

Criptografía y secretos compartidos: Suponga que un par de procesos (por ejemplo un cliente y un servidor concretos) comparten un secreto; es decir, ningún proceso del sistema distribuido aparte de ellos conoce este secreto. Entonces, si se intercambia un mensaje entre estos dos procesos e incluye información que demuestra el conocimiento, por parte del emisor, de este secreto compartido, tenemos que el receptor podrá estar seguro de quién es el que ha emitido el mensaje. Obviamente, es preciso tener cuidado para que el secreto compartido no sea revelado a un enemigo.

La *criptografía* es la ciencia de mantener los mensajes seguros, y la *encriptación* es el proceso de trastocar el mensaje de modo que quede oculto el contenido. La criptografía moderna se basa en algoritmos de encriptación que emplean claves secretas (números grandes difíciles de deducir) para transformar los datos, de forma que sólo se pueda revertir el proceso con la clave de desencriptado correspondiente.

Autenticación: El uso de los secretos compartidos y la encriptación presenta la base de los mensajes de *autenticación*, probando las identidades proporcionadas por sus emisores. La técnica básica de autenticación consiste en incluir en un mensaje un fragmento encriptado que contenga suficiente contenido del mensaje como para garantizar su autenticidad. La porción de autenticación de una petición a un servidor de archivos, pongamos que para leer parte de un archivo, podría incluir una representación de la identidad del principal solicitante, la identidad del archivo y la fecha y hora de la petición, todo encriptado con una clave secreta que comparten el servidor de archivos y el proceso solicitante. El servidor lo desencriptaría y comprobaría que se corresponde con los detalles no encriptados que se especifican en la petición.

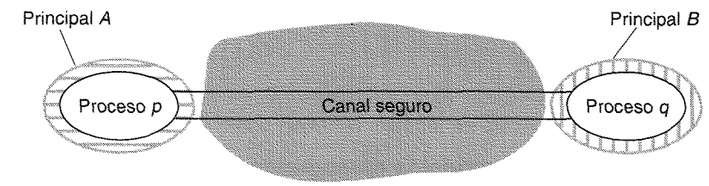


Figura 2.15. Canales seguros.

Canales seguros: La encriptación y la autenticación se emplean para construir canales seguros en forma de capa de servicio sobre los servicios de comunicación existentes. Un canal seguro es un canal de comunicación que conecta un par de procesos, cada uno de los cuales actúa en representación de un principal, según se muestra en la Figura 2.15. Un canal seguro presenta las siguientes propiedades:

- Cada proceso conoce bien la identidad del principal en cuya representación se ejecuta otro proceso. De este modo si un cliente y un servidor se comunican vía un canal seguro, el servidor conoce la identidad del principal tras las invocaciones y puede comprobar sus derechos de acceso antes de realizar una operación. Esto permite que el servidor proteja sus objetos correctamente y permite al cliente estar seguro de estar recibiendo resultados de un servidor que actúa de *buena fe*.
- Un canal seguro asegura la privacidad y la integridad (protección contra la manipulación) de los datos transmitidos por él.
- Cada mensaje incluye un sello de carácter temporal, de tipo físico o lógico, para prevenir el reenvío o la reordenación de los mensajes.

La construcción de canales seguros se discutirá con detalle en el Capítulo 7. Los canales seguros se han convertido en una importante herramienta práctica para asegurar el comercio electrónico y la protección de la comunicación. Las redes privadas virtuales (VPN, que se verán en el Capítulo 3) y el protocolo de capa de sockets segura (SSL, *Secure Sockets Layer*) son ejemplos de canales seguros.

◇ **Otras posibilidades de amenaza de un enemigo.** La Sección 1.4.3 presentó brevemente dos amenazas de seguridad, los ataques de denegación de servicio y el despliegue de código móvil. Veamos otra vez este tipo de ataques desde el punto de vista de las posibilidades que se le ofrecen al enemigo de dificultar las actividades de los procesos:

Denegación de servicio: ésta es una forma de ataque en la que el enemigo interfiere con las actividades de los usuarios autorizados mediante un número excesivo de invocaciones sin sentido sobre servicios o la red, lo que resulta en una sobrecarga de los recursos físicos (ancho de banda, capacidad de procesamiento del servidor). Tales ataques suelen tener el fin de retrasar o impedir las acciones de los otros usuarios. Por ejemplo, podría desconectarse la operación de las cerraduras electrónicas de una edificación por medio de un ataque que sature el computador que las controla con peticiones inválidas.

Código móvil: el código móvil despierta nuevos e interesantes problemas para cualquier proceso que reciba y ejecute código de programas proveniente de algún otro sitio, tal como los anexos a los correos electrónicos mencionados en la Sección 1.4.3. Tal código podría jugar fácilmente el papel de caballo de Troya, pretendiendo cumplir con un objetivo inocente pero, de hecho, incluyendo código que accede o modifica los recursos que están disponibles legítimamente para los procesos del host, pero no para el creador del código. Los métodos mediante los

cuales es posible llevar a cabo tales ataques son muchos y variados, en consecuencia el entorno del host debe construirse cuidadosamente para evitarlos. Muchas de estas cuestiones han sido tenidas en cuenta en Java y otros sistemas de código móvil, pero la historia reciente sobre el tema muestra la desnudez de algunas debilidades bastante embarazosas. Esta cuestión muestra la necesidad de realizar un análisis riguroso sobre el diseño de todos los sistemas seguros.

◇ **Aplicaciones de los modelos de seguridad.** Pudiera pensarse que obtener sistemas distribuidos seguros debiera ser un asunto evidente que implica el control del acceso a los objetos según permisos de acceso predefinidos y el empleo de canales de comunicación seguros. Desgraciadamente, no siempre es éste el caso. El empleo de técnicas de seguridad como la encriptación y el control de acceso conlleva un coste de procesamiento y administración sustancial. El modelo de seguridad esquematizado anteriormente da las bases del análisis y el diseño de sistemas seguros en los que se mantienen los costes al mínimo; pero las amenazas a un sistema distribuido se presentan desde muchos flancos, y se hace necesario un análisis cuidadoso de estas amenazas, que pudieran provenir de todas las fuentes posibles en el entorno de red del sistema, el entorno físico y el entorno humano. Este análisis involucra la construcción de un *modelo de amenazas* que tabula todas las formas de ataque a las que se encuentra expuesto el sistema y una evaluación de los riesgos y consecuencias de cada uno. La efectividad y el coste de las técnicas de seguridad necesarias pueden entonces ser contrapesadas con las amenazas.

2.4. RESUMEN

La mayoría de los sistemas distribuidos se componen de una o más variedades de modelos de arquitectura. El modelo cliente-servidor es el más usado; el Web y otros servicios Internet tales como ftp, news y mail así como el DNS se basan en este modelo, así como el sistema local de archivos y otros más. Los servicios como DNS que tienen un número de usuarios grande y tratan con una gran cantidad de información se basan en múltiples servidores, el uso de particiones sobre los datos y la replicación para facilitar la disponibilidad y la tolerancia frente a fallos. El uso de caché en los clientes y servidores *proxy* se encuentra ampliamente extendido para mejorar las prestaciones de un servicio. En el modelo de procesos de igual a igual, los procesos de aplicaciones distribuidas se comunican uno con otro directamente sin emplear servidores intermediarios.

La posibilidad de mover código de un proceso a otro ha desembocado en algunas variantes del modelo cliente-servidor. El ejemplo más común de esto es el *applet* cuyo código es aportado por un servidor web para ejecutarse en un cliente, proporcionando funcionalidades no disponibles en el cliente y la mejora de ciertas prestaciones debido a la proximidad con el cliente.

La existencia de computadores portátiles, PDA y otros dispositivos digitales y su integración en los sistemas distribuidos permite que los usuarios accedan a servicios locales y de Internet, aunque estén lejos de su computador de sobremesa. La presencia de dispositivos de computación con posibilidades de comunicación inalámbrica en aparatos cotidianos como las máquinas lavadoras o las alarmas antirrobo les permite dar servicios accesibles desde una red inalámbrica en el hogar. Una característica de los dispositivos móviles de los sistemas distribuidos es que pueden estar conectados o desconectados impredeciblemente, conduciendo a requisitos de descubrimiento de servicios mediante los que los servidores móviles puedan ofrecer sus servicios y los clientes móviles buscarlos.

Hemos presentado modelos de interacción, fallo y seguridad que identifican las características comunes de los componentes básicos con que se construyen los sistemas distribuidos. El modelo de interacción tiene que ver con las prestaciones de los procesos y los canales de comunicación, y con la ausencia de un reloj global. También define un sistema asíncrono como uno en el que no hay límites en el tiempo de ejecución de un proceso, el tiempo de reparto de un mensaje, y la deriva de los relojes; siendo así como se comporta Internet.

El modelo de fallo clasifica los fallos de los procesos y los canales de comunicación básicos en un sistema distribuido. El enmascaramiento es una técnica con la que se construye un servicio más fiable sobre otro menos fiable, de modo que se enmascaran algunos fallos que exhibe este último. En particular, se puede construir un servicio de comunicación fiable desde un canal de comunicación básico simplemente enmascarando sus fallos. Por ejemplo, sus fallos por omisión pueden enmascararse mediante la retransmisión de los mensajes perdidos. La integridad es una propiedad de la comunicación fiable, y requiere que un mensaje al ser recibido sea idéntico al que se envió, amén de que no sea recibido dos veces. La validez es otra propiedad; requiere que cualquier mensaje puesto en el búfer de mensajes de salida se reparta eventualmente al búfer de mensajes entrantes.

El modelo de seguridad identifica las posibles amenazas a los procesos y los canales de comunicación en un sistema distribuido abierto. Algunas de estas amenazas se relacionan con la integridad: los usuarios malintencionados pueden modificar o repetir los mensajes. Otros amenazan su privacidad. Otra cuestión de seguridad es la autenticación del principal (usuario o servidor) en cuyo nombre se envía un mensaje. Los canales seguros emplean técnicas criptográficas para asegurar la integridad y la privacidad de los mensajes y para autenticar pares de principales en una comunicación.

EJERCICIOS

2.1. Describa e ilustre la arquitectura cliente-servidor de una de las principales aplicaciones de Internet (por ejemplo el Web, email o netnews).

2.2. Para las aplicaciones discutidas en el Ejercicio 2.1 indique cómo cooperan los servidores al proveer un servicio.

2.3. ¿Cómo están involucradas, en el particionado y/o replicación (o el uso de caché) de los datos de ciertos servidores, las aplicaciones discutidas en el Ejercicio 2.1?

2.4. Un motor de búsqueda es un servidor web que ofrece a los clientes la oportunidad de buscar en ciertos índices almacenados y (concurrentemente) lanzar varios escaladores web para construir y actualizar estos índices. ¿Cuáles son los requisitos de sincronización entre estas actividades concurrentes?

2.5. Sugiera algunas aplicaciones para un modelo entre pares, distinguiendo entre casos en los que el estado de todos necesita ser idéntico y casos que demandan menos consistencia.

2.6. Tabule los tipos de recursos locales que son vulnerables a un ataque por un programa no fiable que se descarga de un lugar remoto y se ejecuta en un computador local.

2.7. Dé ejemplos de aplicaciones donde sea beneficioso emplear código móvil.

2.8. ¿Qué factores afectan el modo de comportamiento de una aplicación que accede a los datos compartidos administrados por un servidor? Describa los remedios disponibles y discuta su utilidad.

2.9. Distinga entre búfer y caché.

2.10. Dé algunos ejemplos de fallos en el hardware y el software de un sistema distribuido que puedan o no ser tolerados mediante el uso de redundancia. ¿En qué punto podemos asegurar que el empleo de redundancia, cuando sea adecuado, hace que el sistema sea tolerante frente a fallos?

2.11. Considere un servidor simple, que realiza peticiones de clientes, sin acceder a otros servidores. Explique por qué no es posible en general establecer un límite al tiempo que invierte un servidor en responder una petición de un cliente. ¿Qué habría que hacer para que el servidor fuera capaz de ejecutar respuestas dentro de un tiempo acotado? ¿Es práctica esta opción?

2.12. Para cada uno de los factores que contribuye al tiempo que se invierte en la transmisión de un mensaje entre dos procesos sobre un canal de comunicación, enumere qué medidas habría que emplear para acotar su contribución al tiempo total. ¿Por qué estas medidas no se incluyen en los sistemas distribuidos de propósito general actuales?

2.13. El servicio del Protocolo de Tiempo de Red se puede utilizar para sincronizar relojes de computadores. Explique por qué, incluso con este servicio, no se puede garantizar un límite para la diferencia de tiempo entre dos relojes.

2.14. Considere dos servicios de comunicación para utilizarse en sistemas distribuidos asíncronos. En el servicio A, los mensajes pueden perderse, duplicarse o retrasarse y la suma de chequeo sólo se aplica a las cabeceras. En el servicio B, los mensajes pueden perderse, retrasarse o distribuirse demasiado rápido para que el receptor pueda recibirlos, pero aquellos que llegan lo hacen en buenas condiciones.

Describa las clases de fallo que se presentan en cada servicio. Clasifique sus características según su efecto sobre la validez y la integridad. ¿Puede el servicio B, tal y como se ha descrito, ser un servicio de comunicación fiable?

2.15. Considere un par de procesos X e Y que emplean el servicio de comunicación B del Ejercicio 2.14 para comunicarse entre sí. Suponga que X es un cliente e Y un servidor y que una *invocación* consta de un mensaje de X a Y, tras lo cual Y obtiene resultado, seguido por un mensaje de respuesta de Y a X. Describa las clases de fallos que pueden presentarse en una invocación.

2.16. Suponga que una lectura básica de disco puede leer a veces valores que difieren de los escritos. Indique el tipo de fallo presentado por una lectura básica de disco. Sugiera cómo puede enmascararse este fallo para producir una forma diferente de fallo benigno. Ahora sugiera cómo enmascarar el fallo benigno.

2.17. Defina la propiedad de integridad de la comunicación fiable e indique todas las amenazas posibles a la integridad para los usuarios y para los componentes del sistema. ¿Qué medidas pueden tomarse para garantizar la propiedad de integridad en presencia de cada una de estas fuentes de ataques?

2.18. Describa las posibles ocurrencias de cada uno de los principales tipos de amenazas a la seguridad (amenazas a procesos, amenazas a canales de comunicación y denegación de servicio) que pueden acontecer en Internet.

3

REDES E INTERCONEXIÓN DE REDES

3.1. Introducción
3.2. Tipos de redes
3.3. Fundamentos de redes
3.4. Protocolos Internet
3.5. Casos de estudio: Ethernet, LAN inalámbrica y ATM
3.6. Resumen

Los sistemas distribuidos utilizan redes de área local, redes de área extendido e interredes para comunicarse. Las prestaciones, fiabilidad, escalabilidad, movilidad y calidad del servicio de las redes subyacentes influyen en el comportamiento de los sistemas distribuidos. Los cambios en las necesidades de los usuarios han producido la irrupción de las redes inalámbricas y las redes de altas prestaciones con calidad de servicio garantizada.

Los fundamentos en los que se basan las redes de computadores incluyen las capas de protocolos, el intercambio de paquetes, el encaminamiento y el flujo de datos. Las técnicas de interconexión de redes hacen posible que se puedan combinar redes heterogéneas para que funcionen como una sola. Internet es el mejor ejemplo de esto; sus protocolos son casi universalmente utilizados en los sistemas distribuidos. Los modos de direccionamiento y encaminamiento usados en Internet han experimentado el impacto de su enorme crecimiento. Tanto es así que ahora se encuentran en proceso de revisión para que puedan soportar el crecimiento futuro y para poder cumplir con los requerimientos de movilidad, seguridad y calidad de servicio demandados.

El diseño de tecnologías específicas de redes se ilustra en los casos de estudio: Ethernet, Modo de Transferencia Asíncrono (ATM, *Asynchronous Transfer Mode*) y el estándar IEEE 802.11 de red inalámbrica.

3.1. INTRODUCCIÓN

Las redes utilizadas por los sistemas distribuidos están compuestas por *medios de transmisión* variados, que incluyen el cable coaxial, la fibra óptica y los canales inalámbricos; *dispositivos hardware*, como *routers*, *switches*, *bridges*, *hubs*, repetidores e interfaces de red; y *componentes software*, entre los que se encuentran las pilas de protocolos, los gestores de comunicaciones y los controladores de dispositivos. Todos estos elementos influyen en la funcionalidad resultante y en las prestaciones disponibles para los sistemas distribuidos y los programas de aplicación que se ejecutan sobre las redes. Llamamos *subsistema de comunicaciones* a la colección de componentes hardware y software que proporcionan las capacidades de comunicación para un sistema distribuido. A cada una de los computadores y de los otros dispositivos que utilizan la red para comunicarse entre sí los denominaremos *hosts*. El término *nodo* se utilizará para referirse a cualquier computador o dispositivo de intercambio asociado a una red.

Internet es un subsistema de comunicaciones singular que proporciona comunicación entre todos los hosts que están conectados a él. Internet está construido a partir de muchas *subredes* empleando una variedad de tecnologías de red distintas en cada caso. Una subred es un conjunto de nodos interconectados, que emplean la misma tecnología para comunicarse entre ellos. La infraestructura de Internet comprende tanto la arquitectura como los componentes hardware y software que hacen posible la integración con éxito de diversas subredes en un único servicio de comunicaciones.

El diseño de un subsistema de comunicaciones está fuertemente influenciado por las características de los sistemas operativos utilizados en los computadores que componen el sistema distribuido, además de por las redes que los interconectan. En este capítulo estudiaremos el impacto de las tecnologías de red en los subsistemas de comunicaciones; los sistemas operativos serán considerados en el Capítulo 6.

Este capítulo está orientado a proporcionar un repaso introductorio de las redes de computadores desde el punto de vista de los requisitos de comunicaciones que los sistemas distribuidos plantean. Aquellos lectores que no estén familiarizados con las redes de computadores deberían considerarlo como soporte para el resto de los contenidos presentados en este libro, ya que encontrarán que este capítulo ofrece un resumen extenso de todos aquellos aspectos de las redes de computadores que son particularmente relevantes para los sistemas distribuidos.

Las redes de computadores fueron concebidas poco después de la invención del computador. Los fundamentos teóricos del intercambio de paquetes fueron descritos en un informe por Leonard Kleinrock [1961]. En 1962, J. C. R. Licklider y W. Clark, que participaban en el desarrollo del primer sistema de tiempo compartido en el MIT en los primeros años sesenta, publicaron un informe en el que se discutía el potencial de la computación interactiva y de la comunicación sobre redes de área extensa, que presagiaba en algunos aspectos a Internet [DEC 1990]. En 1964, Paul Baran produjo un esquema de un diseño práctico de redes de área extensa fiables y efectivas [Baran 1964]. Se puede encontrar más material y enlaces sobre la historia de las redes de computadores e Internet en las siguientes fuentes: [www.isoc.org, Comer 1995, Kurose and Ross 2000].

En el resto de esta sección se presentarán los requisitos de comunicación de los sistemas distribuidos. Daremos un repaso de los tipos de redes en la Sección 3.2 y una introducción a los fundamentos de las redes en la Sección 3.3. La Sección 3.4 tratará específicamente de Internet. Este capítulo concluye con la Sección 3.5 con casos de estudio detallados sobre Ethernet, ATM e IEEE 802.11 (WaveLAN).

3.1.1. LAS REDES Y LOS SISTEMAS DISTRIBUIDOS

Las primeras redes de computadores fueron diseñadas para satisfacer unos pocos, y relativamente sencillos, requisitos de aplicación del tipo transferencia de archivos, conexión a sistemas remotos, correo electrónico y servicios de noticias. El consiguiente desarrollo de los sistemas distribuidos, sobre el que se asientan los programas de aplicación distribuidos que permiten compartir archivos y otros recursos, ha puesto más alto el estándar de prestaciones que satisfagan las necesidades de las aplicaciones interactivas.

Más recientemente, con el crecimiento y comercialización de Internet y la aparición de muchos modos nuevos de uso, se han impuesto requisitos más exigentes en cuanto a fiabilidad, escalabilidad, movilidad, seguridad y calidad de servicio. En esta sección, definiremos y describiremos la naturaleza de cada uno de estos requisitos.

◇ **Prestaciones.** Los parámetros indicadores de las prestaciones de las redes más interesantes para nuestros propósitos son aquellos que afectan a la velocidad con la que los mensajes individuales pueden ser transferidos entre dos computadores interconectados. Éstos son la latencia y la tasa de transferencia punto a punto.

La *latencia* es el intervalo de tiempo que ocurre entre la ejecución de la operación de envío y el instante en que los datos comienzan a estar disponibles en el destino. Puede ser considerada como el tiempo necesario para enviar un mensaje vacío.

La *tasa de transferencia de datos* es la velocidad a la cual se pueden transferir datos entre dos computadores en red, una vez que la transmisión ha sido iniciada; normalmente es medida en bits por segundo.

A partir de estas definiciones, el tiempo requerido por una red para transferir un mensaje de l bits de longitud entre dos computadores es:

Tiempo de transmisión del mensaje = latencia + longitud/tasa de transferencia

La ecuación anterior es válida para mensajes cuya longitud no exceda un máximo que viene determinado por la tecnología de red subyacente. Los mensajes más largos tendrán que ser segmentados y el tiempo de transmisión será la suma del tiempo de transmisión de cada segmento.

La tasa de transferencia de una red viene determinada principalmente por sus características físicas, mientras que la latencia estará determinada básicamente por las sobrecargas del software, los retrasos en el encaminamiento y una componente estadística derivada de los conflictos en el uso de los canales de transmisión. Muchos de los mensajes transferidos entre los procesos que forman parte de los sistemas distribuidos son de pequeño tamaño; por lo que la latencia es a menudo tan importante como la tasa de transferencia a la hora de determinar las prestaciones.

El *ancho de banda total del sistema* de una red es una medida de la productividad (*throughput*), del volumen de tráfico que puede ser transferido a través de la red en un intervalo de tiempo dado. En muchas de las tecnologías de red local, como Ethernet, se utiliza toda la capacidad de transmisión de la red en cada transmisión y el ancho de banda es igual a la tasa de transferencia. En cambio, en la mayoría de las redes de área extensa los mensajes pueden ser transmitidos simultáneamente sobre varios canales diferentes, de modo que el ancho de banda no guarda una relación directa con la tasa de transferencia. Las prestaciones de las redes se deterioran con situaciones de sobrecarga; cuando existen demasiados mensajes en la red al mismo tiempo. El efecto preciso de la sobrecarga en la latencia, la tasa de transferencia y el ancho de banda de una red depende esencialmente de la tecnología de la red.

Considérese ahora las prestaciones de una comunicación cliente-servidor. El tiempo necesario para transmitir un pequeño mensaje de solicitud y recibir una respuesta pequeña en una red local poco cargada entre sistemas PC o UNIX estándar es generalmente de milisegundos. En comparación, el intervalo de tiempo necesario para invocar una operación en un objeto del nivel de aplicación cargado en memoria local de un proceso en ejecución es inferior a un microsegundo. Esto es, a pesar de los avances en las prestaciones de las redes, el tiempo necesario para acceder a recursos compartidos en la misma red local es más de 1.000 veces mayor que el necesario para acceder a

recursos residentes en memoria local. Por otro lado, acceder vía una red de alta velocidad a un servidor web o un servidor de archivos con una gran caché de archivos utilizados puede igualar o superar las prestaciones del acceso a archivos almacenados en un disco local, ya que la latencia de la red y su ancho de banda a menudo superan las prestaciones de un disco duro.

En Internet, las latencias medias de un viaje de ida y vuelta están en el rango de 300 a 600 milisegundos: unas 500 veces más lenta que las redes de área local. La mayor parte de este tiempo se debe a la latencia derivada de los retardos de encaminamiento en los routers y las contenciones en los circuitos de la red.

La Sección 6.5.1 tratará este tema y comparará las prestaciones de las operaciones locales y remotas con más detalle.

◇ **Escalabilidad.** Las redes de computadores son una parte imprescindible de la infraestructura de la sociedad moderna. En la Figura 1.4 se muestra el crecimiento del número de computadores conectados a Internet en un período de veinte años. El tamaño futuro de Internet será comparable con la población del planeta. Resulta creíble esperar que alcance varios miles de millones de nodos y cientos de millones de hosts activos.

Estos cálculos muestran el enorme cambio en tamaño y carga que Internet tendrá que soportar. Las tecnologías de red sobre las que se asienta no están diseñadas incluso ni para soportar la escala actual de Internet, aunque se han comportado de forma suficientemente satisfactoria. Hay planes de algunos cambios sustanciales para el direccionamiento y los mecanismos de encaminamiento, con el fin de dar soporte a la siguiente fase de crecimiento de Internet; tal y como será explicado en la Sección 3.4.

No se dispone de cifras globales sobre el tráfico de Internet, pero se puede estimar el impacto del tráfico sobre las prestaciones a partir de las latencias. En [www.mids.org] se puede encontrar un conjunto interesante de cálculos actuales e históricos de latencias observadas en Internet. Aunque sea frecuente oír referencias al *the world wide wait* (la espera mundial), estos cálculos muestran que se han reducido las latencias en los últimos tiempos hasta alcanzar un tiempo medio de ida y vuelta de 100-150 milisegundos. En ocasiones existen grandes variaciones, con picos de latencias en torno a los 400 milisegundos, pero probablemente éste no es el principal factor causante de los retrasos sufridos por los usuarios del Web. Para aplicaciones cliente-servidor sencillas, como es el Web, deberíamos esperar que el tráfico futuro crezca en proporción al número de usuarios activos. La capacidad de la infraestructura de Internet para vérselas con este crecimiento dependerá de la economía de utilización, en particular las cargas sobre los usuarios y los patrones de comunicación que se dan actualmente, como por ejemplo su grado de localidad.

◇ **Fiabilidad.** Nuestra discusión sobre los modelos de fallos de la Sección 2.3.2 describe el impacto de los errores de comunicación. Muchas aplicaciones son capaces de recuperarse de fallos de comunicación, y por lo tanto no requieren una comunicación libre de errores garantizada. El argumento extremo-a-extremo (véase la Sección 2.2.1) trabaja con la visión de un subsistema de comunicaciones que no proporciona una comunicación totalmente libre de errores; la detección de errores de comunicación y su corrección es a menudo desempeñada mejor por software del nivel de aplicación. La fiabilidad de la mayoría de los medios de transmisión es muy alta. Cuando ocurren errores son normalmente debidos a fallos de sincronización en el software en el emisor o en el receptor (por ejemplo, fallos en el computador receptor al aceptar un paquete) o desbordamientos en el búfer más que fallos en la red.

◇ **Seguridad.** El Capítulo 7 tratará sobre los requerimientos y las técnicas para conseguir seguridad en los sistemas distribuidos. El primer nivel de defensa adoptado por la mayoría de las organizaciones es proteger sus redes y los computadores a ellos conectados mediante un cortafuegos (*firewall*). Un cortafuegos crea un límite de protección entre la red interna de la organización o *intranet*, y el resto de Internet. El propósito del cortafuegos es proteger los recursos en todas los computadores dentro de la organización del acceso por parte de usuarios o procesos externos, y

controlar el uso de recursos del otro lado del cortafuegos por parte de los usuarios dentro de la organización.

Un cortafuegos se ejecuta sobre un *gateway* o pasarela, un computador que se coloca en el punto de entrada de la red interna de una organización. El cortafuegos recibe y filtra todos los mensajes que viajan desde o hacia la organización. Está configurado de acuerdo con la política de seguridad de la organización para permitir que ciertos mensajes entrantes y salientes pasen a través de él, y para rechazar los demás. Volveremos sobre este tema en la Sección 3.4.8.

Para permitir que las aplicaciones distribuidas se puedan mover más allá de las restricciones impuestas por los cortafuegos existe la necesidad de producir un entorno seguro de red en el cual pueda diseminarse un gran número de aplicaciones distribuidas, con autenticación extremo a extremo, privacidad y seguridad. Esta forma de seguridad más detallada y más flexible puede ser conseguida mediante técnicas de criptografía. Éstas se aplican normalmente en un nivel por encima del subsistema de comunicaciones, y se tratarán con más detalle en el Capítulo 7. Los casos excepcionales como la necesidad de proteger componentes de la red como routers contra accesos no autorizados y la necesidad de disponer de enlaces seguros para dispositivos móviles y otros nodos externos que les posibiliten participar en la red interna segura (concepto de red privada virtual o *virtual private network*, VPN) son tratados en la Sección 3.4.8.

◇ **Movilidad.** En el Capítulo 2, presentamos los requisitos que se derivaban de la necesidad de soportar computadores portátiles y dispositivos digitales de mano en los sistemas distribuidos, y mencionamos la necesidad de disponer de redes inalámbricas que permitieran la comunicación continua con esos dispositivos. Pero las consecuencias de la movilidad van más allá de la necesidad de una red inalámbrica. Los dispositivos móviles se desplazan frecuentemente entre distintos lugares y se conectan en puntos de conexión variados. Los modos de direccionamiento y encaminamiento de Internet y de otras redes fueron desarrollados antes de la llegada de los dispositivos móviles, y aunque los mecanismos actuales han sido adaptados y extendidos para soportar cierta movilidad, el esperado crecimiento futuro del uso de los dispositivos móviles harán necesarias nuevas extensiones.

◇ **Calidad de servicio.** En el Capítulo 2, definimos calidad de servicio como la capacidad de cumplir con las restricciones temporales cuando se transmiten y se procesan flujos de datos multimedia en tiempo real. Esto impone unas condiciones más importantes a las redes de computadores. Las aplicaciones que transmiten datos multimedia requieren tener garantizado un ancho de banda y unos límites de latencia en los canales que utilizan. Algunas aplicaciones varían sus demandas dinámicamente y especifican tanto la calidad de servicio aceptable mínima como la óptima deseada. El modo en que se satisfacen y se mantienen esas garantías es el tema tratado en el Capítulo 15.

◇ **Multidifusión (*multicasting*).** La mayoría de las comunicaciones en los sistemas distribuidos se hacen entre pares de procesos, pero a menudo existe también la necesidad de establecer comunicaciones uno a muchos. Aunque esto puede ser simulado enviando mensajes a varios destinos, resulta más costoso de lo necesario, y no posee la característica de tolerancia a fallos requerida por las aplicaciones. Por estas razones, muchas tecnologías de red soportan la transmisión simultánea de mensajes a varios receptores.

3.2. TIPOS DE REDES

Aquí presentaremos los principales tipos de redes utilizados para soportar los sistemas distribuidos: *redes de área local*, *redes de área amplia*, *redes de área metropolitana*, *redes inalámbricas* e *interredes* (comunicación entre redes).

Algunos de los nombres utilizados para nombrar tipos de redes son confusos ya que parece referirse exclusivamente a la extensión física (local, amplia, metropolitana) cuando también hacen referencia a las tecnologías de transmisión física y a los protocolos de bajo nivel. Éstos son diferentes para las redes locales y amplias, aunque alguna tecnología de red desarrollada recientemente como ATM (*Asinchronous Transfer Mode*) resulta adecuada para ambos casos. Las redes inalámbricas también soportan transmisiones de área local y amplia.

Cuando hablamos de comunicación entre redes, por interredes nos referiremos a aquellas redes compuestas de varias redes interconectadas e integradas para proporcionar un único medio de comunicación de datos. Internet es el ejemplo de interred más típico; actualmente está compuesta por cientos de miles de redes de área local, amplia y metropolitana. Describiremos su implementación con algún detalle en la Sección 3.4.

◇ **Redes de área local.** Las redes de área local (*local area networks*) llevan mensajes a velocidades relativamente altas entre computadores conectados a un único medio de comunicaciones: un cable de par trenzado, un cable coaxial o uno de fibra óptica. Un *segmento* es una sección de cable que da servicio a un departamento o a una planta de un edificio y que puede tener varios computadores conectados. Dentro de un segmento no se necesita encaminar mensajes, ya que el medio físico proporciona una conexión directa entre todos los computadores conectados a él. El ancho de banda del segmento se reparte entre los computadores conectados al mismo. Las redes de área local mayores, tales como las que abarcan un campus o un edificio de oficinas, están compuestas de varios segmentos interconectados por conmutadores (*switches*) o concentradores (*hubs*) (véase la Sección 3.3.7). En las redes de área local, el ancho de banda total del sistema es grande y la latencia pequeña, excepto cuando el tráfico de mensajes es muy alto.

En los años 70 se desarrollaron varias tecnologías de redes de área local: Ethernet, *token rings* y *slotted rings*. Cada una de ellas proporciona una solución efectiva y de altas prestaciones, pero Ethernet se ha impuesto como la tecnología dominante para las redes de área amplia. Fue producida originalmente en los primeros años 70 con un ancho de banda de 10 Mbps (millones de bits por segundo) y ha sido extendida a 100 Mbps y a 1 Gbps (gigabits por segundo) en versiones más recientes. Describiremos los principios de funcionamiento de Ethernet en la Sección 3.5.1.

Existe una gran cantidad de redes de área local instaladas, sirviendo en casi todos los entornos de trabajo con dos o más computadores personales o estaciones de trabajo. Sus prestaciones son, en general, adecuadas para la implementación de sistemas y aplicaciones distribuidas. La tecnología Ethernet carece de las garantías necesarias sobre latencia y ancho de banda necesarias para las aplicaciones multimedia. ATM fue desarrollada para ocupar esa demanda, pero su costo ha impedido su implantación en las redes de área local. En su lugar, se han implantado redes Ethernet de alta velocidad que resuelven estas limitaciones de un modo satisfactorio, aunque no sean tan efectivas como ATM.

◇ **Redes de área extensa.** Las redes de área amplia (*wide area networks*) pueden llevar mensajes entre nodos que están a menudo en diferentes organizaciones y quizás separados por grandes distancias, pero a una velocidad menor que las redes LAN. Pueden estar situadas en diferentes ciudades, países o continentes. El medio de comunicación está compuesto por un conjunto de circuitos de enlazados mediante computadores dedicados, llamados *routers* o encaminadores. Éstos gestionan la red de comunicaciones y encaminan mensajes o paquetes hacia su destino. En la mayoría de las redes, las operaciones de encaminamiento introducen un retardo en cada punto de la ruta, por lo que la latencia total de la transmisión de un mensaje depende de la ruta seguida y de la carga de tráfico en los distintos segmentos que atraviese. En las redes actuales, estas latencias pueden ser tan importantes como 0,1 ó 0,5 segundos. Se pueden encontrar estimaciones actuales sobre Internet en [www.mids.org]. La velocidad de las señales electrónicas en la mayoría de los medios es cercana a la velocidad de la luz, y esto impone un límite inferior a la latencia de las transmisiones para las transmisiones de larga distancia. Por ejemplo, el tiempo necesario para que una señal electrónica viaje desde Europa a Australia es aproximadamente de 0,13 segundos y cualquier transmisión que sea encaminada a través de un satélite geoestacionario está sujeta a un retardo de propagación de aproximadamente 0,20 segundos.

Los anchos de banda disponibles en las conexiones a través de Internet varían mucho. En algunos países, se pueden alcanzar velocidades de hasta 1 ó 2 Mbps, pero también son usuales velocidades de 10 a 100 Kbps.

◇ **Redes de área metropolitana.** Las redes de área metropolitana (*metropolitan area networks*) se basan en el gran ancho de banda de los cableados de cobre y de fibra óptica recientemente instalados en pueblos y ciudades para la transmisión de vídeo, voz y otros tipos de datos; cuyas longitudes abarcan hasta unos 50 kilómetros de distancia. Este cableado puede explotarse para proporcionar tasas de transferencia compatibles con las necesidades de los sistemas distribuidos. Varias han sido las tecnologías utilizadas para implementar el encaminamiento en las redes MAN, desde Ethernet hasta ATM. IEEE ha publicado la especificación 802.6 [IEEE 1994], diseñada expresamente para satisfacer las necesidades de las redes MAN, cuyas implementaciones están en marcha. Se podría decir que las redes MAN están en su infancia pero están cerca de alcanzar las características de las redes LAN, aunque pudiendo cubrir mayores distancias que éstas.

Las conexiones de línea de subscripción digital, DSL (*digital subscriber line*), y los módem de cable disponibles en algunas ciudades son un ejemplo de esto. DSL utiliza generalmente conmutadores ATM (véase la Sección 3.5.3) colocados en intercambiadores telefónicos para encaminar datos digitales sobre par trenzado (el mismo cable de cobre que se utiliza para las conexiones telefónicas) hacia la casa o la oficina de los subscriptores a velocidades entre 0,25 y 6,0 Mbps. El uso de par trenzado para las conexiones del subscriptor DSL limita la distancia al conmutador a 1,5 kilómetros. Una conexión de módem por cable utiliza una señalización análoga sobre el cable coaxial de televisión para conseguir velocidades de 1,5 Mbps con un alcance superior que DSL.

◇ **Redes inalámbricas.** La conexión de los dispositivos portátiles y de mano necesitan redes de comunicaciones inalámbricas (*wireless networks*), según se presentó en el Capítulo 2. Recientemente han surgido muchas tecnologías de comunicaciones inalámbricas digitales. Algunas, como la IEEE 802.11 (*WaveLAN*) ofrecen transmisiones de datos entre 2 y 11 Mbps sobre 150 metros, son verdaderas redes LAN inalámbricas, (*wireless local area networks*, WLAN) diseñadas para ser utilizadas en lugar de las redes LAN. Otras están diseñadas para conectar dispositivos móviles a otros dispositivos móviles o fijos muy próximos, por ejemplo para comunicarse con impresoras o con otros computadores de mano o de escritorio. Estas redes se denominan *redes de área personal inalámbricas* (*wireless personal area networks*, WPAN), y ejemplos de las mismas son los enlaces infrarrojos que se incluyen en muchos computadores de mano y portátiles y la tecnología de radio de baja potencia BlueTooth [www.bluetooth.com], que ofrece transmisiones de datos entre 1 y 2 Mbps hasta 10 metros. Muchas redes de telefonía móvil están basadas en tecnologías de redes inalámbricas, incluida la red europea mediante el Sistema Global para Comunicaciones Móviles, GSM (*Global System for Mobile Communication*), utilizada en la mayoría de los países del mundo. En los Estados Unidos, la mayoría de los teléfonos móviles están actualmente basados en la análoga red de radio celular AMPS, sobre la cual se encuentra la red digital de comunicaciones de Paquetes de Datos Digitales Celular, CDPD (*Cellular Digital Packet Data*). Las redes de teléfonos móviles están diseñadas para trabajar sobre áreas amplias (generalmente países o continentes enteros) basándose en conexiones de radio celulares; y por lo tanto, ofrecen posibilidades de conexión a Internet a los dispositivos móviles en grandes áreas. Las redes celulares mencionadas anteriormente ofrecen tasas de datos relativamente bajas, de 9,6 a 19,2 kbps, aunque las redes que les sucederán están siendo planeadas con tasas de transmisión entre 128 y 384 kbps para células de unos cuantos kilómetros, y de 2 Mbps para células más pequeñas.

Dado el restringido ancho de banda disponible y las otras limitaciones de los dispositivos portátiles, tales como pantallas pequeñas, se ha desarrollado un conjunto de protocolos llamado

Protocolo de Aplicación Inalámbrica, WAP (*Wireless Application Protocol*), especialmente para estos dispositivos inalámbricos [www.wapforum.org].

◇ **Interredes.** Una interred es un subsistema de comunicación compuesto por varias redes que se han enlazado juntas para proporcionar unas posibilidades de comunicación ocultando las tecnologías y los protocolos y métodos de interconexión de las redes individuales que las componen.

Las interredes son necesarias para el desarrollo de los sistemas distribuidos abiertos extensibles. Las características de apertura de los sistemas distribuidos implican que las redes utilizadas deberían ser extensibles a un gran número de computadores, mientras que por el contrario las redes individuales tienen restricciones en los espacios de direccionamiento y limitaciones en las prestaciones que las hacen incompatibles para su uso a gran escala. En las interredes, se pueden integrar una gran variedad de tecnologías de red de área local y amplia, posiblemente de distintos fabricantes, para proporcionar la capacidad de trabajo en red necesaria para cada grupo de usuarios. De este modo, las interredes aportan gran parte de los beneficios de los sistemas abiertos a las comunicaciones de los sistemas distribuidos.

Las interredes se construyen a partir de varias redes. Éstas están interconectadas por computadores dedicados llamadas *routers* y por computadores de propósito general llamadas *gateways*, y por un subsistema integrado de comunicaciones producido por una capa de software que soporta el direccionamiento y la transmisión de datos a los computadores a través de la interred. El resultado puede contemplarse como una *red virtual* construida a partir de solapar una capa de interred sobre un medio de comunicación que consiste en varias redes, routers y gateways subyacentes. Internet es el mayor y mejor ejemplo de interred, y los protocolos TCP/IP son un ejemplo de la capa de integración mencionada antes.

◇ **Comparación de redes.** La Figura 3.1 muestra los rangos y las prestaciones de varios tipos de redes comentadas anteriormente. Un punto de comparación adicional relevante para los sistemas distribuidos es la frecuencia y los tipos de fallos que se pueden esperar en cada tipo de red. La fiabilidad de los mecanismos de transmisión es muy alta en todos los tipos, excepto en las redes inalámbricas, donde los paquetes se pierden con frecuencia debido a las interferencias externas. En todos los tipos de redes, las pérdidas de paquetes vienen causadas por los retardos de procesamiento o por los desbordamientos en los destinos. Ésta es, de lejos, la causa más común de pérdida de paquetes.

Los paquetes pueden entregarse en diferente orden al que fueron transmitidos. Esto es especialmente importante en las redes en que los paquetes son encaminados individualmente, principalmente en las redes de área amplia. También se pueden entregar copias duplicadas de paquetes, usualmente causadas por la suposición por parte del emisor de que el paquete fue extraviado. Tanto la retransmisión del paquete como el original llegan a su destino.

Todos los fallos descritos son ocultados por TCP y por otros protocolos llamados protocolos fiables, que hacen posible que las aplicaciones supongan que todo lo que es transmitido será recibido por el destinatario. Existen, sin embargo, buenas razones para utilizar protocolos menos fiables como UDP en algunos casos de sistemas distribuidos, y en aquellas circunstancias en las que los programas de aplicación puedan tolerar los fallos.

	Rango	*Ancho de Banda (Mbps)*	*Latencia (ms)*
LAN	1-2 km	10-1.000	1-10
WAN	mundial	0,010-600	100-500
MAN	2-50 km	1-150	10
LAN inalámbrica	0,15-1,5 km	2-11	5-20
WAN inalámbrica	mundial	0,010-2	100-500
Internet	mundial	0,010-2	100-500

Figura 3.1. Tipos de redes.

3.3. FUNDAMENTOS DE REDES

Las base de las redes de computadores es la técnica de la conmutación de paquetes, desarrollada en los primeros años sesenta. La conmutación de paquetes fue un cambio radical y un paso más allá de las redes de telecomunicación conmutadas que utilizaban los teléfonos y los telégrafos, explotando la capacidad de los computadores de almacenar información mientras está en tránsito. Esto posibilita que paquetes con diferentes destinos compartan un mismo enlace de comunicaciones. Los paquetes se ponen en cola en un búfer y se transmiten cuando el enlace está disponible. La comunicación es asíncrona, ya que los mensajes llegan a sus destinos después de un retardo variable que depende del tiempo que tardaron los paquetes en viajar a través de la red.

3.3.1. TRANSMISIÓN DE PAQUETES

En la mayoría de las aplicaciones de las redes de computadores se necesita transmitir unidades de información o *mensajes*: secuencias de ítems de datos de longitud arbitraria. Antes de que un mensaje sea transmitido es dividido en *paquetes*. La forma más simple de paquete es una secuencia de datos binarios (una secuencia de bits o bytes) de una longitud determinada, junto a la suficiente información para identificar los computadores origen y destino. Los paquetes tienen una longitud limitada:

- Porque así cada computador en la red puede reservar el espacio de almacenamiento suficiente para almacenar el paquete más largo que pueda recibir.
- Para evitar los retardos que podrían ocurrir si se estuviera esperando a que los canales estuvieran libres el tiempo suficiente para enviar un mensaje grande sin dividir.

3.3.2. FLUJOS DE DATOS

Existen excepciones a la comunicación basadas en mensajes. Ya vimos en el Capítulo 2 que las aplicaciones multimedia se basan en la transmisión de flujos, o *caudales*, de datos de audio y vídeo a tasas garantizadas y con latencias limitadas. Estos caudales difieren substancialmente del tipo de tráfico basado en mensajes para los que fue diseñada la transmisión de paquetes. Los caudales de audio y vídeo necesitan anchos de banda mucho mayores que otras formas de comunicación en los sistemas distribuidos.

La transmisión de un caudal de vídeo en tiempo real necesita un ancho de banda de 1,5 Mbps si los datos están comprimidos, o de 120 Mbps si no lo están. Además, el flujo es continuo, al contrario que el tráfico intermitente generado por la típica interacción cliente-servidor. El *tiempo de ejecución* en un elemento multimedia es el tiempo en el cual debe mostrarse (para un elemento de vídeo) o convertirse a audio (para un ejemplo de sonido). Por ejemplo, en un caudal de vídeo con una tasa de 24 marcos por secundo, el marco N tiene que ser ejecutado $N/24$ segundos después del comienzo. Los elementos que lleguen a su destino después de su tiempo de ejecución no son útiles y deben desecharse por el proceso receptor.

La entrega a tiempo de estos flujos de datos depende de la disponibilidad de conexiones con una calidad de servicio garantizada; se deben garantizar el ancho de banda, la latencia y la fiabilidad. Se debe poder establecer un canal desde el origen hasta el destino del caudal multimedia, con una ruta predefinida a través de la red, en el que se reserven en cada nodo perteneciente al mismo un conjunto de recursos apropiados para amortiguar cualquier irregularidad en el flujo de datos a través del canal. Los datos pueden, entonces, pasar a través del canal desde el emisor al receptor a una tasa especificada.

Las redes ATM (véase la Sección 3.5.3) están diseñadas específicamente para proporcionar altos anchos de banda y bajas latencias y para soportar calidad de servicio reservando recursos de la red. El protocolo IPv6 es un nuevo protocolo de Internet que será utilizado en la década próxima (véase la Sección 3.4.4), que incluye la posibilidad de que cada paquete IP que forme parte de un caudal de tiempo real sea identificado como tal y sea tratado separadamente de cualquier otro tipo de datos en el nivel de red.

Los subsistemas de comunicación que proporcionan calidad de servicio garantizada necesitan poder reservar *a priori* recursos de red y forzar esas reservas. El Protocolo de Reserva de Recursos RSVP (*Resource Reservation Protocol*) [Zhang y otros 1993] posibilita a las aplicaciones la negociación de una reserva de ancho de banda para los caudales de tiempo real. El Protocolo de Transporte de Tiempo Real RTP (*Real Time Transport Protocol*) [Schulzrinne y otros 1996] es un protocolo de transferencia de la capa de aplicación que incluye en cada paquete consideraciones sobre el tiempo de ejecución y otras necesidades de sincronización. La disponibilidad de implementaciones efectivas de estos protocolos en Internet dependerá de cambios substanciales en las capas de transporte y de red. El Capítulo 15 tratará en detalle las necesidades de las aplicaciones multimedia.

3.3.3. ESQUEMAS DE CONMUTACIÓN

Una red se compone de un conjunto de nodos conectados a través de circuitos. Para transmitir información entre dos nodos cualquiera se necesita un sistema de conmutación. A continuación vamos a definir los cuatro tipos de conmutación que se utilizan en las redes de computadores.

◇ **Difusión.** La difusión (*broadcast*) es una técnica de transmisión que no involucra conmutación alguna. Toda información es transmitida a todos los nodos, y depende de los receptores decidir si el mensaje va dirigidos a ellos o no. Algunas tecnologías de red LAN como Ethernet se basan en la difusión. Las redes inalámbricas están basadas necesariamente en la difusión, ya que en ausencia de circuitos fijos la difusión llega a todos los nodos agrupados en la misma *celda*.

◇ **Conmutación de circuitos.** Hace algún tiempo, las redes telefónicas eran las únicas redes de telecomunicaciones. Su modo de operar era simple de entender: cuando el emisor marcaba un número, el par de hilos de cobre que iba desde su teléfono hasta la centralita era conectado automáticamente al par de hilos que llevaba al teléfono del receptor. En las llamadas a larga distancia el proceso era similar, sólo que las conexiones eran realizadas a través de muchos conmutadores. A este sistema se le denomina como el sistema telefónico plano antiguo (*plain old telephone system*, POTS). Es una típica red de *conmutación de circuitos*.

◇ **Conmutación de paquetes.** El advenimiento de los dispositivos de conmutación y de la tecnología digital trajo nuevas posibilidades a las telecomunicaciones, básicamente capacidad de procesamiento y de almacenamiento. Esto hizo posible construir redes de comunicaciones de un modo muy diferente. Este nuevo tipo de redes de comunicaciones se denomina de *almacenamiento y reenvío* (*store-and-forward network*). En lugar de iniciar y destruir conexiones para construir circuitos, una red de almacenamiento y reenvío solamente envía paquetes desde el origen hacia el destino. En cada nodo de conmutación se encuentra un computador (allá donde varios circuitos se conectan). Los paquetes que llegan a un nodo se almacenan en la memoria del computador de ese nodo y luego son procesados por un programa que les envía hacia su destino eligiendo uno de los circuitos salientes que llevará al paquete a otro nodo que estará más cerca del destino que el nodo anterior.

Realmente no existe nada nuevo en este modo de trabajar: el sistema postal es una red de almacenamiento y reenvío para las cartas, donde el procesamiento es realizado por humanos o por máquinas en las oficinas de clasificación. Pero los paquetes de datos pueden ser almacenados y procesados por los computadores lo suficientemente rápido como para crear la ilusión de una transmisión instantánea. Al menos, eso era lo que parecía hasta que la gente (incluidos los ingenieros en telefonía) comenzaron a pensar que podría ser una buena idea utilizar las redes de computadores para transmitir audio y vídeo. Las ventajas de esta integración son potencialmente inmensas; una sola red podría ser suficiente para gestionar las comunicaciones de los computadores, los servicios telefónicos, las emisiones de televisión y radio y también aplicaciones nuevas como la teleconferencia. Dado que todo podría ser representado de forma digital, podría ser susceptible de almacenamiento y reenvío. Sin duda, emergerían muchas aplicaciones nuevas de tal sistema de telecomunicación integrado.

◇ **Frame relay.** La transmisión basada en el almacenamiento y el reenvío de paquetes no es instantánea; generalmente se toma desde unas pocas decenas de microsegundos hasta unos pocos milisegundos para encaminar los paquetes en cada nodo de la red, dependiendo del tamaño del paquete, las velocidades del hardware y la cantidad de tráfico. Los paquetes pueden ser encaminados hacia bastantes nodos antes de que alcancen su destino. Internet está basada en gran medida en redes de conmutación de paquetes, y ya se ha visto que incluso paquetes pequeños tardan alrededor de 200 milisegundos en alcanzar sus destinos.

Retardos de esta magnitud son demasiado grandes para aplicaciones como la telefonía, donde para sostener una conversación telefónica sin interferencias sólo se toleran retardos menores de 50 milisegundos. Como los retardos son acumulativos, cuantos más nodos atraviesen el paquete, mayor será el retardo, y como mucha parte del retardo en cada nodo depende de factores inherentes a la técnica de conmutación de paquetes, esto parece un serio obstáculo para la integración.

Pero los ingenieros de telefonía y de redes de computadores no se desaniman fácilmente. Vinieron con otro método de conmutación, el *frame relay* o retransmisión de marcos, que aporta algunas de las ventajas de la conmutación de circuitos a la conmutación de paquetes. El resultado son las redes ATM. Describiremos su funcionamiento en la Sección 3.5.3. Solucionaron el problema del retardo al conmutar los paquetes pequeños (llamados *marcos*, *frames*) según venían, al vuelo. Los nodos de conmutación (que usualmente son procesadores paralelos de propósito específico) encaminan los marcos basándose en el examen de los primeros bits; los marcos no se almacenan como un todo en el nodo, sino que pasan a través de él como pequeños flujos de bits. Como resultado, la tecnología ATM puede transmitir paquetes a través de una red de muchos nodos en unas pocas decenas de microsegundos.

3.3.4. PROTOCOLOS

El término *protocolo* se utiliza para referirse a un conjunto bien conocido de reglas y formatos que se han de utilizar para la comunicación entre procesos que realizan una tarea determinada. La definición de un protocolo tiene dos partes importantes:

- Una especificación de la secuencia de mensajes que se han de intercambiar.
- Una especificación del formato de los datos en los mensajes.

La existencia de protocolos bien conocidos posibilita que los componentes software separados, que formarán parte de sistemas distribuidos, puedan desarrollarse independientemente e implementarse en diferentes lenguajes de programación sobre computadores que quizás tengan diferente representación interna de los datos.

Un protocolo está implementado por dos módulos software ubicados en los computadores del emisor y del receptor. Por ejemplo, un *protocolo de transporte* transmite mensajes de cualquier longitud desde un proceso emisor hacia un proceso receptor. Un proceso que desee transmitir un mensaje a otro proceso efectuará una llamada al módulo del protocolo de transporte, pasándole el mensaje en un cierto formato. Entonces, el software de transporte se pondrá a la tarea de transmitir el mensaje a su destino, subdividiéndolo en paquetes de un tamaño y formato determinados, así éstos podrán ser transmitidos al destino vía el *protocolo de red*, otro protocolo de nivel inferior. El

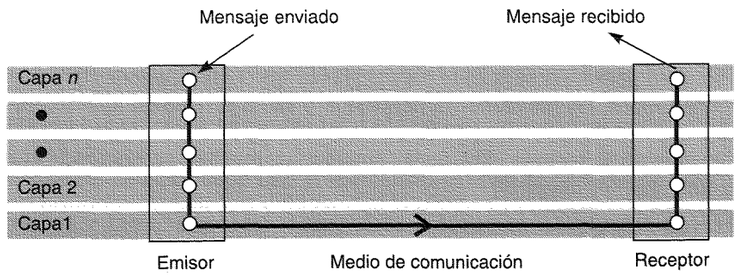


Figura 3.2. Capas conceptuales de un protocolo software.

correspondiente módulo del protocolo de transporte en el computador destino recibirá el paquete de su módulo del protocolo de nivel de red y realizará las transformaciones inversas para regenerar el mensaje antes de entregárselo al proceso receptor.

◇ **Protocolos a capas.** El software de red está organizado en una jerarquía de capas. Cada capa presenta una interfaz a las capas que se colocan sobre ella que extiende las propiedades del sistema de comunicaciones subyacente. Cada capa se representa por un módulo en cada uno de los computadores conectados a la red. La Figura 3.2 ilustra la estructura y el flujo de datos cuando se transmite un mensaje utilizando la pila de protocolos. Cada módulo aparenta comunicarse directamente con el módulo del mismo nivel en otro computador en la red, aunque en realidad los datos no son transmitidos directamente entre los módulos de los protocolos del mismo nivel. En lugar de esto, cada capa de software de red se comunica con los protocolos que están por encima y por debajo de él mediante llamadas a procedimientos.

En el lado del emisor, cada capa (excepto la superior o *capa de aplicación*) acepta ítems de datos en una formato específico de la capa superior, y después de procesarlos, los transforma para encapsularlos según el formato especificado por la capa inferior a la que se los pasa para su procesamiento. La Figura 3.3 ilustra este proceso tal y como se aplica en las cuatro capas superiores del conjunto de protocolos OSI. La figura muestra las cabeceras de los paquetes que tienen la mayoría de los datos relacionados con la red, aunque por claridad se han omitido las colas que se encuentran en algunos tipos de paquetes; también se supone que el mensaje de la capa de aplicación a transmitir es más pequeño que el tamaño máximo admisible por la capa de red subyacente. En caso contrario, debería encapsularse en varios paquetes de la capa de red. En el lado del receptor, las transformaciones de conversión se aplican en las capas inferiores antes de que el mensaje se propa-

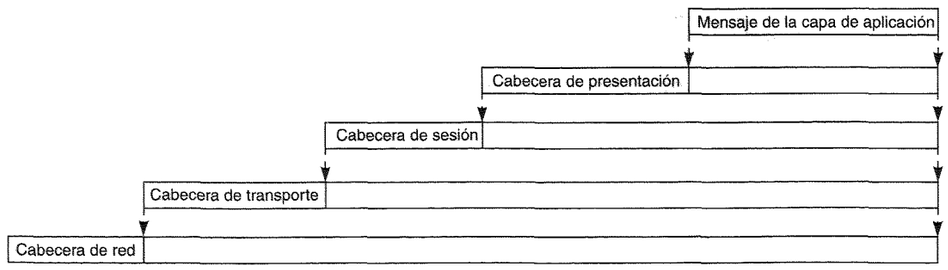


Figura 3.3. La encapsulación según se aplica en los protocolos multicapa.

gue hacia arriba. El tipo de protocolo de la capa superior se incluye en la cabecera de cada capa para posibilitar en cada nivel de la pila del receptor la selección correcta de los componentes software que deben tratar los paquetes.

De este modo cada capa proporciona un servicio a la capa superior y extiende el servicio proporcionado por la capa inferior. En el nivel base se encuentre la *capa física*. Está implementada por un medio de comunicación (cables de cobre o fibra óptica, comunicaciones por satélite o transmisiones de radio) y por circuitos correspondientes que colocan señales en el medio de comunicación en el nodo emisor y las detectan en el nodo receptor. En el nodo receptor los ítems de datos son recibidos y pasan a través de la jerarquía de módulos de software, transformándose en cada capa hasta que estén en una forma adecuada para poder ser transferidos al siguiente proceso receptor.

◇ **Conjuntos de protocolos.** Al conjunto completo de capas de protocolos se le denomina *conjunto de protocolos* o *pila de protocolos*, plasmando con ello la estructura de capas. La Figura 3.4 muestra una pila de protocolos que conforma el Modelo de Referencia para la *Interconexión de Sistemas Abiertos* (*Open System Interconnection*, OSI) de siete capas adoptado por la Organización Internacional de Estándares (*International Standard Organization*, ISO) [ISO 1992]. El Modelo de Referencia OSI fue adoptado para favorecer el desarrollo de estándares de protocolos que pudieran satisfacer los requisitos de los sistemas abiertos.

El propósito de cada nivel en el Modelo de Referencia OSI se resume en la Figura 3.5. Como su nombre indica, es un marco de trabajo para la definición de protocolos y no una definición de un conjunto determinado de ellos. Los protocolos que sigan el modelo OSI deben incluir al menos un protocolo específico en cada una de las siete capas que el modelo define.

Los protocolos por capas proporcionan beneficios substanciales al simplificar y generalizar las interfaces software para el acceso a los servicios de comunicación de las redes, aunque también implican costos significativos en prestaciones. La transmisión de un mensaje de la capa de aplicación vía la pila de protocolos con N capas normalmente involucra N transferencias de control a las capas relevantes en la pila, una de las cuales es una entrada del sistema operativo, y realiza N copias de los datos como parte del mecanismo de encapsulación. Todas estas sobrecargas producen que la tasa de transferencia ente los procesos de aplicación sea mucho menor que el ancho de banda disponible.

La Figura 3.5 incluye ejemplos de protocolos utilizados en Internet, aunque la implementación de Internet no siga el modelo OSI en dos aspectos. Primero, las capas de aplicación, presentación

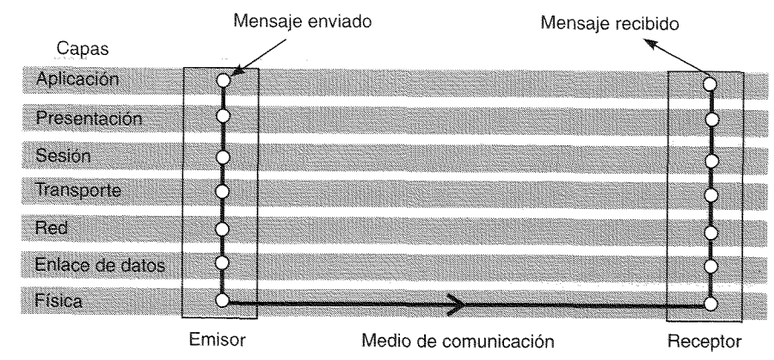


Figura 3.4. Capas de protocolos en el modelo de protocolos de *Interconexión de Sistemas Abiertos* (OSI).

Capa	*Descripción*	*Ejemplos*
Aplicación	Protocolos diseñados para responder a los requisitos de comunicación de aplicaciones específicas, a menudo definiendo la interfaz a un servicio.	HTTP, FTP, SMTP, CORBA IIOP
Presentación	Los protocolos de este nivel transmiten datos en una representación de datos de red independiente de las utilizadas comúnmente en los computadores, que pueden ser distintas. Si se necesitara, la encriptación también se llevaría a cabo en este nivel.	SSL, Representación de datos CORBA
Sesión	En este nivel se implementa la fiabilidad y la adaptación, tales como la detección de fallos y la recuperación automática.	
Transporte	Éste es el nivel más bajo en el que se gestionan mensajes (en lugar de paquetes). Los mensajes son dirigidos a los puertos de comunicaciones asociados a los procesos. Los protocolos de esta capa pueden ser orientados a conexión o no.	TCP, UDP
Red	Transfiere paquetes de datos entre computadores en una red específica. En una WAN o en una interred esto implica la generación de una ruta de paso a través de los routers. En una LAN simple no se necesita encaminamiento.	IP, circuitos virtuales ATM
Enlace de datos	Es responsable de la transmisión de paquetes entre nodos que están conectados directamente por un enlace físico. En una transmisión WAN será entre pares de routers o entre un router y un host. En las LANs es entre cualquier par de hosts.	MAC de Ethernet, transferencia de celdas ATM, PPP
Físico	Los circuitos y el hardware que dirigen la red. Transmite secuencias de datos binarios mediante señales binarias, utilizando modulación en amplitud o en frecuencia de las señales eléctricas (en los circuitos de cables), señales ópticas (en los circuitos de fibra óptica) u otras señales electromagnéticas (en los circuitos de radio o microondas).	Señalización de banda-base Ethernet, ISDN

Figura 3.5. Resumen del protocolo OSI.

y sesión no son claramente distinguibles en la pila de protocolos de Internet. En su lugar, las capas de aplicación y presentación están implementadas o en una única capa de software intermedio (*middleware*), o separadamente dentro de cada aplicación. De esta guisa, CORBA implementa las invocaciones entre objetos y las representaciones de datos en una biblioteca middleware que se incluye en cada proceso de aplicación (véase el Capítulo 17 para más detalles sobre CORBA). Los navegadores web y otras aplicaciones que requieren canales seguros emplean la capa de sockets seguros (*Secure Sockets Layer*, SSL) (véase el Capítulo7) como una biblioteca de procedimientos de una manera similar.

Segundo, la capa de sesión está integrada con la de transporte. Los conjuntos de protocolos interredes incluyen una capa de aplicación, una de transporte y otra de interred. La capa de interred es una capa de red *virtual* que es responsable de la transmisión de los paquetes de la interred al computador destino. El *paquete interred* es la unidad de datos transmitidos sobre la red.

Los protocolos de interred están ocultos en las redes subyacentes, tal y como se muestra en la Figura 3.6. La *capa de interfaz de red* acepta paquetes de interredes y los convierte en paquetes adecuados para su transmisión por los protocolos de red de cada red subyacente.

◇ **Ensamblado de paquetes.** La tarea de dividir los mensajes en paquetes antes de la transmisión y reensamblarlos en el computador destino se realiza normalmente en la capa de transporte.

Los paquetes del protocolo de la capa de red están compuestos por una *cabecera* y por un *campo de datos*. En la mayoría de las tecnologías de red, el campo de datos es de longitud variable, aunque tenga un límite llamado la *unidad máxima de transferencia*, (*maximum transfer unit*, MTU). Si la longitud del mensaje excede la MTU de la capa de red, debe ser fragmentado en tro-

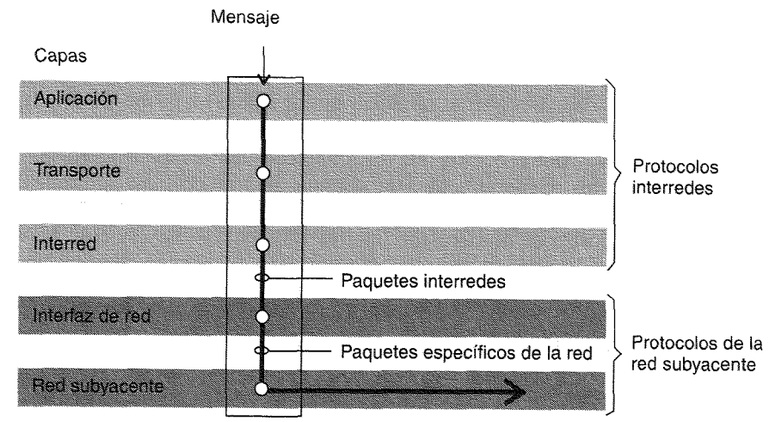


Figura 3.6. Capas de interredes.

zos de tamaño apropiado, y debe ser identificado con una secuencia de números para utilizarla en el reensamblado, y transmitido en múltiples paquetes. Por ejemplo, la MTU de Ethernet es de 1.500 bytes; por lo que no se puede transmitir más que esta cantidad de datos en un solo paquete Ethernet.

La MTU del protocolo IP, que es el utilizado como protocolo de la capa de red en Internet, es inusualmente grande: 64 kbytes (en la práctica se suele utilizar sólo 8 kbytes, ya que algunos nodos no son capaces de gestionar paquetes tan grandes). Independientemente de la MTU adoptada para los paquetes IP, aquellos que sean mayores que la MTU de Ethernet deberán ser fragmentados para su transmisión sobre Ethernet.

◇ **Puertos.** La tarea de la capa de transporte es la de proporcionar un servicio de transporte de mensajes independiente de la red entre pares de *puertos* de red. Los puertos son puntos de destino para la comunicación dentro de un computador definidos por software. Los puertos se asocian a procesos, permitiendo, de este modo, la comunicación de un proceso con otro. Los detalles específicos de la abstracción de los puertos puede variar para proporcionar propiedades adicionales útiles. Nosotros describiremos aquí el direccionamiento de puertos según se da en Internet y en la mayoría de las otras redes. El Capítulo 4 tratará sobre el uso de los puertos.

◇ **Direccionamiento.** La capa de transporte es responsable de la entrega de los mensajes en el destino utilizando para ello una *dirección de transporte*, compuesta por la dirección de red de un computador y por un número de puerto. Una dirección de red es un identificador numérico que identifica de forma única a un computador y posibilita su localización por parte de los nodos responsables del encaminamiento de los datos. En Internet todas los computadores tienen asignado un número IP, que identifica tanto al computador como a la subred a la que está conectada, haciendo posible que los datos sean encaminados a ella desde cualquier otro nodo según se describe en las siguientes secciones. En las redes Ethernet no existen nodos encaminadores; cada host es responsable de reconocer y recuperar los paquetes a él dirigidos.

En Internet existen normalmente varios puertos en cada computador con números bien conocidos, cada uno reservado para dar un servicio de Internet, como HTTP o FTP. Los puertos bien conocidos y las definiciones de los servicios están registrados por una autoridad central [www.iana.org]. Para acceder a un servicio en un determinado host, se envía una petición al puerto correspondiente en el host. Algunos servicios, como FTP, reservan entonces otro puerto nuevo

(con un número privado) y envían el número del nuevo puerto al cliente. El cliente utiliza el nuevo puerto para establecer una conexión TCP, sobre la que se realiza el resto de una transacción o de una sesión. Otros servicios, como HTTP, realizan todas sus transacciones a través de puertos bien conocidos.

Los números de puerto por encima del 1023 están disponibles para el uso general, por ejemplo por nuevos servicios y por procesos nuevos, que necesitan reservar puertos en los que recibir mensajes de los servidores. Generalmente, un sistema distribuido tiene múltiples servidores, que además son distintos dependiendo del tiempo y de las organizaciones. Obviamente, no resulta adecuada una reserva de host o de números de puertos fijos para esos servicios. La solución al problema en el que los clientes localizan servidores arbitrarios en un sistema distribuido se trata en la subsección sobre enlaces del Capítulo 5.

◇ **Entrega de paquetes.** Existen dos aproximaciones a la hora de entregar paquetes por parte de la capa de red:

Entrega de paquetes tipo datagrama: el término *datagrama* se refiere a la semejanza que existe entre este modo de entrega y aquel en el que se entregan los telegramas y las cartas. La característica esencial de los datagramas de red es que la entrega de cada paquete es un proceso de un paso; no requiere ninguna preparación y una vez que el paquete ha sido entregado, la red no guarda ninguna información sobre él. En una red de datagramas, cada miembro de la secuencia de paquetes transmitidos por un host a un destino puede seguir rutas diferentes (si, por ejemplo, la red es capaz de adaptarse a fallos en el funcionamiento o mitigar los efectos de una congestión localizada) y quizás lleguen desordenados.

Cada datagrama contiene la dirección de red completa de los host origen y destino; la última es esencial para el proceso de encaminamiento, según describimos en la siguiente sección. La entrega de datagramas es el concepto en el que se basaron originalmente las redes y puede encontrarse en la mayoría de las redes de computadores en uso hoy en día. La capa de red de Internet (IP), Ethernet y la mayoría de las tecnologías de red locales tanto cableadas como inalámbricas están basadas en la entrega de datagramas.

Entrega de paquetes por circuito virtual: algunos servicios del nivel de red implementan la transmisión de paquetes de un modo análogo a la red telefónica. Se debe conseguir un circuito virtual antes de que los paquetes puedan pasar del host origen A al host destino B. El establecimiento del circuito virtual involucra la identificación de una ruta desde el origen al destino, posiblemente pasando a través de varios nodos intermediarios. En cada nodo a lo largo de la ruta se crea una entrada en la tabla de encaminamiento, indicando qué enlace debe ser utilizado para la siguiente etapa de la ruta.

Una vez que el circuito virtual ha sido configurado, puede ser utilizado para transmitir cualquier número de paquetes. Cada paquete de la capa de red contiene solamente el número del circuito virtual reemplazando de este modo a las direcciones del origen y del destino. Estas direcciones no son necesarias, pues los paquetes se encaminan en los nodos intermedios basándose en el número del circuito virtual. Cuando un paquete alcanza su destino, el origen puede ser determinado a partir del número del circuito virtual.

La analogía con las redes telefónicas no debe ser tomada demasiado al pie de la letra. En las redes POTS una llamada de teléfono tiene como efecto el establecimiento de un circuito físico entre el origen y el destino de la llamada, y los enlaces de voz son reservados para su uso exclusivo. En la entrega de paquetes por medio de circuitos virtuales los circuitos están representados sólo por entradas en las tablas de los nodos de encaminamiento, y los enlaces sobre los que los paquetes son encaminados se utilizan sólo el tiempo necesario para que el paquete sea transmitido; estando disponibles para ser utilizados por otros usuarios el resto del tiempo. Un enlace puede ser empleado, por lo tanto, por muchos circuitos virtuales distintos. La tecnología de circuitos virtuales más importante en uso es ATM; y como ya hemos mencionado (en la Sección 3.3.3), tiene una baja latencia de transmisión de paquetes, lo que es una consecuencia directa de la utilización de los circuitos virtuales. La necesidad de la fase de establecimiento del circuito produce, sin embargo, un pequeño retraso antes de que cualquier paquete pueda llegar a su destino.

La distinción entre la entrega de paquetes por datagramas y por circuitos virtuales no debe confundirse con un par de mecanismos similares que existen en la capa de transporte (transmisión orientada a conexión y no orientada a conexión). Trataremos este apartado en la Sección 3.4.6, en el contexto de los protocolos de transporte de Internet UDP (sin conexión) y TCP (orientado a conexión). Aquí solamente hacemos notar que cada uno de esos modos de transmisión puede ser implementado sobre cualquier tipo de capa de red.

3.3.5. ENCAMINAMIENTO

El encaminamiento es una función necesaria en todas redes excepto en aquellas redes LAN, como Ethernet, que proporcionan conexiones directas entre todos los pares de hosts conectados. En las redes grandes se emplea un *encaminamiento adaptativo*: se reevalúan periódicamente las mejores rutas para comunicar dos puntos de red, teniendo en cuenta el tráfico actual y cualquier fallo como conexiones rotas o routers caídos.

La entrega de los paquetes a sus destinos en una red como la mostrada en la Figura 3.7 es una responsabilidad colectiva de los routers situados en los puntos de conexión. A menos que los hosts origen y destino se encuentren en la misma LAN, el paquete tendrá que ser transmitido en una serie de saltos, pasando a través de los routers. La determinación de las rutas para la transmisión de los paquetes a su destino es responsabilidad del *algoritmo de encaminamiento*, implementado por un programa en la capa de red de cada nodo.

Un algoritmo de encaminamiento tiene dos partes:

1. Tomar decisiones que determinen la ruta seguida por cada paquete que viaja por la red. En las capas de red de conmutación de circuitos como las redes X.25, y del tipo *frame-relay* como las redes ATM, la ruta se determina cuando se establece un circuito virtual o una conexión. En las capas de red de conmutación de paquetes como IP se fija separadamente para cada paquete, y el algoritmo debe ser particularmente simple y eficiente para que no degrade las prestaciones de la red.

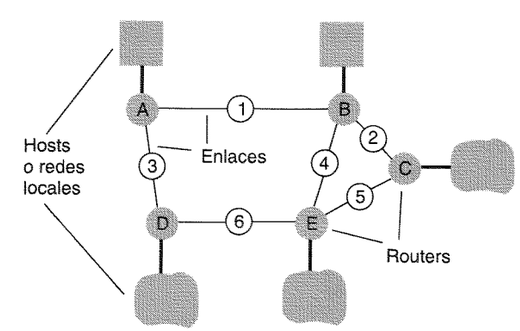


Figura 3.7. Encaminamiento en una red de área amplia.

2. Debe actualizar dinámicamente su conocimiento de la red basándose en la monitorización del tráfico y la detección de cambios de configuración o de fallos. Esta actividad es menos crítica respecto al tiempo, por lo que ser pueden utilizar técnicas de cómputo más lentas o más intensivas.

Ambas actividades están distribuidas a lo largo de la red. Las decisiones de encaminamiento se toman salto a salto, utilizando información local para determinar el siguiente salto a dar por el paquete recién llegado. La información almacenada localmente es actualizada periódicamente por un algoritmo que distribuye información sobre el estado de los enlaces (su carga y sus estados de error).

Nuestra presentación comenzará con la descripción de un algoritmo de encaminamiento relativamente simple. Esto aportará las bases para la discusión del algoritmo de *estado de enlace* de la Sección 3.4.3, que desde 1979 ha sido utilizado como el principal algoritmo de encaminamiento en Internet. El algoritmo que describimos aquí es del tipo *vector de distancias*. El encaminamiento en las redes es un caso particular del problema de encontrar caminos en grafos. El algoritmo del camino más corto de Bellman, publicado bastante antes de que fueran desarrolladas las redes de computadores [Bellman 1957], proporciona la base para el método de vector de distancia. El método de Bellman fue convertido en un algoritmo distribuido capaz de ser implementado en grandes redes por Ford y Fulkerson [1962], y los protocolos basados en su trabajo son referidos a menudo como protocolos *Bellman-Ford*.

La Figura 3.8 muestra las tablas de encaminamiento que se deberían almacenar en cada uno de los routers de la red de la Figura 3.7, suponiendo que la red no tenga ni enlaces ni routers caídos. Cada fila contiene la información de encaminamiento relevante para los paquetes dirigidos a un cierto destino. El campo *enlace* especifica el enlace de salida para los paquetes dirigidos a cada destino. El campo *coste* es un cálculo del vector distancia, o el número de saltos para alcanzar el destino. Para las redes de almacenamiento y reenvío con enlaces de ancho de banda similar, esto proporciona una estimación razonable del tiempo que tardará en llegar el paquete al destino. La información del coste almacenada en las tablas de encaminamiento no se utiliza durante la primera fase del algoritmo de encaminamiento, pero es necesaria para la construcción y el mantenimiento de las tablas de encaminamiento de la segunda fase.

Rutas desde A

Hacia	*Enlace*	*Coste*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Rutas desde B

Hacia	*Enlace*	*Coste*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Rutas desde C

Hacia	*Enlace*	*Coste*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Rutas desde D

Hacia	*Enlace*	*Coste*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	1

Rutas desde E

Hacia	*Enlace*	*Coste*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Figura 3.8. Tablas de encaminamiento para la red de la Figura 3.7.

Las tablas de encaminamiento contienen una entrada por cada posible destino, en la que se muestra el siguiente *salto* que debe dar el paquete hacia su destino final. Cuando un paquete llega a un router, se extrae su dirección destino y se busca en la tabla de encaminamiento. La entrada resultante identifica el enlace de salida que tiene que ser utilizado para encaminar el paquete hacia su destino.

Por ejemplo, cuando un paquete dirigido hacia C es entregado al router A, éste examina la entrada para C en su tabla de encaminamiento. Esta entrada muestra que el paquete debería ser encaminado hacia A por el enlace etiquetado como 1. El paquete llega entonces a B y siguiendo el mismo procedimiento que el utilizado en la tabla de encaminamiento de B, se determina que la ruta de salida de C para ese paquete es el enlace 2. Cuando el paquete llega a C, la entrada de la tabla indica *local* en lugar de un número de enlace. Esto indica que el paquete debe ser entregado en el host local.

Consideremos ahora el modo en que se construyen las tablas de encaminamiento y cómo se mantienen cuando se producen fallos en la red. Es decir, el modo en que se ejecuta la segunda fase del algoritmo de encaminamiento referido anteriormente. Dado que cada tabla de encaminamiento especifica un único salto para cada ruta, la construcción o reparación de la información de encaminamiento puede llevarse a cabo de modo distribuido. Un router intercambia información sobre la red con sus nodos vecinos enviando un resumen de su tabla de encaminamiento utilizando para ello un *protocolo de información de encaminamiento* (*router information protocol*, RIP). Las acciones desarrolladas en un router como consecuencia del protocolo RIP pueden describirse de manera informal como sigue:

1. *Periódicamente y siempre que cambie la tabla local*, enviar la tabla (de manera resumida) a todos los vecinos accesibles. Esto es, enviar un paquete RIP conteniendo una copia de la tabla con cada enlace saliente no caído.
2. *Cuando se recibe una tabla de un router vecino*, si la tabla recibida muestra un nuevo destino, o una ruta mejor (menos costosa) a un destino existente, entonces modificar la tabla local añadiendo la nueva ruta. Si la tabla fue recibida por el enlace *n* y da un coste diferente que la tabla local para la ruta que comienza con el enlace *n*, entonces reemplazar el coste en la tabla local con el nuevo coste. Esto es así porque la tabla nueva fue enviada desde un router más cercano al destino relevante y está por lo tanto más autorizado respecto de las rutas que pasan a través de él.

Este algoritmo se describe de manera más precisa en el programa en pseudocódigo mostrado en la Figura 3.9, donde *Tr* es la tabla recibida desde otro router y *Tl* es la tabla local. Ford y Fulkerson [1962] han demostrado que los pasos descritos anteriormente son suficientes para asegurar que las

```
Envía: Cada t segundos, o cuando cambia Tl, enviar Tl por cada enlace saliente no caído.
Recibe: Siempre que se reciba una tabla Tr por el enlace n:
  para todas las filas Rr en Tr {
    si (Rr.enlace ≠ n){
      Rr.coste = Rr.coste + 1;
      Rr.enlace = n;
      si (Rr.destino no está en Tl) añadir Rr a Tl; // añadir un nuevo destino a Tl
      // si no para todas las filas de Rl en Tl{
        si (Rr.destino = Rl.destino y
            (Rr.coste < Rl.coste o Rl.enlace = n)) Rl = Rr;
            // Rr.coste < Rl.coste: el nodo remoto es una ruta mejor
            // Rl.enlace= n: el nodo remoto está más autorizado
      }
    }
  }
```

Figura 3.9. Pseudocódigo del algoritmo de encaminamiento RIP.

tablas de encaminamiento convergerán en las mejores rutas para cada destino siempre que se dé algún cambio en la red. La frecuencia t con la que se propagan las tablas, incluso si no se producen cambios, está diseñada para asegurar que se mantiene una estabilidad, por ejemplo, en el caso de que se pierdan algunos paquetes RIP. El valor de t en Internet es de 30 segundos.

Para tratar los fallos, cada router comprueba sus enlaces y actúa de la siguiente manera:

Cuando se detecta un enlace caído, se pone el coste a ∞ en todas las entradas de la tabla local que se refieren al enlace caído y se ejecuta la operación *Envía*.

De este modo, el hecho de que un enlace esté roto se representa por un valor de infinito en el costo asociado a los destinos relevantes. Cuando esta información es propagada a los nodos vecinos será procesada según la acción *Recibe* (téngase en cuenta que $\infty + 1 = \infty$) y entonces llegará hasta un nodo que tenga una ruta activa hacia los destinos relevantes, si es que existe. El nodo con la ruta activa en algún momento propagará su tabla, y la ruta activa reemplazará a la caída en todos los nodos.

El algoritmo de vectores de distancia puede mejorarse en varios aspectos: los costes (también conocidos como *métricas*) pueden basarse en los anchos de bandas efectivos de los enlaces; el algoritmo se puede modificar para incrementar su velocidad de convergencia y para evitar estados intermedios indeseables, tales como ciclos, que pueden darse antes de alcanzar la convergencia. El primer protocolo de encaminamiento utilizado en Internet fue un protocolo con esas mejoras llamado RIP-1 y descrito en el RFC 1058 [Hedrick 1988]. Pero las soluciones adoptadas para corregir los problemas causados por la lenta convergencia no son totalmente efectivas, y esto conduce a un encaminamiento ineficaz y a la pérdida de paquetes cuando la red está en un estado intermedio.

Los desarrollos siguientes de los algoritmos de encaminamiento se han orientado hacia el incremento de la cantidad de conocimiento de la red que se almacena en cada nodo. La más importante familia de algoritmos de este tipo se conoce como *algoritmos de estado de enlace* (*link-state algorithms*). Se basan en la distribución y actualización de una base de datos en cada nodo que representa la totalidad o una porción substancial de la red. Cada nodo es responsable de calcular las rutas óptimas para los destinos incluidos en su base de datos. Este cálculo puede ser realizado por varios algoritmos, algunos de los cuales evitan problemas conocidos del algoritmo Bellman-Ford tales como la convergencia lenta y los indeseables estados intermedios. El diseño de los algoritmos de encaminamiento es un tema esencial y nuestro tratamiento del mismo es necesariamente limitado. Para un tratamiento extensivo del encaminamiento en Internet consulte a Huitema [1995] y para más material sobre los algoritmos de encaminamiento en general refiérase a Tanenbaum [1996].

3.3.6. CONTROL DE LA CONGESTIÓN

La capacidad de la red está limitada por las prestaciones de sus enlaces de comunicación y por los nodos de conmutación. Cuando la carga en un enlace o en un nodo se acerca a su capacidad máxima, se forman colas con los mensajes que los hosts están intentando enviar y en los nodos intermedios se almacenan las transmisiones que no se pueden realizar al estar bloqueadas por el tráfico. Si la carga continúa en el mismo nivel alto, las colas seguirán creciendo hasta alcanzar el límite de espacio disponible en cada búfer.

Una vez que un nodo alcanza este estado, no tiene otra opción que desechar los paquetes que le llegan. Como ya hemos comentado anteriormente, la pérdida ocasional de paquetes en el nivel de red es aceptable y puede ser remediada mediante retransmisiones en los niveles superiores. Pero si la tasa de paquetes perdidos y retransmitidos alcanza un determinado nivel, el efecto en el rendimiento de la red puede ser devastador. Es fácil ver por qué sucede esto: si los paquetes son desechados en los nodos intermedios, los recursos de la red que ya han sido consumidos son desperdiciados y las consecuentes retransmisiones requerirán una cantidad similar de recursos para alcanzar el mismo punto de la red. Como regla práctica, cuando la carga de una red excede el 80 % de su capacidad, el rendimiento total tiende a caer como resultado de las pérdidas de paquetes, a menos que se controlen los enlaces cargados en exceso.

En lugar de permitir que los paquetes viajen sin más por la red hasta que llegan a los nodos sobrecargados, donde tendrán que ser desechados, sería mejor almacenarlos en nodos anteriores a los sobrecargados, hasta que la congestión se reduzca. Esto incrementará los retardos de los paquetes, pero no se degradará significativamente el rendimiento de la red. Bajo el nombre de «control de la congestión» se agrupan las técnicas que se diseñan para controlar este aspecto.

En general, el control de la congestión se consigue informando a los nodos a lo largo de la ruta de que se ha producido la congestión, y que, en consecuencia, debería reducirse su tasa de transmisión de paquetes. Para los nodos intermedios, esto implicará el almacenamiento de los paquetes entrantes en cada búfer por un período largo. Para los hosts que son fuente de los paquetes, el resultado quizás sea que los paquetes sean puestos en colas antes de su transmisión, o que se bloquee el proceso que está generándolos hasta que la red pueda admitir los paquetes.

Todas las capas de red basadas en datagramas, incluyendo IP y Ethernet, basan el control del tráfico en métodos de extremo a extremo. Esto es, el nodo emisor debe reducir la tasa a la que transmite los paquetes basándose exclusivamente en la información que recibe del nodo receptor. La información sobre la congestión puede ser enviada al nodo emisor mediante la transmisión explícita de paquetes especiales (llamados paquetes de estrangulamiento, o *choke paquets*) que solicitan una reducción en la tasa de transmisión, o mediante la implementación de un protocolo de control de la transmisión específico (de lo cual toma su nombre el protocolo TCP, cuyo mecanismo se explicará en la Sección 3.4.6), o por la observación de la ocurrencia de las pérdidas de paquetes (si el protocolo es uno de aquellos en el que cada paquete es reconocido).

En algunas redes basadas en circuitos virtuales, la información sobre la congestión puede recibirse en todos los nodos, cada uno de los cuales actuará en consecuencia. Aunque ATM utiliza circuitos virtuales, se basa en la gestión de la calidad de servicio (véase la Sección 3.5.3 y el Capítulo 15) para asegurar que cada circuito puede soportar el tráfico requerido.

3.3.7. INTERCONEXIÓN DE REDES

Existen muchas tecnologías de red con diferentes protocolos de red, enlace y nivel físico. Las redes locales se construyen con tecnologías Ethernet y ATM, las redes de área amplia se construyen sobre otras análogas y con redes de telefonía digital de distintos tipos, enlaces de satélite y redes ATM de área amplia. Los computadores individuales y las redes locales están conectadas a Internet o a intranets mediante módems, enlaces ISDN y conexiones DSL.

Para construir una red integrada (una interred) debemos integrar muchas subredes, cada una de las cuales se basa en una de esas tecnologías de red. Para hacer esto posible se necesita:

1. Un esquema de direccionamiento unificado que posibilite que los paquetes sean dirigidos a cualquier host conectado en cualquier subred.
2. Un protocolo que defina el formato de los paquetes interred y las reglas según las cuales serán gestionados.
3. Componentes de interconexión que encaminen paquetes hacia su destino en términos de dirección interred, transmitiendo los paquetes utilizando subredes con tecnologías de red variadas.

En Internet, la primera condición está resuelta por la utilización de las direcciones IP, el protocolo referido en la segunda condición es el protocolo IP y la tercera condición está soportada por unos componentes llamados *Routers de Internet*. El protocolo IP y el direccionamiento IP están descritos con detalle en la Sección 3.4. Aquí describiremos las funciones de los routers de Internet y algunos otros componentes que se usan para conectar a las redes.

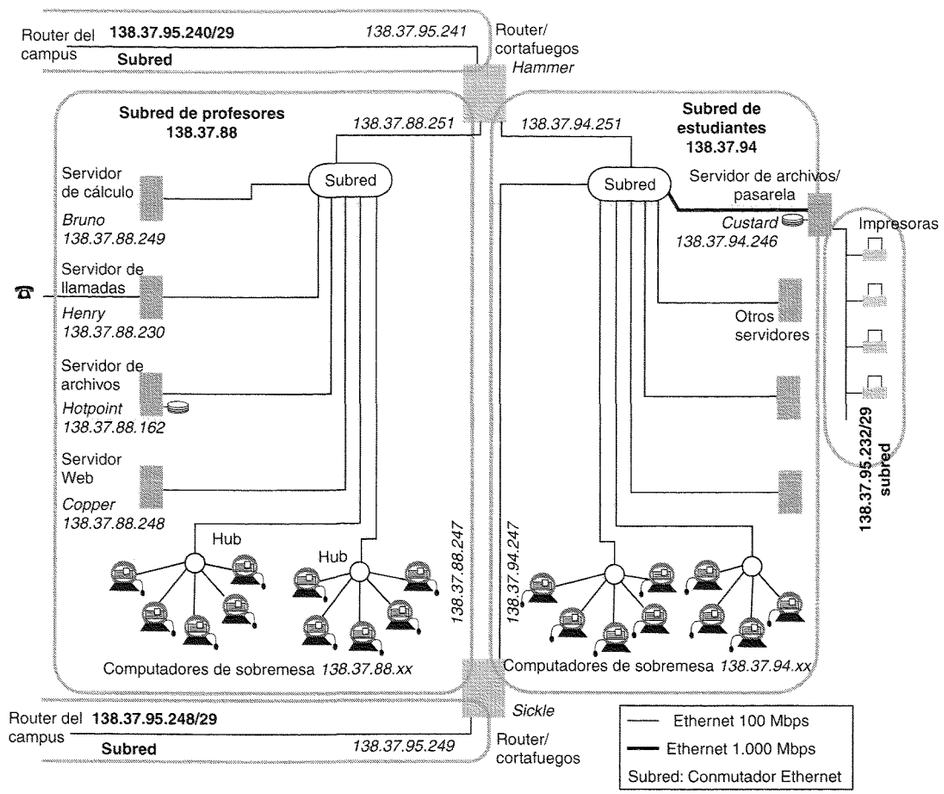


Figura 3.10. Visión simplificada de la red de Informática del QMW.

La Figura 3.10 muestra una pequeña parte de la *intranet* del *Queen Mary and Westfiel College* (QMW) de la Universidad de Londres. Muchos de los detalles que muestra serán explicados en las secciones posteriores. Ahora vamos a fijarnos en que la figura contiene varias subredes interconectadas mediante routers. Las subredes aparecen recuadradas; existen cinco de ellas, tres de las cuales comparten la red IP 138.37.95 (utilizando el esquema de encaminamiento entre dominios sin clases descrito en la Sección 3.4.3). Las direcciones numéricas son direcciones IP y se explicarán más adelante. Los routers están colocados entre varias subredes y tienen una dirección IP por cada una a las que están conectados (las direcciones están colocadas junto a los enlaces).

Los routers (con nombre de host *hammer* y *sickle*) son, de hecho, computadores de propósito general que también atienden a otros fines. Uno de ellos es de servir como cortafuegos; el papel de un cortafuegos está estrechamente ligado con la función de encaminamiento, como describiremos más adelante. La subred 138.37.95.232/29 no está conectada con el resto de la red a nivel IP. Solamente el servidor de archivos *custard* puede acceder a ella para proporcionar servicios de impresión en las impresoras conectadas a la subred, a través de un proceso servidor que gestiona y controla el uso de las mismas.

Todos los enlaces de la Figura 3.10 son Ethernet. El ancho de banda de la mayoría de ellos es 100 Mbps, excepto uno que es de 1.000 Mbps, ya que lleva un gran volumen de tráfico entre el

gran número de computadores utilizados por los estudiantes y *custard*, el servidor de archivos que alberga todos sus archivos.

En la parte de la red mostrada hay dos conmutadores Ethernet y varios concentradores Ethernet. Ambos tipos de componentes resultan transparentes para los paquetes IP. Un concentrador Ethernet es un medio sencillo de conectar juntos distintos segmentos de cable Ethernet, formando una única red Ethernet para el protocolo de red. Todos los paquetes Ethernet recibidos por el concentrador son retransmitidos a todos los segmentos. Un conmutador Ethernet conecta varias redes Ethernet, encaminando los paquetes entrantes sólo a la red Ethernet a la que está conectada el host destino.

◇ **Routers.** Hemos comentado anteriormente que el encaminamiento es necesario en todas las redes excepto en aquellas en las que, como Ethernet y las inalámbricas, todos los hosts están conectados por un medio de transmisión único. La Figura 3.7 mostraba una red con cinco routers conectados por seis enlaces. En una interred, los routers pueden enlazarse mediante conexiones directas, tal y como sucede en la Figura 3.7, o bien pueden estar interconectados a través de subredes, siendo éste el caso de *custard* en la Figura 3.10. En ambos casos, los routers son los responsables de reenviar los paquetes de interred que les llegan hacia las conexiones salientes correctas según se explicó anteriormente, para lo cual se mantienen las tablas de encaminamiento.

◇ **Puentes.** Los puentes (*bridges*) enlazan redes de distintos tipos. Algunos puentes comunican varias redes, y se les denomina puentes/routers porque también efectúan funciones de encaminamiento. Por ejemplo, la red del QMW incluye una conexión troncal (*backbone*) de fibra óptica (*Fibre Distributed Data Interface*, FDDI), no mostrada en la Figura 3.10, que está conectada a las subredes Ethernet de la figura mediante puentes/routers.

◇ **Concentradores.** Los concentradores (*hubs*) son un modo apropiado para conectar hosts y extender los segmentos de Ethernet y otras tecnologías de redes locales de difusión. Tienen varios conectores (generalmente entre 4 y 64), a los que se pueden conectar hosts. También pueden ser utilizados para eludir las limitaciones de distancia en un único segmento y proporcionar un modo de añadir hosts adicionales.

◇ **Conmutadores.** Un conmutador (*switch*) lleva a cabo una función similar a la que desarrolla un router, pero restringida a redes locales, normalmente Ethernet. Según esto, pueden interconectar distintas redes Ethernet separadas, encaminando los paquetes entrantes a la red de salida apropiada. Efectúan su tarea en el nivel del protocolo Ethernet. Comienzan sin ningún conocimiento sobre la interred, y van construyendo sus tablas de encaminamiento basándose en la observación del tráfico, complementando la falta de información con la difusión de solicitudes de información.

La ventaja de los conmutadores sobre los concentradores es que pueden separar el tráfico entrante y transmitirlo sólo hacia la red de salida relevante, reduciendo la congestión en las otras redes a las que están conectados.

◇ **Túneles.** Los puentes y los routers transmiten paquetes de interred sobre una variedad de redes subyacentes, pero se da una situación en la cual el protocolo de red puede quedar oculto para los protocolos superiores sin tener que utilizar un protocolo especial de interred. Cuando un par de nodos conectados a dos redes separadas necesitan comunicarse a través de algún otro tipo de red o sobre un protocolo extraño, pueden hacerlo construyendo un protocolo soterrado o de *túnel* (*tunnelling*). En la Figura 3.11 se muestra el propósito de los túneles, donde se utiliza para posibilitar la migración de Internet al recientemente aprobado protocolo IPv6. El protocolo IPv6 está designado a reemplazar a la versión actual de IP, IPv4, con la que es incompatible. (Tanto IPv4 como IPv6 son descritos en la Sección 3.4.) Durante el período de transición a IPv6 existirán islas de IPv6 en el mar de IPv4. En la figura, A y B son esas islas. En los límites de las islas los paquetes IPv6 son encapsulados en paquetes IPv4 y transportados de ese modo sobre las redes IPv4 intervinientes.

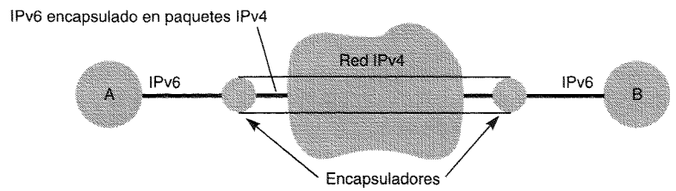


Figura 3.11. Empleo de túneles para la migración a IPv6.

Por lo tanto, un protocolo túnel es una capa de software que transmite paquetes a través de un entorno de red extraño. Por ejemplo, el protocolo MobileIP transmite paquetes IP a hosts móviles en cualquier sitio de Internet construyendo un túnel a ellos desde la red considerada la base de los hosts móviles. Los nodos de red intervinientes no necesitan ser modificados para gestionar el protocolo MobileIP. El protocolo IP de multidifusión se maneja de un modo similar, llegando a los pocos routers que soportan el encaminamiento de la multidifusión IP para determinar las rutas, pero transmitiendo paquetes IP a través de las otras rutas utilizando direcciones IP estándares. El protocolo PPP para la trasmisión de paquetes IP sobre enlaces serie es otro ejemplo de ello.

La analogía siguiente explica la razón de la elección de la terminología y proporciona otro modo de pensar sobre el túnel. Un túnel en una montaña posibilita que una carretera la atraviese y puedan circular los automóviles allá donde de otro modo sería imposible. La carretera es continua y el túnel es trasparente para las aplicaciones (o automóviles). La carretera es el mecanismo de trasporte, y el túnel posibilita que funcione en un entorno extraño.

3.4. PROTOCOLOS INTERNET

Aquí describiremos las principales características del conjunto de protocolos TCP/IP y discutiremos sus ventajas y limitaciones cuando se utilizan en sistemas distribuidos.

Internet surgió después de dos décadas de investigación y desarrollo de redes de área amplia en los Estados Unidos, comenzando en los primeros años setenta con ARPANET, la primera red de computadores a gran escala desarrollada [Leinet y otros 1997]. Una parte importante de esa investigación fue el desarrollo del conjunto de protocolos TCP/IP. TCP es el acrónimo de *Transmisión Control Protocol* (protocolo de control de la transmisión), e IP se refiere a *Internet Protocol* (protocolo de Internet). La extensa adopción de TCP/IP y de los protocolos de aplicación de Internet en las redes de investigación nacionales, y más recientemente en las redes comerciales de muchos países, ha posibilitado que las redes nacionales puedan ser integradas en una sola interred que ha crecido de forma extremadamente rápida hasta alcanzar su tamaño actual, con más de 60 millones de hosts. Existen muchos servicios de aplicación y protocolos de nivel de aplicación (mostrados entre paréntesis en la siguiente lista) basados en TCP/IP, incluyendo el Web (HTTP), el correo electrónico (SMTP, POP), las redes de noticias (NNTP), la transferencia de archivos (FTP), y la conexión remota (Telnet). TCP es un protocolo de transporte; puede ser utilizado para soportar aplicaciones directamente sobre él, o se le pueden superponer capas adicionales de protocolos para proporcionar características adicionales. Por ejemplo, HTTP es usualmente trasportado directamente por TCP, pero cuando se requiere seguridad en la comunicación entre extremos, se coloca sobre él el protocolo *Secure Sockets Layer* (SSL) (descrito en la Sección 7.6.3) para conseguir canales seguros sobre los que enviar los mensajes HTTP.

Los protocolos Internet fueron desarrollados principalmente para soportar aplicaciones simples de área amplia entre computadores dispersos geográficamente, tales como la trasferencia de archivos y el correo electrónico, involucrando comunicaciones con latencias relativamente grandes; pero se han mostrado lo suficientemente eficientes como para soportar los requisitos de muchas aplicaciones distribuidas tanto en redes de área local como de área amplia, y ahora son utilizados casi universalmente en los sistemas distribuidos. La estandarización resultante de los protocolos de comunicación ha producido inmensos beneficios. Las únicas excepciones significativas a la adopción generalizada de la comunicación TCP/IP son:

- La utilización de WAP en las aplicaciones de los dispositivos móviles.
- Protocolos especiales para soportar aplicaciones multimedia basadas en flujos.

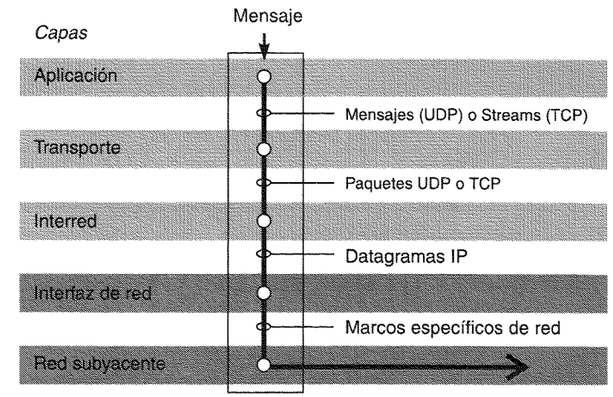


Figura 3.12. Capas TCP/IP.

En la Figura 3.12 se muestra el caso general de las capas de protocolos interred de la Figura 3.6, trasladado al caso específico de Internet. Existen dos protocolos de transporte, TCP (*Transport Control Protocol*) y UDP (*User Datagram Protocol*). TCP es un protocolo fiable orientado a conexión, mientras que UDP es un protocolo de datagramas que no garantiza fiabilidad en la transmisión. El protocolo interred IP (*Internet Protocol*) es el protocolo de *red* subyacente de la red virtual Internet; esto es, los datagramas proporcionan un mecanismo de transmisión básico para Internet y otras redes TCP/IP. Hemos puesto en cursiva a la palabra *red* en la frase anterior porque no es la única capa de red involucrada en las comunicaciones de Internet. Esto se debe a que los protocolos Internet están soportados usualmente por otra tecnología de red, como Ethernet, la cual proporciona una capa de red física que posibilita que los computadores conectados a la misma red intercambien datagramas. La Figura 3.13 muestra la encapsulación de paquetes que se da en la trasmisión de mensajes vía TCP sobre una red Ethernet. Las etiquetas en las cabeceras indican el tipo de protocolo de la capa superior, y son necesarias para que la pila de protocolos receptora desembale correctamente los paquetes. En la capa TCP, el número de puerto del receptor sirve para un propósito semejante, haciendo posible que el componente de software TCP en el host receptor pase el mensaje al correspondiente proceso de la capa de aplicación.

Las especificaciones de TCP/IP [Postel 1981a; 1981b] no especifican las capas por debajo de la capa de datagramas interred; los paquetes IP en la capa interred son transformados en paquetes correspondientes para la transmisión sobre casi cualquier combinación de redes subyacentes o enlaces de datos.

Por ejemplo, IP funcionó inicialmente sobre ARPANET, que constaba de hosts y de una versión temprana de routers (llamados PSE) conectados por enlaces de datos de larga distancia. Hoy en día se utiliza sobre casi todo tipo de tecnología de red conocida, incluida ATM, redes de área

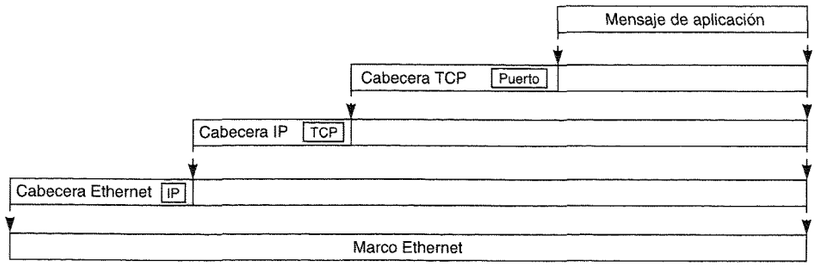


Figura 3.13. Encapsulamiento de un mensaje cuando se transmite vía TCP sobre Ethernet.

local como Ethernet y redes Token Ring. IP se encuentra implementado sobre líneas serie y circuitos telefónicos vía el protocolo PPP [Parker 1992], haciendo posible su utilización en las comunicaciones con módem y otros enlaces serie.

El éxito de TCP/IP se basa en su independencia de la tecnología de transmisión subyacente, haciendo posible construir interredes a partir de varias redes y enlaces de datos heterogéneos. Los usuarios y los programas de aplicación perciben una única red virtual que soporta TCP y UDP, y los constructores de TCP y UDP ven una única red IP virtual, ocultando la diversidad de medios de transmisión. Esta visión es la mostrada en la Figura 3.14.

En las dos secciones siguientes describiremos el esquema de direccionamiento IP y el protocolo IP. El sistema de nombres de dominio (*Domain Name System*, DNS), que es el encargado de la traducción en direcciones IP de nombres de dominio como *www.amazon.com*, *hpl.hp.com*, *stanford.edu* y *qmw.ac.uk*, con los que los usuarios de Internet están tan familiarizados, será presentado en la Sección 3.4.7 y descrito con más profundidad en el Capítulo 9.

La versión de IP actualmente en uso en Internet es IPv4 (desde enero de 1984), y ésta es la versión que nosotros describiremos en las siguientes dos secciones. Pero el rápido crecimiento en el uso de Internet condujo a la publicación de la especificación de una nueva versión (IPv6) que resuelve las limitaciones de direccionamiento de IPv4 y añade nuevas características para satisfacer nuevos requisitos. Describiremos IPv6 en la Sección 3.4.4. Dada la gran cantidad de software que está afectado por el cambio, la migración gradual a IPv6 está planeada para ser llevada a cabo en un período de 10 años o más.

3.4.1. DIRECCIONAMIENTO IP

Quizás uno de los aspectos que ofrecieron un mayor desafío en el diseño de los protocolos de Internet fue la construcción de esquemas de nombres y de direccionamiento para los hosts que pudieran permitir el encaminamiento de los paquetes IP a sus destinos. El esquema utilizado debía satisfacer los siguientes requisitos:

- Debería ser universal: cualquier host debería ser capaz de enviar paquetes a cualquier otro host en Internet.
- Debería ser eficiente en el uso del espacio de direccionamiento: es imposible predecir el tamaño último de Internet y el número de redes y de hosts a considerar. El espacio de direccionamiento debe ser repartido cuidadosamente para asegurarse que las direcciones no se agoten. Entre 1978-1982, cuando se comenzaron a desarrollar las especificaciones para los protocolos TCP/IP, se consideró que con 2^{32}, o lo que es aproximadamente lo mismo, con 4.000 millones de hosts direccionables (aproximadamente la población del mundo en esa

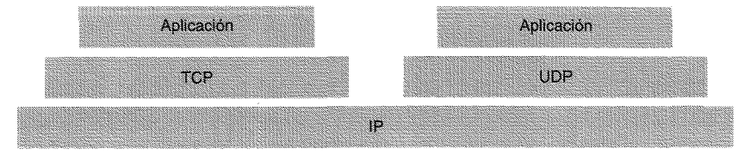


Figura 3.14. Visión conceptual del programador de una interred TCP/IP.

época) se tendrían suficientes direcciones. Se ha demostrado que esta estimación se quedó corta por dos razones:

— La tasa de crecimiento de Internet ha excedido por mucho todas las predicciones.
— El espacio de direccionamiento ha sido reservado y utilizado mucho menos eficientemente de lo esperado.

- El esquema de direccionamiento debe conducir por sí mismo al desarrollo de un esquema de encaminamiento flexible y eficiente, pero las direcciones no deben contener mucha más información de la estrictamente necesaria para que los paquetes sean encaminados a su destino.

El esquema elegido asigna una dirección IP a cada host en Internet; un número de 32 bits formado por un identificador de red, que identifica de forma única a una de las subredes de Internet, y por un identificador de host, que identifica de manera inequívoca al host conectado a esa subred. Esta dirección se coloca en los paquetes IP y es utilizada para encaminarlos al destino.

El diseño adoptado para el espacio de direccionamiento de Internet se muestra en la Figura 3.15. Existen cuatro clases de direcciones Internet: A, B, C y D. La clase D se reservada para las comunicaciones de multidifusión (*multicasting*), que se implementa sólo sobre algunos routers y sobre la que trataremos en la Sección 4.5.1. La clase E contiene un rango de direcciones no asignadas, que están reservadas para usos futuros.

Las direcciones Internet de 32 bits contienen un identificador de red y un identificador de host, normalmente escritos como una secuencia de cuatro números decimales separados por puntos. Cada número representa uno de los cuatro bytes u *octetos* de la dirección IP. Los valores permitidos para cada clase de dirección de red son los indicados en la Figura 3.16.

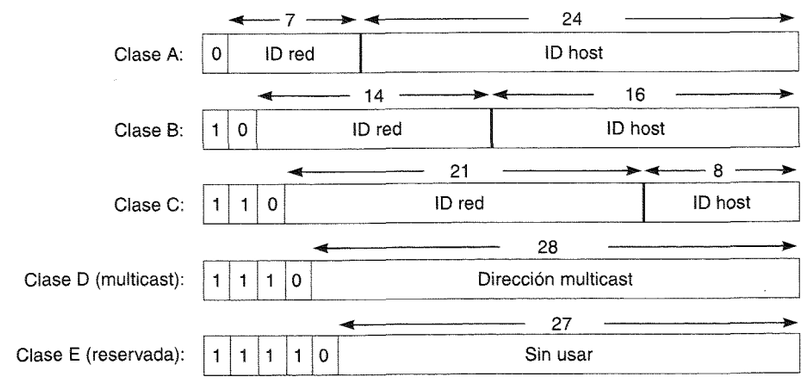


Figura 3.15. Estructura de las direcciones Internet, mostrando los tamaños de los campos en bits.

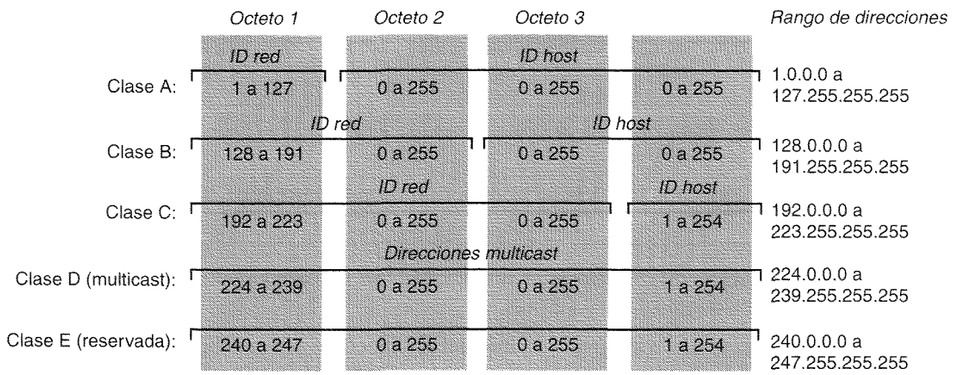


Figura 3.16. Representación decimal de las direcciones Internet.

Se diseñaron tres clases de direcciones para satisfacer los requisitos de los distintos tipos de organizaciones. Las direcciones de Clase A, con una capacidad de 2^{24} hosts en cada subred, están reservadas para grandes redes como la norteamericana NSFNet y otras redes nacionales de área amplia. Las direcciones de Clase B se reservan para organizaciones que gestionan redes con más de 255 computadores, y las direcciones de Clase C se dedican al resto de redes.

Las direcciones Internet con los bits del identificador de host a 0 y todo a 1 (en binario) se utilizan para propósitos especiales. Las direcciones con el identificador de host a 0 se utilizan para referirse a *este host*, y un identificador de host con todos los bits a 1 se utiliza para indicar que se debe difundir un mensaje (*broadcast*) a todos los hosts conectados a la red especificada en la parte del identificador de red de la dirección.

Los identificadores de red son asignados a las organizaciones con redes conectadas a Internet por el *Internet Network Information Center* (NIC). Los identificadores de los hosts para los computadores de cada red conectado a Internet son asignados por el administrador de la red en cuestión. Dado que las direcciones de host incluyen un identificador de red, cualquier computador que esté conectado a más de una red debe tener una dirección de red para cada una de ellas, y siempre que un computador se mueva a una red diferente, debe cambiar su dirección Internet. Estos requisitos pueden conducir a sobrecargas administrativas considerables, por ejemplo en el caso de computadores portátiles.

En la práctica, el esquema de reserva de direcciones IP no se ha mostrado muy efectivo. La principal dificultad es que el administrador de la red no puede predecir fácilmente el crecimiento futuro de sus necesidades de direcciones de host y tienen a sobreestimarlas, solicitando una dirección de Clase B en caso de duda. Alrededor de 1990 ya resultó evidente que según la tasa de reservas en ese momento, el NIC iba a agotar las direcciones IP hacia 1996. Entonces se tomaron dos decisiones. La primera fue el inicio del desarrollo de un nuevo protocolo IP y un nuevo esquema de direccionamiento, cuyo resultado ha sido la especificación de IPv6. La segunda decisión fue modificar radicalmente el modo en que eran reservadas las direcciones IP. El uso del espacio de direcciones IP se volvió más efectivo con un nuevo esquema de reserva y de encaminamiento llamado *encaminamiento interdominio sin clases* (*classless interdomain routing*, CIDR). Describiremos el CIDR en la Sección 3.4.3.

La Figura 3.10 mostraba una parte de la red del Departamento de Informática del *Queen Mary and Westfield College*, en la Universidad de Londres (nombre de dominio: *dcs.qmw.ac.uk*). La red incluye varias subredes del tamaño de las de la Clase C (utilizando CIDR para subdividir una red de Clase B, según se describe en la Sección 3.4.3) en el rango 138.37.88-138.37.95, enlazadas por

routers. Los routers gestionan la entrega de los paquetes IP a todas las subredes. También manejan el tráfico entre las subredes y desde las subredes hacia el resto del mundo.

3.4.2. EL PROTOCOLO IP

El protocolo IP es el encargado de transmitir datagramas desde un host a otro, si fuera necesario, vía routers intermediarios. El formato completo de los paquetes IP es bastante complejo, y sus componentes principales se muestran en la Figura 3.17. Existen varios campos en la cabecera, no mostrados en el diagrama, que se utilizan para la transmisión y en los algoritmos de encaminamiento.

IP proporciona un servicio de entrega que se puede describir como *no fiable* o como el *mejor posible*, *best-effort*, porque no existe garantía de entrega. Los paquetes se pueden perder, ser duplicados, sufrir retrasos o ser entregados en un orden distinto al original, pero esos errores surgen sólo cuando las redes subyacentes fallan o cuando los búferes en el destino están llenos. La única comprobación de errores realizada por IP es la suma de comprobación, *checksum*, de la cabecera, que es asequible de calcular y asegura que no se han detectado alteraciones en los datos bien de direccionamiento o bien de gestión del paquete. No existe comprobación de errores en los datos, lo que evita sobrecargas cuando se cruzan routers, confiando en las comprobaciones para detectar errores de los protocolos de nivel superior, TCP y UDP (un ejemplo práctico del argumento extremo a extremo (Sección 2.2.1).

La capa IP coloca los datagramas IP en paquetes de red adecuados para ser transmitidos por la red subyacente (que podría ser, por ejemplo, Ethernet). Cuando un datagrama IP es mayor que la MTU de la red subyacente, se divide en el origen en paquetes más pequeños y se reensamblan en su destino final. Los paquetes pueden seguir siendo fragmentados para adecuarse a las características de las distintas redes que cruzan en su camino desde el origen al destino. (Cada paquete tiene un identificador de fragmento que hace posible el ensamblado de los paquetes que llegan desordenados.)

La capa IP debe insertar una dirección *física* de red del destino del mensaje antes de confiárselo a la capa inferior. Esa dirección la obtiene del módulo de resolución de direcciones en la capa de Interfaz de Red Internet, que se describe en la siguiente subsección.

◇ **Resolución de direcciones.** El módulo de resolución de direcciones es el responsable de la conversión de las direcciones Internet a direcciones de la red, para una red subyacente dada (a veces llamadas direcciones físicas). Por ejemplo, si la red fuera una red Ethernet, el módulo de resolución de direcciones convertiría las direcciones Internet de 32 bits a direcciones Ethernet de 48 bits.

Esta traducción es dependiente de la tecnología de red utilizada:

- Algunos hosts están conectados directamente a conmutadores de paquetes Internet, con lo que los paquetes IP pueden ser encaminados hacia ellos sin traducción de direcciones.
- Algunas redes de área local permiten que las direcciones de red sean asignadas a los hosts de forma dinámica, y las direcciones pueden ser elegidas de manera que coincidan con la porción del identificador del host de la dirección Internet, de esta forma la traducción es tan sencilla como la extracción del identificación del host de la dirección IP.

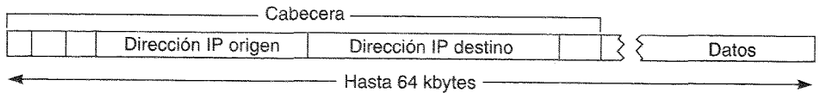


Figura 3.17. Plantilla de un paquete IP.

- Para las redes Ethernet, y para algunas otras redes locales, las direcciones de red de cada computador son establecidas por métodos hardware en las interfaces de red y no guardan ninguna relación con su dirección Internet, de modo que la traducción depende del conocimiento de la correspondencia entre las direcciones IP y las direcciones Ethernet de los hosts en la red Ethernet local.

A continuación, vamos a dar un repaso al método utilizado para la resolución de direcciones en las redes Ethernet. Para hacer posible la entrega de paquetes IP en una Ethernet, cada computador conectado a la red debe implementar el protocolo de resolución de direcciones (*Address Resolution Protocol*, ARP). Considérese primero el caso en el que un host conectado a una Ethernet utiliza IP para transmitir un mensaje a otro host en la misma Ethernet. El módulo software IP en el computador emisor debe traducir la dirección Internet del receptor que encuentra en el paquete IP a la correspondiente dirección Ethernet antes de que el paquete pueda ser entregado. Para hacer eso invoca el módulo ARP del computador emisor.

El módulo ARP mantiene en cada host una *caché* de pares (*dirección IP*, *dirección Ethernet*) que ha ido obteniendo previamente. Si la dirección IP buscada está en la caché, entonces la consulta es respondida inmediatamente. Si no, el módulo ARP transmitirá un paquete de difusión (un paquete de solicitud ARP) a todos los hosts de la red, conteniendo la dirección IP deseada. Cada una de los computadores conectados en la red Ethernet local recibirá el paquete de solicitud ARP y comprobará la dirección IP que lleva para ver si es su propia dirección IP. Si es así, enviará un paquete de respuesta ARP al emisor de la solicitud ARP conteniendo la dirección Ethernet del emisor; en otro caso el paquete de solicitud ARP será ignorado. El módulo ARP añade la nueva correspondencia *dirección IP* → *dirección Ethernet* en la caché local de modo que pueda responder a la misma pregunta en ocasiones posteriores sin la difusión de otra solicitud ARP. Después de un período de tiempo, la caché ARP de cada computador contendrá un par (*dirección IP*, *dirección Ethernet*) para todos los computadores a los que se han enviado paquetes IP. Por lo tanto, las difusiones de paquetes ARP sólo serán necesarias cuando un computador sea conectado por primera vez a la red Ethernet.

◇ **IP trucado.** Como hemos visto, los paquetes IP incluyen la dirección IP del origen, del computador emisor del mismo. Ésta será utilizada, junto con la dirección del puerto encapsulada en el campo de datos (para los paquetes UDP y TCP), para generar la dirección de retorno. Desgraciadamente, no es posible garantizar que la dirección origen dada sea de hecho la dirección del emisor. Un emisor malvado podría sustituir fácilmente una dirección por otra que no sea la suya. Este resquicio ha sido utilizado por varios ataques bien conocidos, incluyendo los ataques de denegación de servicio distribuidos de febrero de 2000 [Farrow 2000], mencionados en el Capítulo 1, Sección 1.4.3. El método utilizado fue generar muchas solicitudes del servicio *ping* a un gran número de computadores situados en varios sitios (*ping* es un servicio simple diseñado para comprobar la disponibilidad de un host). Todos estas solicitudes ping maliciosas contenían en el campo de dirección del emisor la dirección IP del computador objetivo. Las respuestas al ping fueron, por lo tanto, dirigidas contra el objetivo, cuyos búferes de entrada fueron sobrecargados, impidiendo que cualquier paquete IP legítimo pudieran llegar a ellos. Este ataque se verá con más detalle en el Capítulo 7.

3.4.3. ENCAMINAMIENTO IP

La capa IP encamina paquetes desde su origen hasta su destino. Cada router en Internet implementa la capa de software IP para proporcionar un algoritmo de encaminamiento.

◇ **Conexiones troncales.** La topología de Internet está dividida conceptualmente en *sistemas autónomos*, (*autonomous systems*, AS), que están divididos a su vez en *áreas*. Las intranets de la mayoría de las grandes organizaciones como universidades y grandes compañías son consideradas como AS, y normalmente incluyen varias áreas. La intranet del campus mostrada en la Figura 3.10 es un AS y la porción del mismo mostrada es un área. Cada AS representado en el mapa topológico tiene un área troncal. La colección de routers que conectan las áreas no troncales con la troncal y los enlaces que interconectan esos routers se conocen como la conexión troncal o la columna dorsal de la red. Los enlaces troncales (*backbone*) son de un ancho de banda mayor y suelen estar replicados para mayor fiabilidad. Esta estructura jerárquica se utiliza principalmente para la gestión de recursos y el mantenimiento de los componentes. No afecta al encaminamiento de los paquetes IP.

◇ **Protocolos de encaminamiento.** El primer algoritmo de encaminamiento utilizado en Internet, el RIP-1, es una versión del algoritmos de vector de distancias descrito en la Sección 3.3.5. El algoritmo RIP-2 (descrito en el RFC 1388 [Malkin 1993]) fue desarrollado a partir del anterior para cumplir algunos requisitos adicionales, incluyendo el encaminamiento entre dominios sin clases, un mejor encaminamiento multidifusión y la necesidad de autenticar los paquetes RIP para prevenir ataques a los routers.

Como la escala de Internet ha aumentado y la capacidad de los routers ha crecido, existe una tendencia hacia la adopción de algoritmos que no se vean limitados por la convergencia lenta y la potencial inestabilidad propias de los algoritmos de vectores de distancias. Este movimiento condujo a los algoritmos del tipo de estado de enlace mencionados en la Sección 3.3.5, en concreto al algoritmo OSPF (*open shortest path first*). Este protocolo está basado en un algoritmo de búsqueda de caminos de Dijkstra [1959] y ha demostrado converger mucho más rápidamente que el algoritmo RIP.

Debemos hacer notar que la adopción de nuevos algoritmos de encaminamiento en los routers IP puede llevarse a cabo de forma incremental. Un cambio en el algoritmo de encaminamiento implica una nueva versión del protocolo RIP, y un nuevo número de versión que será llevado por todos los paquetes RIP. El protocolo IP no cambia cuando se introduce un nuevo protocolo RIP. Cualquier router RIP encaminará correctamente los paquetes que le lleguen por una ruta, si no óptima, si razonable, independientemente de la versión de RIP que utilice. Pero para que los routers cooperen en la actualización de sus tablas de encaminamiento deben utilizar un algoritmo similar. Para este propósito se definieron las áreas topológicas anteriormente referidas. Dentro de cada área se aplica un único algoritmo de encaminamiento y los routers dentro de un área cooperan para mantener las tablas de encaminamiento. Todavía son habituales los routers que sólo soportan RIP-1, y coexisten con los routers que utilizan RIP-2 y OSPF, ya que los nuevos protocolos incorporan la compatibilidad hacia atrás.

En 1993, observaciones empíricas [Floyd y Johnson 1993] demostraron que los intercambios de información entre los routers RIP que se producen cada 30 segundos producían una periodicidad en las prestaciones de las transmisiones IP. La latencia media para las trasmisiones de paquetes IP mostraba un pico cada 30 segundos. El origen de este efecto fue asociado al comportamiento de los routers que ejecutaban el protocolo RIP y se encontró que cuando recibían un paquete RIP, demoraban la transmisión de cualquier paquete IP entrante hasta que la tabla de encaminamiento estuviera actualizada según los paquetes RIP recibidos. Esto producía que los routers ejecutaran las acciones RIP a trompicones. La corrección recomendada fue que el período de actualización tomara un valor aleatorio entre 15 y 45 segundos.

◇ **Routers por defecto.** Hasta ahora, nuestra discusión sobre los algoritmos de encaminamiento ha hecho suponer que cada router mantiene una tabla de encaminamiento completa mostrando la ruta a cualquier destino (subred o host conectado directamente) en Internet. A la escala actual de Internet esto es claramente imposible (el número de destinos probablemente haya superado ya el millón y crece rápidamente).

Dos posibles soluciones a este problema se vienen rápidamente a la cabeza, y ambas han sido adoptadas en un intento de aliviar los efectos del crecimiento de Internet. La primera es adoptar alguna forma de agrupamiento topológico de las direcciones IP. Antes de 1993, nada se podía hacer para intentar inferir algo sobre la localización física a partir de una dirección IP. En 1993, como parte del movimiento para la simplificación y el ahorro de direcciones IP bajo los auspicios del CIDR, se tomó la decisión de que para futuras reservas, se aplicarían las siguientes reglas:

Las direcciones desde 194.0.0.0 a 195.255.255.255 están en Europa.
Las direcciones desde 198.0.0.0 a 199.255.255.255 están en Norteamérica.
Las direcciones desde 200.0.0.0 a 201.255.255.255 están en América Central y Sudamérica.
Las direcciones desde 202.0.0.0 a 203.255.255.255 están en Asia y el Pacífico.

Dado que esas regiones geográficas también se corresponden con regiones topológicas bien definidas en Internet y sólo unos pocos routers pasarela proporcionan acceso a cada región, esto posibilita una substancial simplificación de las tablas de encaminamiento para aquellos rangos de direcciones. Por ejemplo, un router fuera de Europa puede tener una única entrada en la tabla para el rango de direcciones 194.0.0.0 a 195.255.255.255 que envía todos los paquetes IP destinados dentro de ese rango por la misma ruta al router pasarela europeo más cercano. Hay que hacer constar, que antes de esta decisión, las direcciones IP fueron reservadas sin tomar en cuenta la topología ni la geografía. Muchas de aquellas direcciones están todavía en uso, y la decisión de 1993 no implica reducción alguna en el tamaño de las tablas de encaminamiento relacionadas con ellas.

La segunda solución al problema de la explosión del tamaño de las tablas de encaminamiento es más simple y muy efectivo. Está basada en la observación de que la precisión de la información de encaminamiento puede ser escasa en la mayoría de los routers, siempre que algunos routers clave, aquéllos más cercanos a los enlaces troncales, tengan unas tablas de encaminamiento relativamente completas. La relajación en la precisión toma la forma del destino *por defecto* en las tablas de encaminamiento. La entrada por defecto especifica la ruta a utilizar por todos aquellos paquetes IP cuyos destinos no estén incluidos en la tabla de encaminamiento. Para ilustrar esto, considérense las Figuras 3.7 y 3.8 y supónganse que la tabla de encaminamiento del nodo C se modifica como sigue:

Rutas desde C		
Hacia	*Enlace*	*Coste*
B	2	1
C	local	0
E	5	1
Defecto	5	—

El nodo C ignora a los nodos A y D, por lo que encaminará los paquetes dirigidos a ellos por el enlace 5 hacia E. ¿Cuál es la consecuencia? Los paquetes dirigidos a D alcanzarán su destino sin pérdida de eficiencia en el encaminamiento, pero los paquetes dirigidos hacia A realizarán un salto extra, pasando a través de E y B en su camino. En general, la utilización de rutas por defecto compromete la eficiencia del encaminamiento a favor del tamaño de las tablas. Pero en algunos casos no existe pérdida de eficiencia, especialmente cuando un router está en un espolón, de modo que todos los mensajes salientes deben pasar por un único punto. El esquema de encaminamiento por defecto es ampliamente utilizado en Internet; ningún router almacena las rutas para todos los destinos en Internet.

◇ **Encaminamiento en subredes locales.** Los paquetes dirigidos a la misma red del emisor se transmiten al destino en un único salto, utilizando la parte del identificador del host de la dirección para obtener la dirección del host destino en la red subyacente. La capa IP utiliza ARP para conseguir la dirección de red del destino y entonces encomienda a la red subyacente la transmisión de los paquetes.

Si la capa IP del emisor descubre que el destino está en una red diferente, debe enviar el mensaje al router local. Utiliza ARP para conseguir la dirección de red de la pasarela o del router y la utiliza para que la red subyacente transmita el paquete. Las pasarelas y routers están conectados a dos o más redes y tienen varias direcciones Internet, una para cada red a la que están conectados.

◇ **Encaminamiento interdominio sin clases (CIDR).** La carencia de direcciones IP a la que nos referíamos en la Sección 3.4.1 condujo a la introducción en 1996 de este modo de reserva de direcciones y de gestión de las entradas en las tablas de encaminamiento. El principal problema era la escasez de direcciones de la Clase B, aquéllas para las subredes con más de 255 hosts conectados, mientras que se encontraban disponibles muchas de las direcciones de la Clase C. La solución CIDR (*Classless interdomain routing*) para este problema es reservar un bloque de direcciones C contiguas para aquellas subredes que necesitaban más de 255 direcciones. El esquema CIDR también hacía posible la división del espacio de una dirección de Clase B en múltiples subredes.

El agrupamiento de direcciones de Clase C suena como un paso fácil de realizar, pero a menos que sea acompañado de un cambio en el formato de las tablas de encaminamiento, tendrá un gran impacto en el tamaño de las tablas de encaminamiento y en la eficiencia de los algoritmos que las gestionan. El cambio adoptado fue añadir un campo de *máscara* a las tablas de encaminamiento. La máscara es un patrón de bits utilizado para seleccionar la porción de las direcciones IP que será comparada con las entradas de la tabla de encaminamiento. Esto posibilita que las direcciones host/subred ocupen cualquier parte de la dirección IP, proporcionando más flexibilidad que la aportada por las clases A, B y C. De ahí el nombre de encaminamiento entre dominios *sin clases*. Una vez más, estos cambios en los routers se realizan de modo incremental, por lo que algunos routers utilizarán CIDR y otros utilizarán los viejos algoritmos basados en clases.

Esto funciona porque los nuevos rangos de direcciones de Clase C reservados se asignan módulo 256, de modo que cada rango representa un número íntegro de direcciones de subred de tamaño del de las de Clase C. En el otro extremo, algunas subredes también utilizan CIDR para subdividir el rango de direcciones en una red, de Clase A, B o C. Si la colección de subredes es conectada al resto del mundo únicamente por routers CIDR, entonces el rango de direcciones IP utilizadas dentro de la colección puede ser reservado en subredes individuales en bloques determinados por una máscara binaria de cualquier tamaño.

Por ejemplo, un espacio de direcciones de Clase C puede dividirse en 32 grupos de 8 hosts. La Figura 3.10 contiene un ejemplo de la utilización de CIDR para partir la subred 138.37.95 del tamaño de las de la Clase C en varios grupos de ocho direcciones de hosts que son encaminados de forma diferente. Los grupos separados se expresan mediante la notación 138.37.95.232/29, 138.37.95.248/29, etc. La porción /29 de esas direcciones indica que la máscara de 32 bits asociada tiene 29 unos y 3 ceros al final.

3.4.4. IP VERSIÓN 6

También se buscó una solución más duradera a las limitaciones de direccionamiento de IPv4, lo que condujo al desarrollo y la adopción de una nueva versión del protocolo IP con un espacio de direccionamiento mayor. La IETF se dio cuenta en los primeros años noventa de los problemas que se podrían plantear con las direcciones de 32 bits, e inició un proyecto para el desarrollo de una nueva versión del protocolo IP. En 1994 el IETF adoptó IPv6 y recomendó una estrategia de migración hacia él.

La Figura 3.18 muestra el esquema de las cabeceras IPv6. No nos proponemos cubrir los detalles de su construcción. Aquellos lectores que estén interesados en ellos pueden consultar el

Versión 4 (bits)	Prioridad (4 bits)	Etiqueta de flujo (24 bits)	
Longitud de los datos (16 bits)		Cabecera siguiente (8 bits)	Límite de saltos (8 bits)
Dirección origen (128 bits)			
Dirección destino (128 bits)			

Figura 3.18. Esquema de la cabecera IPv6.

material introductorio de Tanenbaum [1996] o Stallings [1998a], y a Huitema [1998] en el caso de buscar la narración del proceso de diseño de IPv6 y los planes de implantación. Nosotros mostraremos aquí los principales avances implicados por la utilización de IPv6.

Espacio de direccionamiento: las direcciones IPv6 son de 128 bits (16 bytes). Esto proporciona un número realmente astronómico de entidades direccionables: 2^{128}, o aproximadamente 3×10^{38}. Tanenbaum calcula que esto es suficiente para asignar 7×10^{23} direcciones IP por metro cuadrado en toda la superficie de la tierra. De manera más moderada, Huitema calcula que, suponiendo que las direcciones IP sean asignadas de manera tan ineficiente como los números de teléfono, a cada metro cuadrado de la superficie de la Tierra (ya sea tierra o mar) le tocan 1.000 direcciones IP.

El espacio de direccionamiento IPv6 está particionado. No podemos detallar aquí el modo en que se realizan las particiones, pero incluso una de las más pequeñas (una de las cuales podría albergar todo el espacio IPv4, haciendo una correspondencia una a una) es más grande que el espacio total de IPv4. Muchas de las particiones (representando el 72 % del total) están reservadas para funciones todavía no definidas. Dos de las mayores particiones (cada una abarcando 1/8 del espacio de direccionamiento) se reservan para propósitos generales y serán asignadas a nodos normales. Una de ellas está pensada para estar organizada de acuerdo con las localizaciones geográficas de los nodos, y la otra según los detalles organizativos. Esto permite dos estrategias alternativas al agrupamiento de las direcciones para propósitos de encaminamiento (queda por ver cuál será más eficiente o más popular).

Velocidad de encaminamiento: se ha reducido la complejidad de la cabecera básica IPv6 y por lo tanto del procesamiento requerido en cada nodo. No se aplican sumas de comprobación de errores al contenido del paquete (supone carga), y los paquetes no se pueden fragmentar una vez que han comenzado su viaje. Lo primero se considera aceptable ya que los errores se pueden detectar en los niveles más altos (TCP incluye una comprobación de errores de la carga), y lo último se consigue basándose en un mecanismo para determinar la MTU mínima antes de enviar los paquetes.

Tiempo real y otros servicios especiales: los campos de *prioridad* y de *etiqueta de flujo* están relacionados con estos temas. Los flujos multimedia y otras secuencias de datos de tiempo real pueden transmitirse como parte de un flujo identificado. El campo prioridad se puede utilizar junto con el campo de etiqueta de flujo, o de manera independiente, para especificar que ciertos paquetes específicos tienen que ser gestionados más rápidamente o con mayor fiabilidad que otros. Los valores de prioridad en el rango 0 a 8 son para aquellas transmisiones que pueden ser ralentizadas sin producir efectos desastrosos en la aplicación que soportan. Los valores del 8 al 15 se reservan para aquellos paquetes cuya entrega depende del tiempo. Estos paquetes deben ser entregados rápidamente o deben ser desechados, ya que la entrega fuera de tiempo no sirve.

Las etiquetas de flujo sirven para reservar recursos que permitan cumplir con los requisitos de sincronización de los flujos de datos de tiempo real, tales como transmisiones de audio o vídeo en directo. El Capítulo 15 trata estos requisitos y los métodos para la reserva de los recursos para ellos. Por supuesto, los routers y los enlaces de transmisión en Internet tienen unos recursos limitados y el concepto de reserva de éstos para ciertos usuarios y aplicaciones no había sido previamente considerado. El uso de estas posibilidades de IPv6 dependerá del desarrollo de métodos adecuados para la realización y la gestión de las reservas de los recursos.

Evolución futura: la clave para el desarrollo de una futura evolución es el campo de *cabecera siguiente*. Si es distinto de cero, define el tipo de una extensión de la cabecera que está incluida en el paquete. Existen actualmente distintos tipos de extensiones de cabecera que proporcionan datos adicionales para servicios especiales como son los siguientes: información para routers, definición de la ruta, gestión de fragmentos, autentificación, información de la encriptación e información para la gestión en destino. Cada tipo de extensión de cabecera tiene un tamaño específico y un formato definido. Se irán definiendo nuevas extensiones de cabecera según se vayan presentando nuevas necesidades. Si existe alguna extensión de cabecera, sigue a la cabecera básica y precede la carga, e incluye su propio campo de cabecera siguiente, haciendo posible el encadenamiento de múltiples extensiones de cabecera.

Multidifusión (multicast) y monodifusión (anycast): tanto IPv4 como IPv6 soportan la transmisión de paquetes IP a múltiples destinos utilizando para ello una única dirección destino (una del rango reservado para este propósito). Los routers IP son los responsables de encaminar los paquetes a todos los hosts que se han subscrito al grupo identificado por la dirección relevante. Se pueden encontrar más detalles sobre la multidifusión IP en la Sección 4.5.1. Además de esto, IPv6 aporta un nuevo modo de transmisión denominado monodifusión, *anycast*. Este servicio entrega un paquete a al menos uno de los hosts subscritos a la dirección indicada.

Seguridad: hasta ahora, las aplicaciones Internet que precisan autenticación o una transmisión de datos privada tenían que conseguirla mediante el uso de técnicas de criptografía en la capa de aplicación. Según el punto de vista del argumento extremo a extremo este es el lugar adecuado para ello. Si la seguridad es implementada en el nivel IP entonces los usuarios y los desarrolladores dependen de la corrección del código que está implementado en cada uno de los routers a lo largo del camino, y deben confiar en los routers y en los otros nodos intermediarios para gestionar las claves criptográficas.

La ventaja de implementar la seguridad en el nivel IP es que entonces puede ser utilizada sin necesidad de implementaciones conscientes de la seguridad en los programas de aplicación. Por ejemplo, los administradores de sistemas pueden implementar un cortafuegos y aplicar la seguridad a todas las comunicaciones externas sin tener que asumir el costo de encriptación en las comunicaciones internas. Los routers también pueden utilizar los mecanismos de seguridad en el nivel IP para autentificar los mensajes de actualización de las tablas de encaminamiento que intercambian entre ellos.

La seguridad en IPv6 se implementa en las cabeceras de extensión de *autentificación* y de *encriptación de la carga*. Éstas implementan características similares al concepto de canal seguro presentado en la Sección 2.3.3. La carga se encripta y/o firma digitalmente si fuera necesario. En Ipv4 también se dispone de posibilidades similares utilizando un túnel IP entre routers o hosts que implementen la especificación IPSec (*veáse* el RFC 2411 [Thayer 1998]).

◇ **Migración desde IPv4.** Las consecuencias de un cambio en su protocolo básico son profundos para la infraestructura existente de Internet. IP se procesa en la pila de protocolos TCP/IP en

todos los hosts y en el software de todos los routers. Las direcciones IP son manejadas por muchas aplicaciones y programas de utilidad. Todos estos elementos necesitan actualizarse para soportar la nueva versión de IP. Pero el cambio se hace inevitable dado el venidero agotamiento del espacio de direcciones IPv4, y el grupo de trabajo del IETF responsable del IPv6 ha definido una estrategia de migración, que esencialmente consiste en la obtención de *islas* de routers y hosts IPv6 comunicados mediante túneles, que progresivamente se irán juntando en islas mayores. Como ya hemos dicho, los routers y hosts IPv6 no deberían tener dificultad en gestionar un tráfico mixto, ya que las direcciones IPv4 están incluidas en el espacio IPv6.

La teoría de esta estrategia puede ser técnicamente sólida, pero el avance de su implementación está siendo muy lento, quizás porque las razones económicas están en contra suya. Nosotros tenemos que confiar en que la comunidad de Internet llegará a estar suficientemente motivada por la inminente restricción en el crecimiento de Internet, como para realizar las inversiones necesarias para actualizar los protocolos software de routers y hosts, modificar las aplicaciones y cualquiera otras consecuencias de la migración.

3.4.5. IP MÓVIL

Los computadores móviles, tales como portátiles y computadores de mano, se conectan a Internet en diferentes localizaciones según se mueven. En la oficina un portátil puede conectarse a una red Ethernet local, llegando a Internet a través de un router, o quizás puede estar conectado a Internet en cualquier otro sitio vía un teléfono móvil mientras está en un coche o en el tren. El usuario deseará disponer de acceso a servicios como el correo electrónico y el Web en cualquiera de esas situaciones.

El simple acceso a servicios no hace imprescindible que un computador móvil retenga una única dirección, y puede adquirir una nueva dirección IP en cada lugar; éste es el propósito del Protocolo de Configuración Dinámica de Host (*Dynamic Host Configuration Protocol*, DHCP), el cual hace posible que un computador recién conectado adquiera del servidor DHCP una dirección IP temporal y las direcciones de los recursos locales tales como un servidor de nombres de dominio, DNS. También necesitará descubrir qué servicios locales (tales como impresión, entrega de correo, etc.) están disponibles en el sitio que visita. Los servicios de exploración son un tipo de servicios de nombres que asisten a este proceso de descubrimiento; y se describen en el Capítulo 9.

En el portátil existen muchos archivos y demás recursos a los cuales pueden necesitar acceder otros, o el portátil pudiera estar ejecutando una aplicación distribuida como un servicio de monitorización compartido que recibe notificaciones de eventos concretos tales como que los recursos que un usuario reserva sobrepasa un umbral predeterminado. Si un computador móvil debe permanecer accesible a los clientes y aplicaciones de recursos compartidos, cuando se mueve entre redes locales y redes inalámbricas, debe conservar una dirección IP única, pero el encaminamiento IP está basado en las subredes. Las subredes son ubicaciones físicas, y el encaminamiento correcto de paquetes hacia ellas depende de su localización en la red.

Una solución al problema anterior es el IP móvil. La solución se implementa de forma transparente, de modo que la comunicación IP continúa normalmente cuando un computador se mueve entre subredes en localizaciones diferentes. Se basa en la reserva permanente de una dirección IP normal para cada host móvil en una subred en su dominio *casa*.

Cuando el host móvil está conectado a su base, los paquetes se encaminan hacia él de modo habitual. Cuando está conectado a Internet en cualquier otro sitio, dos procesos agente toman la responsabilidad de reencaminar. Estos agentes son el agente de casa (*home agent*, HA) y el agente foráneo (*foreign agent*, FA). Estos procesos discurren sobre computadores en el sitio base y en la ubicación actual del host móvil.

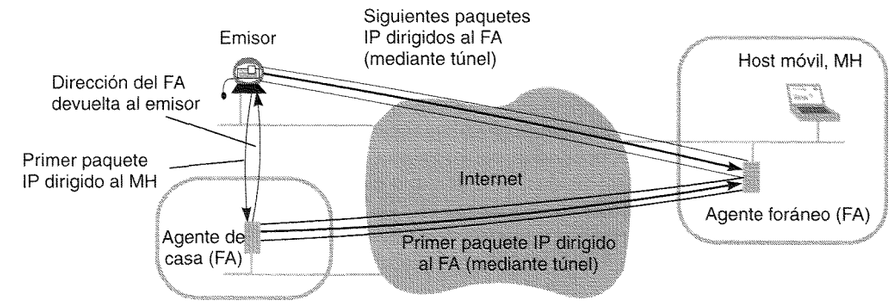


Figura 3.19. Mecanismo de encaminamiento de IP móvil.

El HA es el responsable de almacenar el conocimiento actualizado de la localización del host móvil (la dirección IP en la que se le puede encontrar). Esto se realiza con la colaboración del host móvil mismo. Cuando un host móvil deja su casa, debe informar al HA y el HA se da cuenta de la ausencia del host móvil. Durante la ausencia se comportará como un *proxy*; y para poder hacerlo, les dice a los routers locales que borren cualquier registro que guardaran sobre la dirección IP del host móvil. Mientras el HA actúa como proxy, responderá a las solicitudes ARP referidas a la dirección IP del host móvil con su propia dirección de red como si fuera la dirección de red del host móvil.

Cuando el host móvil llega a un nuevo sitio, informa al FA de ese sitio. El FA le asigna una dirección IP provisional, una dirección IP nueva temporal en la subred local. Entonces el FA contacta con el HA proporcionándole la dirección IP base del host móvil y la dirección que se ha reservado para él.

La Figura 3.19 muestra el mecanismo de encaminamiento de IP móvil. Cuando un paquete IP dirigido a la dirección base del host móvil es recibido en la red local base, es encaminado al HA. El HA encapsula el paquete IP en un paquete IP móvil y lo envía al FA. El FA desempaqueta el paquete IP original y se lo entrega al host móvil a través de la red local a la que está conectado. El método por el que tanto el HA como el FA reencaminan el paquete original hacia el destino apropiado es un ejemplo de la técnica del túnel descrita en la Sección 3.3.7.

El HA también envía la dirección provisional del host móvil al emisor original. Si el emisor es capaz de gestionar IP móvil, tomará en cuenta esa dirección y la utilizará para las subsecuentes comunicaciones con el host móvil, evitando las sobrecargas de reencaminar vía el HA. Si no es así, entonces ignorará el cambio de dirección y las siguientes comunicaciones continuarán siendo reencaminadas a través del HA.

La solución IP móvil es efectiva, pero escasamente eficiente. Sería preferible una solución que tratara a los hosts móviles como ciudadanos de primera clase, permitiéndoles vagar sin avisar antes y encaminándoles los paquetes sin utilizar túneles o reencaminamiento. Hay que darse cuenta que esta aparentemente difícil hazaña es exactamente lo que está resuelto en las redes de telefonía celular, donde los teléfonos móviles no cambian sus números según van moviéndose entre células o incluso entre países. En lugar de eso, simplemente notifican a la estación base de la célula local su presencia cada cierto tiempo.

3.4.6. TCP Y UDP

Tanto TCP como UDP aportan unas características de comunicación a Internet que la convierten en útil para los programas de aplicación. Los desarrolladores de aplicaciones quizás deseen dispo-

ner de otros tipos de servicio de transporte, como por ejemplo el tiempo real o la seguridad, pero estos servicios requieren más soporte de la capa de red que el que aporta IPv4. TCP y UDP pueden verse, desde nivel de programación de aplicaciones, como un fiel reflejo de las posibilidades que IPv4 tiene que ofrecer. IPv6 es otra historia, ya que aunque continuará soportando tanto TCP como UDP, incluye capacidades que no pueden ser aprovechadas adecuadamente por TCP y UDP. Esto puede ser una buena oportunidad para introducir tipos de servicio de transporte adicionales y explotar esas nuevas posibilidades, una vez que el despliegue de IPv6 sea lo suficientemente amplio para justificar su desarrollo.

El Capítulo 4 describe las características de TCP y de UDP desde el punto de vista de los desarrolladores de programas distribuidos. Ahora seremos breves, y solamente describiremos la funcionalidad que añaden a IP.

◇ **Uso de puertos.** La primera característica a señalar es que, mientras que IP soporta comunicaciones entre pares de computadores (identificadas por sus direcciones IP), TCP y UDP, como protocolos de transporte, deben proporcionar comunicación proceso a proceso. Esto es llevado a cabo utilizando los puertos. Los *números de puerto* se utilizan para dirigir los mensajes hacia los procesos desplegados en un computador, y sólo son válidos para ese computador. Un número de puerto es un entero de 16 bits. Una vez que un paquete IP ha sido entregado en el host destino, la capa TCP-UDP los despacha hacia el proceso indicado en el número de puerto.

◇ **Características de UDP.** UDP es casi una réplica en el nivel de transporte de IP. Cada datagrama UDP se encapsula en un paquete IP. Tiene una cabecera corta que incluye los números de puerto origen y destino (las direcciones correspondientes a los hosts están en la cabecera IP), un campo de longitud y otro de suma de comprobación. UDP no ofrece garantía de entrega. Ya hemos comentado que los datagramas IP pueden desecharse en caso de congestión o error en la red. UDP no añade ningún mecanismo adicional de fiabilidad excepto la suma de comprobación, opcional. Si el campo de suma de comprobación es distinto de cero, el host receptor calcula el valor de comprobación para el contenido del paquete y lo compara con el valor recibido en el paquete, desechándolo en caso de que no coincidan.

Por lo tanto, UDP proporciona un modo de transmitir mensajes de hasta 64 kbytes de tamaño (el máximo permitido por IP) entre pares de procesos (o desde un proceso hacia varios en el caso de datagramas dirigidos a una dirección IP de multidifusión), con un coste adicional y un retardo de transmisión mínimos sobre aquellos debidos a la transmisión IP. No existe un coste asociado a la configuración y no necesita mensajes de reconocimiento. Por ello, su uso está restringido a aquellas aplicaciones y servicios que no requieran una entrega fiable de mensajes simples o múltiples.

◇ **Características de TCP.** TCP proporciona un servicio de transporte mucho más sofisticado. Proporciona entrega fiable de secuencias de bytes arbitrariamente grandes por vía de la abstracción de la programación basada en streams. La garantía de fiabilidad implica la entrega al proceso receptor de todos los datos confiados al protocolo TCP por el proceso emisor y en el mismo orden. TCP está orientado a conexión. Antes de transferir cualquier dato, el proceso emisor y el receptor deben cooperar para establecer un canal de comunicaciones bidireccional. La conexión es simplemente un acuerdo extremo a extremo para realizar una transmisión fiable de datos; los nodos intermediarios como los routers no tienen conocimiento de las conexiones TCP, y los paquetes IP que transfieren los datos de una transmisión TCP no tienen que seguir necesariamente el mismo camino.

La capa TCP incluye mecanismos adicionales (implementados sobre IP) para cumplir con el compromiso de fiabilidad. Éstos son:

Secuenciación: el proceso emisor TCP divide los flujos en una secuencia de segmentos de datos que se transmiten como paquetes IP. A cada segmento TCP se le asocia un número de secuencia. Proporciona el número de byte correspondiente al primer byte del segmento dentro del stream. El receptor utiliza los números de secuencia para ordenar los segmentos recibidos antes de colocarlos en el stream de entrada del proceso receptor. No puede colocarse ningún segmento en el flujo de entrada hasta que todos los segmentos con números de secuencia inferiores hayan sido recibidos y colocados en el stream, de modo que los segmentos que lleguen desordenados debe reposar en un búfer hasta que lleguen sus predecesores.

Control del flujo: el emisor tiene cuidado de no saturar al receptor o a los nodos intermedios. Esto se consigue con un sistema de acuses de recibo por segmento. Siempre que el receptor recibe un segmento correctamente, almacena su número de secuencia. De vez en cuando el receptor envía al emisor un reconocimiento indicando el número de secuencia mayor de los recibidos junto con un *tamaño de ventana*. Si existe un flujo de datos inverso (del receptor al emisor), los reconocimientos se incluyen en los segmentos de datos normales, en otro caso se transmiten como segmentos de reconocimiento. El campo de tamaño de ventana en el segmento de reconocimiento indica la cantidad de datos que el emisor tiene permiso para enviar antes del siguiente reconocimiento.

Cuando una conexión TCP se utiliza para la comunicación con un programa remoto interactivo, los datos pueden producirse en pequeñas cantidades pero de manera muy continuada. Por ejemplo, la entrada de teclado puede producir sólo unos pocos caracteres por segundo, pero los caracteres deberían ser enviados lo suficientemente rápido como para que el usuario pueda ver el resultado de su tecleo. Esto puede ser resuelto imponiendo un tiempo límite T en almacenamiento local (generalmente de 0,5 segundos). Con este sencillo esquema, un segmento es enviado al receptor siempre que los datos hayan estado esperando en el búfer de salida durante T segundos o el contenido del búfer haya alcanzado el límite de la MTU. Este esquema de almacenamiento no añade más que T segundos al retardo interactivo. Nagle ha descrito otro algoritmo que produce menos tráfico y es más efectivo para algunas aplicaciones interactivas [Nagle 1984]. El algoritmo de Nagle es utilizado en muchas implementaciones TCP. La mayoría de las implementaciones TCP son configurables, de modo que permiten a la aplicación modificar el valor de T o bien seleccionar un algoritmo de almacenaje de entre los disponibles.

Dada la poca fiabilidad de las redes inalámbricas y la frecuente pérdida de paquetes, estos mecanismos de control del flujo no son especialmente adecuados para las comunicaciones inalámbricas. Ésta es una de las razones para la adopción de un mecanismo de transporte diferente en la familia de protocolos WAP, para la comunicación móvil de área amplia. A pesar de esto, el número de implementaciones de TCP en redes inalámbricas es también importante, y se han propuesto modificaciones al mecanismo TCP para este propósito [Balakrishnan y otros 1995, 1996]. La idea es implementar un componente que soporte TCP en la estación base inalámbrica (la pasarela entre las redes cableada e inalámbrica). El componente TCP fisgará en los segmentos TCP tanto entrantes como salientes de la red inalámbrica, retransmitiendo todo segmento dirigido a la red inalámbrica que no sea reconocido rápidamente por el receptor móvil, y solicitando la retransmisión de los segmentos provenientes de la red inalámbrica cuando se detecten agujeros en los números de secuencia.

Retransmisión: el emisor registra los números de secuencia de los segmentos que envía. Cuando recibe un reconocimiento asume que los segmentos aludidos han sido recibidos satisfactoriamente, por lo que pueden ser borrados de los búferes de salida. Cualquier segmento que no haya sido reconocido en un tiempo límite fijado, será retransmitido por el emisor.

Almacenamiento: el búfer de entrada en el receptor se utiliza para equilibrar el flujo entre el emisor y el receptor. Si el proceso receptor genera operaciones *recibe* más despacio que el emisor genera operaciones *envía*, la cantidad de datos en el búfer puede crecer. La información normalmente se extraerá del búfer antes de que éste se llene, aunque al final el búfer puede

desbordarse y los segmentos entrantes serán desechados, sin registrarse su llegada. Por lo tanto, su llegada no será reconocida y el emisor se verá obligado a retransmitirlos de nuevo.

Suma de comprobación: cada segmento lleva una suma de comprobación (*checksum*) que cubre tanto la cabecera como los datos del segmento. Si un segmento recibido no produce una suma de comprobación igual a la que lleva asociada se desecha.

3.4.7. NOMBRES DE DOMINIO

El diseño y la implementación del sistema de nombres de dominio (*Domain Name System*, DNS) se describirá con detalle en el Capítulo 9; nosotros daremos aquí un breve repaso para completar el tratamiento de los protocolos Internet. Internet soporta un esquema que utiliza nombres simbólicos tanto para referirse a hosts como a redes, tales como *binkley.cs.mcgill.ca* o *essex.ac.uk*. Las entidades nombradas están organizadas según una jerarquía de nombres. Las entidades nombradas se llaman *dominios* y los nombres simbólicos que reciben se llaman *nombres de dominio*. Los dominios están organizados en una jerarquía que pretende reflejar su estructura organizativa. La jerarquía de nombres es totalmente independiente de la disposición física de las redes que forman Internet. Los nombres de dominio se eligen de modo que sean fáciles de manejar por los usuarios humanos, pero deben ser traducidos a las direcciones Internet (IP) correspondientes antes de que puedan ser utilizados como identificadores en las comunicaciones. Esto es responsabilidad de un servicio específico, el DNS. Los programas de aplicación le pasan al DNS peticiones de traducción para que convierta los nombres de dominio en direcciones Internet.

DNS está implementado como un proceso servidor que puede ejecutarse en los computadores host en cualquier sitio de Internet. Existen al menos dos servidores DNS en cada dominio y a menudo son más de dos. Los servidores en cada dominio almacenan un mapa parcial del árbol de nombres de dominio bajo su dominio. Deben almacenar al menos la porción consistente en todos los nombres de dominio y de host dentro de su dominio, aunque con frecuencia contienen una porción mayor del árbol. Los servidores DNS gestionan las peticiones de traducción de nombres de dominio fuera de su porción del árbol enviando peticiones a los servidores DNS en los dominios relevantes, procediendo a resolver los nombres en los segmentos recursivamente de derecha a izquierda. La traducción resultante se almacena entonces en el servidor que originó la petición original para su uso en futuras solicitudes de resolución de nombres en el mismo dominio, de modo que sean tratadas sin involucrar a otros servidores. El DNS no funcionaría sin la utilización intensiva del almacenamiento, ya que debería consultarse los servidores de nombres *raíz* en casi todos los casos, creando un cuello de botella en el acceso al servicio.

3.4.8. CORTAFUEGOS

Casi todas las organizaciones necesitan acceder a Internet para proporcionar servicios a sus clientes y a otros usuarios externos, y para posibilitar que sus usuarios internos pueden acceder a información y a servicios tanto interiores como exteriores. Los computadores en la mayoría de las organizaciones son muy diversos, funcionan bajo sistemas operativos distintos y ejecutan aplicaciones variadas. La seguridad del software es incluso más variable; algunos programas pueden incluir lo último en seguridad mientras que el resto tendrá implementado un nivel de seguridad básico o no tendrá seguridad alguna que aplicar para confiar en las comunicaciones entrantes o para conseguir una comunicación saliente privada, cuando sea necesario. En resumen, en una red interna o *intranet* con muchos computadores conectados y una gran variedad de software instalado es inevitable que algunas partes del sistema tengan alguna debilidad que las expone a ataques a su seguridad. Las formas de ataque se describirán en el Capítulo 7.

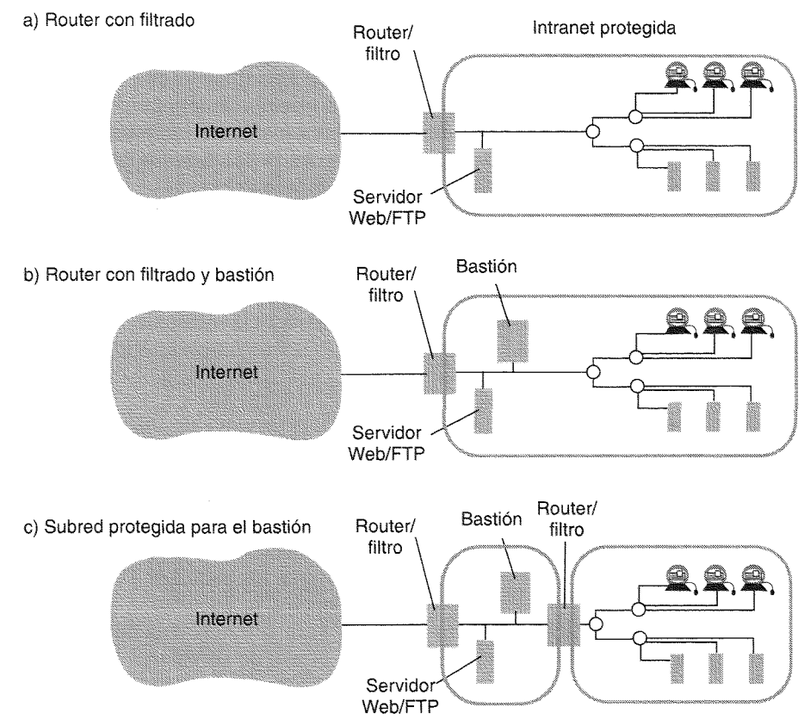


Figura 3.20. Configuraciones de cortafuegos.

El propósito de un cortafuegos es supervisar y controlar todas las comunicaciones entrantes y salientes de una intranet. Un cortafuegos consta de un conjunto de procesos que actúan como una pasarela hacia la intranet (Figura 3.20(a)), aplicando una política de seguridad determinada por la organización.

Los objetivos de la política de seguridad de un cortafuegos incluyen todos o algunos de los enumerados a continuación:

Control de servicios: para determinar qué servicios en los hosts internos son accesibles desde el exterior y para rechazar cualquier otra petición de servicio. También se controlan las peticiones de servicios al exterior y sus respuestas. Estas acciones de filtrado pueden basarse en el contenido de los paquetes IP y en las peticiones TCP y UDP que contienen. Por ejemplo, las peticiones HTTP pueden rechazarse a menos que sean dirigidas al host que albergue el servidor web oficial.

Control del comportamiento: para prevenir comportamientos que infrinjan las políticas de la organización, sean antisociales o no tengan un propósito legítimo y, por lo tanto, sean sospechosos de formar parte de un ataque. Algunas de estas acciones de filtrado pueden ser aplicadas a nivel IP o a nivel TCP, pero otras pueden requerir la interpretación de mensajes al más alto nivel. Por ejemplo, filtrar un ataque de *spam* en el correo electrónico puede necesitar examinar la dirección del remitente en la cabecera del mensaje o incluso el contenido del mismo.

Control de usuarios: la organización puede querer distinguir entre sus usuarios, permitiéndoles algún acceso a servicios exteriores pero inhibiendo a otros hacer lo mismo. Un ejemplo de control de usuarios, que quizás sea más aceptable socialmente que otros, es impedir que se pueda instalar software excepto a aquellos que son miembros del equipo de administradores de sistemas, para prevenir infecciones de virus o para mantener estándares de software. Este ejemplo en concreto puede ser difícil de llevar a cabo en la práctica sin llegar a impedir el uso del Web por los usuarios normales.

Otro ejemplo de control de usuarios es la gestión de los accesos telefónicos y de otras conexiones proporcionadas a los usuarios que se encuentran ausentes. Si el cortafuegos es también el host de las conexiones vía módem, puede autentificar los usuarios en el momento de conectarse y requerir el uso de un canal seguro para las comunicaciones (para prevenir la escucha, y el enmascaramiento y otros ataques en las conexiones externas). Ésta es la función de la tecnología de red privada virtual, *Virtual Private Network* (VPN), descrita en la subsección siguiente.

La política tiene que expresarse en términos de operaciones de filtrado que se llevarán a cabo por los procesos de filtrado que operan en varios niveles diferentes:

Filtrado de paquetes IP: Éste es un proceso de filtrado que examina los paquetes IP individuales. Puede tomar decisiones basándose en las direcciones origen y destino. También puede examinar el campo de *tipo de servicio* de los paquetes IP e interpretar el contenido del paquete basándose en el tipo. Por ejemplo, puede filtrarse los paquetes TCP basándose en el número de puerto al que van dirigidos, y como los servicios se encuentran generalmente en puertos bien conocidos, esto posibilita que los paquetes sean filtrados dependiendo del servicio implicado. Por ejemplo, en muchos lugares se prohíbe el uso de servidores NFS por parte de usuarios externos.

Por razones de prestaciones, el filtrado IP se lleva a cabo por un proceso dentro del núcleo del sistema operativo del router. Si se utilizan varios cortafuegos, el primero de ellos puede marcar ciertos paquetes para un examen posterior más exhaustivo, permitiendo que pasen los paquetes *limpios*. Es posible filtrar secuencias de paquetes IP, por ejemplo para prevenir el acceso a un servidor FTP antes de que se haya validado la entrada en él.

Pasarela TCP: una pasarela TCP comprueba todas las peticiones de conexión TCP y las transmisiones de segmentos. Cuando se ha instalado una pasarela, se pueden controlar todos los establecimientos de conexiones TCP y los segmentos TCP pueden ser analizados para comprobar su corrección (algunos ataques de denegación de servicio utilizan segmentos TCP mal formados para interrumpir los sistemas operativos de los clientes). Siempre que sea preciso, pueden encaminarse a una pasarela del nivel de aplicación para comprobar su contenido.

Pasarela del nivel de aplicación: una pasarela del nivel de aplicación actual como un proxy para un proceso de aplicación. Por ejemplo, se puede desear una política que permita que ciertos usuarios internos establezcan conexiones Telnet a ciertos hosts externos. Cuando un usuario ejecuta el programa Telnet en su computador local, intenta establecer una conexión TCP con un host remoto. La petición es interceptada por la pasarela TCP. La pasarela TCP arranca el proceso denominado *proxy Telnet* al que redirige la conexión TCP original. Si el proxy aprueba la operación Telnet (el usuario está autorizado para utilizar el host solicitado) establece otra conexión con el host solicitado y a partir de ese instante retransmite todos los paquetes TCP en ambas direcciones. Un proceso proxy similar podría ejecutarse en nombre de cada cliente Telnet, y otros procesos proxy equivalentes podrían ser utilizados para FTP y otros servicios.

Un cortafuegos normalmente se compone de varios procesos trabajando en protocolos de diferentes niveles. Es habitual emplear más de un computador en el papel de cortafuegos para conseguir mejores prestaciones y cierta tolerancia a fallos. En todas las configuraciones descritas a continuación e ilustradas en la Figura 3.20 tratamos el caso de un servidor Web/FTP sin protección. Éste solamente alberga información pública que no necesita protección frente al acceso público, y su software de servidor asegura que sólo los usuarios internos autorizados puedan actualizarla.

El filtrado de paquetes IP se realiza normalmente por un router (un computador con al menos dos direcciones de red en redes IP distintas). Éste ejecuta un proceso RIP, un proceso de filtrado de paquetes IP junto con el menor número de procesos más que sea posible. El router/filtro debe ejecutar solamente software fiable de modo que se garantice la aplicación de las políticas de filtrado. Esto implica asegurarse de que ningún proceso del tipo caballo troyano se ejecute en él y que el software de filtrado y encaminamiento no ha sido modificado o falsificado. La Figura 3.20(a) muestra una configuración de cortafuegos simple que confía exclusivamente en el filtrado IP y emplea un único router para ello. La configuración de la red de la Figura 3.20 incluye dos router/filtros que actúan como cortafuegos de este tipo. En esta configuración, existen dos router/filtros por razones de prestaciones y fiabilidad. Ambos obedecen a la misma política y el segundo no incrementa la seguridad del sistema.

Cuando se necesitan pasarelas TCP y del nivel de aplicación, normalmente se ejecutan en un computador separado, conocido como *bastión* (el término es originario de la construcción de castillos fortificados; corresponde a una torre de vigilancia sobresaliente desde la cual el castillo podía ser defendido o desde la cual los defensores podían negociar con aquellos que deseaban entrar). Un computador bastión es un host que está colocado dentro de la intranet protegida por un router/filtro IP y ejecuta las pasarelas TCP y del nivel de aplicación (Figura 3.20(b)). Como el router/filtro, el bastión debe ejecutar exclusivamente software fiable. En una intranet bien protegida se deben usar proxies para acceder a todos los servicios externos. Los lectores estarán familiarizados con la utilización de proxies en los accesos web. Éstos son un ejemplo del uso de proxies de cortafuegos; a menudo están construidos de modo que integran un servidor de caché web (descrito en el Capítulo 2). Éstos y otros proxies requieren bastantes recursos tanto de procesamiento como de almacenamiento.

La seguridad puede mejorarse empleando dos router/filtros en serie, con el bastión y los servidores públicos colocados en una subred separada que enlaza los dos router/filtros (Figura 3.20(c)). Esta configuración tiene varias ventajas respecto a la seguridad:

- Si la política del bastión es estricta, las direcciones IP de los hosts en la intranet no tienen por qué conocerse desde el mundo exterior, y las direcciones exteriores tampoco necesitan ser conocidas por los nodos interiores, ya que todas las comunicaciones externas pasan a través de los proxies del bastión, que tiene acceso a ambas.
- Si el primer router/filtro IP es atacado o comprometido, el segundo, que es invisible desde fuera de la intranet y, por lo tanto, menos vulnerable, sigue extrayendo y rechazando los paquetes IP indeseables.

◇ **Redes privadas virtuales.** La redes privadas virtuales (*virtual private network*, VPN) extienden el límite de protección del cortafuegos más allá de la intranet local utilizando canales seguros protegidos criptográficamente en el nivel IP. En la Sección 3.4.4, presentamos brevemente las extensiones de seguridad IP disponibles para IPv4 e IPv6 con el túnel IPsec [Thayer 1998]. Éstas son las bases de la implementación de VPN. Pueden utilizarse para usuarios externos individuales o para implementar conexiones seguras entre intranets localizadas en diferentes lugares utilizando enlaces Internet públicos.

Por ejemplo, un miembro de la plantilla de una organización puede necesitar conectarse a la intranet de la organización a través de un proveedor de acceso a Internet. Una vez conectado, debería disponer de las mismas posibilidades que cualquier usuario al otro lado del cortafuegos. Esto se puede conseguir si el host local implementa seguridad IP. El host local almacena una o más claves

criptográficas que comparte con el cortafuegos, y que son utilizadas para establecer un canal seguro en el momento de la conexión. Los mecanismos de canal seguro se describen con detalle en el Capítulo 7.

3.5. CASOS DE ESTUDIO: ETHERNET, LAN INALÁMBRICA Y ATM

Hasta aquí hemos discutido los principios involucrados en la construcción de las redes de computadores y hemos descrito IP, la *capa de red virtual* de Internet. Para completar este capítulo, en esta sección describiremos los fundamentos y las implementaciones de tres redes actuales.

En los primeros años ochenta, el instituto norteamericano de ingenieros eléctricos y electrónicos (*US Institute of Electrical and Electronics Engineeres*, IEEE) creó un comité para especificar una serie de estándares de red local (el comité 802 [IEEE 1990]), y sus subcomités han producido una serie de especificaciones que se han convertido en la clave de los estándares de LAN. En la mayoría de los casos, los estándares se basan en estándares industriales previos que habían surgido de las investigaciones realizadas en los años setenta. En la Figura 3.21 se muestran los subcomités relevantes y los estándares que han publicado hasta la fecha.

Ellos difieren en prestaciones, eficiencia, fiabilidad y costo, pero todos proporcionan un ancho de banda relativamente grande sobre distancias cortas y medias. El estándar IEEE 802.3 Ethernet ha ganado de largo la batalla en el mercado de LAN cableadas, y lo describimos en la Sección 3.5.1 como representante de las tecnologías de redes LAN cableadas. Aunque existen implementaciones de Ethernet para varios anchos de banda, el principio de operación es idéntico en todas ellas.

El estándar IEEE 802.5 de Anillo con Paso de Testigo (*Token Ring*) resultó un competidor significativo en gran parte de los años noventa, ofreciendo ventajas sobre Ethernet en eficiencia y en garantías de ancho de banda, pero ahora ha desaparecido del mercado. Los lectores interesados en una breve descripción de esta interesante tecnología de red LAN pueden encontrar una en www.cdk3.net/networking.

El estándar IEEE 802.4 de Bus con Paso de Testigo (*Token Bus*) fue desarrollado para entornos industriales con requisitos de tiempo real y todavía se emplea en esos entornos. El estándar IEEE 802.6 de Área Metropolitana cubre distancias de hasta 50 kilómetros y está pensado para usarse en redes que abarquen ciudades pequeñas y grandes. Descripciones más detalladas y comparaciones de los cuatro estándares IEEE 802.3-802.6 se pueden encontrar en Tanenbaum [1996].

El estándar IEEE 802.11 de LAN Inalámbrica surgió algo tarde pero ahora ocupa una posición significativa en el mercado con productos de Lucent (WaveLAN) y otros vendedores, y se va a convertir en más importante de todos con el advenimiento de los dispositivos de cómputo ubicuos y móviles. El estándar IEEE 802.11 está diseñado para soportar comunicaciones a velocidades de hasta 11 Mbps sobre distancias de hasta 150 metros entre dispositivos equipados con transmisores/receptores inalámbricos simples. Describimos sus principios de funcionamiento en la Sección 3.5.2. Se pueden encontrar más detalles sobre las redes IEEE 802.11 en Crow y otros [1997] y Kurose y Ross [2000].

Núm. IEEE	*Nombre*	*Referencia*
802.3	Redes CSMA/CD (Ethernet)	[IEEE 1985a]
802.4	Redes de Bus con Testigo	[IEEE 1985b]
802.5	Redes de Anillo con Testigo	[IEEE 1985c]
802.6	Redes de Área Metropolitana	[IEEE 1994]
802.11	Redes de Área Local Inalámbricas	[IEEE 1999]

Figura 3.21. Estándares de red IEEE 802.

La tecnología ATM surgió de un gran esfuerzo de estandarización de industrias de telecomunicaciones e informática en el final de los años ochenta y principio de los años noventa [CCITT 1990]. Su propósito es proporcionar una tecnología digital de redes de área amplia con un gran ancho de banda adecuada para aplicaciones de telefonía, datos y multimedia (audio y vídeo de alta calidad). Aunque el despegue ha sido más lento de lo esperado, ATM es ahora la tecnología para las redes de área amplia de muy alta velocidad. También se piensa que puede ser un sustituto de Ethernet en aplicaciones de LAN, pero su éxito en ese mercado ha sido menor debido a la dura competencia con las Ethernet de 100 Mbps y 1.000 Mbps, disponibles a menor costo. Describimos los principios de funcionamiento de ATM en la Sección 3.5.3. En Tanenbaum [1996] y en Stallings [1998a] se pueden encontrar más detalles sobre ATM y sobre otras redes de alta velocidad.

3.5.1. ETHERNET

Ethernet fue desarrollada por el centro de investigación Palo Alto de Xerox en 1973 [Metcalfe y Boggs 1976; Shoch y otros 1982; 1985] como parte del programa de investigación llevado a cabo sobre estaciones de trabajo personales y sistemas distribuidos. La red Ethernet piloto fue la primera red de área local de alta velocidad, demostrando la factibilidad y utilidad de las redes locales de alta velocidad para enlazar computadores en un mismo lugar, permitiéndoles comunicarse a altas velocidades de transmisión con bajas tasas de error y sin retardos de conmutación. El prototipo original Ethernet funcionaba a 3 Mbps. Los sistemas Ethernet están disponibles ahora con anchos de banda en el rango 10 Mbps a 1.000 Mbps. Muchas redes propietarias han sido implementadas utilizando el mismo método básico de operación con características de coste/prestaciones adecuadas para una variedad de aplicaciones. En su forma más barata, los mismos principios de funcionamiento se utilizan para conectar microcomputadores de bajo coste con velocidades de transmisión de 100 a 200 kbps.

Describiremos los principios de funcionamiento de Ethernet 10 Mbps especificados en el estándar IEEE 802.3 [IEEE 1985a]. Ésta fue la primera tecnología de red de área local ampliamente extendida, y probablemente hoy, todavía la tecnología predominante, aunque la variante de 100 Mbps sea ahora la elegida más habitualmente para las instalaciones nuevas. Daremos un repaso a las variaciones en la tecnología de transmisión y en el ancho de banda para redes Ethernet disponibles en la actualidad. Para una descripción comprensible de Ethernet en todas sus variantes, véase Spurgeon [2000].

Una red Ethernet es una línea de conexión en bus simple o ramificado que utiliza un medio de transmisión consistente en uno o más segmentos contiguos de cable enlazados por concentradores o repetidores. Los concentradores y los repetidores son dispositivos sencillos que enlazan trozos de cable, posibilitando que las señales pasen a través suyo. Se pueden enlazar varias Ethernet en el nivel de protocolo de red Ethernet por conmutadores o puentes Ethernet. Los conmutadores y los puentes funcionan al mismo nivel que los marcos Ethernet, encaminándolos a las redes Ethernet adyacentes cuando su destino está allí. Las redes Ethernet enlazadas aparecen como una única red para los protocolos de las capas superiores, tales como IP (véase la Figura 3.10, donde las subredes IP 138.37.88 y 138.37.94 se componen cada una de varias redes Ethernet enlazadas por elementos marcados como *Eswitch*). En concreto, el protocolo ARP (Sección 3.4.2) es capaz de hacer corresponder direcciones IP con direcciones Ethernet sobre un conjunto de redes Ethernet enlazadas; cada solicitud ARP se difunde a todas las redes enlazadas dentro de la subred.

Los modos de funcionamiento de las redes Ethernet son definidos por la frase *acceso múltiple sensible a la portadora, con detección de colisiones* (*carrier sensing, multiple accesss with collision detection*, CSMA/CD) y pertenecen a la clase de redes de contienda en bus. Los métodos de contienda utilizan un medio de transmisión único para enlazar todos los hosts. El protocolo que gestiona el acceso al medio se llama protocolo de control de acceso al medio (*medium access con-*

trol, MAC). Dado que un único enlace conecta todos los hosts, el protocolo MAC combina en una única capa las funciones de un protocolo de enlace de datos (responsable de la transmisión de paquetes sobre los enlaces de comunicación) y de un protocolo de red (responsable de la entrega de los paquetes a los hosts).

◇ **Difusión de paquetes.** El método de comunicación en las redes CSMA/CD es difundir paquetes de datos en el medio de transmisión. Todas las estaciones están escuchando continuamente el medio en busca de paquetes dirigidos a ellas. Cualquier estación que desee transmitir un mensaje difundirá por el medio uno o más paquetes (llamados marcos en la especificación Ethernet). Cada paquete contiene la dirección de la estación destino, la dirección de la estación emisora y una secuencia variable de bits que representa el mensaje que está siendo transmitido. La transmisión de datos se efectúa a 10 Mbps (o a velocidades mayores en las redes Ethernet 100 Mbps y 1.000 Mbps) y la longitud de los paquetes varía entre 46 y 1.518 bytes, por lo que el tiempo de transmisión de un paquete sobre una red Ethernet de 10 Mbps es de 50-200 microsegundos, dependiendo de su longitud. La MTU (unidad máxima de transferencia) está fijada en 1.518 bytes por el estándar IEEE, aunque no existe razón técnica alguna para fijar un límite, excepto la necesidad de limitar los retardos causados por la contienda.

La dirección de la estación destino normalmente se refiere a una interfaz de red única. El controlador del hardware en cada estación recibe una copia de todos los paquetes. Compara la dirección destino en cada paquete con la dirección local grabada físicamente, ignorando los paquetes dirigidos a otras estaciones y pasando aquellos cuya dirección destino coincida con la dirección local. La dirección destino también puede especificar una dirección de difusión o de multidifusión. Las direcciones ordinarias se distinguen de las de difusión y multidifusión por el bit de mayor orden (0 y 1, respectivamente). Una dirección consistente de todos unos está reservada para utilizarse como dirección de difusión y se utiliza cuando un mensaje tiene que ser recibido por todas las estaciones en la red. Esto se utiliza, por ejemplo, para implementar el protocolo ARP de resolución de direcciones IP. Cualquier estación que reciba un paquete con la dirección de difusión la pasará hacia su host local. Una dirección de multidifusión especifica una forma limitada de difusión que será recibida por un grupo de estaciones cuyas interfaces de red han sido configuradas para recibir paquetes con esa dirección de multidifusión. No todas las implementaciones de interfaces de red Ethernet pueden reconocer direcciones de multidifusión.

El protocolo de red Ethernet (que es responsable de la transmisión de paquetes Ethernet entre pares de hosts) está implementado en el hardware de la interfaz Ethernet; se requiere de un protocolo software para la capa de transporte y para las que están por encima de ella.

◇ **Esquema del paquete Ethernet.** Los paquetes (o más correctamente, marcos) transmitidos por las estaciones en Ethernet tienen la siguiente estructura:

bytes: 7	*1*	*6*	*6*	*2*	*46 ⩽ longitud ⩽ 1.500*	*4*
Preámbulo	S	Dirección destino	Dirección origen	Longitud de los datos	Datos a transmitir	Suma de comprobación

Además de las direcciones de origen y de destino ya mencionadas, los marcos incluyen un prefijo de 8 bytes fijo, un campo de longitud, un campo de datos y uno de suma de comprobación. El prefijo se utiliza para la sincronización del hardware y consiste en un preámbulo de 7 bytes, cada uno conteniendo el patrón de bits 10101010 seguido de un único byte de arranque (S en el diagrama) con el patrón 10101011.

A pesar de que la especificación no permite que convivan más de 1.024 estaciones en una única Ethernet, las direcciones ocupan seis bytes, proporcionando 2^{48} diferentes direcciones. Esto hace posible que toda interfaz hardware Ethernet reciba una dirección única de su fabricante, asegurándose que todas las estaciones en cualquier conjunto de Ethernet interconectadas tengan una dirección única. El IEEE actúa como autoridad de reserva para las direcciones Ethernet, reservando rangos de 48 bits a los fabricantes de interfaces hardware.

El campo de datos contiene el mensaje que está siendo transmitido o parte de él (si la longitud del mismo excede de los 1.500 bytes). El límite inferior de 46 bytes en el campo de datos asegura una longitud de paquete mínima de 64 bytes, la cual es necesaria para garantizar que las colisiones serán detectadas en todas las estaciones de la red, según se explica a continuación.

La secuencia de comprobación del marco es una suma de comprobación que genera e inserta el emisor y que sirve para validar los paquetes en el receptor. Los paquetes que producen sumas de comprobación incorrectas son simplemente desechados por la capa de enlace de datos en la estación receptora. Esto es otro ejemplo del argumento de extremo a extremo: para garantizar la transmisión de un mensaje, se debe utilizar un protocolo de la capa de transporte como TCP, con recepción de reconocimientos de cada paquete y retransmisión de cualquier paquete no reconocido. La incidencia de la corrupción de los datos en las redes locales es tan pequeña que el uso de este método de recuperación cuando se requiere garantía en la entrega es enteramente satisfactorio y hace posible que se pueda utilizar un protocolo de transporte menos costoso como UDP cuando no se necesite garantizar la entrega.

◇ **Colisiones de paquetes.** Incluso durante el relativamente corto período de tiempo que lleva la transmisión de un paquete, existe una probabilidad finita de que dos estaciones en la red estén intentando transmitir mensajes de forma simultánea. Si una estación intenta transmitir un paquete sin comprobar que el medio está siendo utilizado por otra estación, puede ocurrir una colisión.

Ethernet tiene tres mecanismos para tratar esta posibilidad. El primero es llamando detección de portadora; la interfaz hardware en cada estación escucha buscando la presencia de una señal en el medio (conocida como portadora por analogía con la difusión de señales de radio). Cuando una estación desea transmitir un paquete, espera hasta que no se detecte señal alguna en el medio y entonces comienza a transmitir.

Desgraciadamente, la detección de la portadora no previene todas las colisiones. Queda la posibilidad de que se produzca una colisión debido al tiempo finito τ que tarda una señal insertada en el medio de transmisión en un punto, en alcanzar todos los demás puntos (la velocidad de transmisión electrónica es aproximadamente 2×10^8 metros por segundo). Considere dos estaciones A y B que están listas para transmitir paquetes casi al mismo tiempo. Si A comienza a transmitir primero, B puede escuchar y no encontrar ninguna señal en el medio en cualquier instante de tiempo $t < \tau$ después de que A haya comenzado la transmisión. Por lo tanto, B comienza a transmitir, *interfiriendo* la transmisión de A. Ambos paquetes, el de A y el de B, se destruirán como consecuencia de la interferencia.

La técnica utilizada para reponerse de tales interferencias se llama *detección de colisiones*. Siempre que una estación está transmitiendo un paquete a través de su puerto de salida hardware, también escucha en su puerto de entrada y compara las dos señales. Si difieren, entonces es que se ha producido una colisión. Cuando esto sucede la estación deja de transmitir y produce una *señal de interferencia* para asegurarse de que todas las estaciones reconocen la colisión. Como ya hemos comentado, se necesita una longitud de paquete mínima para asegurarse de que se detectan todas las colisiones. Si dos estaciones transmiten aproximadamente de forma simultánea desde los extremos de la red, no se darán cuenta de que se ha producido una colisión durante 2τ segundos (ya que el primer emisor debe estar transmitiendo aun cuando reciba la segunda señal). Si los paquetes que transmiten tardan menos de τ en ser difundidos, la colisión no será detectada, ya que cada estación emisora no verá el otro paquete hasta que ha terminado de transmitir el propio, mientras que las estaciones en los puntos intermedios recibirán ambos paquetes simultáneamente, produciendo una corrupción de los datos.

Después de la señal de interferencia, todas las estaciones que están transmitiendo y escuchando cancelan la transmisión del paquete actual. Entonces, las estaciones transmisoras tendrán que intentar transmitir otra vez sus paquetes. Y es ahora cuando aparece una nueva dificultad. Si las estaciones involucradas en la colisión intentan retransmitir sus paquetes inmediatamente después de la señal de interferencia, con toda probabilidad se producirá otra colisión. Para evitar esto se utiliza una técnica llamada *marcha atrás* (*back-off*). Cada una de las estaciones envueltas en una colisión esperará un tiempo $n\tau$ antes de intentar la retransmisión. El valor de n es un entero aleatorio elegido independientemente por cada estación y limitado por una constante L definida en el software de red. Si a pesar de esto se produce otra colisión, se duplica el valor de L y el proceso se repite si es necesario hasta 10 veces.

Finalmente, la interfaz hardware en la estación receptora calcula la secuencia de comprobación y la compara con la suma de comprobación transmitida en el paquete. Utilizando todas estas técnicas, las estaciones conectadas a Ethernet son capaces de gestionar el uso del medio de transmisión sin ningún control centralizado o sincronización.

◇ **Eficiencia de Ethernet.** La eficiencia de una red Ethernet es la relación entre el número de paquetes transmitidos satisfactoriamente y el número máximo teórico de paquetes que podrían haber sido transmitidos sin colisiones. Se ve afectada por el valor τ ya que el intervalo de 2τ segundos que comienza después del envío de un paquete es la *ventana de oportunidad* para las colisiones (no pueden ocurrir colisiones después de los 2τ segundos tras el comienzo de la transmisión de un paquete). También está afectada por el número de estaciones en la red y su nivel de actividad.

Para un cable de 1 kilómetro el valor de τ es menor que 5 microsegundos y la probabilidad de que se produzcan colisiones es suficientemente pequeña para asegurar una eficiencia alta. Ethernet puede conseguir una utilización del canal de entre el 80 y el 95 %, aunque los retardos debidos a la contienda se hacen notar cuando se supera una utilización del 50 %. Dado que la carga es variable, es imposible *garantizar* la entrega de un mensaje dado dentro de un tiempo fijo, ya que la red puede estar completamente cargada cuando el mensaje esté listo para ser transmitido. Pero la *probabilidad* de transferir el mensaje con un retardo dado es tan buena, o mejor que con otras tecnologías de red.

Ciertas medidas empíricas de prestaciones de una Ethernet en el PARC Xerox y registradas por Shoch y Hupp [1980] confirman este análisis. En la práctica, la carga en las redes Ethernet utilizadas en los sistemas distribuidos varían mucho. Muchas redes se utilizan principalmente para interacciones asíncronas cliente-servidor, y la mayoría del tiempo éstas trabajan sin estaciones esperando a transmitir y con una utilización del canal cercana a 1. Las redes que soportan un gran volumen de acceso a datos desde un gran número de usuarios muestran una mayor carga, y aquellas que transmiten caudales multimedia tienen tendencia a ser desbordadas en cuanto se transmiten unos pocos caudales concurrentemente.

◇ **Implementaciones físicas.** La descripción anterior define el protocolo de la capa de acceso al medio para todas las redes Ethernet. La amplia adopción de esta tecnología por parte de un gran mercado ha permitido que se pueda disponer de hardware a muy bajo costo que ejecuta los algoritmos necesarios para su implementación, y que se incluya como parte estándar de muchos computadores de sobremesa y de consumo.

Se pueden encontrar un amplio rango de implementaciones físicas sobre la base de Ethernet que ofrecen distintas prestaciones y relaciones calidad/precio explotando las crecientes prestaciones del hardware. Las variaciones se pueden dar en los diferentes medios de transmisión: cable coaxial, par trenzado (similar al utilizado en telefonía) y fibra óptica, con límites diferentes en el rango de transmisión; y en el uso de velocidades de señal cada vez mayores, obteniendo anchos de banda mayores y, generalmente, menores rangos de transmisión. El IEEE ha adoptado varios estándares para las implementaciones de la capa física, y utiliza un esquema de denominación para

distinguir entre ellos. Se utilizan nombres como 10Base5 y 100BaseT. Éstos tienen la siguiente forma:

$<R><B><L>$ donde: R = tasa de datos en Mbps
B = tipo de medio de transmisión de la señal (banda base o banda ancha)
L = longitud máxima del segmento en metros/100 o T (jerarquía de par trenzado)

Hemos construido una tabla con el ancho de banda y el rango máximo de varias configuraciones estándares y tipos de cable actualmente disponibles.

	10Base5	*10BaseT*	*100BaseT*	*1.000BaseT*
Velocidad de datos	10 Mbps	10 Mbps	100 Mbps	1.000 Mbps
Longitud máx. de los segmentos				
Par trenzado (UTP)	100 m	100 m	100 m	25 m
Cable coaxial (STP)	500 m	500 m	500 m	25 m
Fibra multimodo	2.000 m	2.000 m	500 m	500 m
Fibra monomodo	25.000 m	25.000 m	20.000 m	2.000 m

Las configuraciones que terminan con la designación T están implementadas con cable UTP (par trenzado no apantallado, *Unshielded Twisted Pair*), y están organizadas como una jerarquía de concentradores con computadores en las hojas del árbol. En ese caso, las longitudes de segmento dadas en nuestra tabla son dos veces la distancia máxima permitida desde un computador a un concentrador.

3.5.2. LAN INALÁMBRICA IEEE 802.11

En esta sección resumiremos las características especiales del trabajo con redes sin hilos que se pueden conseguir con la tecnología de red LAN inalámbrica y explicamos cómo las trata el estándar IEEE 802.11. El estándar IEEE 802.11 extiende el principio del acceso múltiple sensible a la portadora (CSMA) utilizado por la tecnología Ethernet (IEEE 802.3) para adecuarse a las características de la comunicación sin hilos. El estándar 802.11 está dirigido a soportar comunicaciones entre computadores separados unos 150 metros entre sí a velocidades de hasta 11 Mbps.

La Figura 3.22 ilustra una porción de una intranet que incluye una LAN inalámbrica. Varios dispositivos móviles inalámbricos se comunican con el resto de la intranet a través de una estación base que es el *punto de acceso* a la red LAN cableada. Una red sin hilos que se conecta al mundo a través de un punto de acceso en una LAN convencional se conoce como una *red de infraestructura*.

Una configuración alternativa para la conexiones de red inalámbricas es conocida como una *red ad hoc* (*al caso*). Las redes *ad hoc* no incluyen un punto de acceso o una estación base. Se construyen *al vuelo* como resultado de la detección mutua de dos o más dispositivos móviles con interfaces inalámbricas en las cercanías. Una red *ad hoc* puede darse, por ejemplo, cuando dos o más usuarios de computadores portátiles establecen una conexión con cualquier estación disponible. Ellos podrían compartir archivos lanzando un proceso de servidor de archivos en cualquiera de las máquinas.

Las estaciones en las redes IEEE 802.11 utilizan como medio de transmisión señales de radio (en la banda de los 2,4 GHz) o señales de infrarrojos. La versión radio del estándar ha centrado la

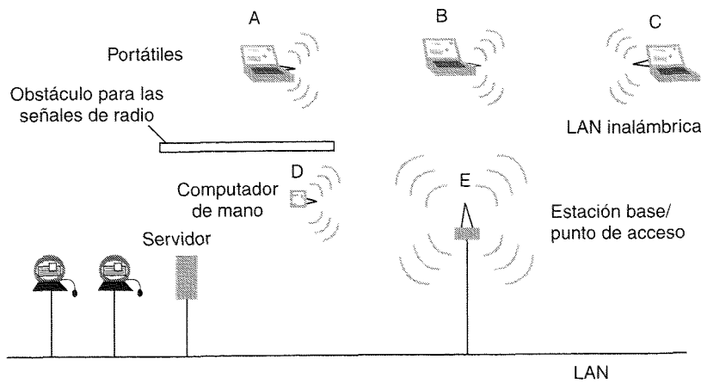


Figura 3.22. Configuración de una LAN inalámbrica.

mayor parte de la atención comercial y será la que describiremos. Utilizan técnicas de selección de frecuencia y de saltos de frecuencia para evitar las interferencias externas y mutuas entre redes LAN inalámbricas independientes, que no detallaremos aquí. Nos centraremos, en cambio, en los cambios que son necesarios realizar al mecanismo CSMA/CD para posibilitar las transmisiones de difusión que serán utilizadas en las transmisiones de radio.

Como Ethernet, el protocolo MAC del 802.11 ofrece iguales oportunidades a todas las estaciones para utilizar el canal de transmisión, y cualquier estación puede transmitir directamente a cualquier otra. Un protocolo MAC controla la utilización del canal por varias estaciones. Como en el caso de Ethernet, la capa MAC también realiza funciones tanto de la capa de enlace de datos como de la capa de red, entregando los paquetes de datos a los hosts en la red.

Cuando utilizamos ondas de radio en lugar de cables como medio de transmisión surgen varios problemas. Estos problemas se derivan del hecho de que los mecanismos de detección de portadora y de colisiones empleados en Ethernet son efectivos sólo cuando la potencia de la señal es aproximadamente la misma a lo largo de toda de la red.

Volvemos a recordar que el propósito de la detección de la portadora es determinar cuándo el medio está libre en todos los puntos entre las estaciones emisoras y receptoras, mientras que la detección de colisiones determina cuándo el medio en los alrededores del receptor está libre de interferencias mientras dura la transmisión. Dado que la potencia de la señal no es uniforme a lo largo del espacio en el que las redes LANs inalámbricas trabajan, la detección de la portadora y de las colisiones pueden fallar de los siguientes modos:

Estaciones ocultas: la detección de la portadora puede fallar en la detección de la transmisión de otra estación. Esto se muestra en la Figura 3.22. Si el computador de mano D está transmitiendo a la estación base E, el portátil A puede que no sea capaz de detectar la señal de D debido a la obstrucción de las señales de radio mostrada. Entonces, A puede comenzar a transmitir, produciendo una colisión en E, a menos que se tomen medidas para prevenirlo.

Atenuación: debido a la ley del inverso del cuadrado de la propagación de las ondas electromagnéticas, la potencia de las señales de radio disminuye rápidamente con la distancia al transmisor. Las estaciones dentro de una LAN inalámbrica pueden encontrarse fuera del alcance de otras estaciones en la misma LAN. Así, en la Figura 3.22, el portátil A puede ser incapaz de detectar una transmisión de C, aunque cada uno de ellos pueda transmitir perfectamente a B o a E. La atenuación imposibilita tanto la detección de la portadora como de las colisiones.

Enmascaramiento de colisiones: desgraciadamente, la técnica de *escuchar* utilizada en Ethernet para detectar las colisiones no es muy efectiva en las redes vía radio. Dada la ley del inverso del cuadrado referida antes, una señal generada localmente será siempre mucho más potente que cualquier señal originada en cualquier otro lugar, y tapará a la transmisión remota. Por eso, si los portátiles A y C transmitieran simultáneamente a E, ninguno de ellos detectaría la colisión, pero E recibiría una transmisión inutilizable.

A pesar de su propensión al fallo, la detección de portadora no se excluye del documento IEEE 802.11; aunque es aumentada por la adición al protocolo MAC del mecanismo de *reserva de ventana*. El esquema resultante se denomina acceso múltiple sensible a la portadora con eliminación de las colisiones (*carrier sensing multiple access with collision avoidance*, CSMA/CA).

Cuando una estación está preparada para transmitir comprueba el medio. Si no detecta ninguna portadora puede suponer que se cumple una de las siguientes condiciones:

1. El medio está disponible.
2. Una estación fuera de alcance está en el proceso de solicitar una ventana.
3. Una estación fuera de alcance está utilizando una ventana que había reservado previamente.

El protocolo de reserva de ventana lleva asociado el intercambio de un par de mensajes cortos (marcos) entre el emisor y el receptor. El primero es un marco de solicitud para enviar (*request to send*, RTS) del emisor al receptor. El mensaje RTS especifica una duración para la ventana solicitada. El receptor responde con un marco libre para enviar (*clear to send*, CTS) repitiendo la duración de la ventana. El efecto de este intercambio es el siguiente:

- Las estaciones dentro del rango del emisor capturarán el marco RTS y tomarán nota de la duración.
- Las estaciones dentro del rango del receptor capturarán el marco CTS y tomarán nota de la duración.

Como resultado, todas las estaciones dentro del rango del emisor y del receptor se abstendrán de transmitir durante la duración de la ventana solicitada, dejando libre el canal para que el emisor transmita un marco de datos de la longitud apropiada. Finalmente, la recepción satisfactoria del marco de datos es reconocida por el receptor para ayudar a tratar el problema de las interferencias externas del canal. Las características aportadas por la reserva de ventana al protocolo MAC ayudan a evitar las colisiones de las siguientes formas:

- Los marcos CTS ayudan a evitar los problemas de atenuación y de ocultación de estaciones.
- Los mensajes RTS y CTS son cortos, por lo que el riesgo de colisión es pequeño. Si se detecta una colisión o un marco RTS no produce otro marco CTS, se utiliza un período aleatorio de marcha atrás, como en Ethernet.
- Cuando los marcos RTS y CTS han sido correctamente intercambiados, no deberían producirse colisiones con los siguientes marcos de datos y de reconocimiento, a menos que una atenuación intermitente hubiera impedido a una tercera parte recibir cualquiera de los primeros.

◇ **Seguridad.** La privacidad y la integridad de las comunicaciones son un tema de interés obvio para las redes sin hilos. Cualquier estación dentro del rango de alcance y equipado con un receptor/transmisor puede encontrar el modo de conectarse a una red, o si esto falla, puede espiar las transmisiones entre otras estaciones. El estándar IEEE 802.11 contempla estos problemas. Se requiere un intercambio de autentificación para cada estación que se conecte a la red mediante el cual se demuestre el conocimiento de una clave compartida. Está basado en un mecanismo de autentificación de clave compartida similar al que se describirá en el Capítulo 7. Resulta efectivo para prevenir que cualquier estación que no tenga acceso a la clave compartida pueda conectarse a la red.

La prevención del espionaje se consigue mediante un esquema de encriptación simple. Éste enmascara el contenido de los flujos de datos transmitidos al combinarlo con una secuencia de números aleatorios utilizando una operación XOR al nivel de bits. La secuencia comienza desde una clave compartida y puede ser reproducida y utilizada para revelar los datos originales por cualquier estación que conozca la clave. Éste es un ejemplo de la técnica de cifrado de flujos descrita en la Sección 7.3.

3.5.3. REDES DE MODO DE TRANSFERENCIA ASÍNCRONO

Las redes ATM se han desarrollado para transportar una gran variedad de datos incluyendo datos multimedia como voz o vídeo. Es una red rápida de conmutación de paquetes basada en un método de encaminamiento de paquetes conocido como retransmisión de celdas, o *cell relay*, que puede funcionar mucho más rápidamente que la conmutación de paquetes tradicional. Consigue más velocidad evitando el control de flujo y la comprobación de errores en los nodos intermedios de una transmisión. Los enlaces y nodos de una transmisión deben presentar, por lo tanto, una probabilidad pequeña de que se produzcan datos corruptos. Otro factor que afecta a las prestaciones es que el tamaño de las unidades de datos transmitidas es pequeño y de tamaño fijo, lo que reduce el tamaño del búfer y el retardo debido a las colas en los nodos intermedios además de su complejidad. ATM trabaja en modo conectado, pero una conexión sólo puede ser establecida si se dispone de los recursos suficientes. Una vez que se ha establecido una conexión, se puede garantizar su calidad (esto es, las características de ancho de banda y de latencia).

ATM es una tecnología de conmutación de datos que puede implementarse sobre las redes de telefonía existentes, siendo éstas síncronas. Cuando ATM se asienta sobre una red de enlaces digitales síncronos de alta velocidad como aquellos especificados para las redes ópticas síncronas, (*Syncrhronous Optical Network*, SONET) [Omidyar y Aldridge 1993], produce una red de paquetes digital de alta velocidad mucho más flexible con muchas conexiones virtuales. Cada conexión virtual ATM proporciona garantía sobre el ancho de banda y sobre la latencia. Los circuitos virtuales resultantes pueden utilizarse para soportar un gran rango de servicios con velocidades variables. Éstos incluyen voz (32 kbps), fax, servicios de sistemas distribuidos, vídeo y televisión de alta definición (100-150 Mbps). El estándar ATM [CCITT 1990] recomienda la provisión de circuitos virtuales con tasas de transferencia de datos de hasta 155 ó 622 Mbps.

Las redes ATM también se pueden implementar en el *modo nativo* directamente sobre fibra óptica, cobre u otro medio de transmisión, permitiendo anchos de banda de hasta varios gigabits por segundo con la tecnología de fibra óptica actual. Éste es el modo en el cual se emplea en las redes de área local y metropolitana.

Los servicios ATM están estructurados en tres capas, representados por los bloques más oscuros de la Figura 3.23. La *capa de adaptación ATM* es una capa extremo a extremo implementada sólo en los hosts emisor y receptor. Se dedica a dar servicio a los protocolos de las capas superiores como TCP/IP y X25 por encima de la capa ATM. Las diferentes versiones de la capa de adaptación pueden proporcionar funciones diferentes de adaptación para ajustarse a las necesidades de los diferentes protocolos de niveles superiores. Incluirán algunas funciones comunes como el ensamblado o desensamblado de paquetes que serán utilizadas por los protocolos de nivel superior.

La *capa ATM* proporciona un servicio orientado a la conexión que transmite paquetes de longitud fija llamados *celdas*. Una conexión consiste en una secuencia de canales virtuales dentro de caminos virtuales. Un canal virtual, *virtual channel* (VC), es una asociación lógica unidireccional entre dos extremos de un enlace en un camino físico desde el origen al destino. Un camino virtual, *virtual path* (VP) es un paquete de canales virtuales que están asociados con un camino físico entre dos nodos conmutadores. Los caminos virtuales están pensados para ser utilizados como soporte de conexiones semipermanentes entre pares de extremos. Los canales virtuales se reservan dinámicamente cuando se establecen las conexiones.

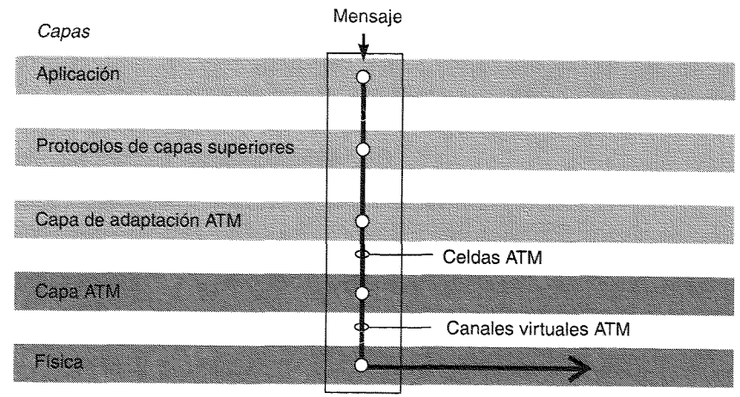


Figura 3.23. Capas del protocolo ATM.

Los nodos en una red ATM pueden jugar tres papeles distintos:

- *Hosts*, que envían y reciben mensajes.
- *Conmutadores de VP*, que almacenan las tablas que muestran la correspondencia entre los caminos entrantes y salientes.
- *Conmutadores VP/VC*, que almacenan tablas similares tanto para los caminos virtuales como para los canales virtuales.

Una celda ATM tiene una cabecera de 5 bytes y un campo de datos de 48 bytes (Figura 3.24). El campo de datos siempre se envía lleno, incluso cuando los datos no ocupan todo el espacio. La cabecera contiene un identificador de un canal virtual y de identificador de camino virtual, que juntos proporcionan la información necesaria para encaminar la celda a través de la red. El identificador de camino virtual se refiere a un camino virtual particular sobre el enlace físico en el que la celda es transmitida. El identificador de canal virtual se refiere a un canal virtual específico dentro del camino virtual. Además se utilizan otros campos en la cabecera para indicar el tipo de celda, la prioridad de pérdida y los límites de la celda.

Cuando una celda ATM llega a un conmutador VP, se utiliza el identificador de camino virtual de la cabecera para buscar en la tabla de encaminamiento el correspondiente camino de salida físico (véase la Figura 3.25). Un conmutador VP/VC puede efectuar un encaminamiento similar basándose tanto en los identificadores VP como VC.

Hay que hacer notar que los identificadores VP y VC se definen localmente. Este esquema tiene la ventaja de que no existe necesidad de identificadores globales dentro de toda la red, lo cual podría implicar el manejo de números muy grandes. Un esquema de direccionamiento global introduciría sobrecargas administrativas e implicaría la utilización de cabeceras y tablas en los conmutadores capaces de almacenar más información.

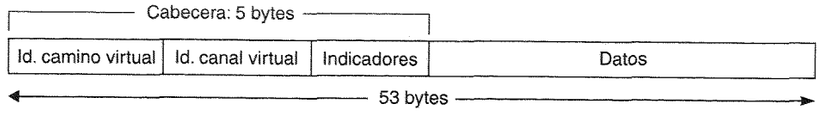


Figura 3.24. Esquema de una celda ATM.

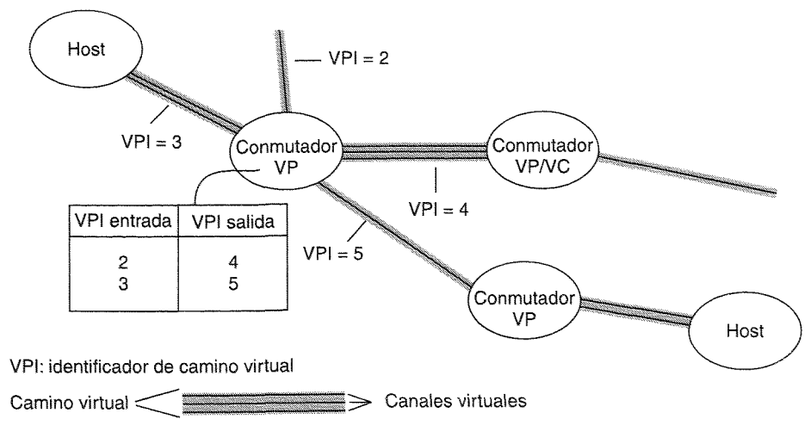


Figura 3.25. Conmutación de caminos virtuales en una red ATM.

ATM proporciona un servicio con latencias pequeñas: el retardo de conmutación es de unos 25 microsegundos por conmutador, dando, por ejemplo, una latencia de 250 microsegundos cuando un mensaje pasa a través de diez conmutadores. Esto se ajusta bien a nuestros requisitos estimados de prestaciones para los sistemas distribuidos (véase la Sección 3.2), sugiriendo que una red ATM podría soportar comunicación entre procesos e interacciones cliente-servidor con unas prestaciones similares a, o mejores que, las que están disponibles ahora en las redes de área local. También se dispone de canales con un ancho de banda muy alto, y con calidad de servicio garantizada que son perfectos para transmitir flujos de datos multimedia a velocidades de hasta 600 Mbps. En las redes ATM puras se pueden alcanzar gigabits por segundo.

3.6. RESUMEN

Nos hemos centrado en los conceptos y en las técnicas de las redes que se necesitan como base para los sistemas distribuidos y nos hemos aproximado a ellos desde el punto de vista de un diseñador de sistemas distribuidos. Las redes de paquetes y los protocolos a capas son la base de las comunicaciones en los sistemas distribuidos. Las redes de área local se basan en la difusión de paquetes en un medio común; y Ethernet es la tecnología dominante. Las redes de área amplia se basan en la conmutación de paquetes para encaminar los paquetes hacia sus destinos a través de la red conectada. El encaminamiento es el mecanismo clave y son varios los algoritmos de encaminamiento utilizados, de los cuales el de vectores de distancia es el más básico pero efectivo. Es necesario un control de la congestión para prevenir el desbordamiento de los búferes en el receptor y en los nodos intermedios.

Las interredes se construyen colocando una capa *virtual* de protocolo interred sobre la colección de redes unidas por los routers. Los protocolos de Internet TCP/IP hacen posible que los computadores en Internet se comuniquen con cualquier otro de una forma uniforme, independientemente de que se encuentren en la misma red de área local o en países diferentes. Los estándares de Internet incluyen varios protocolos del nivel de aplicación que resultan adecuados para su uso en aplicaciones distribuidas a gran escala. IPv6 tiene el espacio de direccionamiento necesario para la

evolución futura de Internet y aporta los medios para satisfacer los requisitos de las nuevas aplicaciones tales como calidad de servicio y seguridad.

Los usuarios móviles tienen soporte de IP móvil en su deambular por áreas amplias, y por las LAN inalámbricas basadas en el IEEE 802.11 para una conectividad local. ATM ofrece un ancho de banda muy alto en comunicaciones asíncronas basadas en circuitos virtuales con calidad de servicio garantizada.

EJERCICIOS

3.1. Un cliente envía un mensaje de solicitud de 200 bytes a un servicio, que produce una respuesta conteniendo 5.000 bytes. Estime el tiempo total necesario para completar la operación en cada uno de los siguientes casos, según las suposiciones de prestaciones enumeradas a continuación:

i) Utilizando una comunicación de datagramas no orientada a conexión (por ejemplo, UDP).
ii) Utilizando una comunicación orientada a conexión (por ejemplo, TCP).
iii) El proceso servidor está en la misma máquina que el cliente.
[Latencia por paquete (local o remoto, tanto en el enviar como en el recibir): 5 ms
Tiempo de establecimiento de conexión (sólo TCP): 5 ms
Tasa de transferencia de datos: 10 Mbps
MTU: 1.000 bytes
Tiempo de procesado de solicitud en el servidor: 2 ms
Supóngase que la red está poco cargada.]

3.2. Internet es demasiado grande para que cualquier router pueda almacenar la información de encaminamiento para todos los nodos. ¿Cómo resuelve el esquema de encaminamiento de Internet este problema?

3.3. ¿Cuál es el cometido de un conmutador Ethernet? ¿Qué tablas mantiene?

3.4. Construya una tabla similar a la de la Figura 3.5 describiendo las tareas realizadas por el software del protocolo en cada capa cuando las aplicaciones Internet y el conjunto de protocolos TCP/IP están implementados sobre una red Ethernet.

3.5. ¿Cómo se ha aplicado el argumento de extremo a extremo [Saltzer y otros 1984] en el diseño de Internet? Considere el modo en que afectará a la viabilidad del *World Wide Web* la utilización de un protocolo de circuitos virtuales en lugar de IP.

3.6. ¿Cómo se puede estar seguro de que no hay dos computadores en Internet con la misma dirección IP?

3.7. Compare las comunicaciones no orientadas a conexión (UDP) y orientadas a conexión (TCP) para implementar cada uno de los siguientes protocolos del nivel de aplicación o del nivel de presentación:

i) Acceso virtual a terminales (por ejemplo, Telnet).
ii) Transferencia de archivos (por ejemplo, FTP).
iii) Localización de usuarios (por ejemplo, rwho, finger).
iv) Mostrar información (por ejemplo HTTP).
v) Invocación de procedimientos remotos.

3.8. Explique cómo es posible que una secuencia de paquetes transmitidos a través de una red de área amplia llegue a su destino en un orden diferente al utilizado en su envío. ¿Por qué esto no puede suceder en una red local? ¿Puede suceder en una red ATM?

3.9. Un problema específico que debe ser resuelto en los protocolos de acceso remoto a terminales como Telnet es la transmisión de eventos excepcionales como señales de finalización (*kill signals*) desde el *terminal* al host adelantándose a datos transmitidos previamente. Las señales de finalización deberían alcanzar su destino antes que cualquier transmisión en marcha. Discuta la solución de este problema con protocolos orientados y no orientados a conexión.

3.10. ¿Cuáles son las desventajas de utilizar la difusión (*broadcast*) a nivel de red para localizar recursos

i) En una única Ethernet?
ii) En una intranet?

¿En qué sentido mejora la multidifusión Ethernet a la difusión?

3.11. Sugiera un esquema que mejore el IP móvil cuando se accede a un servidor web desde un dispositivo móvil que algunas veces está conectado a Internet desde un teléfono móvil y otras veces está conectado por cable en varias localizaciones físicas.

3.12. Muestre la secuencia de cambios que sufrirían las tablas de encaminamiento de la Figura 3.8 (de acuerdo con el algoritmo RIP de la Figura 3.9) después de que se hubiera roto el enlace 3 de la Figura 3.7.

3.13. Utilice el diagrama de la Figura 3.13 como base para ilustrar la segmentación y encapsulación de una solicitud a un servidor HTTP y de la consiguiente respuesta. Suponga que la petición es un mensaje HTTP corto, pero que la respuesta incluye al menos 2.000 bytes de HTML.

3.14. Considere el uso de TCP en un cliente de terminal remoto Telnet. ¿Cómo debería ser almacenada en el cliente la entrada por teclado? Investigue los algoritmos de Nagle y de Clark [Nagle 1984, Clark 1982] para el control de flujo y compárelos con el algoritmo simple descrito en la página 97 cuando TCP es utilizado por (a) un servidor web, (b) una aplicación Telnet, (c) una aplicación gráfica remota con continuas entradas desde el ratón.

3.15. Construya un esquema de red similar al de la Figura 3.10 para la red local de su institución o empresa.

3.16. Describa el modo en que debería configurar un cortafuegos para proteger la red local de su institución o empresa. ¿Qué solicitudes entrantes y salientes debería interceptar?

3.17. ¿Cómo descubre un computador recién instalado y conectado a una red Ethernet la dirección IP de los servidores locales? ¿Cómo traduce esas direcciones a direcciones Ethernet?

3.18. ¿Pueden los cortafuegos prevenir los ataques de denegación de servicio como los descritos en el Epígrafe 3.4.2? ¿Qué otros métodos están disponibles para tratar este tipo de ataques?

4

COMUNICACIÓN ENTRE PROCESOS

4.1. Introducción
4.2. La interfaz de programación de aplicaciones para los protocolos de Internet
4.3. Representación externa de datos y empaquetado
4.4. Comunicación cliente-servidor
4.5. Comunicación en grupo
4.6. Caso de Estudio: comunicación entre procesos en UNIX
4.7. Resumen

Este capítulo está dedicado a estudiar las características de los protocolos para la comunicación entre procesos en un sistema distribuido, ya sea por sí mismos, o como soporte para la comunicación entre objetos.

La interfaz de programación de aplicaciones (API) de Java para la comunicación entre procesos en Internet proporciona comunicación tanto por datagramas como por *streams*. Se tratarán ambas, junto a una discusión sobre sus modelos de fallo. Éstas proveen bloques constructivos alternativos para los protocolos de comunicación.

Se presentarán protocolos para la representación de colecciones de objetos mediante mensajes y referencias a objetos remotos.

También se discutirá la construcción de protocolos que soporten los dos patrones de comunicación más comúnmente utilizados en los programas distribuidos:

- Comunicación cliente-servidor: en la que los mensajes de petición y respuesta proporcionan la base para la invocación remota de métodos o de procedimientos.
- Comunicación de grupo: en la que el mismo mensaje se envía a varios procesos.

Se presentará la comunicación entre procesos UNIX como un caso de estudio.

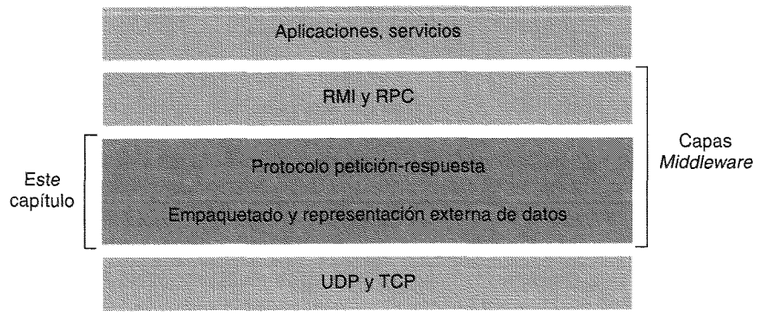


Figura 4.1. Capas de Middleware.

4.1. INTRODUCCIÓN

Tanto este capítulo como el siguiente están relacionados con el *middleware*. Este primer capítulo se centra en el nivel central que se encuentra resaltado en la Figura 4.1. La capa situada sobre ésta, que se estudia en el Capítulo 5, se encargará de integrar la comunicación en un paradigma de lenguaje de programación, proporcionando por ejemplo, la invocación de métodos remotos (*remote method invocation*, RMI) o la llamada de procedimientos remotos (*remote procedure call*, RPC). La *invocación de métodos remotos*, permite que un objeto invoque un método en otro objeto situado en un procedimiento remoto. Ejemplos de sistemas para la invocación remota son CORBA y Java RMI. De un modo similar, una *llamada a un procedimiento remoto*, permite que un cliente ejecute un procedimiento de un servidor remoto.

El Capítulo 3 hablaba de los protocolos de la capa de transporte de Internet, TCP y UDP, sin decir nada sobre cómo el middleware y los programas de aplicación pueden utilizar esos protocolos. La siguiente sección de este capítulo presenta las características de la comunicación entre procesos y discute UDP y TCP desde el punto de vista del programador, presentado la interfaz Java para estos dos protocolos, junto con una discusión de su modelo de fallo. La última sección de este capítulo presenta, como caso de estudio, la interfaz para *sockets* UNIX del tipo UDP y TCP.

La interfaz del programa de aplicación para UDP proporciona un abstracción del tipo *paso de mensajes*, la forma más simple de comunicación entre procesos. Esto hace que el proceso emisor pueda transmitir un mensaje simple al proceso receptor. Los paquetes independientes que contienen estos mensajes se llaman *datagramas*. Tanto en Java como en cada API UNIX, el emisor especifica el destino utilizando un zócalo, conector o *socket* (una referencia indirecta a un puerto particular utilizada por el proceso destino en el computador destino).

La interfaz del programa de aplicación de TCP proporciona la abstracción de un *flujo* (*stream*) de dos direcciones entre pares de procesos. La información intercambiada consiste en un stream de ítems de datos sin límites entre mensajes. Los streams son un bloque básico para la construcción de la comunicación productor-consumidor [Bacon 1998]. Un productor y un consumidor forman un par de procesos en los cuales el papel del primero es producir ítems de datos y el papel del segundo es consumirlos. Los ítems de datos enviados por el productor al consumidor se colocan en una cola a su llegada hasta que el consumidor esté en disposición de recibirlos. El consumidor debe esperar cuando no haya datos disponibles. El productor debe esperar si se llena el almacenamiento utilizado para guardar la cola con los ítems de datos.

La tercera sección de este capítulo trata sobre el modo en que los objetos y las estructuras de datos utilizadas en los programas de aplicación se traducen a una forma adecuada para ser enviada en mensajes a través de la red, teniendo en cuenta el hecho de que computadores diferentes pueden usar diferentes representaciones para los datos simples. También aborda una representación adecuada para las referencias de objetos en un sistema distribuido.

Las secciones cuarta y quinta de este capítulo tratan sobre el diseño de protocolos capaces de soportar la comunicación cliente-servidor y en grupos. Los protocolos petición-respuesta están diseñados para soportar una comunicación cliente-servidor tanto bajo la forma RMI como RPC. Los protocolos de multidifusión en grupos están diseñados para soportar la comunicación entre grupos. La multidifusión en grupos es una forma de comunicación entre procesos en la cual un proceso de un grupo de procesos transmite el mismo mensaje a todos los miembros del mismo grupo.

Las operaciones de paso de mensajes se pueden utilizar para construir protocolos que soporten estructuras particulares de procesos y patrones de comunicación, por ejemplo la invocación de métodos remotos. Examinando los patrones de roles y de comunicación es posible diseñar protocolos de comunicación adecuados basados en intercambios eficientes evitando redundancias. En particular, estos protocolos especializados no deberían incluir acuses de recibo redundantes. Por ejemplo, en las comunicaciones petición-respuesta, en general se considera redundante el reconocimiento de un mensaje de petición, ya que el mensaje de respuesta sirve como acuse de recibo de la petición. Si un protocolo más especializado necesita acusar el recibo al emisor, o cualquier otra característica particular, será dotado de esas operaciones especializadas. La idea es añadir funciones especializadas sólo donde sean necesarias, con el propósito de conseguir protocolos que utilicen un mínimo de intercambios de mensajes.

4.2. API PARA LOS PROTOCOLOS DE INTERNET

En esta sección, estudiaremos las características generales de la comunicación entre procesos para después considerar los protocolos de Internet como un ejemplo de la misma, explicando el modo en que los programadores pueden utilizarlos, ya sea por medio de mensajes UDP o streams TCP.

La Sección 4.2.1 repasa las operaciones *envía* y *recibe* presentadas en la Sección 2.3.2 y discute el modo de sincronización y de direccionamiento en un sistema distribuido. La Sección 4.2.2 presenta los *sockets*, que se utilizan en la interfaz de programación de aplicaciones TCP y UDP. La Sección 4.2.3 trata sobre UDP y su API Java. La Sección 4.2.4 hace lo mismo con TCP y su API Java. Las API Java son orientadas al objeto pero se parecen a aquellas diseñadas originalmente en el sistema operativo UNIX BSD 4.x de Berkeley, que se tratan en la Sección 4.6. Los lectores que estudien los ejemplos de programación de esta sección deberían consultar la documentación Java en línea o a Flanagan [1997] para una especificación completa de las clases involucradas, que están en el paquete *java.net*.

4.2.1. LAS CARACTERÍSTICAS DE LA COMUNICACIÓN ENTRE PROCESOS

El paso de mensajes entre un par de procesos se puede basar en dos operaciones: *envía* y *recibe*, definidas en función del destino y del mensaje. Para que un proceso se pueda comunicar con otro, el proceso envía un mensaje (una secuencia de bytes) a un destino y otro proceso en el destino recibe el mensaje. Esta actividad implica la comunicación de datos desde el proceso emisor al proceso receptor y puede implicar además la sincronización de los dos procesos. La Sección 4.2.3 aporta las definiciones para las operaciones *envía* y *recibe* de la API Java de los protocolos de Internet.

◇ **Comunicación síncrona y asíncrona.** A cada destino de mensajes se asocia una cola. Los procesos emisores producen mensajes que serán añadidos a las colas remotas mientras que los pro-

cesos receptores eliminarán mensajes de las colas locales. La comunicación entre los procesos emisor y receptor puede ser síncrona o asíncrona. En la forma *síncrona*, los procesos receptor y emisor se sincronizan con cada mensaje. En este caso, tanto *envía* como *recibe* son operaciones *bloqueantes*. A cada *envía* producido, el proceso emisor se bloquea hasta que se produce el correspondiente *recibe*. Cuando se invoca un *recibe*, el proceso se bloquea hasta que llega un mensaje.

En la forma de comunicación *asíncrona*, la utilización de la operación *envía* es *no bloqueante*, de modo que el proceso emisor puede continuar tan pronto como el mensaje haya sido copiado en el búfer local, y la transmisión del mensaje se lleva a cabo en paralelo con el proceso emisor. La operación *recibe* puede tener variantes bloqueantes y no bloqueantes. En la variante no bloqueante, el proceso receptor sigue con su programa después de invocar la operación *recibe*, la cual proporciona un búfer que será llenado en un segundo plano, pero el proceso debe ser informado por separado de que su búfer ha sido llenado, ya sea por el método de encuesta o mediante una interrupción.

En un entorno como Java, que soporta múltiples hilos en un único proceso, el *recibe* bloqueante tiene pocas desventajas, puesto que puede ser invocado por un hilo mientras que el resto de hilos del proceso permanecen activos, y la simplicidad de sincronizar los hilos receptores con el mensaje entrante es una ventaja substancial. La comunicación no bloqueante parece ser más eficiente, pero implica una complejidad extra en el proceso receptor asociada con la necesidad de capturar el mensaje entrante fuera de su flujo de control. Por estas razones, los sistemas actuales no proporcionan la forma no bloqueante de *recibe*.

◇ **Destinos de los mensajes.** El Capítulo 3 explicaba que en los protocolos Internet, los mensajes son enviados a direcciones construidas por pares (*dirección Internet, puerto local*). Un puerto local es el destino de un mensaje dentro de un computador, especificado como un número entero. Un puerto tiene exactamente un receptor pero puede tener muchos emisores. Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes. Cualquier proceso que conozca el número de puerto apropiado puede enviarle un mensaje. Generalmente, los servidores hacen públicos sus números de puerto para que sean utilizados por los clientes.

Si el cliente utiliza una dirección Internet fija para referirse a un servicio, entonces ese servicio debe ejecutarse siempre en el mismo computador para que la dirección se considere válida. Esto se puede evitar utilizando alguna de las siguientes aproximaciones que proporcionan transparencia de ubicación:

- Los programas cliente se refieren a los servicios por su nombre y utilizan un servidor de nombres o enlazador (véase la Sección 5.2.5) para trasladar esos nombres a ubicaciones del servidor en tiempo de ejecución. Esto permite reubicar los servicios en otro lugar, pero no migrarlos (cambiarlos de sitio mientras el sistema está en ejecución).
- El sistema operativo, por ejemplo Mach (véase el Capítulo 18), proporciona identificadores para los destinos de los mensajes independientes de la localización, haciéndoles corresponder con direcciones de más bajo nivel utilizadas para entregar los mensajes en los puertos, permitiendo tanto la migración como la reubicación de los servicios en otro lugar.

Una alternativa al uso de los puertos sería la entrega de los mensajes directamente a los procesos, tal y como sucedía en el sistema V [Cheriton 1984]. Sin embargo, los puertos tienen la ventaja de que proporcionan varios puntos de entrada alternativos para un proceso receptor. En algunas aplicaciones, es muy útil ser capaz de entregar el mismo mensaje a los miembros de un conjunto de procesos. Por lo tanto, algunos sistemas IPC proporcionan la capacidad de enviar mensajes a grupos de destinos, ya sean procesos o puertos. Por ejemplo, Chorus [Rozier y otros 1990] proporciona grupos de puertos.

◇ **Fiabilidad.** El Capítulo 2 definía una comunicación fiable en términos de validez e integridad. En lo que concierne a la propiedad de validez, se dice que un servicio de mensajes punto a punto es fiable si se garantiza que los mensajes se entregan a pesar de poder dejar caer o perder un número *razonable* de ellos. Por el contrario, puede decirse que un servicio de mensajes punto a punto es no fiable si no se garantiza la entrega de los mensajes ante la pérdida o eliminación incluso de un solo paquete. Respecto a la integridad, los mensajes deben llegar sin corromperse ni duplicarse.

◇ **Ordenación.** Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión, esto es, en el orden en el que fueron transmitidos por el emisor. La entrega de mensajes desordenados, por esas aplicaciones, es considerada como un fallo.

4.2.2. SOCKETS

Ambas formas de comunicación (UDP y TCP) utilizan la abstracción de *sockets*, que proporciona los puntos extremos de la comunicación entre procesos. Los sockets (conectores) se originan en UNIX BSD aunque están presentes en la mayoría de las versiones de UNIX, incluido Linux y también Windows NT y Macintosh OS. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, según se muestra en la Figura 4.2. Para los procesos receptores de mensajes, su conector debe estar asociado a un puerto local y a una de las direcciones Internet del computador donde se ejecuta. Los mensajes enviados a una dirección de Internet y a un número de puerto concretos, sólo pueden ser recibidos por el proceso cuyo conector esté asociado con esa dirección y con ese puerto. Los procesos pueden utilizar un mismo conector tanto para enviar como para recibir mensajes. Cada computador permite un gran número (2^{16}) de puertos posibles, que pueden ser usados por los procesos locales para recibir mensajes. Cada proceso puede utilizar varios puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos del mismo computador. Los procesos que utilizan multidifusión IP son una excepción ya que si comparten puertos (véase la Sección 4.5.1). No obstante, cualquier cantidad de procesos puede enviar mensajes a un mismo puerto. Cada conector se asocia con un protocolo concreto, que puede ser UDP o TCP.

◇ **API de Java para las direcciones Internet.** Como los paquetes IP que subyacen a TCP y UDP se envian a direcciones Internet, Java proporciona una clase, *InetAddress*, que representa las direcciones Internet. Los usuarios de esta clase se refieren a los computadores por sus nombres de host en el Servicio de Nombres de Dominio (*Domain Name Service*, DNS) (véase la Sección 3.4.7). Por ejemplo, se pueden crear instancias de *InetAddress* que contienen direcciones de Internet invocando a un método estático de *InetAddress* con un nombre DNS de host como argumento. El método utiliza DNS para conseguir la correspondiente dirección Internet. Por ejemplo, para conseguir un objeto que represente la dirección Internet de un host cuyo nombre DNS es *bruno.dcs.qmw.ac.uk*, utilice:

InetAddress unComputador = InetAddress.getByName(«bruno.dcs.qmw.ac.uk»);

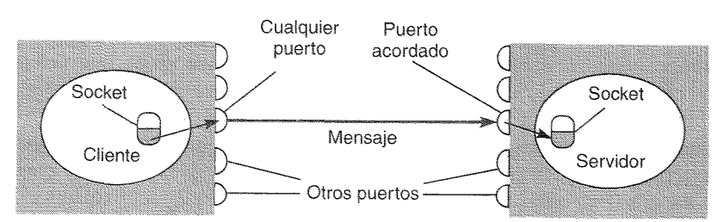


Figura 4.2. Sockets y puertos.

Este método puede lanzar una excepción del tipo *UnknownHostException* (host desconocido). Hay que destacar que el usuario de la clase no necesita dar explícitamente una dirección Internet. De hecho, la clase encapsula los detalles de la representación de las direcciones Internet. Así la interfaz para esta clase no depende del número de bytes utilizado para representar las direcciones Internet (4 bytes en IPv4 y 16 bytes en IPv6).

4.2.3. COMUNICACIÓN DE DATAGRAMAS UDP

Un datagrama enviado por UDP se transmite desde un proceso emisor a un proceso receptor sin acuse de recibo ni reintentos. Si algo falla, el mensaje puede no llegar a su destino. Se transmite un datagrama, entre procesos, cuando uno lo *envía*, y el otro lo *recibe*. Cualquier proceso que necesite enviar o recibir mensajes debe crear, primero, un conector asociado a una dirección Internet y a un puerto local. Un servidor enlazará su conector a un *puerto de servidor* (uno que resulte conocido a los clientes de modo que puedan enviarle mensajes). Un cliente ligará su conector a cualquier puerto local libre. El método *recibe* devolverá, además del mensaje, la dirección Internet y el puerto del emisor, permitiendo al receptor enviar la correspondiente respuesta.

Los párrafos siguientes tratan algunos aspectos referentes a la comunicación de datagramas:

Tamaño del mensaje: el proceso receptor necesita especificar una cadena de bytes de un tamaño concreto sobre la cual se almacenará el mensaje recibido. Si el mensaje es demasiado grande para dicha cadena, será truncado a la llegada. La capa subyacente IP permite paquetes de hasta 2^{16} bytes, incluyendo tanto las cabeceras como el mensaje. Sin embargo, la mayoría de los entornos imponen una restricción a su talla de 8 kilobytes. Cualquier aplicación que necesite mensajes mayores que el máximo debe fragmentarlos en trozos de ese tamaño. Generalmente, una aplicación establecerá un tamaño adecuado para su uso que no resulte excesivamente grande.

Bloqueo: la comunicación de datagramas UDP utiliza operaciones de envío, *envía*, no bloqueantes y recepciones, *recibe*, bloqueantes. La operación *envía* devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP, que son las responsables de su entrega en el destino. A la llegada, el mensaje será colocado en una cola del conector que está enlazado con el puerto de destino. El mensaje podrá obtenerse de la cola de recepción mediante una invocación pendiente o futura del método *recibe* sobre ese conector. Si no existe ningún proceso ligado al conector destino, los mensajes serán descartados.

El método *recibe* produce un bloqueo hasta que se reciba un datagrama, a menos que se haya establecido un tiempo límite (*timeout*) asociado al conector. Si el proceso que invoca el método *recibe* tiene que realizar otras tareas mientras espera el mensaje, debería realizarlas en un hilo separado. Los hilos se explicarán en el Capítulo 6. Por ejemplo, cuando un servidor recibe un mensaje de un cliente, el mensaje puede especificar una serie de tareas a realizar, en cuyo caso el servidor utilizará hilos separados para hacer el trabajo encomendado y para esperar el siguiente mensaje de los demás clientes.

Tiempo límite de espera: el método *recibe* con bloqueo indefinido es adecuado para servidores que están esperando recibir peticiones desde sus clientes. Pero en algunos programas, no resulta apropiado que un proceso que ha invocado una operación *recibe* deba esperar indefinidamente, sobre todo en aquellas situaciones en las que el potencial emisor puede haber caído o se haya podido perder el mensaje esperado. Para tratar estos casos se pueden fijar tiempos límites de espera (*timeouts*) en los conectores. Resulta difícil elegir el intervalo del timeout, aunque debería ser suficientemente grande en comparación con el tiempo adecuado de transmisión de un mensaje.

Recibe de cualquiera: el método *recibe* no especifica el origen de los mensajes. En su lugar, al invocar *recibe* aceptamos mensajes dirigidos a su conector desde cualquier origen. El método *recibe* devuelve la dirección Internet y el puerto del emisor, permitiendo al receptor comprobar de dónde viene el mensaje. Es posible vincular un socket con un puerto y una dirección Internet remotas particulares, en cuyo caso el conector sólo podrá recibir y enviar mensajes con esa dirección.

◇ **Modelo de fallo.** El Capítulo 2 presentaba un modelo de fallo para los canales de comunicación y definía una comunicación fiable en términos de dos propiedades: integridad y validez. La propiedad de integridad requiere que los mensajes no se corrompan ni se dupliquen. La utilización de una suma de comprobación asegura que la probabilidad de que un mensaje esté corrompido sea despreciable. El modelo de fallo puede utilizarse para proponer un modelo de fallo para los datagramas UDP, que padece de las siguientes debilidades:

Fallos de omisión: los mensajes pueden desecharse ocasionalmente, ya sea porque se ha producido un error detectado por la suma de comprobación o porque no queda espacio en el búfer ya sea en el origen o en el destino. Para simplificar muestra discusión, incluiremos los fallos por omisión en el emisor y los fallos por omisión (véase la Figura 2.11) en el receptor como fallos de omisión en el canal de comunicaciones.

Ordenación: algunas veces, los mensajes se entregan en desorden con respecto a su orden de emisión.

Las aplicaciones que utilizan datagramas UDP dependen de sus propias comprobaciones para conseguir la calidad que necesitan respecto a la fiabilidad de la comunicación. Puede construirse un servicio de entrega fiable a partir de uno que adolece de fallos de omisión mediante la utilización de acuses de recibo. La Sección 4.4 discute el modo en que se pueden construir protocolos de petición-respuesta fiables para comunicaciones cliente-servidor sobre UDP.

◇ **Utilización de UDP.** Para algunas aplicaciones, resulta aceptable utilizar un servicio que sea susceptible de sufrir fallos de omisión ocasionales. Por ejemplo, el Servicio de Nombres de Dominio en Internet (*Domain Name Service*, DNS) está implementado sobre UDP. Los datagramas UDP son, en algunas ocasiones, una elección atractiva porque no padecen las sobrecargas asociadas a la entrega de mensajes garantizada. Existen tres fuentes principales para esa sobrecarga:

1. La necesidad de almacenar información de estado en el origen y en el destino.
2. La transmisión de mensajes extra.
3. La latencia para el emisor.

Las razones de estas sobrecargas se explicarán en la Sección 4.2.4.

◇ **API Java para datagramas UDP.** La API Java proporciona una comunicación de datagramas por medio de dos clases: *DatagramPacket* y *DatagramSocket*.

DatagramPacket: esta clase proporciona un constructor que crea una instancia compuesta por una cadena de bytes que almacena el mensaje, la longitud del mensaje y la dirección Internet y el número de puerto local del conector destino, tal y como sigue:

Paquete del datagrama

cadena de bytes conteniendo el mensaje	longitud del mensaje	dirección Internet	número de puerto

Las instancias de *DatagramPacket* podrán ser transmitidas entre procesos cuando uno las *envía*, y el otro las *recibe*.

Esta clase proporciona otro constructor para cuando se recibe un mensaje. Sus argumentos especifican la cadena de bytes en la que alojar el mensaje y la longitud de la misma. Un mensaje recibido se coloca en un *DatagramPacket* junto a su longitud y la dirección Internet y el número del puerto del conector emisor. El mensaje puede recuperarse del *DatagramPacket* mediante el método *getData*. Los métodos *getPort* y *getAddress* acceden, respectivamente, al puerto y a la dirección Internet.

DatagramSocket: esta clase maneja conectores para enviar y recibir datagramas UDP. Proporciona un constructor que toma un número de puerto como argumento, apropiado para los procesos que necesitan utilizar un puerto concreto. También proporciona un constructor sin argumentos que permite que el sistema elija un puerto de entre los que estén libres. Estos constructores pueden lanzar una excepción *SocketException* si el puerto ya está siendo usado o si se especifica un puerto reservado (por debajo del 1.024) cuando se ejecuta sobre UNIX.

La clase *DatagramSocket* proporciona varios métodos que incluyen los siguientes:

send y *receive*: estos métodos sirven para transmitir datagramas entre un par de conectores. El argumento de *send* es una instancia de *DatagramPacket* conteniendo el mensaje y su destino. El argumento de *receive* es un *DatagramPacket* vacío en el que colocar el mensaje, su longitud y su origen. Tanto el método *send* como *receive* pueden lanzar una excepción *IOException*.

setSoTimeout: este método permite establecer un tiempo de espera límite. Cuando se fija un límite, el método *receive* se bloquea durante el tiempo fijado y después lanza una excepción *InterruptedIOException*.

connect: este método se utiliza para conectarse a un puerto remoto y a una dirección Internet concretos, en cuyo caso el conector sólo podrá enviar y recibir mensajes de esa dirección.

La Figura 4.3 muestra el programa de un cliente que crea un conector, envía un mensaje a un servidor en el puerto 6.789 y después espera una respuesta. Los argumentos para el método *main* proporcionan un mensaje y el nombre DNS de host del servidor. El mensaje se convierte a una cadena

```
import java.net.*;
import java.io.*;
public class ClienteUDP{
  public static void main(String args[ ]){
    // args proporciona el mensaje y el nombre del servidor
    try {
        DatagramSocket unSocket = new DatagramSocket( );
        byte [ ] m = args[0].getBytes( );
        InetAddress unHost = InetAddress.getByName(args[1]);
        int puertoServidor = 6789;
        DatagramPacket peticion =
                new DatagramPacket(m, args[0].length( ), unHost, puertoServidor);
        unSocket.send(peticion);
        byte [ ] bufer = new byte[1000];
        DatagramPacket respuesta = new DatagramPacket(bufer, bufer.length);
        unSocket.receive(respuesta);
        System.out.println("Respuesta: " + new String(respuesta.getData( )));
        unSocket.close( );
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
    }catch (IOException e){System.out.println("IO: " + e.getMessage( ));}
  }
}
```

Figura 4.3. Un cliente UDP enviando un mensaje a un servidor y recogiendo su respuesta.


```
import java.net.*;
import java.io.*;
public class ServidorUDP{
  public static void main(String args[ ]){
    try {
        DatagramSocket unSocket = new DatagramSocket(6789);
        byte [ ] bufer = new byte[1000];
        while (true){
          DatagramPacket peticion = new DatagramPacket(bufer, bufer.length);
          unSocket.receive(peticion);
          DatagramPacket respuesta = new DatagramPacket(peticion.getData( ),
            peticion.getLength( ), peticion.getAddress( ), peticion.getPort( ));
          unSocket.send(respuesta);
        }
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
    }catch (IOException e) {System.out.println("IO: " + e.getMessage( ));}
  }
}
```

Figura 4.4. Un servidor UDP recibe peticiones y las devuelve al cliente de forma repetitiva.

de bytes, y el nombre DNS del host a la correspondiente dirección Internet. La Figura 4.4 muestra el programa para el correspondiente servidor, el cual crea un conector ligado a su puerto de servidor (6.789) y entonces espera repetidamente a los mensajes de petición de los clientes, a los cuales responde mandando de vuelta el mismo mensaje.

4.2.4. COMUNICACIÓN DE STREAMS TCP

La API para el protocolo TCP, que es originaria de UNIX BSD 4.x, proporciona la abstracción de un flujo de bytes (*stream*) en el que pueden escribirse y desde el que pueden leerse datos. La abstracción de *stream* oculta las siguientes características de la red:

Tamaño de los mensajes: la aplicación puede elegir la cantidad de datos que quiere escribir o leer del stream. El conjunto de datos puede ser muy pequeño o muy grande. La implementación del flujo TCP subyacente decide cuántos datos recoge antes de transmitirlos como uno o más paquetes IP. En el destino, los datos son proporcionados a la aplicación según los va solicitando. Las aplicaciones pueden forzar, si fuera necesario, que los datos sean enviados de forma inmediata.

Mensajes perdidos: el protocolo TCP utiliza un esquema de acuse de recibo de los mensajes. Como un ejemplo de un esquema simple (no utilizado por TCP), el extremo emisor almacena un registro de cada paquete IP enviado y el extremo receptor acusa el recibo de todos los paquetes IP que le llegan. Si el emisor no recibe dicho acuse de recibo dentro de un plazo de tiempo fijado, volverá a transmitir el mensaje. El esquema de desplazamiento de ventana [Comer 1991], más sofisticado, reduce drásticamente el número de mensajes de reconocimiento necesarios.

Control del flujo: el protocolo TCP intenta ajustar las velocidades de los procesos que leen y escriben en un stream. Si el escritor es demasiado rápido para el lector, entonces será bloqueado hasta que el lector haya consumido una cantidad suficiente de datos.

Duplicación y ordenación de los mensajes: a cada paquete IP se le asocia un identificador, que hace posible que el receptor pueda detectar y rechazar mensajes duplicados, o que pueda reordenar los mensajes que lleguen desordenados.

Destinos de los mensajes: un par de procesos en comunicación establecen una conexión antes de que puedan comunicarse mediante un stream. Una vez que se ha establecido la comunicación, los procesos simplemente leen o escriben en el stream sin tener que preocuparse de las direcciones Internet ni de los números de puerto. El establecimiento de la conexión implica una petición de conexión, *connect*, desde el cliente al servidor, seguida de una aceptación, *accept*, desde el servidor al cliente antes de que cualquier comunicación pueda tener lugar. Esto puede suponer una sobrecarga considerable para una única petición y una única respuesta.

El API para la comunicación por streams supone que en el momento de establecer una conexión, uno de ellos juega el papel de cliente y el otro juega el papel de servidor, aunque después se comuniquen de igual a igual. El rol de cliente implica la creación de un conector, de tipo stream, sobre cualquier puerto y la posterior petición de conexión con el servidor en su puerto de servicio. El papel del servidor involucra la creación de un conector de escucha ligado al puerto de servicio y la espera de clientes que soliciten conexiones. El conector para escuchar mantiene una cola de peticiones de conexión. En el modelo de sockets, cuando un servidor acepta una conexión, crea un nuevo conector para mantener la comunicación con el cliente, mientras que se reserva el conector del puerto de servicio para escuchar las peticiones de conexión de otros clientes. En el final de este capítulo, en el caso de estudio UNIX se pueden encontrar más detalles sobre las operaciones mencionadas *connect* y *accept*.

El par de conectores, el del cliente y el del servidor, se conectan por un par de streams, uno en cada dirección. Así, cada conector tiene su propio stream de entrada y de salida. Uno de los procesos del par puede enviar información al otro escribiendo en su stream de salida, y el otro puede obtener la información leyendo de su stream de entrada.

Cuando una aplicación cierra un conector, indica que no va a escribir nada más en su stream de salida. Cualquier dato que se encuentre en su búfer de salida será enviado al otro extremo del stream y puesto en la cola del conector destino con una indicación de que el stream ha sido roto. El proceso en el destino puede leer los datos en la cola, pero cualquier lectura después de que la cola esté vacía resultará en una indicación de final de stream. Cuando un proceso finaliza su ejecución o falla, todos sus conectores se cierran y cualquier proceso que intente comunicarse con él descubrirá que la conexión se ha roto.

Los siguientes párrafos discuten aspectos importantes relacionados con la comunicación de streams:

Concordancia de ítems de datos: los dos procesos que se comunican necesitan estar de acuerdo en el tipo de datos transmitidos por el stream. Por ejemplo, si un proceso escribe un *int* seguido de un *double*, el proceso receptor debe interpretarlo como un *int* seguido de un *double*. Cuando un par de procesos no coopera correctamente en el uso del stream, el proceso lector puede encontrarse con problemas cuando interprete los datos o puede bloquearse debido a que se encuentre una cantidad insuficiente de datos en el stream.

Bloqueo: los datos escritos en un stream se almacenan en un búfer en el conector destino. Cuando un proceso intenta leer datos de un canal de entrada, o bien extraerá los datos de la cola o bien se bloqueará hasta que existan datos disponibles. El proceso que escribe los datos en el stream resultará bloqueado por el mecanismo de control de stream de TCP si el conector del otro lado intenta almacenar en la cola de entrada más información de la permitida.

Hilos: cuando un servidor acepta una conexión, generalmente crea un nuevo hilo con el que comunicarse con el nuevo cliente. La ventaja de utilizar un hilo separado para cada cliente es que el servidor puede bloquearse a la espera de entradas sin afectar a los otros clientes. En un entorno en el cual no se disponga de hilos, una alternativa es comprobar si existen datos accesibles en el stream antes de intentar leerlos; por ejemplo, en un entorno UNIX se puede utilizar para ello la llamada al sistema *select*.

◇ **Modelo de fallo.** Para satisfacer la propiedad de integridad de una comunicación fiable, los streams TCP utilizan una suma de comprobación para detectar y rechazar los paquetes corruptos, y utilizan un número de secuencia para detectar y eliminar los paquetes duplicados. Con respecto a la propiedad de validez, los streams TCP utilizan *timeouts* y retransmisión de los paquetes perdidos. Por lo tanto, los mensajes tienen garantizada su entrega incluso cuando alguno de los paquetes subyacentes se haya perdido.

Pero si la pérdida de paquetes sobrepasa un cierto límite, o la red que conecta un par de procesos en comunicación está severamente congestionada, el software TCP responsable de enviar los mensajes no recibirá acuses de recibo de los paquetes enviados y después de un tiempo declarará rota la conexión. Por esto TCP no proporciona comunicación fiable, ya que no garantiza la entrega de mensajes en presencia de algún tipo de problema.

Cuando una conexión está rota, se notificará al proceso que la utiliza siempre que intente leer o escribir. Esto tiene los siguientes efectos:

- Los procesos que utilizan la conexión no distinguen entre un fallo en la red y un fallo en el proceso que está en el otro extremo de la conexión.
- Los procesos comunicantes no pueden saber si sus mensajes recientes han sido recibidos o no.

◇ **Utilización de TCP.** Muchos de los servicios utilizados se ejecutan sobre conexiones TCP, con números de puerto reservados. Entre ellos se encuentran los siguientes:

HTTP: El protocolo de transferencia de hipertexto se utiliza en comunicación entre un navegador y un servidor web; se tratará de él más tarde en este capítulo.

FTP: El protocolo de transferencia de archivos permite leer los directorios de un computador remoto y transferir archivos entre los computadores de una conexión.

Telnet: La herramienta Telnet proporciona acceso a un terminal en un computador remoto.

SMTP: El protocolo simple de transferencia de correo se utiliza para mandar correos electrónicos entre computadores.

◇ **El API Java para los streams TCP.** La interfaz Java para los streams TCP está constituida por las clases *ServerSocket* y *Socket*.

ServerSocket: esta clase está diseñada para ser utilizada por un servidor para crear un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes. Su método *accept* toma una petición *connect* de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar *accept* es una instancia de *Socket*, un conector que da acceso a streams para comunicarse con el cliente.

Socket: esta clase es utilizada por el par de procesos de una conexión. El cliente utiliza un constructor para crear un conector, especificando el nombre DNS de host y el puerto del servidor. Este constructor no sólo crea el conector asociado con el puerto local, sino que también se conecta con el computador remoto especificado en el puerto indicado. Puede lanzar una excepción *UnknownHostException* si el nombre de host no es correcto, o una excepción *IOException* si se da un error de entrada y salida.

La clase *Socket* proporciona los métodos *getInputStream* y *getOutputStream* para acceder a los dos streams asociados con un conector. El tipo de datos devueltos por esos métodos son *InputStream* y *OutputStream*, respectivamente (clases abstractas que definen métodos para leer y escribir bytes). Los valores de retorno se pueden utilizar como argumentos de constructores para gestionar adecuadamente los streams de entrada y de salida del programa. Nuestro ejemplo utiliza *DataInputStream* y *DataOutputStream*, los cuales permiten utilizar representaciones binarias de los tipos primitivos de datos para ser leídos y escritos de una forma independiente de la máquina.

```
import java.net.*;
import java.io.*;
public class ClienteTCP{
  public static void main(String args[ ]){
  // los argumentos proporcionan el mensaje y el nombre del host destino
    try{
      int puertoServicio= 7896;
      Socket s = new Socket(args[1], puertoServicio);
      DataInputStream entrada = new DataInputStream(s.getInputStream());
      DataOutputStream salida =
              new DataOutputStream(s.getOutputStream( ));
      salida.writeUTF(args[0]); //UTF es una codificación de Strings, ir a Sec. 4.3
      String datos = entrada.readUTF( );
      System.out.println("Recibido: "+ datos);
      s.close( );
    }catch (UnknownHostException e)
          System.out.println("Socket: "+ e.getMessage( ));
    }catch (EOFException e){System.out.println("EOF: "+ e.getMessage( ));
    }catch (IOException e){System.out.println("IO: "+ e.getMessage( ));}
  }
}
```

Figura 4.5. Un cliente TCP realiza una conexión a un servidor, envía una petición y recibe una respuesta.

La Figura 4.5 muestra un programa cliente donde se le da como argumento al método *main* un mensaje y el nombre DNS del host del servidor. El cliente crea un conector ligado al nombre del host y al puerto 7896. Obtiene *DataInputStream* y *DataOutputStream* de los streams de los conectores de entrada y de salida respectivamente, y entonces escribe el mensaje en su stream de salida y espera a leer la respuesta en el stream de entrada. El programa del servidor de la Figura 4.6 abre un conector de servidor en su puerto de servicio (7896) y escucha las peticiones de conexión, *connect*. Cuando llega una, crea un nuevo hilo para comunicarse con el cliente. El nuevo hilo crea un *DataInputStream* y un *DataOutputStream* para los flujos de entrada y salida de su conector, y entonces espera a leer el mensaje del cliente y lo escribe de vuelta.

Como nuestro mensaje está compuesto por una cadena de caracteres, el cliente y el servidor utilizan el método *writeUTF* de *DataOutputStream* para escribirlo en el stream de salida, y el método *readUTF* de *DataInputStream* para leerlo del stream de entrada. UTF es un código que representa cadenas en un formato particular, descrito en la Sección 4.3.

Cuando un proceso ha cerrado su conector, no le será posible utilizar sus streams de entrada y de salida. El proceso al que le ha enviado datos puede leerlos de su cola, pero cualquier lectura después de que la cola esté vacía resultará en una excepción *EOFException*. Los intentos de utilizar un conector cerrado o de escribir en un stream roto tienen como resultado una excepción *IOException*.

4.3. REPRESENTACIÓN EXTERNA DE DATOS Y EMPAQUETADO

La información almacenada dentro de los programas en ejecución se representa mediante estructuras de datos (por ejemplo, por conjuntos de objetos interrelacionados) mientras que la información transportada en los mensajes consiste en secuencias de bytes. Independientemente de la forma de comunicación utilizada, las estructuras de datos deben ser *aplanadas* (convertidas a una secuencia


```
import java.net.*;
import java.io.*;
public class ServidorTCP{
  public static void main(String args[]){
    try{
        int puertoServicio= 7896;
        ServerSocket escuchandoSocket = new ServerSocket(puertoServicio);
        while (true) {
            Socket socketCliente = escuchandoSocket.accept( );
            Conexion c = new Conexion(socketCliente);
        }
    }catch (IOException e) {System.out.println(«Escuchando: »+ e.getMessage( ));}
  }
}
class Conexion extends Thread {
    DataInputStream entrada;
    DataOutputStream salida;
    Socket socketCliente;
    public Conexion (Socket unSocketCliente) {
      try {
        socketCliente = unSocketCliente;
        entrada = new DataInputStream(socketCliente.getInputStream( ));
        salida = new DataOutputStream(socketCliente.getOutputStream());
        this.start( );
      }catch(IOException e)
          {System.out.println("Conexión :"+ e.getMessage());}
    }
    public void run(){
      try {          // un servidor eco
        String datos = entrada.readUTF( );
        salida.writeUTF(datos);
        socketCliente.close( );
      }catch (EOFException e) {System.out.println("EOF: "+ e.getMessage( ));
      }catch (IOException e) {System.out.println("IO: "+ e.getMessage( ));}
    }
}
```

Figura 4.6. Un servidor TCP establece una conexión para cada cliente y les reenvía las peticiones.

de bytes) antes de su transmisión y reconstruidas en el destino. Los tipos de datos primitivos transmitidos en los mensajes pueden tener valores de muchos tipos distintos, y no todos los computadores almacenan los valores primitivos, tales como los enteros, en el mismo orden. También es diferente en diferentes arquitecturas la representación de los números en coma flotante. Otro problema es el conjunto de códigos utilizado para representar los caracteres: por ejemplo, UNIX utiliza la codificación ASCII con un byte por carácter, mientras que el estándar Unicode permite representar textos en la mayoría de los lenguajes y utiliza dos bytes por carácter. Existen dos variantes en la ordenación de enteros: la llamada *big-endian*, en la que el byte más significativo va el primero, y la llamada *little-endian*, en la que va el último.

Para hacer posible que dos computadores puedan intercambiar datos se puede utilizar uno de los dos métodos siguientes:

- Los valores se convierten a un formato externo acordado antes de la transmisión y se revierten al formato local en la recepción; si los dos computadores son del mismo tipo y lo saben, se puede omitir la transformación al formato externo.

- Los valores se transmiten según el formato del emisor, junto con una indicación del formato utilizado, y el receptor los convierte si es necesario.

Hay que hacer notar, sin embargo, que los bytes no son alterados durante la transmisión. Para soportar RMI o RPC, cualquier tipo de dato que pueda ser pasado como un argumento o devuelto como resultado debe ser capaz de ser aplanado y cada uno de los tipos de datos primitivos representados en una representación de datos acordada. Al estándar acordado para la representación de estructuras de datos y valores primitivos se denomina *representación externa de datos*.

El *empaquetado* (*marshalling*) consiste en tomar una colección de ítems de datos y ensamblarlos de un modo adecuado para la transmisión en un mensaje. El *desempaquetado* (*unmarshalling*) es el proceso de desensamblado en el destino para producir una colección equivalente de datos. Por lo tanto, empaquetar consiste en traducir las estructuras de datos y los valores primitivos en una representación externa de datos. Similarmente, desempaquetar consiste en generar los valores primitivos desde la representación de datos externa y reconstruir las estructuras de datos.

Se tratarán las dos alternativas para la representación externa de datos y el empaquetado:

- La representación común de datos de CORBA, que incumbe a una representación externa para los datos estructurados y primitivos que pueda ser pasada como argumento y resultado de la invocación remota de métodos en CORBA. Puede ser utilizada por una gran variedad de lenguajes de programación (véase el Capítulo 17).
- La serialización de objetos Java, que está relacionada con el aplanado y la representación externa de datos de cualquier objeto simple o un árbol de objetos que tienen que ser transmitidos en un mensaje o almacenados en disco. Es de uso exclusivo de Java.

En ambos casos, el empaquetado y el desempaquetado son llevados a cabo por una capa de middleware sin ninguna participación del programador de la aplicación. Debido a que el empaquetado requiere la consideración de todos los detalles de bajo nivel de la representación de los componentes primitivos de los objetos compuestos, el proceso es propenso a fallar cuando se hace a mano. La eficiencia es otro criterio que resulta deseable en el diseño de los procedimientos de empaquetado generados de forma automática.

Las dos aproximaciones tratadas aquí empaquetan los tipos de datos primitivos de forma binaria. Una alternativa es empaquetar todos los objetos a transmitir en texto ASCII, lo cual es relativamente simple de implementar, pero implica un empaquetado generalmente más grande. El protocolo HTTP, descrito en la Sección 4.4, es un ejemplo de esta última aproximación.

Aunque estamos interesados en el uso de empaquetado para los argumentos y para los resultados de los RMI y RPC, resulta que tienen un uso más general en la conversión de las estructuras de datos u objetos en una forma adecuada para la transmisión en mensajes o para su almacenamiento en archivos.

4.3.1. REPRESENTACIÓN COMÚN DE DATOS DE CORBA (CDR)

CORBA CDR es la representación externa de datos definida en CORBA 2.0 [OMG 1998a]. CDR puede representar todos los tipos de datos que se pueden utilizar como argumentos o como resultados en las invocaciones remotas de CORBA. Consta de 15 tipos primitivos, que incluyen los tipos *short* (16-bit), *long* (32-bit), *unsigned short* (sin signo), *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), y *any* (el cual puede representar cualquier tipo básico o compuesto); junto con un abanico de tipos compuestos, que se describen en la Figura 4.7. Cada argumento o resultado en una invocación remota se representa como una secuencia de bytes en el mensaje de la invocación o en el del resultado.

Tipos primitivos: CDR define una representación tanto para el orden big-endian como para el little-endian. Los valores se transmiten en el orden del emisor, que se especifica en cada mensaje. El receptor lo traduce si es necesario a un orden diferente. Por ejemplo, un *short* de 16-bit ocupa dos bytes en el mensaje, y en el orden big-endian los bits más significativos ocupan el primer byte y los menos significativos el segundo. Cada valor primitivo se coloca en una posición en la secuencia de bytes de acuerdo con su tamaño. Supóngase que se indexa la secuencia de bytes desde cero. Entonces un valor primitivo cuyo tamaño son n bytes (donde $n = 1, 2, 4$ u 8) será añadido a la secuencia en una posición que sea un múltiplo de n en el flujo de bytes. Los valores de coma flotante siguen el estándar IEEE, en el cual el signo, el exponente y la parte fraccionaria están en los bytes 0-n para el orden big-endian y en el orden inverso para el little-endian. Los caracteres se representan por un conjunto de caracteres acordado entre el cliente y el servidor.

Tipos compuestos: los valores primitivos que componen cada tipo compuesto se añaden a la secuencia en un orden particular, según se muestra en la Figura 4.7.

Tipo	*Representación*
sequence	longitud (unsigned long —entero largo sin signo—) seguida de los elementos en orden
string	longitud (unsigned long) seguida de los caracteres en orden (también puede tener caracteres anchos —2 bytes—)
array	elementos de la cadena en orden (no se especifica la longitud porque es fija)
struct	en el orden de declaración de los componentes
enumerated	unsigned long (los valores son especificados por el orden declarado)
union	etiqueta de tipo seguida por el miembro seleccionado

Figura 4.7. Tipos compuestos CDR de CORBA.

La Figura 4.8 muestra un mensaje en CDR CORBA que contiene los tres campos de una estructura, *struct*, cuyos respectivos tipos son *string*, *string* y *unsigned long*. La figura muestra la secuencia de bytes con cuatro bytes en cada fila. La representación de cada cadena está compuesta por un *unsigned long* representando la longitud, seguido por los caracteres en la cadena. Para simplificar, supongamos que cada carácter ocupa justo un byte. Los datos de longitud variable se rellenan con ceros de modo que tengan una forma estándar, haciendo que sean comparables los datos empaquetados o su suma de comprobación. Hay que hacer notar que cada *unsigned long*, que ocupa cuatro bytes, comienza en una posición que es múltiplo de cuatro. La figura no distingue entre los órdenes

Posición en la secuencia de bytes	← 4 bytes →	*Notas sobre la representación*
0-3	5	*Longitud del string*
4-7	"Pére"	*«Pérez»*
8-11	"z___"	
12-15	6	*Longitud del string*
16-19	"Madr"	*«Madrid»*
20-23	"id__"	
24-27	1934	*Unsigned long*

La forma aplanada representa una estructura *Persona* con el valor: {«Pérez», «Madrid», 1934}

Figura 4.8. Mensaje CDR CORBA.

big-endian y little-endian. Aunque el ejemplo de la Figura 4.8 es simple, el CDR de CORBA puede representar cualquier estructura de datos que se pueda componer a partir de datos primitivos y compuestos, siempre que no se utilicen punteros.

Otro ejemplo de una representación externa de datos es el estándar XDR de Sun, el cual está especificado en el RFC 1832 [Srinivasan 1995b] y descrito en www.cdk3.net/ipc. Fue desarrollado por Sun para utilizarlo en el intercambio de mensajes entre clientes y servidores en Sun NFS (véase el Capítulo 8).

El tipo de un elemento de datos no acompaña a la representación de los datos en el mensaje, ni en el CDR de CORBA ni el XDR de Sun. Esto es porque se supone que tanto el emisor como el receptor tienen un conocimiento común del orden y de los tipos de datos de los ítems en el mensaje. En particular para RMI o RPC, cada invocación de método pasa argumentos de unos tipos concretos, y el resultado es un valor de un tipo dado.

◇ **Empaquetado en CORBA.** Las operaciones de empaquetado se pueden generar automáticamente a partir de las especificaciones de los tipos de datos de los ítems que tienen que ser transmitidos en un mensaje. Los tipos de las estructuras de datos y los tipos de los ítems de datos básicos están descritos en CORBA IDL (véase la Sección 17.2.3), que proporciona una notación para describir los tipos de los argumentos y los resultados de los métodos RMI. Por ejemplo, podríamos utilizar CORBA IDL para describir la estructura de datos en el mensaje de la Figura 4.8 como sigue:

```
struct Persona
    string nombre;
    string lugar;
    long año;
};
```

La interfaz del compilador CORBA (véase el Capítulo 5) genera las operaciones de empaquetado y desempaquetado apropiadas para los argumentos y el resultado de los métodos remotos a partir las definiciones de los tipos de sus parámetros y resultados.

4.3.2. SERIALIZACIÓN DE OBJETOS EN JAVA

En Java RMI, tanto los objetos como los datos primitivos pueden ser pasados como argumentos y resultados de la invocación de métodos. Un objeto es una instancia de una clase Java. Por ejemplo, la clase Java equivalente a *struct Persona* definida en IDL de CORBA podría ser:

```
public class Persona implements Serializable {
    private String nombre;
    private String lugar;
    private int año;
    public Persona(String unNombre, String unLugar, int unAño) {
        nombre = unNombre;
        lugar = unLugar;
        año = unAño;
    }
    //seguido por los métodos para acceder a los campos
}
```


La clase de arriba declara implementar la interfaz *Serializable*, la cual no tiene métodos. Declarar que una clase implementa la interfaz *Serializable* (que viene dada en el paquete *java.io*) tiene el efecto de permitir que sus instancias sean serializables.

En Java, el término serialización se refiere a la actividad de aplanar un objeto o un conjunto relacionado de objetos para obtener una forma lineal adecuada para ser almacenada en disco o para ser transmitida en un mensaje, por ejemplo, como argumento o como resultado de un RMI. La deserialización consiste en restablecer el estado de un objeto o un conjunto de objetos desde su estado lineal. Se asume que el proceso que realiza la deserialización no tiene conocimiento previo de los tipos de los objetos en la forma lineal. Por lo tanto, debe incluirse en la forma lineal alguna información sobre la clase de cada objeto. Esta información posibilita al receptor la carga de la clase apropiada cuando un objeto es deserializado.

La información sobre la clase se compone del nombre de la clase y un número de versión. El número de versión está pensado para cambiar y reflejar con este cambio modificaciones importantes en la clase. Puede ser establecido por el programa o calculado de forma automática como un valor dependiente del nombre de la clase, sus campos, métodos e interfaces. El proceso que deserializa un objeto puede comprobar que tiene la versión correcta de la clase.

Los objetos Java pueden contener referencias a otros objetos. Cuando un objeto es serializado, todos los objetos a los que referencia son serializados con él para asegurarse de que cuando el objeto sea reconstruido, todas sus referencias pueden ser rellenadas en el destino. Las referencias se serializan como apuntadores (*handlers*) (en este caso, el apuntador es una referencia a un objeto dentro de la forma serializada, por ejemplo, el siguiente número en una secuencia de enteros positivos). El procedimiento de serialización debe asegurarse de que existe una correspondencia 1-1 entre las referencias y los apuntadores. También debe asegurarse de que cada objeto sea escrito sólo una vez (en la segunda o subsecuentes ocurrencias de un objeto, se escribirá el apuntador en lugar del objeto).

Para serializar un objeto, se escribe la información de su clase, seguida de los tipos y los nombres de los campos. Si los campos pertenecen a una clase nueva, entonces también se escribe la información de la clase, seguida de los nombres y los tipos de sus campos. Este procedimiento recursivo continúa hasta que se haya escrito la información de todas las clases necesarias y de los nombres y tipos de sus campos. Cada clase recibe un apuntador, y ninguna clase se escribe más de una vez en el flujo de bytes (en su lugar se escribe su apuntador cuando sea necesario).

Los contenidos de los campos que sean tipos de datos primitivos, como enteros, caracteres, booleanos, bytes y enteros largos, se escriben en un formato binario transportable utilizando métodos de la clase *ObjetctOutputStream*. Las cadenas de caracteres y los caracteres se escriben con los métodos llamados *writeUTF* utilizando el Formato de Transferencia Universal (*Universal Transfer Format*, UTF), que hace que los caracteres ASCII sean representados sin cambios (en un byte) mientras que los caracteres Unicode se representan por varios bytes. Las cadenas de caracteres se preceden por el número de bytes que ocupan en el flujo.

Como ejemplo, considérese la serialización del siguiente objeto:

```
Persona p = new Persona(«Pérez», «Madrid», 1934);
```

La forma serializada correspondiente se muestra en la Figura 4.9, la cual omite los valores de los apuntadores y de los marcadores de tipo que indican los objetos, clases, cadenas y otros objetos en la forma serializada completa. El primer campo (1934) es un entero que tiene una longitud fija, el segundo y el tercer campo son cadenas de caracteres que están precedidas por sus longitudes.

Para utilizar la serialización Java, por ejemplo para serializar el objeto *Persona*, hay crear una instancia de la clase *ObjectOutputStream* e invocar a su método *writeObject*, pasándole el objeto *Persona* como argumento. Para deserializar un objeto desde un flujo de datos, hay que abrir un *ObjectInputStream* sobre el flujo y utilizar su método *readObject* para reconstruir el objeto origi-

Valores serializados				*Explicación*
Persona	Número de versión de 8-bytes		a0	*nombre de la clase, número de versión*
3	int año	java.lang.String nombre:	java.lang.String lugar:	*número, tipo y nombre de las variables de instancia*
1934	5 Pérez	6 Madrid	a1	*valores de las variables de instancia*

La auténtica forma serializada contiene marcadores de tipo adicionales; a0 y a1 son apuntadores

Figura 4.9. Muestra de una serialización Java.

nal. La utilización de este par de clases es similar al uso de las clases *DataOutputStream* y *DataInputStream* mostrado en las Figuras 4.5 y 4.6.

La serialización y la deserialización de los argumentos y los resultados de una invocación remota son llevadas a cabo generalmente de forma automática por el middleware, sin ninguna participación del programador. Si fuera necesario, los programadores con requisitos especiales pueden escribir sus propias versiones de los métodos que leen y escriben objetos. Para encontrar el modo de hacerlo y conseguir más información sobre la serialización en Java, lea el curso sobre serialización de objetos [java.sun.com II]. Otro modo en el que el programador puede modificar los efectos de la serialización es declarando aquellas variables que no deberían ser serializadas como *transient*. Ejemplo de cosas que no deberían ser serializadas son las referencias a recursos locales como archivos o sockets.

◇ **El uso de la reflexión.** El lenguaje Java soporta la *reflexión*, que es la habilidad de preguntar sobre las propiedades de una clase, tales como los nombres y los tipos de sus campos y métodos. Esto también hace posible que se creen las clases a partir de su nombre, y crear un constructor para una clase dada con unos argumentos dados. La reflexión hace posible hacer la serialización y la deserialización de una manera totalmente genérica. Esto significa que no hay necesidad de generar funciones de empaquetado especiales para cada tipo de objeto, como se describió anteriormente para CORBA. Para encontrar más información sobre la reflexión, véase Flanagan [1997].

La serialización de objetos Java utiliza la reflexión para encontrar el nombre de la clase del objeto a serializar y los nombres, tipos y valores de sus variables de instancia. Esto es todo lo que se necesita en la forma serializada.

Para la deserialización, se utiliza el nombre de la clase en la forma serializada para crear un nuevo constructor con los tipos de argumentos correspondientes a aquellos especificados en la forma serializada. Finalmente, se utiliza el nuevo constructor para crear una nueva instancia del objeto con aquellos valores leídos de la forma serializada.

4.3.3. REFERENCIAS A OBJETOS REMOTOS

Cuando un cliente invoca un método en un objeto remoto, se envía un mensaje de invocación al proceso servidor que alberga el objeto remoto. Este mensaje necesita especificar el objeto particular cuyo método se va a invocar. Una *referencia a un objeto remoto* es un identificador para un objeto remoto que es válida a lo largo y ancho de un sistema distribuido. En el mensaje de invocación se incluye una referencia a objeto remoto que especifica cuál es el objeto invocado. El Capítulo 5 explica que las referencias a objetos remotos también se pasan como argumentos y se devuelven como resultados de la invocación de métodos remotos, que cada objeto remoto tiene una única referencia a objeto remoto y que las referencias a objetos remotos pueden compararse para saber si se refieren al mismo objeto remoto. Ahora trataremos sobre la representación externa de las referencias a objetos remotos.

32 bits	*32 bits*	*32 bits*	*32 bits*	
dirección Internet	número de puerto	tiempo	número de objeto	interfaz de objeto remoto

Figura 4.10. Representación de una referencia de objeto remoto.

Las referencias a objetos remotos deber generarse de modo que se asegure su unicidad sobre el espacio y el tiempo. En general, existirán varios procesos alojando objetos remotos, de modo que las referencias a objetos remotos deben ser únicas entre todos los procesos en los computadores de un sistema distribuido. Es importante que la referencia al objeto remoto no sea reutilizada incluso después de que el objeto remoto asociado a una referencia a objeto remoto haya sido borrado, ya que los potenciales invocadores podrían tener referencias remotas obsoletas. Cualquier intento de invocar objetos borrados debería producir un error en lugar de permitir el acceso a un objeto diferente.

Existen varios modos de asegurarse que una referencia a un objeto remoto es única. Un modo es construir una referencia a objeto remoto concatenando la dirección Internet de su computador y el número de puerto del proceso que lo creó junto con el instante de tiempo de su creación y un número de objeto local. El número de objeto local se incrementa cada vez que el proceso crea un objeto.

El número de puerto junto al tiempo constituyen un identificador único en ese computador. Con esta aproximación, las referencias a objetos remotos podrían representarse según un formato como el mostrado en la Figura 4.10. En las implementaciones más simples de RMI, los objetos remotos viven en el proceso que los crea y sobreviven sólo mientras que el proceso continúa en ejecución. En estos casos, la referencia a objeto remoto puede utilizarse como una dirección del objeto remoto. En otras palabras, los mensajes de invocación se envían a la dirección Internet de la referencia remota, y dentro del computador al proceso identificado por el número de puerto.

Para permitir que los objetos remotos sean recolocados en distintos procesos en computadores diferentes, las referencias de objetos remotos no debieran utilizarse como dirección del objeto remoto. La Sección 17.2.4 tratará sobre un formato de referencia de objetos remotos que permite que los objetos puedan ser activados en diferentes servidores a lo largo de su tiempo de vida.

El último campo de la referencia de objeto remoto mostrada en la Figura 4.10 contiene cierta información sobre la interfaz del objeto remoto, por ejemplo el nombre de la interfaz. Esta información es relevante para cualquier proceso que reciba una referencia de objeto remoto como argumento o resultado de una invocación remota, ya que necesitará conocer los métodos que ofrece el objeto remoto. Este punto se tratará de nuevo en la Sección 5.2.5.

4.4. COMUNICACIÓN CLIENTE-SERVIDOR

Esta forma de comunicación está orientada a soportar los roles y el intercambio de mensajes de las interacciones típicas cliente-servidor. En el caso normal, la comunicación petición-respuesta es síncrona, ya que el proceso cliente se bloquea hasta que llega la respuesta del servidor. Esta comunicación también puede ser fiable ya que la respuesta del servidor es, en efecto, un acuse de recibo para el cliente. La comunicación cliente-servidor asíncrona es una alternativa que puede ser útil en situaciones donde los clientes pueden recuperar las respuestas más tarde (veáse la Sección 6.5.2).

En los siguientes párrafos se describen los intercambios cliente-servidor en términos de las operaciones *envía* y *recibe* del API Java para datagramas UDP, aunque muchas implementaciones actuales utilicen streams TCP. Un protocolo construido sobre datagramas evita las sobrecargas asociadas con el protocolo de streams TCP. En concreto:

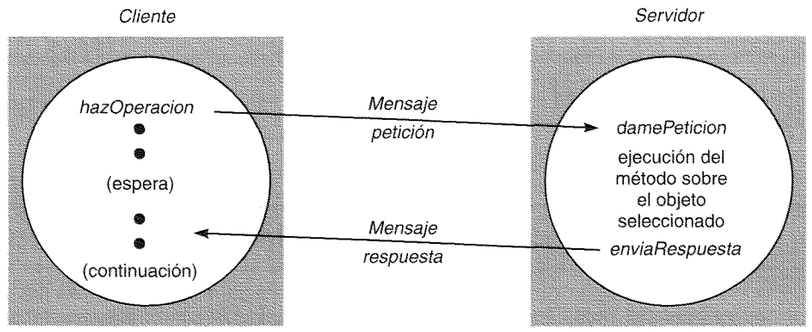


Figura 4.11. Comunicación cliente-servidor.

- Los reconocimientos son redundantes, ya que las peticiones son seguidas por las respuestas.
- El establecimiento de la conexión involucra dos pares de mensajes extra además del par necesario para la petición de conexión y la consiguiente respuesta.
- El control del flujo es redundante en la mayoría de las invocaciones, ya que éstas pasan solamente una pequeña cantidad de información sobre argumentos y resultados.

◇ **El protocolo petición-respuesta.** El siguiente protocolo está basado en un trío de primitivas de comunicación: *hazOperacion*, *damePeticion*, *enviaRespuesta*, según se muestra en la Figura 4.11. Es de esperar que la mayoría de los sistemas RMI y RPC estén soportados por un protocolo similar. El que describimos aquí se construye a medida para dar soporte RMI en la que se le pasa una referencia a un objeto remoto sobre el que hay que invocar el método indicado en el mensaje de petición.

Este protocolo de petición-respuesta, diseñado especialmente, hace corresponder a cada petición una respuesta. Podría estar diseñado para obtener ciertas garantías de entrega. Si se utilizan datagramas UDP, las garantías de entrega deben venir dadas por el protocolo petición-respuesta, donde el mensaje de respuesta del servidor sirve como acuse de recibo del mensaje de petición del cliente. La Figura 4.12 muestra las tres primitivas de comunicación.

La primitiva *hazOperacion* se utiliza en los clientes para invocar las operaciones remotas. Sus argumentos especifican el objeto remoto y el método a invocar, junto con información adicional (argumentos) requerida por el método. Su resultado es una respuesta RMI. Se supone que el cliente que usa *hazOperacion* empaqueta los argumentos en una cadena de bytes y desempaqueta el resultado de la cadena de bytes que le es devuelta. El primer argumento de *hazOperacion* es una instancia de la clase *RemoteObjectRef*, que representa las referencias de los objetos remotos, como por ejemplo la mostrada en la Figura 4.10. Esta clase proporciona los métodos para conseguir la dirección Internet y el puerto del servidor del objeto remoto. La primitiva *hazOperacion* envía un mensaje de petición al servidor cuya dirección Internet y puerto se especifican en la referencia de objeto remoto dada como argumento. Después de enviar el mensaje de petición, *hazOperacion* invoca el método *recibe* para conseguir el mensaje respuesta, del que extrae el resultado y lo devuelve a su invocador. El invocador de *hazOperacion* se bloquea hasta que el objeto remoto en el servidor ejecuta la operación solicitada y transmite el mensaje respuesta al proceso cliente.

public byte[ ] hazOperacion (RemoteObjectRef o, int idMetodo, byte [ ] argumentos)
envía un mensaje de petición al objeto remoto y recibe la respuesta.
Los argumentos especifican el objeto remoto, el método a invocar y los argumentos de ese método.

public byte[ ] damePeticion( );
adquiere una petición del cliente a través del puerto del servidor.

public void enviaRespuesta(byte [ ] respuesta, InetAddress hostCliente, int puertoCliente);
envía el mensaje de respuesta al cliente a su dirección Internet y a su puerto.

Figura 4.12. Operaciones del protocolo petición-respuesta.

La primitiva *damePeticion* se usa en el servidor para hacerse con las peticiones de servicio, según se muestra en la Figura 4.11. Cuando el servidor ha invocado el método sobre el objeto especificado, utiliza el método *enviaRespuesta* para mandar el mensaje de respuesta al cliente. Cuando el cliente recibe el mensaje de respuesta, desbloquea la operación *hazOperacion* y continúa la ejecución el programa cliente.

La información transmitida en un mensaje de petición y de respuesta se muestra en la Figura 4.13. El primer campo indica cuándo el mensaje es una petición o una respuesta. El segundo campo, *idPeticion*, contiene un identificador de mensajes. Una primitiva *hazOperacion* en el cliente genera un *idPeticion* para cada mensaje de petición, y el servidor lo copia en el correspondiente mensaje de respuesta. Esto hace posible que *hazOperacion* pueda comprobar que un mensaje de respuesta es el resultado de la petición actual, no de una invocación anterior que se ha retrasado. El tercer campo es una referencia de objeto remoto empaquetada según la forma mostrada en la Figura 4.10. El cuarto campo es un identificador del método a invocar, por ejemplo, los métodos en una interfaz pueden numerarse 1, 2, 3, ...; si el cliente y el servidor utilizan un lenguaje común que soporte la reflexión, entonces se puede colocar en este campo una representación del método mismo (en Java, se puede colocar en este campo una instancia de *Method*).

Identificadores de mensaje: Cualquier esquema que involucre la gestión de mensajes para proporcionar propiedades adicionales como la entrega fiable de mensajes o una comunicación petición-respuesta necesita que cada mensaje tenga un identificador único por el que pueda ser referenciado. Un identificador de mensaje tiene dos partes:

1. Un identificador de petición, *idPeticion*, que proporciona el proceso emisor de una sucesión de enteros creciente.
2. Un identificador para el proceso emisor, por ejemplo, su puerto y su dirección Internet.

La primera parte hace que el identificador sea único para el emisor, y la segunda lo hace único para el sistema distribuido. (La segunda parte se puede obtener independientemente, por ejemplo, si se usa UDP, desde el mensaje recibido.)

Cuando el valor del campo *idPeticion* alcanza el valor máximo para un entero sin signo (por ejemplo $2^{32} - 1$) vuelve a cero. La única restricción impuesta es que el tiempo de vida de un identificador de mensaje deberá ser mucho menor que el tiempo que se necesita para dar la vuelta a esa secuencia de enteros.

tipoMensaje	*int (0= Petición, 1= Respuesta)*
idPeticion	*int*
referenciaObjeto	*RemoteObjectRef*
idMetodo	*int o Method*
argumentos	*// cadena de bytes*

Figura 4.13. Estructura de un mensaje petición-respuesta.

◇ **Modelo de fallos del protocolo petición-respuesta.** Si las tres primitivas *hazOperacion*, *damePeticion*, *enviaRespuesta* se implementan con datagramas UDP, adolecerán de los mismos fallos de comunicación que cualquier otro ejemplo de aplicación de UDP. Esto es:

- Sufrirán fallos de omisión.
- No se garantiza que los mensajes lleguen en el orden de emisión.

Además, el protocolo puede padecer el fallo de los procesos (véase la Sección 2.3.2). Presuponemos que los procesos pueden caer. Es decir, cuando se detienen, permanecen detenidos (no provocan un comportamiento bizantino).

En aquellas ocasiones en las que un servidor falle o se elimine un mensaje de petición o de respuesta, *hazOperacion* utiliza un *timeout* para esperar el mensaje de respuesta del servidor. La acción a tomar cuando se supera el tiempo límite de espera depende de las garantías de entrega ofrecidas.

Tiempos de espera límite: Existen varias opciones respecto a lo que la primitiva *hazOperacion* puede hacer después de superar un timeout. La opción más simple es devolver inmediatamente el control indicando al cliente que la primitiva *hazOperacion* ha fracasado. Ésta no es la aproximación normal; se puede alcanzar un timeout debido a la pérdida de un mensaje de petición o de respuesta y, en el último caso, la operación se habrá realizado. Para compensar la posibilidad de la pérdida de los mensajes, *hazOperacion* manda el mensaje de petición de forma repetida hasta que bien obtiene una respuesta o bien está razonablemente seguro que el retraso se debe a una falta de respuesta del servidor, más que a mensajes perdidos. En algún momento, cuando *hazOperacion* devuelva el control indicará al cliente mediante una excepción que no ha recibido resultado alguno.

Eliminación de mensajes de petición duplicados: En los casos en los que el mensaje de petición es retransmitido, el servidor puede recibir más de uno. Por ejemplo, el servidor puede recibir el primer mensaje de petición pero precisará más tiempo que el timeout del cliente para ejecutar el comando y devolver la respuesta. Esto puede llevar a que el servidor ejecute una operación mas de una vez para la misma petición. Para evitarlo, se diseñó el protocolo para reconocer sucesivos mensajes (del mismo cliente) con el mismo identificador de petición y para eliminar los duplicados. Si el servidor no hubiera enviado aún la contestación, no necesita realizar ninguna operación especial, transmitirá la respuesta cuando termine de ejecutar la operación.

Pérdida de mensajes de respuesta: Si el servidor ya ha enviado la respuesta cuando recibe una petición duplicada necesitará ejecutar otra vez la operación para obtener el resultado, a menos que haya almacenado el resultado de la ejecución original. Algunos servidores pueden ejecutar sus operaciones más de una vez y obtener el mismo resultado cada vez. Una *operación idempotente* es una operación que puede ser llevada a cabo repetidamente con el mismo efecto que si hubiera sido ejecutada exactamente una sola vez. Por ejemplo, una operación que añade un elemento a un conjunto es una operación idempotente porque siempre tendrá el mismo efecto cada vez que se realice, mientras que la operación de añadir un elemento a una secuencia no es una operación idempotente. Un servidor cuyas operaciones sean todas idempotentes no tendrá que tomar medidas especiales para evitar que se ejecuten más de una vez.

Historial: En aquellos servidores que necesitan retransmitir las respuestas sin tener que volver a ejecutar las operaciones, es imprescindible la utilización de un historial. El término *historial* o *histórico* se utiliza para referirse a una estructura que contiene el registro de los mensajes de respuesta que han sido transmitidos. Una entrada en el historial contiene un identificador de petición, un mensaje y un identificador del cliente al que fue enviado. Su propósito es permitir que el servidor pueda retransmitir los mensajes de respuesta cuando los clientes se lo soliciten. Un problema asociado con el uso del historial es el coste de almacenamiento. Un historial se volverá demasiado grande a menos que el servidor pueda decidir cuándo no será necesario seguir almacenando un mensaje para su retransmisión.

Nombre	*Mensajes enviados por*		
	Cliente	*Servidor*	*Cliente*
R	*Petición*		
RR	*Petición*	*Respuesta*	
RRA	*Petición*	*Respuesta*	*Confirmación de la respuesta*

Figura 4.14. Protocolos de intercambio RPC.

Como los clientes pueden hacer una sola petición al tiempo, el servidor puede interpretar que cada petición es un reconocimiento de la respuesta previa. Por lo tanto, el historial sólo necesita contener el último mensaje de respuesta enviado a cada cliente. Sin embargo, el volumen de mensajes de respuesta en el historial de un servidor puede ser un problema cuando tiene un gran número de clientes. En particular, cuando un cliente termina, no reconoce la última respuesta recibida, por lo que los mensajes en el historial serán normalmente descartados después de que pase un período de tiempo.

Protocolo de intercambio de RPC: En la implementación de los distintos tipos de RPC se utilizan tres protocolos, con diferentes semánticas en presencia de fallos de comunicación. Inicialmente fueron identificados por Spector [1982]:

- El protocolo *petición (R)*.
- El protocolo *petición-respuesta (RR)*.
- El protocolo *petición-respuesta-confirmación de la respuesta (RRA)*.

Los mensajes basados en estos protocolos se resumen en la Figura 4.14. El protocolo R (*request*) puede utilizarse cuando el procedimiento no tiene que devolver ningún valor y el cliente no necesita confirmación de que el procedimiento ha sido ejecutado. El cliente puede seguir inmediatamente después de haber enviado el mensaje de petición ya que no tiene que esperar un mensaje de respuesta. El protocolo RR (*request-reply*) es útil en la mayoría de los intercambios cliente-servidor, pues se basa en el protocolo petición-respuesta. No se necesitan mensajes especiales de acuse de recibo, ya que los mensajes de respuesta del servidor sirven como confirmación de las peticiones del cliente. Similarmente, una subsiguiente llamada desde un cliente podría ser considerada como un reconocimiento del mensaje de respuesta del servidor.

El protocolo RRA (*request-reply-acknowledge reply*) está basado en el intercambio de tres mensajes: petición-respuesta-confirmación de la respuesta. El mensaje de reconocimiento de la respuesta contiene el *idPeticion* del mensaje de respuesta que reconoce. Esto hace posible que el servidor pueda descartar las entradas de su historial. La llegada de un *idPeticion* en un mensaje de reconocimiento será interpretada como el acuse de recibo de la recepción de todos los mensajes de respuesta con identificadores de petición menores, por lo que la pérdida de un mensaje de reconocimiento de respuesta no resulta dañina. Aunque el intercambio implica un mensaje adicional, no necesita bloquear al cliente, ya que el reconocimiento puede transmitirse después de que la respuesta haya sido entregada al cliente, aunque esto requiere procesamiento y recursos de red. El Ejercicio 4.22 sugiere una optimización al protocolo RRA.

◇ **Utilización de streams TCP para implementar el protocolo petición-respuesta.** En la sección dedicada a los datagramas mencionamos que a menudo resulta difícil decidir un tamaño apropiado para el búfer en el cual se van a almacenar los datagramas recibidos. En el protocolo petición-respuesta, esto se aplica a los búferes utilizados por el servidor para recibir los mensajes de petición de los clientes y por el cliente para recibir las respuestas. La limitada longitud de los datagramas (normalmente 8 kilobytes) no se puede considerar como adecuada para su uso en siste-

mas RMI transparentes, ya que los argumentos o resultados de los procedimientos pueden ser cualquier tamaño.

Una de las razones para elegir la implementación de los protocolos petición-respuesta sobre streams TCP es el deseo de evitar la implementación de protocolos multipaquete, permitiendo la transmisión de argumentos y resultados de cualquier tamaño. En particular la serialización de objetos Java es un protocolo de streams que permite enviar los argumentos y los resultados entre el cliente y el servidor sobre streams, haciendo posible la transmisión fiable de colecciones de objetos de cualquier tamaño. Si se utiliza el protocolo TCP, se está asegurando que los mensajes de petición y de respuesta serán entregados de manera fiable, de modo que no es necesario un protocolo petición-respuesta para tratar los mensajes de retransmisión y filtrar los duplicados, y los historiales. Además el mecanismo de control de flujo permite transmitir argumentos y resultados grandes sin tomar medidas especiales para evitar el desbordamiento en el destino. Por todo esto, se elige el protocolo TCP para implementar los protocolos petición-respuesta dado que puede simplificar su implementación. Si se envían sucesivas peticiones y respuestas entre el mismo par de cliente y servidor sobre el mismo flujo, no es necesaria la sobrecarga del establecimiento de la conexión en cada invocación remota. También se puede reducir la sobrecarga debida a los mensajes de reconocimiento si un mensaje de respuesta sigue inmediatamente a otro de petición.

Algunas veces, la aplicación no necesita de todas las posibilidades ofrecidas por TCP, y se puede implementar sobre UDP un protocolo más eficiente, especialmente construido. Por ejemplo, como hemos mencionado anteriormente, Sun NFS no necesita mensajes de tamaño ilimitado, ya que transmite mensajes constituidos por bloques de tamaño fijo entre el cliente y el servidor. Además, sus operaciones son idempotentes, de modo que no importa si las operaciones se ejecutan más de una vez a la hora de retransmitir los mensajes de respuesta perdidos, haciendo innecesario mantener un historial.

◇ **HTTP: un ejemplo de protocolo petición-respuesta.** El Capítulo 1 presentó el Protocolo de Transferencia de Hipertexto (*HyperTex Transfer Protocol*, HTTP) utilizado por los navegadores web para realizar peticiones a los servidores web y para recibir las respuestas de ellos. Valga recordar, como recapitulación, que los servidores web gestionan recursos implementados de diferentes modos:

- Como datos, por ejemplo el texto de una página HTML, una imagen o la clase de un applet.
- Como programa, por ejemplo los programas *cgi* y los servlets (véase [java.sun.com III]) que puede ser ejecutado en el servidor web.

Las peticiones de los clientes especifican un URL que incluye el nombre DNS del host del servidor web y un número de puerto opcional, además del identificador de un recurso en el servidor.

HTTP es un protocolo que especifica los mensajes involucrados en un intercambio petición-respuesta, los métodos, argumentos y resultados y las reglas para representar (empaquetar) todo ello en los mensajes. Soporta un conjunto fijo de métodos (GET, PUT, POST, etc.) que son aplicables a todos los recursos. Al contrario de los protocolos anteriores, cada objeto tiene sus propios métodos. Además de invocar métodos sobre recursos web, el protocolo permite la negociación de contenidos y una autenticación del tipo *clave de acceso*.

Negociación del contenido: las peticiones de los clientes pueden incluir información sobre qué tipo de representación de datos pueden aceptar (por ejemplo lenguaje o tipo de medio), haciendo posible que el servidor pueda elegir la representación más apropiada para el usuario.

Autenticación: se utilizan credenciales y desafíos para conseguir una autenticación del estilo *clave de acceso*. En el primer intento de acceso al área protegida por palabra clave, el servidor responde con un desafío aplicable al recurso. El Capítulo 7 explica los desafíos. Cuando el cliente recibe el desafío pide al usuario un nombre y una palabra de paso, y se los envía al servidor en las siguientes solicitudes. HTTP se implementa sobre TCP. En la versión original del protocolo, cada interacción cliente-servidor se componía de los siguientes pasos:

- El cliente solicita una conexión al puerto del servidor por defecto o a otro especificado en la petición aceptada por el servidor.
- El cliente envía un mensaje de petición al servidor.
- El servidor envía un mensaje de respuesta al cliente.
- Se cierra la conexión.

Sin embargo, la necesidad de establecer y cerrar una conexión para cada intercambio petición-respuesta resulta cara, sobrecargando al servidor y enviando demasiados mensajes por la red. Teniendo en cuenta que los navegadores generalmente hacen múltiples peticiones al mismo servidor, la siguiente versión del protocolo (HTTP 1.1: véase el RFC 2616 [Fielding y otros 1999]) utiliza *conexiones persistentes*; conexiones que permanecen abiertas durante una serie de intercambios petición-respuesta entre el cliente y el servidor. Una conexión persistente puede ser cerrada por el cliente o por el servidor en cualquier instante enviando una indicación al otro participante. Los servidores cerrarán una conexión persistente cuando ha estado inactiva durante un cierto intervalo de tiempo. Es posible que un cliente pueda recibir un mensaje del servidor indicándole que la conexión se cierra mientras se está en el transcurso del envío de otra petición o respuestas. En estos casos, el navegador volverá a enviar las peticiones sin que el usuario tenga que verse implicado en ello, ya que las operaciones involucradas son idempotentes. Por ejemplo, el método GET descrito a continuación es idempotente. Cuando estén implicadas operaciones no idempotentes, el navegador deberá consultar al usuario sobre los pasos a dar.

Las peticiones y las respuestas se empaquetan en los mensajes como cadenas de caracteres ASCII, aunque los recursos que pueden representarse como secuencias de bytes quizás tengan que ser comprimidos. El uso del texto como representación de datos externa simplifica el uso del HTTP por parte de los programadores de aplicaciones que trabajan directamente con el protocolo. En este contexto, una representación textual no añade mucha longitud a los mensajes.

Los recursos considerados como datos se proporcionan en forma de estructuras de tipo MIME tanto en los argumentos como en los resultados. *Multipurpose Internet Mail Extensions* (Extensiones de Correo Electrónico Multipropósito, MIME) es un estándar para enviar mensajes de correo electrónico compuestos por varias partes conteniendo a la vez, por ejemplo, texto, imágenes y sonido. Los datos van precedidos por su *tipo Mime*, de modo que el receptor podrá saber cómo gestionarlos. Un *tipo Mime* especifica un tipo y un subtipo, por ejemplo, *text/plain*, *text/html*, *image/gif*, *image/jpeg*. Los clientes también pueden especificar los tipos Mime que están dispuestos a aceptar.

Métodos HTTP: Cada petición de un cliente especifica el nombre de un método que habrá de ser aplicado al recurso en el servidor y el URL de dicho recurso. El mensaje de respuesta indica el estado de la petición. Las peticiones y las respuestas pueden contener también datos, el contenido de un formulario o la salida de un programa ejecutado en el servidor web. Los métodos considerados son los siguientes:

GET: pide el recurso cuyo URL se da como argumento. Si el URL se refiere a datos, entonces el servidor responderá enviando de vuelta los datos indicados por el URL. Si el URL se refiere a un programa, entonces el servidor web ejecutará el programa y devolverá su salida al cliente. Se pueden añadir argumentos al URL; por ejemplo, un GET se puede utilizar para enviar el contenido de un formulario a un programa *cgi* que los tomará como entrada. La operación GET puede condicionarse a la fecha de modificación del recurso indicado. El método GET también puede configurarse para obtener parte de los datos.

HEAD: esta petición es idéntica a GET, sólo que no devuelve datos. Sin embargo, devuelve toda la información sobre los datos, como el tiempo de la última modificación, su tipo o tamaño.

POST: especifica el URL de un recurso (por ejemplo un programa) que puede tratar los datos aportados con la petición. El procesamiento llevado a cabo sobre los datos depende del programa especificado en el URL. Este método está diseñado para:

- Proporcionar un bloque datos (por ejemplo los obtenidos en un formulario) a un proceso de gestión de datos como un servlet o un programa *cgi*.
- Enviar un mensaje a un tablón de anuncios, lista de correo o grupo de noticias.
- Modificar una base de datos con una operación de añadir registro.

PUT: indica que los datos aportados en la petición deben ser almacenados con la URL aportada como su identificador, ya sea como una modificación de datos existentes o como la creación de un recurso nuevo.

DELETE: el servidor borrará el recurso identificado por el URL. El servidor no siempre permitirá esta función, en cuyo caso se devolverá una indicación de fallo.

OPTIONS: el servidor proporciona al cliente una lista de métodos aplicables a un URL (por ejemplo, *GET*, *HEAD*, *PUT*) y sus requisitos especiales.

TRACE: el servidor envía de vuelta el mensaje de petición. Se utiliza en procesos de depuración.

Las peticiones descritas anteriormente pueden ser interceptadas por un servidor proxy (véase la Sección 2.2.2). Las respuestas a *GET* y *HEAD* pueden ser capturadas y almacenadas en un caché por los servidores proxy.

Contenido del mensaje: Cada mensaje HTTP *Request* especifica el nombre de un método, el URL de un recurso, la versión del protocolo, algunas cabeceras y un cuerpo de mensaje opcional. La Figura 4.15 muestra el contenido de un mensaje HTTP *Request* cuyo método es GET. Cuando la URL específica es un recurso de datos, el método GET no incluye cuerpo del mensaje. Un proxy necesita el URL completo, tal y como se muestra en la figura, aunque tiene sus ventajas enviar sólo la ruta en el caso de un servidor origen, dado que se envían menos datos.

Los campos de la cabecera contienen modificadores de la petición e información sobre el cliente, como las condiciones sobre la última fecha de modificación del recurso o los tipos de contenido aceptables (por ejemplo texto HTML o imágenes JPEG). Se puede utilizar un campo de autorización para proporcionar los credenciales del cliente bajo la forma de un certificado que especifica sus derechos de acceso al recurso.

Un mensaje *Reply* especifica la versión del protocolo, un código de estado y su *razón*, algunas cabeceras y un cuerpo de mensaje opcional, como muestra la Figura 4.16. El *código de estado* y la *razón* proporcionan un informe sobre el éxito o cualquier otra situación asociada al cumplimiento de la petición. El primero es un entero de tres dígitos que deben ser interpretados por el programa cliente, y el último es una frase de texto que debe ser entendida por la persona usuaria. Los campos de cabecera se utilizan para pasar información adicional sobre el servidor o sobre el acceso al recurso. Por ejemplo, si la petición requiere autenticación, el estado de la respuesta indica este extremo y un campo de la cabecera contiene el desafío. Algunas informaciones de estado devueltas tienen efectos más complejos. En concreto, una respuesta con el estado 303 le dice al navegador que busque un URL diferente, el cual viene especificado en un campo de la cabecera de la respuesta. Está indicado para ser utilizado por los programas activados por una petición POST cuando el programa necesita redirigir al navegador hacia un recurso concreto.

método	*URL*	*versión HTTP*	*cabeceras*	*cuerpo del mensaje*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/1.1		

Figura 4.15. Mensaje HTTP *Request*.

versión HTTP	*código de estado*	*razón*	*cabeceras*	*cuerpo del mensaje*
HTTP/1.1	200	OK		datos del recurso

Figura 4.16. Mensaje HTTP *Reply*.

El cuerpo del mensaje en los mensajes de petición o en los de respuesta contiene los datos asociados con la URL especificada en la petición. El cuerpo del mensaje tiene sus propias cabeceras especificando información sobre los datos, como la longitud, el tipo Mime, el conjunto de caracteres, la codificación del contenido y la última fecha de modificación. El campo del tipo Mime concreta el tipo de los datos, por ejemplo *image/jpeg* o *text/plain*. El campo de codificación especifica el algoritmo de compresión utilizado.

4.5. COMUNICACIÓN EN GRUPO

El intercambio de mensajes entre iguales no es mejor modelo para la comunicación entre un proceso y un grupo de procesos, como por ejemplo se da en el caso de un servicio implementado por varios procesos en diferentes computadores, quizás para proporcionar tolerancia a fallos o mejorar la disponibilidad. Resulta más apropiada una *operación de multidifusión*; ésta es una operación que envía un único mensaje desde un proceso a cada uno de los miembros de un grupo de procesos, normalmente de modo que la pertenencia al grupo resulte transparente para el emisor. Existe un abanico de posibilidades respecto al comportamiento deseado de la multidifusión. La más simple no proporciona garantía alguna sobre la entrega del mensaje y el mantenimiento del orden de la emisión de los mismos.

Los mensajes de multidifusión proporcionan una infraestructura para construir sistemas distribuidos con las siguientes características:

1. *Tolerancia a fallos basada en servicios replicados*: un servicio replicado consta de un grupo de servidores. Las solicitudes de los clientes se dirigen a todos los miembros del grupo y cada uno de ellos realiza la misma operación. Incluso cuando varios de los miembros fallen, los clientes todavía serán servidos.
2. *Búsqueda de los servidores de descubrimiento en redes espontáneas*: la Sección 2.2.3 introdujo los servicios de descubrimiento en las redes espontáneas. Los servidores y los clientes pueden utilizar la multidifusión mensajes para localizar los servicios disponibles, y registrar sus interfaces, o para buscar las interfaces de otros servicios en el sistema distribuido.
3. *Mejores prestaciones basada en datos replicados*: los datos se replican para incrementar las prestaciones de un servicio; en algunos casos se colocan réplicas de los datos en los computadores de los usuarios. Cada vez que se producen cambios en los datos, el nuevo valor se comunica a los procesos que gestionan las réplicas de los datos mediante multidifusión.
4. *Propagación de las notificaciones de eventos*: la multidifusión a un grupo puede utilizarse para notificar a los procesos que algo ha sucedido. Por ejemplo, un sistema de noticias podría advertir a los usuarios interesados cada vez que se dirige un nuevo mensaje a un grupo de noticias particular. El sistema Jini utiliza multidifusión para informar a los usuarios interesados que se ha descubierto un nuevo servicio.

A continuación presentamos la multidifusión IP y después revisaremos si las necesidades de multidifusión mostradas arriba se pueden satisfacer por la multidifusión IP. Para aquellos casos en que no sea así, propondremos algunas propiedades añadidas a las de los protocolos de comunicación de grupos proporcionados por la multidifusión IP.

4.5.1. MULTIDIFUSIÓN IP. UNA IMPLEMENTACIÓN DE LA COMUNICACIÓN EN GRUPO

Esta sección discute la multidifusión IP y presenta el API de Java correspondiente a través de la clase *MulticastSocket*.

◇ **Multidifusión IP.** La multidifusión IP se construye sobre el protocolo Internet, IP. Tenga en cuenta que los paquetes IP se dirigen a los computadores; mientras que los puertos pertenecen a los niveles TCP y UDP. La multidifusión IP permite que el emisor transmita un único paquete IP a un conjunto de computadores que forman un grupo de multidifusión. El emisor no tiene que estar al tanto de las identidades de los receptores individuales y del tamaño del grupo. Los grupos de multidifusión se especifican utilizando las direcciones Internet de la clase D (véase la Figura 3.16); esto es, una dirección cuyos primeros cuatro bits son 1110 en IPv4.

El convertirse en miembro de un grupo de multidifusión permite al computador recibir los paquetes IP enviados al grupo. La pertenencia a los grupos de multidifución es dinámica, permitiéndose que los computadores se apunten o se borren a un número arbitrario de grupos en cualquier instante. Es posible enviar datagramas a un grupo de multidifusión sin pertenecer a él.

En el nivel de programación de aplicación, la multidifusión sólo es accesible vía UDP. Un programa de aplicación ejecuta multidifusión enviando datagramas UDP con la dirección de multidifusión y el número de puerto de la forma usual. Es posible apuntarse a un grupo de multidifusión haciendo que su conector pertenezca al grupo, habilitándolo para recibir los mensajes dirigidos al grupo. En el nivel IP, un computador pertenece a un grupo de multidifusión cuando uno o más de sus procesos tienen conectores que pertenezcan a ese grupo. Cuando llega un mensaje de multidifusión a un computador, envía copias del mismo a los conectores locales que pertenecen a la dirección de multidifusión destino del mensaje y están limitadas por el número de puerto especificado. Los siguientes detalles son específicos de IPv4:

Routers multidifusión: los paquetes IP pueden multidifundirse tanto en la red local como en toda Internet. La multidifusión local utiliza la capacidad de multidifusión de la red local, como por ejemplo una Ethernet. La multidifusión dirigida a Internet hace uso de las posibilidades de multidifusión de los routers (encaminadores), los cuales reenvían los datagramas únicamente a otros routers de redes con miembros de ese grupo, donde serán multidifundidos a los miembros locales. Para limitar la distancia de propagación de un datagrama de multidifusión, el emisor puede especificar el número de routers que puede cruzar; llamado tiempo de vida (*time to live*, TTL). Para entender cómo conocen los routers qué otros routers tienen miembros de un grupo de multidifusión véase Comer [1995].

Reserva de direcciones de multidifusión: las direcciones de multidifusión se pueden reservar de forma temporal o permanentemente. Existen grupos permanentes incluso cuando no tengan ningún miembro; y sus direcciones son asignadas arbitrariamente por la autoridad de Internet en el rango 224.0.0.1 a 224.0.0.255. Por ejemplo, la primera dirección corresponde a todos los hosts multidifusión.

El resto de las direcciones de multidifusión están disponibles para su uso por parte de grupos temporales, los cuales deben ser creados antes de su uso y dejar de existir cuando todos los miembros los hayan dejado. Cuando se crea un grupo temporal, se necesita una dirección de multidifusión libre para evitar los conflictos con otros grupos existentes. El protocolo de multidifusión IP no resuelve este problema. Cuando los usuarios sólo necesitan comunicarse de forma local, ponen el TTL a un valor pequeño, haciendo difícil que entre en conflicto con la dirección de otros grupos. Sin embargo, los programas que tienen que utilizar multidifusión IP a través de Internet necesitan una solución a este problema. El programa de directorio de sesiones (sd) sirve para arrancar o unirse a una sesión de multidifusión [mice.ed.ac.uk, session directory]. Proporciona una herramienta con una interfaz interactiva que permite que los usuarios


```
import java.net.*;
import java.io.*;
public class participanteMultidifusion{
  public static void main(String args[ ]){
  // los argumentos dan el mensaje y la dirección del grupo (por ejemplo "228.5.6.7")
    try {
      InetAddress grupo = InetAddress.getByName(args[1]);
      MulticastSocket s = new MulticastSocket(6789);
      s.joinGroup(grupo);
      byte [ ] m = args[0].getBytes( );
      DatagramPacket mensajeSalida =
          new DatagramPacket(m, m.length, grupo, 6789);
      s.send(mensajeSalida);
          // conseguir mensajes de otros miembros del grupo
      byte [ ] bufer = new byte[1000];
      for (int i=0; i<3; i++) {
          DatagramPacket mensajeEntrada =
                new DatagramPacket(bufer, bufer.length);
          s.receive(mensajeEntrada);
          System.out.println("Recibido:" + new String(mensajeEntrada.getData( )));
      }
      s.leaveGroup(grupo);
    }catch (SocketException e){System.out.println("Socket:" + e.getMessage( ));
    }catch (IOException e){System.out.println("IO:" + e.getMessage( ));}
  }
}
```

Figura 4.17. Un participante en multidifusión se apunta a un grupo y envía y recibe datagramas.

puedan detectar las sesiones de multidifusión existentes y anunciar su propia sesión, especificando el tiempo y la duración de la reserva (elige una dirección de multidifusión para nuevas sesiones).

◇ **Modelo de fallo para la multidifusión de datagramas.** La multidifusión de datagramas sobre la multidifusión IP tiene las mismas características de fallo que los datagramas UDP; esto es, sufren de fallos de omisión. El efecto en una multidifusión es que no se garantiza que los mensajes sean entregados a cualquier miembro particular del grupo en presencia de incluso un único fallo de omisión. Esto es, alguno, pero no todos los miembros del grupo pueden recibir el mensaje. Esto puede denominarse multidifusión *no fiable*, ya que no existe garantía de que un mensaje sea entregado a cualquier miembro del grupo. La multidifusión fiable se discutirá en el Capítulo 11.

◇ **API Java para la multidifusión IP.** El API Java para la multidifusión IP proporciona una interfaz de datagramas para la multidifusión IP a través de la clase *MulticastSocket*, que es una subclase de *DatagramSocket* con la capacidad adicional de ser capaz de pertenecer a grupos de multidifusión. La clase *MulticastSocket* proporciona dos constructores alternativos, permitiendo crear los conectores utilizando un número de puerto local concreto (por ejemplo el 6789, como en la Figura 4.17), o cualquier puerto local libre. Un proceso puede pertenecer a un grupo de multidifusión con una dirección de multidifusión dada invocando el método *joinGroup* de su conector de multidifusión. Como consecuencia, el conector pertenecerá a un grupo de multidifusión en un puerto dado y recibirá los datagramas enviados por los procesos en otros computadores a ese grupo en ese puerto. Un proceso puede dejar un grupo dado invocando el método *leaveGroup* de su conector multidifusión.

En el ejemplo de la Figura 4.17, los argumentos del método *main* especifican un mensaje a ser multidifundido y la dirección del grupo de multidifusión (por ejemplo, *«228.5.6.7»*). Después de

apuntarse a ese grupo, el proceso hace una instancia de *DatagramPacket* conteniendo el mensaje y lo envía a través de su conector multidifusión al grupo de multidifusión en el puerto 6789. Después de esto, intenta recibir tres mensajes de multidifusión de sus compañeros mediante su conector, el cual también pertenece al grupo en el mismo puerto. Cuando se ejecutan varias instancias de este programa simultáneamente en diferentes computadores, todos ellos pertenecen al mismo grupo, y cada uno de ellos debería recibir su propio mensaje y los mensajes enviados por lo demás miembros del grupo que se han apuntado después de él.

El API Java permite fijar el TTL para un conector multidifusión mediante el método *setTimeToLive*. El valor por defecto es 1, permitiendo que la multidifusión se propague sólo en la red local.

Una aplicación implementada sobre multidifusión IP puede utilizar más de un puerto. Por ejemplo, la aplicación MultiTalk [mbone], que permite que los miembros de su grupo mantengan conversaciones basadas en texto, tiene un puerto para enviar y recibir datos, y otro para intercambiar datos de control.

4.5.2. FIABILIDAD Y ORDEN EN MULTIDIFUSIÓN

La sección previa ha fijado el modelo de fallos de la multidifusión IP. Esto es, que sufre de fallos de omisión. En la multidifusión en redes de área local que utilizan las capacidades de multidifusión de la red para permitir que un único datagrama llegue a múltiples receptores, cualquiera de esos receptores puede perder un mensaje debido a que su búfer se encuentre lleno. También se puede perder un datagrama enviado de un router multidifusión a otro, impidiendo de este modo que todos los receptores más allá de ese router reciban el mensaje.

Otro factor es que cualquier proceso puede malograrse. Si falla un router de multidifusión, los miembros del grupo al otro lado del router no recibirán el mensaje multidifusión, aunque los miembros locales lo hagan.

Otra cuestión es el orden. Los paquetes enviados sobre IP en una interred no llegan necesariamente en el orden en que fueron enviados, con el efecto posible de que algunos miembros del grupo reciban los datagramas del mismo emisor en un orden diferente a otros miembros del grupo. Además, los mensajes enviados por dos procesos diferentes no llegarán necesariamente en el mismo orden a todos los miembros del grupo.

◇ **Algunos ejemplos de los efectos de la fiabilidad y del orden.** Consideremos ahora el efecto de los fallos de la multidifusión IP en cuatro ejemplos de uso de la replicación comentada en la Sección 4.5.

1. *Tolerancia a fallos basada en la replicación de servicios*: considérese un servicio replicado que consta de un grupo de servidores que arrancan en el mismo estado inicial y siempre ejecutan las mismas operaciones en el mismo orden, de forma que sean consistentes unos con otros. Estas aplicaciones de multidifusión necesitan que todas o ninguna de las réplicas reciban cada petición de operación, si una de ellas pierde la petición, se encontrará en un estado inconsistente con respecto a los demás. En la mayoría de los casos, se requerirá que todos los miembros reciban los mensajes de petición en el mismo orden.
2. *Búsqueda de los servidores de descubrimiento en redes espontáneas*: cualquier proceso que quiera localizar los servidores de descubrimiento manda una petición de multidifusión a intervalos periódicos después de arrancar, y una pérdida ocasional de una petición no representa un problema. De hecho, Jini utiliza multidifusión IP en su protocolo de multidifusión para encontrar los servidores de descubrimiento. Esto está descrito en Arnold y otros [1999].

3. *Mejores prestaciones mediante datos replicados*: considere el caso en el que los datos replicados son distribuidos a través de mensajes multidifusión, en lugar de distribuir las operaciones sobre los datos. El efecto de los mensajes perdidos y la inconsistencia en la ordenación dependerá del método de replicación y de la importancia de que todas las réplicas estén actualizadas. Por ejemplo, las réplicas de los grupos de noticias no tienen que ser necesariamente consistentes con las demás en cualquier instante de tiempo (los mensajes pueden incluso aparecer en orden diferente, pero los usuarios pueden asumirlo).
4. *Propagación de las notificaciones de eventos*: esta particular aplicación determina la calidad necesaria de la multidifusión. Por ejemplo, el servicio de anuncios de Jini informa a las partes interesadas sobre los nuevos servicios disponibles utilizando multidifusión IP haciendo anuncios a intervalos frecuentes.

Estos ejemplos sugieren que algunas aplicaciones requieren un protocolo de multidifusión más fiable que la multidifusión IP. En concreto, existe la necesidad de la *multidifusión fiable*; en la cual cualquier mensaje transmitido o bien es recibido por todos los miembros de un grupo, o no lo recibe ninguno de ellos. Los ejemplos también sugieren que algunas aplicaciones tienen requerimientos fuertes de ordenación, que en el caso más estricto se denomina *multidifusión totalmente ordenada*, en la cual todos los mensajes transmitidos a un grupo llegan a todos los miembros en el mismo orden.

El Capítulo 11 definirá y mostrará cómo implementar la multidifusión *fiable* y varias garantías de orden útiles, incluyendo la multidifusión totalmente ordenada.

4.6. CASO DE ESTUDIO: COMUNICACIÓN ENTRE PROCESOS EN UNIX

Las primitivas IPC en las versiones UNIX BSD 4.x son proporcionadas como llamadas al sistema que son implementadas como una capa sobre los protocolos Internet TCP y UDP. Los destinos de los mensajes se especifican como direcciones de conectores; una dirección de conector consta de una dirección Internet y un número de puerto local.

Las operaciones de comunicación entre procesos se basan en la abstracción de conectores descrita en la Sección 4.2.2. Como se describía allí, los mensajes se colocan en una cola en el conector emisor hasta que el protocolo de red los transmite, y allí permanecen hasta que llega el correspondiente reconocimiento, si es que el protocolo así lo requiere. Cuando llega un mensaje, se coloca en una cola en el conector receptor hasta que el proceso destinatario realiza la correspondiente llamada al sistema para recogerlo.

Cualquier proceso puede crear un conector para utilizarlo en la comunicación con otro proceso. Esto se realiza invocando a la llamada al sistema *socket*, cuyos argumentos especifican el dominio de comunicación (normalmente Internet), el tipo (datagrama o flujo) y en algunas ocasiones un protocolo particular. El protocolo (por ejemplo, TCP o UDP) es seleccionado normalmente por el sistema dependiendo de si el tipo de comunicación es de datagramas o de streams.

La llamada *socket* devolverá un descriptor mediante el cual refererirse al conector en las subsecuentes llamadas al sistema. El conector pervive hasta que se cierra (*close*) o mientras exista algún proceso con su descriptor. Se puede utilizar un par de conectores para la comunicación en ambas direcciones o en cualquier dirección entre procesos en el mismo o en computadores diferentes.

Antes de poder comunicar un par de procesos, el receptor debe *enlazar* su descriptor de conector con una dirección de conector. El emisor también debe enlazar su descriptor de conector con una dirección de conector si necesita responder. Para este propósito se utiliza la llamada al sistema *bind*; cuyos argumentos son el descriptor del conector y una referencia a una estructura que contiene la dirección del conector con la que se va a ligar el conector. Una vez que un conector está ligado a una dirección, ésta no puede cambiar.

Podría parecer más razonable tener una única llamada al sistema tanto para crear un conector como para ligarlo a una dirección, tal y como sucede en el API Java. La supuesta ventaja de tener dos llamadas separadas es que pueden ser útiles los conectores sin dirección.

Algunas direcciones son públicas en el sentido de que pueden ser utilizadas como destinos por cualquier proceso. Después de que un proceso ha ligado su conector a una dirección de conector, el conector puede ser apuntado indirectamente por otro proceso refiriéndose a la dirección de conector apropiada. Cualquier proceso, por ejemplo un servidor que planea recibir mensajes a través de su conector, debe primero enlazar ese conector a una dirección de conector y hacer que la dirección sea conocida por los potenciales clientes.

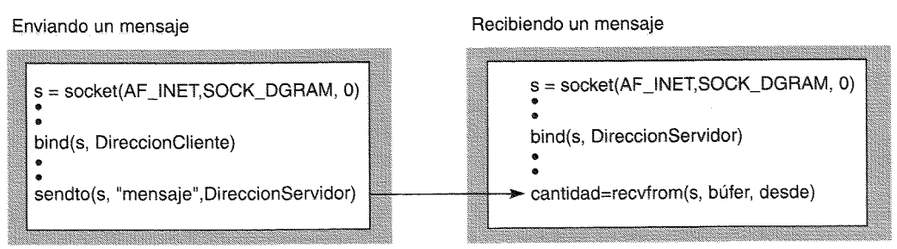
4.6.1. COMUNICACIÓN DE DATAGRAMAS

Para poder enviar datagramas, se identifica el par de conectores cada vez que se realiza una comunicación. Esto se logra en el proceso emisor utilizando su descriptor de conector local y la dirección del conector receptor cada vez que envía un mensaje.

Esto se muestra en la Figura 4.18, en la que se han simplificado los detalles de los argumentos.

- Ambos procesos utilizan la llamada *socket* para crear un conector y conseguir un descriptor para él. El primer argumento de *socket* especifica el dominio de comunicación como el dominio Internet, y el segundo argumento indica que se va a necesitar una comunicación de datagramas. El último argumento de la llamada *socket* puede utilizarse para especificar un protocolo particular, pero si se pone a cero produce que el sistema seleccione el protocolo adecuado (UDP en este caso).
- Ambos procesos, entonces, utilizan la llamada *bind* para enlazar sus conectores a las direcciones de conector. El proceso emisor enlaza su conector a una dirección de conector dada por cualquier puerto local libre. El proceso receptor enlaza su conector a una dirección de conector que contiene un puerto de servidor que debe ser conocido por el emisor.
- El proceso emisor utiliza la llamada *sendto*, con argumentos que especifican el socket a través del cual va a ser enviado el mensaje, el mensaje mismo y (una referencia a la estructura que contiene) la dirección del conector destino. La llamada *sendto* pasa el mensaje a las capas subyacentes UDP e IP y devuelve el número de bytes enviados. Como hemos solicitado un servicio de datagramas, el mensaje es transmitido a su destino sin acuse de recibo. Si el mensaje es demasiado grande para ser enviado, se produce un error que provoca la devolución del control (y el mensaje no será transmitido).
- El proceso receptor utiliza la llamada *recvfrom* con argumentos que especifican el conector local en el que se recibirá el mensaje, las direcciones de memoria en las que se almacenará el mensaje y (una referencia a la estructura que contiene) la dirección de conector del conector emisor. La llamada *recvfrom* tomará el primer mensaje en la cola del conector, o si la cola está vacía esperará a que llegue uno.

La comunicación se produce exclusivamente cuando un *sendto* en un proceso dirige su mensaje al conector utilizado por *recvfrom* en otro proceso. En una comunicación cliente-servidor, no existe necesidad de que los servidores tengan conocimiento previo de las direcciones de los conector de los clientes, ya que la operación *recvfrom* proporciona la dirección del emisor con cada mensaje que entrega. Las propiedades de la comunicación de datagramas en UNIX es la misma que la descrita en la Sección 4.2.3.



DireccionServidor y *DireccionCliente* son direcciones de conectores

Figura 4.18. Conectores utilizados para datagramas.

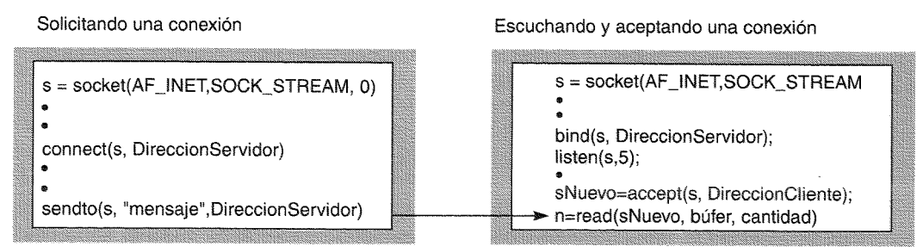
4.6.2. COMUNICACIÓN CON STREAMS

Antes de utilizar el protocolo de streams, dos procesos deben establecer una conexión entre sus pares de conectores. El acuerdo es asimétrico porque uno de los conectores estará escuchando peticones de conexión y el otro estará preguntando por una conexión, según se describe en la Sección 4.2.4. Una vez que un par de conectores han sido conectados, pueden utilizarse para transmitir datos en ambas o en cualquier dirección. Esto es, se comportan como streams en los que cualquier dato disponible se lee inmediatamente en el mismo orden en que fue escrito y no existen indicaciones de fronteras entre los mensajes. Sin embargo, existe una cola limitada en el conector receptor y el receptor se bloqueará si la cola está vacía; y el emisor se bloqueará si la cola está llena.

Para la comunicación entre clientes y servidores, los clientes solicitan conexiones y los servidores que escuchan esas peticiones las aceptan. Cuando una conexión es aceptada, automáticamente UNIX crea un nuevo conector y lo empareja con el conector del cliente de modo que el servidor puede continuar escuchando las peticiones de conexión de los demás clientes a través del conector original. Un par de conector de stream conectados se pueden utilizar en una comunicación de stream hasta que se cierre la conexión.

La comunicación de streams se muestra en la Figura 4.19, en la que se han simplificado los detalles relativos a los argumentos. La figura no muestra al servidor cerrando el conector en el que escucha. Normalmente, un servidor primero escucha y acepta una conexión y entontes crea un nuevo proceso para comunicarse con el cliente. Mientras tanto, continúa escuchando en el proceso original.

- El servidor, o proceso en escucha, utiliza primero la operación *socket* para crear un conector de flujo y después la operación *bind* para enlazar su conector a la dirección del conector de servidor. El segundo argumento de la llamada al sistema *socket* es SOCK_STREAM, para indicar que se requiere una comunicación de streams. El tercer argumento se deja a cero, de



DireccionServidor y *DireccionCliente* son direcciones de conectores

Figura 4.19. Conectores utilizados para streams.

modo que se seleccione el protocolo TCP/IP de forma automática. Utiliza la operación *listen* para escuchar en su conector las peticiones de conexión de los clientes. El segundo argumento de *listen* especifica el número máximo de peticiones de conexión que admite la cola de espera de ese conector.

- El servidor utiliza la llamada al sistema *accept* para aceptar una conexión solicitada por un cliente y obtener un nuevo conector para la comunicación con el cliente. El conector original continúa en uso para aceptar futuras peticiones de conexión de otros clientes.
- El proceso cliente utiliza la operación *socket* para crear un conector de stream y entonces utiliza la llamada al sistema *connect* para solicitar una conexión a través de la dirección de conector del proceso en escucha. Como la llamada al sistema *connect* enlaza automáticamente el conector con una dirección de conector, no es necesario un enlace previo.
- Después de que se ha establecido una conexión, ambos procesos pueden utilizar las operaciones *write* y *read* en sus respectivos conectores para enviar y recibir secuencias de bytes a través de la conexión. La operación *write* es similar a la operación de escritura en archivos. Especifica el mensaje a enviar al conector. Confía el mensaje a los protocolos TCP/IP subyacentes y devuelve el número de bytes enviados. La operación *read* recibe caracteres en su búfer y devuelve el número de bytes recibidos.

Las propiedades de la comunicación de streams en UNIX son las mismas que las descritas en la Sección 4.2.4.

4.7. RESUMEN

La primera sección de este capítulo muestra que existen dos alternativas a la hora de construir los bloques con los que elaborar los protocolos de Internet. Existe una interesante relación de compromiso entre estos dos protocolos: UDP proporciona la posibilidad de un simple paso de mensajes que adolece de fallos de omisión pero no lleva asociada penalización en las prestaciones. En el otro lado, en buenas condiciones, TCP garantiza la entrega de mensajes pero a cambio de tener mensajes adicionales y una mayor latencia y costos de almacenamiento.

La segunda sección aborda dos estilos alternativos de empaquetamiento de datos. CORBA y sus predecesores eligen empaquetar los datos que serán utilizados por receptores con un conocimiento previo de sus componentes. Por el contrario, cuando Java serializa datos, incluye la descripción completa del contenido, permitiendo que los receptores puedan reconstruirlo exclusivamente a partir del contenido. Otra gran diferencia es que CORBA necesita una especificación de los tipos de datos a ser empaquetados (en IDL) para generar una especificación de los tipos de datos de los elementos de información a empaquetar, mientras que Java utiliza la reflexión tanto para serializar datos como para deserializarlos a partir de la forma serializada.

La sección dedicada a los protocolos petición-respuesta muestra que se puede construir un protocolo específico para sistemas distribuidos efectivo basándose sobre datagramas UDP. El mensaje de respuesta se considera como un reconocimiento del mensaje de petición, evitando de este modo las sobrecargas de los mensajes de reconocimiento adicionales. El protocolo se puede hacer más fiable si fuera necesario. Tal como es, no existe garantía de que el envío de un mensaje de petición desencadene la ejecución de un método (para algunas aplicaciones esto puede resultar suficiente). Pero se puede conseguir una fiabilidad adicional haciendo uso de la identificación de los mensajes y de la retransmisión de los mensajes, de modo que nos aseguremos de que los métodos serán ejecutados. Para servicios con operaciones idempotentes, esto es suficiente. Sin embargo, otras aplicaciones necesitan que los mensajes de respuesta sean transmitidos sin tener que volver a ejecutar el método solicitado. Esto se puede conseguir utilizando un historial. Esto demuestra que resulta una buena idea construir distintos protocolos que se ajusten a las necesidades de diferentes clases de aplicaciones en lugar de construir un único protocolo superfiable para su uso general, ya que esto último puede conducir a peores prestaciones en el caso de uso general, en el que raramente se dan fallos.

Los mensajes de multidifusión se utilizan en la comunicación entre los miembros de un grupo de procesos. El protocolo multidifusión IP proporciona un servicio de multidifusión tanto para redes de área local como para Internet. Esta forma de multidifusión tiene el mismo esquema de fallos que los datagramas UDP, pero aunque sufra de fallos de omisión es una herramienta útil para muchas aplicaciones de multidifusión. Otras aplicaciones tienen unos requisitos más exigentes; en particular aquellas en la que la multidifusión tiene que ser atómica, esto es, debe producirse una entrega a todos o a ninguno. Los requisitos más restrictivos relacionados con la multidifusión están asociados a la entrega ordenada de los mensajes, el más exigente de todos exige que todos los miembros del grupo reciban los mensajes en el mismo orden.

EJERCICIOS

4.1. ¿Resulta razonablemente útil que un puerto tenga varios receptores?

4.2. Un servidor crea un puerto que utiliza para recibir peticiones de sus clientes. Discuta los problemas de diseño concernientes a las relaciones entre el nombre de este puerto y los nombres utilizados por los clientes.

4.3. Los programas de la Figuras 4.3 y 4.4 están disponibles en cdk3.net/ipc. Utilícelos para hacer un prototipo que pruebe las condiciones en las que los datagramas son, en ocasiones, desechados. Consejo: el programa cliente debería ser capaz de variar el número y el tamaño de los mensajes que envía; el servidor debería detectar cuándo se ha perdido un mensaje de un cliente particular.

4.4. Utilice el programa de la Figura 4.3 para construir un programa cliente que lea repetidamente un línea de entrada del usuario, la envíe a un servidor en un datagrama UDP, y entonces reciba un mensaje del servidor. El cliente asociará un tiempo de espera límite con su conector, de modo que pueda informar al usuario cuando el servidor no responda. Pruebe este cliente con el servidor de la Figura 4.4.

4.5. Los programas de las Figuras 4.5 y 4.6 están disponibles en cdk3.net/ipc. Modifíquelos de modo que el cliente obtenga de forma repetida una línea del usuario y la escriba en el flujo que el servidor leerá repetidamente, imprimiendo el resultado de la lectura. Compare los datos enviados utilizando un datagrama UDP y un stream.

4.6. Utilice los programas desarrollados en el Ejercicio 4.5 para probar el efecto causado sobre el emisor cuando el receptor se cae y viceversa.

4.7. El protocolo Sun XDR empaqueta los datos convirtiéndolos al estándar big-endian antes de la transmisión. Discuta las ventajas e inconvenientes de este método comparándolo con el CDR de CORBA.

4.8. El protocolo Sun XDR alinea cada valor primitivo en posiciones múltiplos de cuatro bytes, mientras que el CORBA CDR alinea un valor primitivo de tamaño n en posiciones múltiplo de n. Discuta ambos métodos con respecto a la elección de los tamaños ocupados por los valores primitivos.

4.9. ¿Por qué no existe información explícita sobre los tipos de datos en CORBA CDR?

4.10. Escriba un algoritmo en pseudocódigo para describir el procedimiento de serialización descrito en la Sección 4.3.2. El algoritmo debería mostrar cuándo son definidos o sustituidos los apuntadores para las clases y las instancias. Describa la forma serializada que su algoritmo producirá al serializar una instancia de la siguiente clase *Pareja*.

```
clase Pareja implements Serializable{
    private Persona uno;
    private Persona dos;
    public Pareja(Persona a, Persona b) {
        uno = a;
        dos = b;
    }
}
```

4.11. Escriba un algoritmo en pseudocódigo para describir la deserialización de la forma serializada producida por el algoritmo definido en el Ejercicio 4.10. Consejo: utilice la reflexión para crear una clase a partir de su nombre, para crear un constructor a partir de los tipos de sus parámetros y para crear una nueva instancia de un objeto a partir del constructor y de los valores de los argumentos.

4.12. Defina una clase cuyas instancias representen referencias a objetos remotos. Debería contener información similar a la mostrada en la Figura 4.10 y debería proveer de los métodos de acceso necesitados por el protocolo petición-respuesta. Explique cómo serán utilizados cada uno de los métodos de acceso por el protocolo. Justifique el tipo elegido para el campo que contiene información sobre la interfaz del objeto remoto.

4.13. Defina una clase cuyas instancias representen los mensajes de petición y respuesta mostrados en la Figura 4.13. La clase debería poseer un par de constructores, uno para los mensajes de petición y otro para los de respuesta, mostrando el modo en que se asigna el identificador de petición. También debería proporcionar un método para empaquetarse a sí misma en una cadena de bytes y para desempaquetarse desde la cadena de bytes y convertirse en una instancia.

4.14. Programe cada una de las tres operaciones del protocolo petición-respuesta de la Figura 4.12, utilizando comunicación UDP, pero sin añadir ninguna medida para conseguir tolerancia a fallos. Debería utilizar las clases definidas en los Ejercicios 4.12 y 4.13.

4.15. Muestre un esquema de implementación de un servidor mostrando el modo en que se utilizan las operaciones *damePeticion* y *envíaRespuesta* por un servidor que crea un nuevo hilo para ejecutar cada petición de un cliente. Indique el modo en que el servidor copiará la *idPeticion* del mensaje de petición en el mensaje de respuesta y cómo obtendrá la dirección IP y el puerto del cliente.

4.16. Defina una nueva versión del método *hazOperacion* que establezca un timeout sobre la espera del mensaje de respuesta. Después de cumplido este tiempo, retransmitirá el mensaje de petición *n* veces. Si todavía no recibe ninguna respuesta, informará al invocador.

4.17. Describa un escenario en el cual un cliente pudiera recibir una respuesta de una llamada anterior.

4.18. Describa los modos en el que los protocolos petición-respuesta ocultan la heterogeneidad de los sistemas operativos y de las redes de computadores.

4.19. Discuta si las siguientes operaciones son idempotentes o no:

- Apretar el botón de solicitud de un ascensor.
- Escribir datos en un archivo.
- Añadir datos a un archivo.

¿Puede ser una condición necesaria para la idempotencia el que la operación no deba estar asociada con ningún estado?

4.20. Explique las opciones de diseño que son relevantes para minimizar la cantidad de datos de respuesta almacenados en un servidor. Compare las necesidades de almacenamiento cuando se utilizan protocolos RR y RRA.

4.21. Suponga que se está utilizando el protocolo RRA. ¿Cuánto tiempo deberían retener los servidores los mensajes de datos respuesta no reconocidos? ¿Deberían los servidores enviar las respuestas repetidamente en un intento de recibir un reconocimiento?

4.22. ¿Por qué el número de mensajes intercambiados en un protocolo puede tener más influencia en las prestaciones que la cantidad de datos enviados? Diseñe una variante del protocolo RRA en el que los acuses de recibo vayan adheridos, esto es, transmitidos en el mismo mensaje que la siguiente petición que resulte apropiada, y de otro modo enviados como un mensaje separado. (Consejo: utilice un cronómetro extra en el cliente.)

4.23. La multidifusión IP proporciona un servicio que adolece de fallos de omisión. Construya un prototipo (puede estar basado en el programa de la Figura 4.17) con el que descubrir las condiciones bajo las cuales un mensaje de multidifusión es desechado a veces por uno de los miembros del grupo. El prototipo debería estar diseñado para permitir procesos emisores múltiples.

4.24. Esboce el diseño de un esquema que utiliza la retransmisión de mensajes con multidifusión IP para solucionar el problema de los mensajes perdidos. Su esquema deberá tener en cuenta los siguientes puntos:

i) Existen múltiples emisores.
ii) Generalmente sólo una pequeña porción de mensajes son desechados.
iii) En contra de lo que sucede en el protocolo petición-respuesta, los receptores no tienen necesariamente que enviar un mensaje dentro de un rango de tiempo.

Suponga que los mensajes no desechados llegan en el orden de emisión.

4.25. Su solución al Ejercicio 4.24 debería haber solucionado el problema de los mensajes desechados en la multidifusión IP. ¿En qué sentido su solución difiere de la definición de multidifusión fiable?

4.26. Contemple un escenario en el cual las multidifusiones enviadas por clientes diferentes son entregadas en órdenes diferentes a los miembros de dos grupos. Suponga que se está utilizando alguna forma de retransmisión de mensajes, pero que esos mensajes que no son desechados llegan en el orden de emisión. Sugiera el modo en que los receptores pueden remediar esta situación.

4.27. Defina la semántica y diseñe un protocolo para una forma de interacción petición-respuesta para un grupo, utilizando por ejemplo multidifusión IP.

5

OBJETOS DISTRIBUIDOS E INVOCACIÓN REMOTA

5.1. Introducción
5.2. Comunicación entre objetos distribuidos
5.3. Llamada a un procedimiento remoto
5.4. Eventos y notificaciones
5.5. El caso de estudio Java RMI
5.6. Resumen

En este capítulo presentamos la comunicación entre objetos distribuidos mediante invocación de métodos remotos (RMI). Los objetos que pueden recibir invocaciones de métodos remotos se denominan objetos remotos e implementan una interfaz remota. Debido a la posibilidad de fallo independiente por parte del que invoca y los objetos invocados, RMI tiene una semántica de llamada diferente de la de las llamadas locales. Pueden realizarse de modo muy parecido a las invocaciones locales, pero la transparencia total no tiene por qué ser necesariamente deseable. El código para empaquetar y desempaquetar los argumentos y el envío de los mensajes de petición y respuesta puede generarse automáticamente por un compilador de interfaces desde la definición de la interfaz remota.

La llamada a un procedimiento remoto es a RMI lo que una llamada a procedimiento es a una invocación a un objeto. Se describe brevemente y se ilustra mediante un caso de estudio con Sun RPC.

Los sistemas distribuidos basados en eventos permiten a los objetos suscribirse a eventos que ocurren en objetos remotos de interés y volver en sí para recibir las notificaciones cuando ocurren tales eventos. Los eventos y las notificaciones proporcionan una forma de comunicación entre objetos heterogéneos de modo asíncrono. La especificación de eventos distribuidos Jini se presenta como un caso de estudio.

El empleo de RMI se ilustra en un caso de estudio con Java RMI.

El Capítulo 17 contiene un caso de estudio sobre CORBA que incluye CORBA RMI y el Servicio de Eventos de CORBA.

5.1. INTRODUCCIÓN

Este capítulo trata de los modelos de programación para aplicaciones distribuidas; esto es, aquellas aplicaciones que se componen de programas cooperantes corriendo en procesos distintos. Tales programas necesitan ser capaces de invocar operaciones en otros procesos, que a menudo residen en computadores diferentes. Para lograr esto, algunos modelos de programación familiares han sido extendidos para aplicarlos a los programas distribuidos:

- El primero, y quizás mejor conocido de entre ellos, fue la extensión del modelo de llamada a procedimiento convencional al modelo de *llamada a procedimiento remoto*, que permite a programas cliente llamar a procedimientos de programas servidores lanzados en procesos separados, y generalmente en computadores distintos al del cliente.
- Más recientemente, el modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una *invocación a un método remoto* (RMI, *remote method invocation*). RMI es una extensión de la invocación a métodos locales que permite que un objeto que vive en un proceso invoque los métodos de un objeto que reside en otro proceso.
- El modelo de programación basado en eventos permite a los objetos recibir notificaciones de los eventos que ocurren en otros objetos en los que se ha manifestado el interés. Este modelo ha sido extendido para permitir escribir programas distribuidos basados en eventos.

Nótese que usamos el término *RMI* para referirnos a una invocación de un método remoto de forma genérica; no debemos confundir esto con ejemplos particulares de invocación de un método remoto como Java RMI. La mayoría del software de sistemas distribuidos actual se escribe en lenguajes con orientación a objetos, y RPC puede entenderse con relación a RMI. Así este capítulo se concentra en los paradigmas basados en RMI y eventos, aplicables a objetos distribuidos. La comunicación entre objetos distribuidos se presenta en la Sección 5.2, seguida por una discusión del diseño e implementación de RMI. En la Sección 5.5 se da un caso de estudio sobre Java RMI. RPC se discute en el contexto de un caso de estudio de Sun RPC en la Sección 5.3. Los eventos y las notificaciones distribuidas se discuten en la Sección 5.4. En el Capítulo 17 se dará un caso de estudio más sobre CORBA.

◇ **Middleware.** Al software que proporciona un modelo de programación sobre bloques básicos arquitectónicos, a saber: procesos y paso de mensajes, se le denomina middleware. La capa de middleware emplea protocolos basados en mensajes entre procesos para proporcionar abstracciones de un nivel mayor, tales como invocaciones remotas y eventos, como podemos ver en la Figura 5.1. Por ejemplo, la abstracción de invocación de un método remoto se basa en el protocolo petición-respuesta discutido en la Sección 4.4.

Un aspecto importante del middleware es que proporciona transparencia de ubicación e independencia de los detalles de los protocolos de comunicación, los sistemas operativos y el hardware

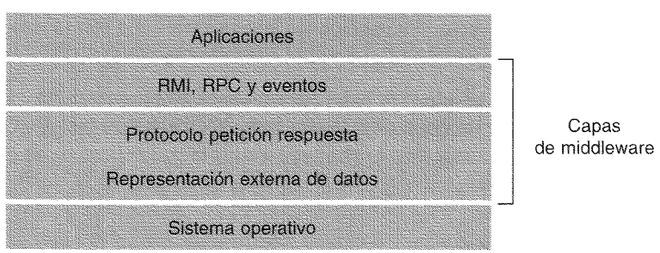


Figura 5.1. Capa de middleware.

ware de los computadores. Algunas formas de middleware permiten que los componentes separados estén escritos en diferentes lenguajes de programación.

Transparencia frente a ubicación: En RPC, el cliente que llama a un procedimiento no puede discernir si el procedimiento se ejecuta en el mismo proceso o en un proceso diferente, posiblemente en otro computador. El cliente tampoco necesita conocer la ubicación del servidor. Análogamente, en RMI el objeto que realiza la invocación no podrá decirnos si el objeto que invoca es local o no, y tampoco requiere conocer su ubicación. Asimismo, en los programas distribuidos basados en eventos, los objetos que generan eventos y los objetos que reciben notificaciones de esos eventos tampoco necesitan estar al corriente de sus ubicaciones respectivas.

Protocolos de comunicación: Los protocolos que dan soporte a las abstracciones del middleware son independientes de los protocolos de transporte subyacentes. Por ejemplo el protocolo petición-respuesta puede estar implementado tanto sobre UDP como sobre TCP.

Hardware de los computadores: En la Sección 4.3 se describen dos estándares comunes para la representación externa de datos. Se emplean en el empaquetado y desempaquetado de mensajes. Éstos ocultan las diferencias de arquitectura en el hardware, como el ordenamiento de los bytes.

Sistemas operativos: Las abstracciones de mayor nivel que provee la capa de middleware son independientes de los sistemas operativos subyacentes.

Utilización de diversos lenguajes de programación: Diversos middleware se diseñan para permitir que las aplicaciones distribuidas sean escritas en más de un lenguaje de programación. En particular CORBA (véase el Capítulo 17) permite a los clientes escritos en un lenguaje invocar métodos en objetos que viven en programas servidores escritos en otro lenguaje. Esto se obtiene empleando un *lenguaje de definición de interfaz* o IDL (*interface definition language*) para definir interfaces. IDL se discute en la siguiente sección.

5.1.1. INTERFACES

La mayoría de los lenguajes de programación modernos proporcionan medios para organizar un programa en conjuntos de módulos que puedan comunicarse unos con otros. La comunicación entre los módulos se puede realizar mediante llamadas a procedimientos entre los módulos o accediendo directamente a las variables de otro módulo. Para controlar las interacciones posibles entre los módulos, se define explícitamente una *interfaz* para cada módulo. Los módulos se implementan de forma que se oculte toda la información excepto aquella que se haga disponible a través de su interfaz. De este modo, mientras la interfaz permanezca inalterada, la implementación podrá cambiar sin afectar a los usuarios del módulo.

◇ **Las interfaces en los sistemas distribuidos.** En un programa distribuido, los módulos pueden lanzarse en procesos separados. No es posible para un módulo que se ejecuta en un proceso acceder a las variables de un módulo que está en otro proceso. Asimismo, la interfaz de un módulo escrita para RPC o RMI no puede especificar el acceso directo a variables. Advierta que las interfaces en IDL de CORBA pueden especificar atributos, lo que parece violar esta regla. Sin embargo, los atributos no son accedidos directamente sino mediante ciertos procedimientos de escritura y lectura que se añaden automáticamente a la interfaz.

Los mecanismos de paso de parámetros, por ejemplo la llamada por valor y la llamada por referencia, utilizados en las llamadas a procedimientos locales, no son adecuados cuando el que llama y el procedimiento llamado están en procesos diferentes. La especificación de un método o procedimiento en la interfaz de un módulo en un programa distribuido describe los parámetros como *entrada* o *salida* o, a veces, ambos. Los parámetros de *entrada* se pasan al módulo remoto mediante el envío de los valores de los argumentos en el mensaje de petición y posteriormente se

proporcionan como argumentos a la operación que se ejecutará en el servidor. Los parámetros de *salida* se devuelven en el mensaje de respuesta y se sitúan como la respuesta de la llamada o reemplazando los valores de las correspondientes variables argumento en el entorno de la llamada. Cuando se proporciona un parámetro tanto como para entrada como para salida, el valor debe transmitirse tanto en los mensajes de la petición como en los de la respuesta.

Otra diferencia entre los módulos locales y remotos es que los punteros en un proceso dejan de ser válidos en el remoto. En consecuencia, no pueden pasarse punteros como argumentos o como valores retornados como resultado de las llamadas a los módulos remotos.

Los próximos dos párrafos discuten las interfaces empleadas en el modelo cliente-servidor original para RPC y en el modelo de objetos distribuidos para RMI:

Interfaces de servicio: En el modelo cliente-servidor, cada servidor proporciona un conjunto de procedimientos disponibles para su empleo por los clientes. Por ejemplo, un servidor de archivos proporcionará procedimientos para leer y escribir archivos. El término *interfaz de servicio* se emplea para referirse a la especificación de los procedimientos que ofrece un servidor, y define los tipos de los argumentos de entrada y salida de cada uno de los procedimientos.

Interfaces remotas: En el modelo de objetos distribuidos, una *interfaz remota* especifica los métodos de un objeto que están disponibles para su invocación por objetos de otros procesos, y define los tipos de los argumentos de entrada y de salida de cada uno de ellos. Sin embargo, la gran diferencia es que los métodos en las interfaces remotas pueden pasar objetos como argumentos y como resultados de los métodos. Además, también se pueden pasar referencias a objetos remotos; lo cual no debe confundirse con los punteros, que se refieren a ubicaciones específicas de la memoria. (La Sección 4.3.3 describía los contenidos de las referencias de los objetos remotos.)

Ni las interfaces de servicio ni las interfaces remotas pueden especificar un acceso directo a variables. En este último, está prohibido el acceso directo a las variables de las instancias de un objeto.

◇ **Lenguajes de definición de interfaces.** Se puede integrar un mecanismo RMI con un lenguaje de programación concreto si incluye una notación apropiada para definir interfaces, que permita relacionar los parámetros de entrada y de salida con el uso habitual de los parámetros en ese lenguaje. Java RMI es un ejemplo en el que se ha añadido un mecanismo RMI a un lenguaje de programación con orientación a objeto. Esta aproximación es útil cuando se puede escribir cada parte de una aplicación distribuida en el mismo lenguaje. También es conveniente dado que permite al programador emplear un solo lenguaje para las invocaciones locales y remotas.

Sin embargo, muchos servicios útiles se encuentran escritos en C++ y otros lenguajes. Sería beneficioso, que pudieran acceder a ellos remotamente, muchos otros programas escritos en gran variedad de lenguajes, incluyendo Java. Los *lenguajes de definición de interfaces* (IDL) están diseñados para permitir que los objetos implementados en lenguajes diferentes se invoquen unos a otros. Un IDL proporciona una notación para definir interfaces en la cual cada uno de los parámetros de un método se podrá describir como de *entrada* o de *salida* además de su propia especificación de tipo.

La Figura 5.2 muestra un ejemplo simple en IDL de CORBA. La estructura *Persona* es la misma que la empleada para ilustrar el empaquetado en la Sección 4.3.1. La interfaz denominada *ListaPersonas* especifica los métodos disponibles para el RMI en un objeto remoto que implemente esa interfaz. Por ejemplo el método *añadePersona* especifica su argumento como *in*, que significa que es un argumento de *entrada*; y el método *damePersona* que recupera una instancia de *Persona* mediante *nombre* especifica su segundo argumento como *out*, que significa que es un argumento de *salida*. Nuestros casos de estudio incluyen CORBA IDL como un ejemplo de un IDL para RMI (véase el Capítulo 17) y Sun XDR como un IDL para RPC.

Otros ejemplos incluyen el lenguaje de definición de interfaz para el sistema RPC en el Entorno de Computación Distribuida de OSF (DCE, *Distributed Computing Environment*) [OSF 1997],


```
// En el archivo Persona.idl
struct Persona {
    string nombre;
    string lugar;
    long año;
};
interface ListaPersonas {
    readonly attribute string nombrelista;
    void añadePersona(in Persona p);
    void damePersona(in string nombre, out Persona p);
    long número();
};
```

Figura 5.2. Ejemplo en CORBA IDL.

que emplea la sintaxis del lenguaje C y se denomina IDL; y DCOM IDL que se basa en DCE IDL [Box 1998] y que se emplea en el Modelo de Objetos Componentes Distribuido (*Distributed Component Object Model*) de Microsoft.

5.2. COMUNICACIÓN ENTRE OBJETOS DISTRIBUIDOS

El modelo basado en objetos para un sistema distribuido presentado en el Capítulo 1 extiende el modelo soportado por los lenguajes de programación con orientación al objeto para hacerlo aplicable a los objetos distribuidos. Esta sección aborda el tema de la comunicación entre objetos distribuidos mediante RMI. El material se presenta bajo los siguientes encabezados:

El modelo de objeto: una breve revisión de los aspectos relevantes del modelo de objetos, adecuado para el lector con un conocimiento básico de un lenguaje de programación con orientación al objeto, por ejemplo Java o C++.

Objetos distribuidos: una presentación de los sistemas distribuidos basados en objetos, donde se argumenta la conveniencia del modelo de objetos para los sistemas distribuidos.

El modelo de objetos distribuido: una discusión de las extensiones necesarias del modelo de objetos para dar soporte a los objetos distribuidos.

Cuestiones de diseño: un conjunto de razonamientos sobre las alternativas de diseño:

1. Las invocaciones locales se ejecutan exactamente una vez, pero, ¿qué semánticas de invocación adecuadas son posibles para las invocaciones remotas?
2. ¿Cómo puede hacerse que la semántica de un RMI sea similar a la invocación de métodos locales, y qué diferencias no podemos eliminar?

Implementación: una explicación de cómo podría diseñarse una capa de middleware sobre el protocolo petición-respuesta para dar soporte a RMI entre aplicaciones con objetos distribuidos al nivel de aplicación.

Compactación automática de la memoria distribuida: una presentación de un algoritmo para la compactación automática de la memoria (*garbage collection*) que sea adecuada para su uso con la implementación de RMI.

5.2.1. EL MODELO DE OBJETOS

Un programa orientado al objeto, por ejemplo en Java o C++, consta de un conjunto de objetos que interaccionan entre ellos, cada uno de los cuales consiste en un conjunto de datos y un conjun-

to de métodos. Un objeto se comunica con otro objeto invocando sus métodos, generalmente pasándole argumentos y recibiendo resultados. Los objetos pueden encapsular sus datos y el código de sus métodos. Algunos lenguajes, por ejemplo Java y C++, permiten que los programadores definan objetos en los que las variables de sus instancias estén accesibles de modo directo. Pero para su uso en un sistema distribuido de objetos, los datos de un objeto sólo deberían ser accesibles mediante sus métodos.

◇ **Referencias a objetos.** Se puede acceder a los objetos mediante referencias de objetos. Por ejemplo, en Java, una variable que aparente contener un objeto en realidad maneja una referencia a ese objeto. Para invocar un método en un objeto, se proporciona la referencia del objeto y el nombre del método, junto a cualquier argumento que sea necesario. El objeto cuyo método se invoca se denomina unas veces el *objetivo* (*target*) y otras el *receptor* (*receiver*). Las referencias a objetos son valores de primera clase, lo que significa que, por ejemplo, pueden ser asignadas a variables, pasadas como argumentos y devueltas como resultados de métodos.

◇ **Interfaces.** Una interfaz proporciona una definición de las signaturas de un conjunto de métodos (es decir, los tipos de sus argumentos, valores devueltos y excepciones) sin especificar su implementación. Un objeto proporcionará una interfaz particular si su clase contiene código que implemente los métodos de esa interfaz. En Java, una clase puede implementar varias interfaces, y los métodos de una interfaz pueden ser implementados por cualquier clase. Una interfaz también define un tipo que puede usarse para declarar el tipo de las variables o de los parámetros y valores devueltos de los métodos. Observe que las interfaces no tienen constructores.

◇ **Acciones.** La acción en un programa orientado al objeto se inicia en un objeto que invoca un método de otro objeto. Una invocación puede incluir información adicional (argumentos) necesaria para llevar a cabo el método. El receptor ejecuta el método apropiado y entonces devuelve el control al objeto que lo invoca, a veces aportando un resultado. Una invocación a un método puede tener dos efectos:

1. Puede cambiar el estado del receptor.
2. Pueden tener lugar más invocaciones sobre métodos de otros objetos.

Dado que una invocación puede traer consigo más invocaciones de métodos en otros objetos, una acción es una cadena de invocaciones de métodos relacionados, cada uno de los cuales retorna eventualmente. Esta explicación no tiene en cuenta las excepciones.

◇ **Excepciones.** Los programas pueden encontrarse muchos tipos de errores y condiciones inesperadas de diversa gravedad. Durante la ejecución de un método, pueden descubrirse muchos problemas diferentes: por ejemplo, valores inconsistentes en las variables de un objeto, o fallo en los intentos de leer o escribir sobre archivos o sockets de red. Cuando los programadores necesitan insertar comprobaciones en su código para tratar con todos los casos inusuales o erróneos posibles, se hace a costa de la claridad del código del caso normal. Las excepciones proporcionan una forma limpia de tratar con las condiciones de error sin complicar el código. Además, cada cabecera de método lista explícitamente como excepciones las condiciones de error que pudiera encontrar, permitiendo a los usuarios del método tratar con ellas. Puede decirse que un bloque de código lanza (*throw*) una excepción cuando quiera que aparezcan condiciones o errores inesperados. Esto conlleva que el control pase a otro bloque de código que capture (*catch*) la excepción. El control no vuelve al lugar donde se lanzó la excepción.

◇ **Compactación automática de la memoria.** Se hace necesario proporcionar mecanismos de liberación del espacio ocupado por aquellos objetos que ya no se necesitan. Un lenguaje, por ejemplo Java, que pueda detectar automáticamente cuándo un objeto ya no es accesible puede recuperar el espacio y ponerlo a disposición de su asignación para otros objetos. Este proceso se denomina *compactación automática de la memoria*. Cuando un lenguaje (por ejemplo C++) no da soporte para la compactación automática de la memoria, el programador debe encargarse de la liberación del espacio asignado a los objetos. Ésta puede ser una gran fuente de errores.

5.2.2. OBJETOS DISTRIBUIDOS

El estado de un objeto consta de los valores de sus variables de instancia. En el paradigma de la programación basada en objetos, el estado de un programa se encuentra fraccionado en partes separadas, cada una de las cuales está asociada con un objeto. Dado que los programas basados en objetos están fraccionados lógicamente, la distribución física de los objetos en diferentes procesos o computadores de un sistema distribuido es una extensión natural.

Los sistemas de objetos distribuidos pueden adoptar la arquitectura cliente-servidor. Es este caso los objetos están gestionados por servidores, y sus clientes invocan sus métodos utilizando una invocación de métodos remota. En RMI, la petición del cliente de invocación de un método de un objeto se envía en un mensaje al servidor que gestiona el objeto. La invocación se lleva a cabo ejecutando un método del objeto en el servidor y el resultado se devuelve al cliente en otro mensaje. Para permitir las cadenas de invocaciones relacionadas, se permite que los objetos en los servidores se conviertan en clientes de objetos de otros servidores.

Los objetos distribuidos pueden asumir los otros modelos arquitectónicos que se han descrito en el Capítulo 2. Por ejemplo, los objetos pueden estar replicados para obtener los beneficios usuales de tolerancia a fallos y mejora de prestaciones, y los objetos pueden migrar con vista a mejorar sus prestaciones y disponibilidad.

El disponer de objetos clientes y servidores en diferentes procesos promueve la encapsulación. Esto es, el estado de un objeto está accesible sólo para los métodos del objeto, lo que quiere decir que no es posible que los métodos no autorizados actúen sobre el estado. Por ejemplo, la posibilidad de RMI concurrente desde objetos en diferentes computadores implica que se pueda acceder concurrentemente a un objeto. De este modo, aparece la posibilidad de accesos conflictivos.

Sin embargo, el hecho de que los datos sobre un objeto sólo sean accesibles por sus propios métodos permite a los objetos proporcionar métodos para protegerse contra accesos incorrectos. Por ejemplo, se puede emplear primitivas de sincronización tales como variables de condición para proteger el acceso a sus variables de instancia.

Otra ventaja de tratar el estado compartido de un programa distribuido como un conjunto de objetos es que se pueda acceder a un conjunto de objetos vía RMI o también copiarse en una caché local y ser accedidos directamente, dado que la implementación de la clase está disponible localmente.

El hecho de que los objetos sean accedidos únicamente mediante sus métodos nos da otra ventaja para los sistemas heterogéneos en los que se pueden usar diferentes formatos de datos en diferentes lugares; estos formatos pasarán inadvertidos para aquellos clientes que usan RMI para acceder a los métodos de los objetos.

5.2.3. EL MODELO DE OBJETOS DISTRIBUIDO

Esta sección discute extensiones al modelo de objetos para hacerlo aplicable a los objetos distribuidos. Cada proceso contiene un conjunto de objetos, algunos de los cuales pueden recibir tanto invocaciones locales como remotas, mientras que los otros objetos sólo pueden recibir invocaciones locales, como se muestra en la Figura 5.3. Las invocaciones de métodos entre objetos en diferentes procesos, tanto si es en el mismo computador o no, se conocen como *invocaciones de métodos remotas*. Las invocaciones de métodos entre objetos del mismo proceso son invocaciones de métodos locales.

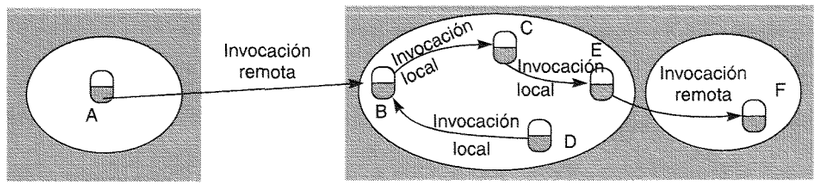


Figura 5.3. Invocaciones de métodos locales y remotas.

A los objetos que pueden recibir invocaciones remotas los conocemos como *objetos remotos*. En la Figura 5.3, los objetos B y F son objetos remotos. Todos los objetos pueden recibir invocaciones locales, a pesar de que sólo puedan recibirlas de otros objetos que posean referencias a ellos. Por ejemplo, el objeto C puede tener una referencia al objeto E de modo que puede invocar uno de sus métodos. Los dos conceptos fundamentales siguientes son el corazón del modelo de objetos distribuidos:

Referencia de objeto remoto: otros objetos pueden invocar los métodos de un objeto remoto si tienen acceso a su *referencia de objeto remoto*. Por ejemplo, en la Figura 5.3 deberá estar disponible para A una referencia a un objeto remoto sobre B.

Interfaz remota: cada objeto remoto tiene una *interfaz remota* que especifica cuáles de sus métodos pueden invocarse remotamente. Por ejemplo, los objetos B y F deben tener interfaces remotas.

Los siguientes párrafos discuten las referencias a objetos remotos, las interfaces remotas y otros aspectos sobre el modelo de objetos distribuido.

◇ **Referencias a objetos remotos.** La noción de referencia a objeto se extiende para permitir que cualquier objeto que pueda recibir un RMI tenga una referencia a objeto remoto. Una referencia a objeto remoto es un identificador que puede usarse a lo largo de todo un sistema distribuido para referirse a un objeto remoto particular único. Su representación, que generalmente es diferente de la de las referencias a objetos locales, se discute en la Sección 4.3.3. Las referencias a objetos remotos son análogas a las locales en cuanto a que:

1. El objeto remoto donde se recibe la invocación de método remoto se especifica mediante una referencia a objeto remoto.
2. Las referencias a objetos remotos pueden pasarse como argumentos y resultados de las invocaciones de métodos remotos.

◇ **Interfaces remotas.** La clase de un objeto remoto implementa los métodos de su interfaz remota, por ejemplo las instancias públicas en Java. Los objetos en otros procesos pueden invocar solamente los métodos que pertenezcan a su interfaz remota, como se muestra en la Figura 5.4. Los objetos locales pueden invocar los métodos en la interfaz remota así como otros métodos implementados por un objeto remoto.

El sistema CORBA proporciona un lenguaje de definición de interfaces (IDL), que permite definir interfaces remotas. En la Figura 5.2 se muestra un ejemplo de una interfaz remota definida en CORBA IDL. Las clases de los objetos remotos y los programas de los clientes pueden implementarse en cualquier lenguaje como C++ o Java para lo cual se dispone de un compilador IDL. Los clientes CORBA no necesitan emplear el mismo lenguaje que el objeto remoto para invocar sus métodos remotamente.

En Java RMI, las interfaces remotas se definen de la misma forma que cualquier interfaz en Java. Adquieren su capacidad de ser interfaces remotas al extender una interfaz denominada *Remote*.

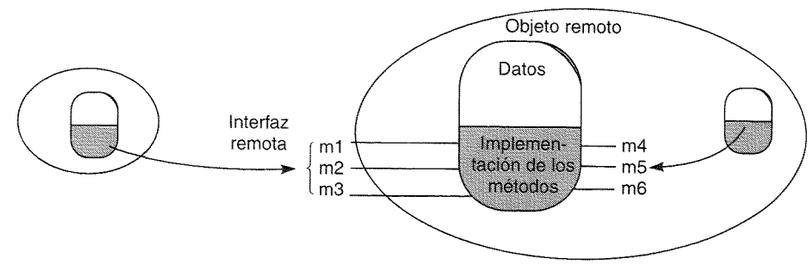


Figura 5.4. Un objeto remoto y su interfaz remota.

Ambos CORBA IDL (véase la Sección 17.2.3) y Java soportan herencia múltiple para sus interfaces. Esto es, se permite que una interfaz extienda una o más interfaces.

◇ **Acciones en un sistema de objetos distribuido.** Como en el caso no distribuido, una acción se inicia mediante la invocación de un método, que pudiera resultar en consiguientes invocaciones sobre métodos de otros objetos. Pero en el caso distribuido, los objetos involucrados en una cadena de invocaciones relacionadas pueden estar ubicados en procesos o en computadores diferentes. Cuando una invocación cruza los límites de un proceso o un computador, se emplea una RMI, y la referencia remota al objeto se hace disponible para hacer posible la RMI. En la Figura 5.3, el objeto A necesita poseer una referencia a objeto remoto para el objeto B. Las referencias a un objeto remoto pueden obtenerse como resultado de una invocación a un objeto remoto. Por ejemplo, el objeto A en la Figura 5.3 podría obtener una referencia remota al objeto F desde el objeto B.

◇ **Compactación automática de memoria en un sistema de objetos distribuido.** Si un lenguaje, por ejemplo Java, soporta compactación automática de memoria, entonces cualquier sistema RMI asociado debiera permitir la compactación automática de memoria para objetos remotos. La compactación automática de memoria, distribuida, se logra usualmente mediante la cooperación entre el compactador automático de memoria local y un módulo adicional que realiza una forma de compactación automática de memoria, distribuida, basada generalmente en una cuenta de referencias. La Sección 5.2.6 describirá tal esquema en detalle.

◇ **Excepciones.** Cualquier invocación remota puede fallar por razones relativas a que el objeto invocado está en un proceso o computador diferente de la del objeto que lo invoca. Por ejemplo, el proceso que contiene el objeto remoto pudiera malograrse o estar demasiado ocupado para responder, o pudiera perderse el mensaje resultante de la invocación. Es así, que una invocación a un método remoto debiera ser capaz de lanzar excepciones tales como *timeouts* debidos a la distribución así como aquellos lanzados durante la ejecución del método invocado. Ejemplos de esto último son: un intento de lectura pasado el fin de un archivo, o un acceso a un archivo sin los permisos adecuados.

CORBA IDL proporciona una notación para las excepciones específicas del nivel de aplicación, y el sistema subyacente genera excepciones estándar cuando ocurren errores debidos a la distribución. Los programas clientes CORBA deben ser capaces de gestionar las excepciones. Por ejemplo, un programa cliente C++ empleará los mecanismos de excepciones de C++.

5.2.4. CUESTIONES DE DISEÑO PARA RMI

La sección anterior sugería que RMI es una extensión natural a la invocación de métodos locales. En esta sección, discutiremos dos cuestiones de diseño que aparecen al realizar esta extensión:

- A pesar de que las invocaciones locales se ejecutan exactamente una sola vez, pudiera no ser siempre éste el caso para las invocaciones de métodos remotos. Se discutirán las alternativas.
- El nivel de transparencia deseable para RMI.

En el resto de la Sección 5.2, nos referimos a los procesos que alojan a los objetos remotos como servidores y a los procesos que alojan a los que los invocan como clientes. Los servidores también podrán ser clientes.

◇ **Semántica de la invocación RMI.** En la Sección 4.4 se discutieron los protocolos petición-respuesta, donde se mostró que se podía implementar *hazOperación* de diferentes formas que proporcionen diferentes garantías de reparto. Las opciones principales son:

Reintento del mensaje de petición: donde se retransmite el mensaje de petición hasta que, o bien se recibe una respuesta o se asume que el servidor ha fallado.

Filtrado de duplicados: cuando se emplean retransmisiones, si se descartan las peticiones duplicadas en el servidor.

Retransmisión de resultados: si se mantiene una historia de los mensajes de resultados para permitir retransmitir los resultados perdidos sin reejecutar las operaciones en el servidor.

Las combinaciones de estas opciones conducen a una variedad de semánticas posibles acerca de la fiabilidad de las invocaciones remotas tal y como las ve el invocante. La Figura 5.5 muestra las opciones de interés, con sus nombres correspondientes para las semánticas de invocación que producen. Observe que para las invocaciones de métodos locales, la semántica es *exactamente una vez*, que significa que todo método se ejecuta exactamente una vez. Las semánticas de invocación se definen como sigue:

Semántica de invocación *pudiera ser*: Con la semántica de invocación *pudiera ser*, el que invoca no puede decir si un método se ha ejecutado una vez, o ninguna en absoluto. La semántica *pudiera ser* aparece cuando no se aplica ninguna medida de tolerancia ante fallos. Ésta puede padecer de los siguientes tipos de fallo:

- Fallos de omisión si se pierde la invocación o el mensaje con el resultado.
- Fallos por caída cuando el servidor que contiene el objeto remoto falla.

Si el mensaje con el resultado no se recibe tras un timeout y no hay reintentos, no hay certeza de si se ha ejecutado el método. Si se pierde el mensaje de invocación, entonces el método no se habrá ejecutado. Por otro lado, puede haberse ejecutado el método y haberse perdido el mensaje con el resultado. Un fallo por caída puede ocurrir tanto antes como después de ejecutarse el método. Además, en un sistema asíncrono, el resultado de ejecutar el método podría llegar después del timeout. La semántica *pudiera ser* es útil sólo en aplicaciones donde es aceptable que haya invocaciones fallidas.

Medidas de tolerancia a fallos			*Semánticas de invocación*
Retransmisión de mensaje de petición	*Filtrado de duplicados*	*Reejecución del procedimiento o retransmisión de la respuesta*	
No	No procede	No procede	*Pudiera ser*
Sí	No	Reejecutar el procedimiento	*Al menos una vez*
Sí	Sí	Retransmitir respuesta	*Como máximo una vez*

Figura 5.5. Semánticas de invocación.

Semántica de invocación *al menos una vez*: Con la semántica de invocación *al menos una vez*, el invocante recibe un resultado, en cuyo caso el invocante sabe que el método se evaluó al menos una vez, a menos que se reciba una excepción informando que no se recibe ningún resultado. La semántica de invocación *al menos una vez* puede alcanzarse mediante la retransmisión de los mensajes de petición, que enmascara los fallos por omisión de los mensajes de invocación del resultado. La semántica *al menos una vez* puede padecer los siguientes tipos de fallo:

- Fallos por caída cuando el servidor que contiene el objeto remoto falla.
- Fallos arbitrarios. En casos donde el mensaje de invocación se retransmite, el objeto remoto puede recibirlo y ejecutar el método más de una vez, provocando que se almacenen o devuelvan valores posiblemente erróneos.

El Capítulo 4 definía *operación idempotente* como aquella que se puede realizar repetidamente con el mismo efecto que si hubiera sido realizada exactamente una sola vez. Las operaciones no idempotentes pueden tener el efecto equivocado si se realizan más de una sola vez. Por ejemplo, una operación para incrementar un balance bancario en 10 unidades debería realizarse de una vez; ¡si tuviera que repetirse, el balance crecería y crecería! Si los objetos de un servidor se pudieran diseñar para que todos los métodos de sus interfaces remotas fueran operaciones idempotentes, entonces la semántica *al menos una vez* sería aceptable.

Semántica *como máximo una vez*: Con la semántica de invocación *como máximo una vez*, el invocante recibe bien un resultado, en cuyo caso el invocante sabe que el método se ejecutó exactamente una vez, o una excepción que le informa de que no se recibió el resultado, de modo que el método se habrá ejecutado o una vez o ninguna en absoluto. La semántica de invocación *como máximo una vez* puede obtenerse utilizando todas las medidas de tolerancia frente a fallos. Como en el caso anterior, al usar reintentos se enmascara cualquier fallo por omisión de los mensajes de invocación y del resultado. Las medidas adicionales de tolerancia frente a fallos previenen los fallos arbitrarios al asegurar que para cada RMI no se ejecuta el método más que una sola vez. Tanto Java RMI como CORBA observan la semántica de invocación *como máximo una vez*, pero CORBA permite emplear la semántica *pudiera ser* para los métodos que no devuelven resultados. Sun RPC proporciona la semántica de llamadas *al menos una vez*.

◇ **Transparencia.** Los creadores de RPC, Birrell y Nelson [1984], intentaban que las llamadas a procedimientos remotos fueran lo más parecido posible a las llamadas locales, sin distinción alguna entre la llamada a procedimiento remoto o local. Todas las llamadas necesarias a procedimientos de empaquetado y paso de mensajes se ocultaban del programador que empleaba la llamada. A pesar de que los mensajes de petición se retransmitían tras un timeout, se hace de modo transparente al que llama; para hacer que la semántica de llamada a procedimiento remoto fuera como la de llamada a procedimiento local. Esta noción de transparencia se extendió para aplicarla a los objetos distribuidos, pero implica la ocultación no sólo del empaquetado y el paso de mensajes sino también de la tarea de ubicación y contacto con un objeto remoto. Como un ejemplo, Java RMI hace que las invocaciones a métodos remotos sean como las llamadas a los locales permitiendo que se emplee la misma sintaxis.

Sin embargo, las invocaciones remotas son más vulnerables a fallos que las locales, puesto que involucran una red, otro computador y otro proceso. Cualquiera que sea la semántica de invocación empleada, siempre es posible que no se reciba resultado alguno, y en caso de fallo es imposible distinguir entre el fallo de la red y el fallo del proceso remoto. Esto requiere que los objetos que hacen invocaciones remotas sean capaces de recuperarse de tales situaciones.

La latencia de una invocación remota está en varios órdenes de magnitud mayor que la de una local. Esto sugiere que los programas que emplean invocaciones remotas deban ser capaces de tener en cuenta este factor, quizás minimizando sus interacciones remotas. Los diseñadores de Argus [Liskov y Scheifler 1982] sugirieron que el invocante debiera ser capaz de malograr una llamada a

un procedimiento remoto que tomara demasiado tiempo de forma que no tuviera efecto sobre el servidor. Para permitir esto, el servidor necesitaría poder restaurar las cosas al estado en que estaban cuando se llamó al procedimiento. Estas cuestiones se discutirán en el Capítulo 12.

Waldo y otros [1994] mencionan que la diferencia entre los objetos locales y remotos debería expresarse en la interfaz remota, para permitir que los objetos reaccionaran de forma consistente frente a fallos parciales. Otros sistemas avanzan más lejos que éste, argumentando que la sintaxis de una llamada remota debería ser diferente de la llamada local: en el caso de Argus, el lenguaje se extendió para que las operaciones remotas fueran explicitadas por el programador.

La elección de si las invocaciones remotas deben ser transparentes también está accesible para los diseñadores de un IDL. Por ejemplo, en CORBA, una invocación remota lanza una excepción cuando el cliente es incapaz de comunicarse con un objeto remoto. Esto requiere que el programa cliente maneje tales excepciones, permitiéndole tratar con tales fallos. Un IDL también proporciona una infraestructura para especificar la semántica de llamada de un método. Esto puede ayudar al diseñador del objeto remoto; por ejemplo, si se elige la semántica de llamada *al menos una vez* para evitar las sobrecargas de la semántica *como máximo una vez*, las operaciones del objeto se diseñan para que sean idempotentes.

El consenso actual parece ser que las invocaciones remotas deben ser transparentes en el sentido que la sintaxis de la invocación remota sea la misma que en la invocación local, pero la diferencia entre objetos remotos y locales debe indicarse en sus interfaces. En el caso de Java RMI, los objetos remotos pueden distinguirse porque por el hecho de que implementan la interfaz *Remote* y lanzan *RemoteExceptions*. Los programadores de un objeto remoto cuya interfaz esté especificada en un IDL también deben estar al tanto de la diferencia. El conocimiento de que un objeto está pensado para un acceso remoto tiene otra implicación para su diseñador: debiera ser capaz de mantener consistente su estado en presencia de accesos concurrentes desde múltiples clientes.

5.2.5. IMPLEMENTACIÓN DE RMI

En una invocación de método remoto están involucrados varios objetos y módulos separados. Esto se muestra en la Figura 5.6, en la que un objeto A del nivel de aplicación invoca un método en un objeto remoto B del nivel de aplicación para el cual dispone de una referencia de objeto remoto. Esta sección discute los papeles de cada uno de los componentes mostrados en esa figura, tratando primero con los módulos de comunicación y referencias remotas y después con el software RMI que se ejecuta sobre ellas.

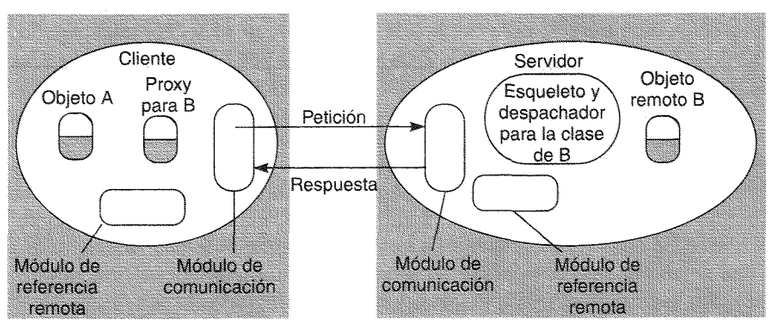


Figura 5.6. El papel de un proxy y un esqueleto en la invocación de métodos remotos.

El resto de esta sección trata de los siguientes temas relacionados: la generación de un proxy, enlace de nombres con referencias a objetos remotos, la activación y desactivación de objetos y la ubicación de objetos desde sus referencias a objetos remotos.

◇ **Módulo de comunicación.** Los dos módulos cooperantes realizan el protocolo de petición-respuesta, que retransmite los mensajes de *petición* y *respuesta* entre el cliente y el servidor. Los contenidos de los mensajes *petición* y *respuesta* se muestran en la Figura 4.13. El módulo de comunicación emplea sólo los tres primeros elementos, que especifican el tipo de mensaje, su *idPeticion* y la referencia remota del objeto que se invoca. La *idMetodo* y todo el empaquetado y desempaquetado es cuestión del software RMI que se discutirá más adelante. Los módulos de comunicación son responsables conjuntamente de proporcionar una semántica de invocación, por ejemplo *como máximo una vez*.

El módulo de comunicación en el servidor selecciona el distribuidor para la clase del objeto que se invoca, pasando su referencia local, que se obtiene del módulo de referencia remota en respuesta al identificador de objeto remoto en el mensaje *petición*. El papel del distribuidor se discute bajo el software RMI más adelante.

◇ **Módulo de referencia remota.** Un módulo de referencia remota es responsable de traducir las referencias entre objetos locales y remotos, y de crear referencias a objetos remotos. Para soportar sus responsabilidades, el módulo de referencia remota de cada proceso tiene una *tabla de objetos remotos* que almacena la correspondencia entre referencias a objetos locales en ese proceso y las referencias a objetos remotos (cuyo ámbito es todo el sistema). La tabla incluye:

- Una entrada para todo objeto remoto implementado por el proceso. Por ejemplo, en la Figura 5.6, el objeto remoto B estará registrado en la tabla del servidor.
- Una entrada para cada proxy local. Por ejemplo, en la Figura 5.6 el proxy para B estará registrado en la tabla de ese cliente.

El papel del proxy se discute bajo el siguiente párrafo sobre software de RMI. Las acciones del módulo de referencia remota ocurren como sigue:

- Cuando se pasa un objeto remoto por primera vez, como argumento o resultado, se le pide al módulo de referencia remota que cree una referencia a un objeto remoto, que se añade a su tabla.
- Cuando llega una referencia a un objeto remoto, en un mensaje de petición o respuesta, se le pide al módulo de referencia remota la referencia al objeto local correspondiente, que se referirá bien a un proxy o a un objeto remoto. En el caso de que el objeto remoto no esté en la tabla, el software RMI crea un nuevo proxy y pide al módulo de referencia remota que lo añada a la tabla.

Los componentes del módulo software de RMI llaman a este módulo cuando realizan el empaquetado y desempaquetado de las referencias a objetos remotos. Por ejemplo, cuando llega un mensaje de petición, se emplea su tabla para encontrar qué objeto local se va a invocar.

◇ **El software de RMI.** Éste consiste en una capa de software entre los objetos del nivel de aplicación y los módulos de comunicación y de referencia remota. Los papeles de los objetos de middleware mostrados en la Figura 5.6 son como sigue:

Proxy: el papel del proxy es hacer que la invocación al método remoto sea transparente para los clientes, y para ello se comporta como un objeto local para el que invoca; pero en lugar de ejecutar la invocación, dirige el mensaje al objeto remoto. Oculta los detalles de la referencia al objeto remoto, el empaquetado de los argumentos, el desempaquetado de los resultados y el envío y recepción de los mensajes desde el cliente. Hay un proxy para cada objeto remoto del que el cliente disponga de una referencia de objeto remoto. La clase de un proxy implementa los métodos de la interfaz remota del objeto remoto al que representa. Esto asegura que las

invocaciones al método remoto son adecuadas según el tipo del objeto remoto. Sin embargo, el proxy las implementa de modo bastante diferente. Cada método del proxy empaqueta una referencia hacia el objeto remoto, su propio *idMetodo* y sus argumentos en un mensaje de *petición* y lo envía al objetivo, espera el mensaje de *respuesta*, lo desempaqueta y devuelve los resultados a quién lo invocó.

Distribuidor: cada servidor tiene un distribuidor y un esqueleto para cada clase que represente a un objeto remoto. En nuestro ejemplo, el servidor tiene un distribuidor y un esqueleto para la clase del objeto remoto B. El distribuidor recibe el mensaje de *petición* desde el módulo de comunicación. Emplea el *idMetodo* para seleccionar el método apropiado del esqueleto, pasándole el mensaje de *petición*. El distribuidor y el proxy emplean los mismos métodos de asignación de cada *idMetodo* para los métodos de la interfaz remota.

Esqueleto: la clase de un objeto remoto tiene un *esqueleto*, que implementa los métodos de la interfaz remota. Se encuentran implementados de forma muy diferente de los métodos del objeto remoto. Un método del esqueleto desempaqueta los argumentos del mensaje de *petición* e invoca el método correspondiente en el objeto remoto. Espera la consumación de la invocación y después empaqueta el resultado, junto con las excepciones producidas, en un mensaje de *respuesta* para el método de envío del proxy.

Las referencias a objetos remotos se empaquetan de la forma indicada en la Figura 4.10, que incluye información sobre la interfaz remota del objeto remoto, por ejemplo el nombre de la interfaz remota o la clase del objeto remoto. Esta información permite determinar la clase proxy de modo que pueda crearse un nuevo proxy cuando se necesite. Por ejemplo, puede generarse el nombre de la clase proxy añadiendo «_proxy» al nombre de la interfaz remota.

◇ **Generación de las clases para cada proxy, distribuidor y esqueleto.** Las clases para cada proxy, distribuidor y esqueleto empleado en RMI se generan automáticamente mediante un compilador de interfaces. Por ejemplo, en Orbix, una implementación de CORBA, se generan las interfaces para las clases en CORBA IDL, y se emplea el compilador de interfaces para generar las clases para cada proxy, distribuidor y esqueleto en C++. Para Java RMI, el conjunto de métodos que se ofrecen en un objeto remoto se define como una interfaz Java que se implementa conjuntamente con la clase del objeto remoto. El compilador de Java RMI genera las clases proxy, distribuidor y esqueleto desde la clase del objeto remoto.

◇ **Programas cliente y servidor.** El programa servidor contiene las clases para los distribuidores y esqueletos, junto con las implementaciones de las clases de todos los objetos remotos a que da soporte. Las últimas se denominan a menudo clases servidoras. Además, el programa servidor contiene una sección de *inicialización* (por ejemplo en un método *main* en Java o C++). La sección de *inicialización* es responsable de crear e iniciar al menos uno de los objetos remotos que se alojarán en el servidor. Los objetos remotos adicionales se pueden crear en respuesta a las peticiones de los clientes. La sección de *inicialización* puede, también, registrar algunos de sus objetos remotos en un enlazador (*binder*) (véase el siguiente párrafo). Generalmente, sólo se registrará un objeto remoto, que será utilizado para acceder al resto.

El programa cliente contendrá las clases de cada proxy para todos los objetos remotos que invoque. Puede utilizar un enlazador para buscar las referencias a los objetos remotos.

Métodos factoría: Ya dimos cuenta de que las interfaces de objetos remotos no pueden incluir constructores. Esto viene a decir que los objetos remotos no pueden crearse mediante una invocación remota sobre los constructores. Los objetos remotos se crean bien en la sección de *inicialización* o en los métodos diseñados a tal efecto en la interfaz remota. El término *método factoría* se emplea a menudo para hacer referencia a un método que crea objetos remotos, y un *objeto factoría* es un objeto con métodos factoría. Cualquier objeto remoto que necesite ser capaz de crear un objeto remoto nuevo bajo demanda del cliente debe proporcionar métodos remotos en su interfaz remota con este fin. A tales métodos se los llama métodos factoría, aunque en realidad son métodos como los demás.

◇ **El enlazador.** Los programas cliente requieren generalmente algún modo de obtener una referencia a un objeto remoto para al menos uno de los objetos remotos alojados en el servidor. Por ejemplo, en el objeto de la Figura 5.3, el objeto A requeriría una referencia a un objeto remoto B. Un *enlazador* (*binder*) en un sistema distribuido es un servicio separado que da soporte a una tabla que contiene relaciones con nombres textuales y referencias a objetos remotos. Se emplea, por parte de los servidores, para registrar sus objetos remotos mediante su nombre y, por parte de los clientes, para buscarlos por nombre. El Capítulo 17 contiene una discusión del Servicio de Nombres de CORBA. El enlazador Java, RMIregistry, se discute brevemente en el caso de estudio sobre Java RMI en la Sección 5.5.

◇ **Hilos del servidor.** Cuando un objeto ejecuta una invocación remota, su ejecución puede conducir a otras invocaciones de métodos de otros objetos remotos, que pueden tomarse su tiempo en volver. Para evitar que la ejecución de una invocación remota retrase la ejecución de otra, los servidores suelen asignar un hilo de ejecución separado para cada invocación remota. Cuando esto ocurre, el diseñador de la implementación de un objeto remoto debe tener en cuenta los efectos de las ejecuciones concurrentes sobre su estado.

◇ **Activación de objetos remotos.** Algunas aplicaciones requieren que su información sobreviva durante largos períodos de tiempo. Sin embargo, no resulta práctico que los objetos que representan esta información se mantengan en ejecución durante períodos de tiempo ilimitados, particularmente cuando no se usan durante todo ese tiempo. Para evitar el potencial gasto superfluo de recursos debido a la ejecución de todos los servidores que gestionan los objetos remotos, los servidores podrán arrancarse cuando sean necesitados por los clientes, así como se hace para los servicios estándar de tipo TCP como FTP, que se lanzan bajo demanda de un servicio denominado *Inetd*. Los procesos que lanzan los procesos servidores que alojan objetos remotos se denominan *activadores* por las siguientes razones.

Un objeto remoto se dice que está *activo* cuando está disponible para su invocación en el interior de un proceso en ejecución, mientras que se denomina *pasivo* (o inactivo) si no está activo actualmente pero puede activarse. Un objeto pasivo consta de dos partes:

1. La implementación de sus métodos.
2. Su estado en forma empaquetada.

La *activación* consiste en la creación de un objeto activo desde el objeto pasivo correspondiente mediante la creación de una nueva instancia de su clase y la iniciación de sus variables de instancia desde el estado almacenado. Los objetos pasivos pueden activarse bajo demanda, por ejemplo, cuando son invocados por otros objetos.

Un *activador* es responsable de:

- Registrar los objetos pasivos que estén disponibles para su activación, lo que implica el almacenamiento de los nombres de los servidores conjuntamente con el URL o nombre de archivo de un objeto pasivo correspondiente.
- Arrancar procesos de servicio con nombre y activar los objetos remotos de su interior.
- Mantener la pista de las ubicaciones de los servidores de los objetos remotos que ya han sido activados.

El caso de estudio CORBA describe su activador, que se denomina el repositorio de implementación. Java RMI emplea un activador en cada computador servidor, que es responsable de activar los objetos sobre ese computador.

◇ **Almacenes de objetos persistentes.** Un objeto cuya vida se encuentra garantizada entre procesos de activación se denomina *objeto persistente*. Los objetos persistentes suelen estar ges-

tionados por los almacenes de objetos persistentes, que almacenan su estado en forma empaquetada en el disco. Los ejemplos incluyen el servicio de objetos persistentes de CORBA (véase la Sección 17.3) y Persistent Java [Jordan 1996, java.sun.com IV].

En general, un almacén de objetos persistente administrará cantidades muy grandes de objetos persistentes, que se almacenarán sobre disco hasta el momento de su uso. Se activarán cuando sus métodos sean invocados por otros objetos. La activación se suele diseñar de modo transparente; es decir, el que invoca no será capaz de decir si el objeto estaba ya en memoria o tuvo que ser activado antes de que se invocara su método. Los objetos persistentes cuya presencia en memoria principal no sea ya necesaria pueden ser desactivados. En la mayoría de los casos, los objetos se almacenan en el almacén de objetos persistente en cualquier momento en que se encuentren en un estado consistente, para lograr tolerancia a fallos. El almacén de objetos persistente requiere una estrategia para decidir cuándo desactivar los objetos. Por ejemplo, pudiera hacerlo en respuesta a una petición del programa que activó los objetos, por ejemplo, al final de una transacción o cuando el programa acaba. Los almacenes de objetos persistentes suelen intentar optimizar la desactivación almacenando sólo aquellos objetos que hayan sido modificados desde la última vez en que fueron salvados.

Los almacenes de objetos persistentes suelen permitir que haya conjuntos de objetos persistentes relacionados con nombres legibles tales como un camino o un URL. En la práctica, cada nombre legible está asociado con la raíz de un conjunto de objetos persistentes conectados.

Hay dos aproximaciones para saber si un objeto es persistente o no:

- El almacén de objetos persistente mantiene algunas raíces persistentes y cualquier objeto que sea alcanzable desde una raíz persistente se define para ser persistente. Esta aproximación es la que emplea Persistent Java y PerDiS [Ferreira y otros 2000]. Los almacenes de objetos persistentes que utilizan esta aproximación hacen uso de un compactador automático de memoria para disponer de los objetos que no estén siendo referenciados por las raíces persistentes.
- El almacén de objetos persistente proporciona algunas clases sobre las que se basa la persistencia (los objetos persistentes pertenecen a sus subclases). Por ejemplo, en Arjuna [Parrington y otros 1995], los objetos persistentes se basan en clases C++ que proporcionan transacciones y recuperación. Los objetos no requeridos deben borrarse explícitamente.

Algunos almacenes de objetos persistentes, por ejemplo PerDiS y Khazana [Carter y otros 1998] permiten que los objetos sean activados en múltiples cachés locales a los usuarios, en lugar de en los servidores. En este caso, se requiere un protocolo de consistencia de caché; el Capítulo 16 discutirá una variedad de modelos de consistencia.

◇ **Ubicación de objetos.** La Sección 4.3.3 describía una forma de referencia a objeto remoto que contiene la dirección Internet y el número del puerto del proceso que creó el objeto remoto como una forma de garantizar la unicidad. Esta forma de referencia a objeto remoto puede utilizarse como una dirección de objeto remoto en tanto que el objeto remoto permanezca en el mismo proceso durante el resto de su existencia. Pero algunos objetos remotos existirán en un conjunto de procesos diferentes, posiblemente en computadores diferentes, durante su período de vida. En este caso, una referencia a un objeto remoto no puede actuar como una dirección. Los clientes que realicen invocaciones requieren tanto una referencia a objeto remoto como una dirección a la que enviar las invocaciones.

Un *servicio de localización* ayuda a los clientes a localizar objetos remotos desde sus referencias a objeto remoto. Emplea una base de datos que relaciona las referencias a objeto remoto con sus situaciones actuales probables; los lugares son probables porque un objeto podría haber migrado de nuevo desde la última vez que se le encontró. Por ejemplo, el sistema Clouds [Dasgupta y otros 1991] y el sistema Emerald [Jul y otros 1988] emplean un esquema de caché/difusión en el que en cada computador un servicio de localización guarda una pequeña caché de relaciones referencia de objeto remoto y ubicación. Si una referencia a un objeto remoto está en la caché, se intenta la invocación con esa dirección, que fallará si el objeto ha sido movido. Para localizar un objeto que se ha movido o cuya ubicación no aparece en la caché, el sistema difunde una petición. Este esquema puede mejorarse mediante el empleo de apuntadores de localización hacia delante, que contienen pistas de la nueva ubicación de un objeto.

5.2.6. COMPACTACIÓN AUTOMÁTICA DE MEMORIA

El objetivo de un compactador automático de memoria es asegurar que mientras alguien posea una referencia a un objeto remoto o local, el objeto en sí mismo seguirá existiendo, pero tan pronto como no haya ningún objeto que haga referencia a él, se cobra dicho objeto y se recupera la memoria que empleaba.

Describiremos el algoritmo distribuido de compactación automática de memoria de Java, que es similar al descrito por Birrell y otros [1995]. Se basa en el recuento de las referencias. Cuando un proceso obtiene una referencia a un objeto remoto, se crea un proxy que permanecerá allí tanto tiempo como sea necesario. El proceso donde vive este objeto (su servidor) deberá estar informado del nuevo proxy en el cliente. Más tarde, cuando no hubiera tal proxy en el cliente, se informará al servidor. El compactador automático de memoria distribuido trabaja en cooperación con el compactador automático de memoria local como sigue:

- Cada proceso servidor mantiene un conjunto de procesos que guardan referencias a objetos remotos para cada uno de sus objetos remotos; por ejemplo, *B.titulares* es el conjunto de procesos clientes (máquinas virtuales) que tienen algún proxy del objeto B. (En la Figura 5.6, este conjunto incluirá los procesos cliente que se muestran.) Este conjunto puede almacenarse en una columna adicional de la tabla de objetos remotos.
- Cuando un cliente C recibe una referencia a un objeto remoto particular por primera vez, B, realiza una invocación *añadeRef(B)* al servidor de ese objeto remoto y entonces crea un proxy; el servidor añade C a *B.titulares*.
- Cuando el compactador automático de memoria de un cliente C advierte que ya no es posible acceder al proxy de un objeto remoto B, se hace una invocación a *eliminaRef(B)* en el servidor correspondiente y destruye el proxy; el servidor elimina C de *B.titulares*.
- Cuando *B.titulares* está vacío, el compactador automático de memoria local del servidor reclamará el espacio ocupado por B a menos que haya algún objeto local que sea titular de alguna referencia.

Este algoritmo está pensado para ser llevado a cabo mediante una comunicación emparejada petición-respuesta con una semántica del tipo *al menos una vez* entre los módulos de referencias remotas en los procesos; no requiere ninguna sincronización global. Observe también que las invocaciones extras realizadas en beneficio del algoritmo de compactación automática de memoria no afectan a las invocaciones RMI normales; sólo aparecen cuando se crea o elimina algún proxy.

Existe la posibilidad de que algún cliente realice una invocación *eliminaRef(B)* a la vez que otro cliente realice una invocación a *añadeRef(B)*. Si la operación *eliminaRef* llega primero y *B.titulares* se encuentra vacío, el objeto remoto B podría ser borrado antes de que llegara *añadeRef*. Para evitar esta situación, si el conjunto *B.titulares* está vacío en el momento en que se transmite una referencia al objeto remoto, se añade una entrada temporal hasta que llega *añadeRef*.

El algoritmo de compactación automática de memoria distribuido tolera los fallos de comunicación usando la siguiente aproximación. Las operaciones *añadeRef* y *eliminaRef* son idempotentes. En el caso de que una llamada a *añadeRef(B)* devuelva una excepción (que indicaría que el método se ejecutó una vez o ninguna), el cliente no creará un proxy pero hará una llamada a *removeRef(B)*. El efecto de *eliminaRef* es correcto tanto si tiene éxito como si no *añadeRef*. El caso de que falle *eliminaRef* se trata mediante concesiones (*leases*), según se describe en el párrafo próximo.

El algoritmo de compactación automática de memoria distribuida tolera el fallo de los procesos clientes. Para obtener esto, los servidores *ceden* sus objetos a los clientes durante un período de tiempo limitado. El período de concesión comienza cuando el cliente realiza una invocación *añadeRef* al servidor. Acaba bien cuando expira el plazo o cuando el cliente realiza una invocación *eliminaRef* al servidor. La información almacenada por el servidor, concerniente a cada concesión, contiene el identificador de la máquina virtual del cliente y el período de concesión. Los clientes son los responsables de pedir la renovación de sus concesiones antes de que éstas expiren.

◇ **Concesiones en Jini.** El sistema distribuido Jini incluye una especificación para las concesiones [Arnold y otros 1999] que puede utilizarse en variedad de situaciones cuando un objeto ofrezca un recurso a otro objeto, como por ejemplo cuando los objetos remotos ofrecen referencias a otros objetos. Los objetos que ofrecen tales recursos padecen el riesgo de tener que mantener los recursos aun cuando los usuarios ya no estén interesados o sus programas pudieran haber finalizado. Para evitar protocolos complicados para descubrir si los usuarios de los recursos están aún interesados, los recursos se ofrecen por un período de tiempo limitado. La cesión del uso de un recurso durante un período de tiempo se denomina *concesión*. El objeto que ofrece el recurso lo mantendrá hasta que expire el tiempo de la concesión. Los usuarios de los recursos son responsables de pedir su renovación cuando ésta expira.

El período de una concesión puede negociarse entre el que cede y el que toma la cesión, a pesar de que esto no ocurre con las concesiones utilizadas en Java RMI. Un objeto que represente una concesión implementa la interfaz *Lease*. Ésta contiene información sobre el período de la concesión y los métodos que permiten la renovación o la cancelación de la concesión. El que cede devuelve una instancia de la clase *Lease* cuando proporciona un recurso a otro objeto.

5.3. LLAMADA A UN PROCEDIMIENTO REMOTO

Una llamada a un procedimiento remoto es muy similar a una invocación a un método remoto, en la que un programa cliente llama a un procedimiento de otro programa en ejecución en un proceso servidor. Los servidores pueden ser clientes de otros servidores para permitir cadenas de RPC. Según se mencionó en la introducción a este capítulo, un proceso servidor define en su *interfaz de servicio* los procedimientos disponibles para ser llamados remotamente. RPC, como RMI, puede implementarse para ofrecer alguna de las semánticas de invocación discutidas en la Sección 5.2.4; generalmente se elige *al menos una vez* o *como máximo una vez*. RPC se implementa usualmente sobre un protocolo petición-respuesta como el que se discutió en la Sección 4.4, que se encuentra simplificado por la omisión de referencias a objetos remotos en la parte de los mensajes de petición. Los contenidos de los mensajes de petición y respuesta son los mismos que los que se mostraron para RMI en la Figura 4.13, excepto que se omite el campo *ReferenciaObjeto*.

El software que soporta RPC se muestra en la Figura 5.7. Es similar al mostrado en la Figura 5.6 excepto que no se requieren módulos de referencias remotas, dado que las llamadas a procedimientos no tienen que ver con objetos y referencias a objetos. El cliente que accede a un servicio incluye un *procedimiento de resguardo* para cada procedimiento en la interfaz de servicio. El papel de un procedimiento de resguardo es similar al de un proxy. Se comporta como un procedimiento local del cliente, pero en lugar de ejecutar la llamada, empaqueta el identificador del procedimiento y los argumentos en un mensaje de petición, que se envía vía su módulo de comunicación al servidor. Cuando llega el mensaje de respuesta, desempaqueta los resultados. El proceso servidor contiene un distribuidor junto a un procedimiento de resguardo de servidor y un procedimiento de servicio para cada procedimiento de la interfaz de servicio. El distribuidor selecciona uno de los procedimientos de resguardo según el identificador de procedimiento del mensaje de petición. Un *procedimiento de resguardo de servidor* es como un método de esqueleto en el que se desem-

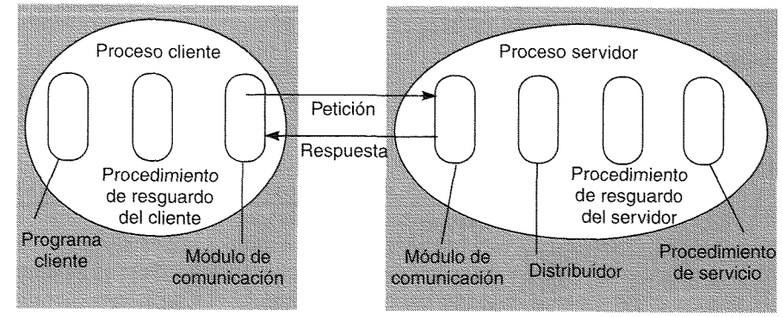


Figura 5.7. Papel de los procedimientos de resguardo de cliente y servidor en RPC.

paquetan los argumentos en el mensaje de petición, se llama al procedimiento de servicio correspondiente y se empaquetan los datos con el resultado para el mensaje de respuesta. Los procedimientos de servicio implementan los procedimientos en la interfaz de servicio.

Los procedimientos de resguardo del cliente y del servidor y el distribuidor pueden generarse desde la definición de la interfaz del servicio por medio de un compilador de interfaces.

5.3.1. CASO DE ESTUDIO SUN RPC

RFC 1831 [Srinivasan 1995] describe Sun RPC, que se diseñó para la comunicación cliente-servidor en el sistema de archivos en red Sun NFS. Sun RPC se denomina a menudo ONC (Computación de Red A —*Open Network Computing*—) RPC. Se proporciona como parte de los varios sistemas operativos de Sun y otros del tipo UNIX, y también está disponible con otras instalaciones de NFS. Los diseñadores tienen la opción de utilizar llamadas a procedimientos remotos sobre UDP o TCP. Cuando se utiliza Sun RPC sobre UDP, la longitud de los mensajes de petición y respuesta se encuentra limitada, teóricamente a 64 kilobytes, aunque más a menudo se encuentran en la práctica limitaciones a 8 ó 9 kilobytes. Utiliza la semántica de llamada *al menos una vez*. Como opción existe la posibilidad de difusión de RPC.

El sistema Sun RPC proporciona un lenguaje de interfaz denominado XDR y un compilador de interfaces llamado *rpcgen* cuyo uso está orientado al lenguaje de programación C.

◇ **Lenguaje de definición de interfaz.** El lenguaje Sun XDR, diseñado originalmente para especificar representaciones externas de datos, se extendió para convertirse en un lenguaje de definición de interfaces. Puede utilizarse para definir una interfaz de servicio para Sun RPC especificando un conjunto de definiciones de procedimiento junto a las definiciones de tipos que las soportan. La notación es bastante primitiva en comparación con la usada por CORBA IDL o Java. En particular:

- La mayoría de los lenguajes permiten especificar nombres de interfaces, pero Sun RPC no; en su lugar, hay que proporcionar un número de programa y un número de versión. Los números de programa pueden obtenerse de una autoridad central para permitir que cualquier programa tenga un número propio único. Los números de versión cambian cuando cambia la signatura de algún procedimiento. Tanto el número de programa como el de versión se envían en el mensaje de petición, de modo que el cliente y el servidor puedan concretar la versión que están usando.

```
const MAX = 1000;
typedef int IdentificadorArchivo;
typedef int ApuntadorArchivo;
typedef int Longitud;
struct Datos {
    int longitud;
    char bufer[MAX];
};
struct argumentosEscribe {
    IdentificadorArchivo f;
    ApuntadorArchivo posicion;
    Datos datos;
};
struct argumentosLee {
    IdentificadorArchivo f;
    ApuntadorArchivo posicion;
    Longitud longitud;
};

program LEEESCRIBEARCHIVO {
    version VERSION {
        void ESCRIBE(argumentosEscribe)= 1
        Data LEE(argumentosLee)= 2;
        }= 2;
}= 9999;
```

Figura 5.8. Interfaz de archivos en Sun XDR.

- Una definición de procedimiento especifica una signatura o firma de un procedimiento y un número de procedimiento. El número de procedimiento se emplea como un identificador en los mensajes de petición. Hubiera sido posible que el compilador de interfaces generara los identificadores de procedimiento.
- Sólo se permite un parámetro de entrada. De este modo, los procedimientos que requieran varios parámetros deberán incluirlos como componentes de una sola estructura.
- Los parámetros de salida de un procedimiento se devuelven como un solo resultado.
- La signatura del procedimiento consta de un tipo de resultado, el nombre del procedimiento y el tipo del parámetro de entrada. Tanto el tipo del resultado como el de entrada pueden ser bien un solo valor o una estructura conteniendo varios valores.

Por ejemplo, véase la definición XDR de la Figura 5.8 de una interfaz con un par de procedimientos para leer y escribir archivos. El número del programa es 9999 y el número de la versión es 2. El procedimiento *LEE* (ver la línea 2) toma como parámetro de entrada una estructura con tres componentes que especifican un identificador de archivo, una posición en el archivo y el número de bytes pedidos. Su resultado es una estructura que contiene el número de bytes devueltos y los datos del archivo. El procedimiento *ESCRIBE* (ver la línea 1) no tiene resultado. Los procedimientos *ESCRIBE* y *LEE* reciben los números 1 y 2. El número 0 se reserva para un procedimiento nulo, que se genera automáticamente y está pensado para comprobar si un servidor está disponible.

El lenguaje de definición de la interfaz proporciona una notación para definir constantes, definiciones de tipo, estructuras, tipos enumerados, uniones y programas. Las definiciones de tipo (*typedef*), estructuras (*struct*), tipos enumerados (*enum*) emplean la sintaxis del lenguaje C. El compilador de interfaces *rpcgen* puede emplearse para generar, desde la definición de interfaz, lo siguiente:

- El procedimiento de resguardo del cliente.

- El procedimiento *main* del servidor, el distribuidor y el procedimiento de resguardo del servidor.
- Los procedimientos de empaquetado y desempaquetado XDR para su empleo por el distribuidor y los procedimientos de resguardo del cliente y el servidor.

◇ **Enlazado.** Sun RPC lanza un servicio de enlazado denominado *enlazador de puertos* (*port mapper*) en un número de puerto bien conocido de cada computador. Cada ejemplar del enlazador de puertos almacena el número de programa, el número de versión y el número de puerto en uso por cada servicio que se ejecuta localmente. Cuando arranca un servidor, registra su número de programa, número de versión y número de puerto frente al enlazador de puertos. Cuando arranca el cliente, encuentra el puerto del servidor mediante una petición remota al enlazador de puertos del servidor huésped, especificando el número del programa y el número de la versión.

Cuando un servicio tiene múltiples ejemplares en ejecución sobre diferentes computadores, las instancias pueden usar diferentes números de puerto para recibir las peticiones de los clientes. Si un cliente necesita multidifundir una petición a todas las instancias de un servicio que están utilizando diferentes números de puerto, no podrá utilizar un mensaje de difusión directo con este fin. La solución es que los clientes efectúen una multidifusión de llamadas a procedimiento remoto mediante una difusión a todos los enlazadores de puertos, especificando los números de programa y versión. Cada enlazador de puertos dirigirá las llamadas a los programas de servicio apropiados, si hubiera alguno.

◇ **Autenticación.** Los mensajes de petición y respuesta de Sun RPC proporcionan campos adicionales que permiten pasar información de autenticación entre el cliente y el servidor. Por ejemplo, en el estilo de autenticación UNIX, las credenciales incluyen el *uid* y el *gid* del usuario. Se pueden construir mecanismos de control de acceso sobre la información de autenticación que se ofrece a los procedimientos de servicio a través de un segundo argumento. El programa servidor se hace responsable de poner en práctica el control de acceso decidiendo si ejecutar o no cada llamada a procedimiento según la información de autenticación. Por ejemplo, si el servidor es un servidor de archivos NFS, puede comprobar si el usuario tiene suficientes derechos de acceso para llevar a cabo la operación requerida sobre un archivo.

Es posible dar soporte a diferentes protocolos de autenticación. Éstos incluyen:

- Ninguno.
- Al estilo UNIX, como se ha comentado.
- De forma en que se establece una clave compartida para firmar los mensajes RPC.
- Al estilo de autenticación de Kerberos (véase el Capítulo 7).

Un campo en la cabecera RPC indica cuál de ellos se está empleando.

En el RPC 2203 [Eisler y otros 1997] se describe una aproximación más genérica a la seguridad. Proporciona privacidad e integridad a los mensajes RPC, al mismo tiempo que autenticación. Permite que el cliente y el servidor negocien un contexto de seguridad en el que o bien no se aplica seguridad o bien se requiere seguridad, en cuyo caso se podrá aplicar tanto integridad de los mensajes como privacidad, conjuntamente o por separado.

◇ **Programas cliente y servidor.** En www.cdk3.net/RMI se dispone de material adicional sobre Sun RPC. Incluye programas cliente y servidor correspondientes a la interfaz definida en la Figura 5.8.

5.4. EVENTOS Y NOTIFICACIONES

La idea bajo el uso de eventos es que un objeto pueda reaccionar a un cambio que ocurre en otro objeto. Las notificaciones y los eventos son esencialmente asíncronos y son determinados por sus

receptores. En particular, en las aplicaciones interactivas, las acciones que el usuario realiza sobre los objetos, por ejemplo al manipular un botón con el ratón o al introducir texto en una caja de texto mediante el teclado, se ven como *eventos* que provocan cambios en los objetos que mantienen el estado de la aplicación. Los objetos responsables de mostrar el aspecto del estado actual son *notificados* cuando cambia el estado.

Los sistemas distribuidos basados en eventos extienden el modelo local de eventos al permitir que varios objetos en diferentes ubicaciones puedan ser notificados de los eventos que tienen lugar en un objeto. Emplean el paradigma *publica-suscribe*, en el que un objeto que genera eventos *publica* el tipo de eventos que ofrece para su observación por otros objetos. Los objetos que desean recibir notificaciones de un objeto que haya publicado sus eventos se *suscriben* a los tipos de eventos que sean de interés para ellos. Los objetos que representan los eventos se denominan *notificaciones* o *anuncios*. Las notificaciones se pueden almacenar, enviarse en los mensajes, consultarse y aplicarse en variedad de órdenes a diferentes cosas. Cuando un anunciante experimenta un evento, los suscriptores que hayan expresado interés en ese tipo de evento recibirán las notificaciones. La acción de suscribirse a un tipo particular de evento se denomina también *registrar el interés* por ese tipo de evento.

Los eventos y las notificaciones pueden emplearse en una gran variedad de aplicaciones diferentes, por ejemplo para comunicar que se añade una forma a un dibujo, una modificación a un documento, el hecho de que una persona haya entrado o dejado una sala, o que una parte del equipamiento o un libro etiquetado electrónicamente se encuentra en un nuevo lugar. Los dos últimos ejemplos se hacen posibles con el uso de distintivos activos o dispositivos empotrados (véase la Sección 2.2.3).

Los sistemas distribuidos basados en eventos presentan dos características importantes:

Heterogéneos: cuando se emplean notificaciones como mecanismo de comunicación entre objetos distribuidos, es posible hacer funcionar conjuntamente aquellos componentes del sistema distribuido que no han sido diseñados con características de interoperabilidad. Todo lo que se requiere es que los objetos generadores de eventos publiquen los tipos de eventos que ofrecen, y que otros objetos se suscriban a los eventos y proporcionen una interfaz para recibir las notificaciones. Por ejemplo, Bates y otros [1996] describen cómo se pueden emplear los sistemas basados en eventos para conectar componentes heterogéneos en Internet. Éste, describe un sistema en el que las aplicaciones son advertidas de las ubicaciones y actividades de sus usuarios, tales como utilizar los computadores, las impresoras o libros marcados electrónicamente. Los autores prevén su uso futuro en el contexto de una red doméstica con operaciones como: *si los niños vuelven a casa, conecta la calefacción general.*

Asíncronos: las notificaciones se envían asíncronamente desde los objetos generadores de eventos a todos los objetos que se hayan suscrito a ellos para prevenir que el que los anunciantes necesiten sincronizarse con los suscriptores; es preciso desacoplar los anunciantes de los suscriptores. Mushroom [Kindberg y otros 1996] es un sistema distribuido basado en eventos diseñado para dar soporte al trabajo colaborativo, en el que la interfaz de usuario muestra objetos que representan a usuarios y objetos de información tales como documentos y libros de notas en el interior de espacios de trabajo llamados *lugares de red*. El estado de cada lugar se encuentra replicado en los computadores de los usuarios que concurren en ese lugar. Los eventos se emplean para describir cambios sobre los objetos y cambios en el foco de interés de un usuario. Por ejemplo, un evento podría especificar que un usuario concreto ha entrado o salido de un lugar, o ha realizado una acción particular sobre un objeto. Cada réplica de cualquier objeto para el que sean relevantes ciertos tipos de eventos se subscribe a ellos y recibe notificaciones cuando éstos ocurran. Pero los suscriptores se encuentran desacoplados de los objetos que experimentan los eventos, dado que en momentos diferentes los usuarios activos son diferentes.

Una situación en la que pueden ser útiles los eventos es la que se muestra en el siguiente ejemplo sobre una sala de contratación:

◇ **Sistema simple de una sala de contratación.** Considere un sistema simple de una sala de contratación cuya tarea es permitir que los tratantes hagan uso de los computadores para consultar la información más reciente sobre los precios del mercado de las mercancías con que comercian. El precio del mercado para una sola mercancía conocida se representa mediante un objeto con varias variables de instancia. La información llega a la sala de contratación desde diferentes fuentes externas en forma de actualizaciones de algunos o todas las variables de instancia de los objetos que representan las mercancías y es recolectada por procesos que llamamos proveedores de información. Los tratantes están interesados solamente en aquellas mercancías en que están especializados. Un sistema de sala de contratación podría modelarse mediante procesos con dos tareas diferentes:

- Un proceso proveedor de información recibe continuamente nueva información comercial desde una sola fuente externa y le aplica los objetos de la mercancía apropiada. Cada una de las actualizaciones a un objeto de mercancía se contempla como un evento. El objeto de mercancía que experimenta tales eventos notifica a todos los tratantes que se hayan suscrito a la mercancía correspondiente. Habrá un proceso proveedor de información separado para cada fuente externa.
- Un proceso tratante crea un objeto para representar cada mercancía que el usuario pida visualizar. Este objeto local se suscribe al objeto que representa esa mercancía en el proveedor de información relevante. Entonces recibe toda la información enviada al anterior mediante las notificaciones y la muestra al usuario.

La comunicación de las notificaciones se muestra en la Figura 5.9.

◇ **Tipos de eventos.** Una fuente de eventos puede generar eventos de uno o más *tipos* diferentes. Cada evento tiene *atributos* que especifican información sobre ese evento, tal como el nombre

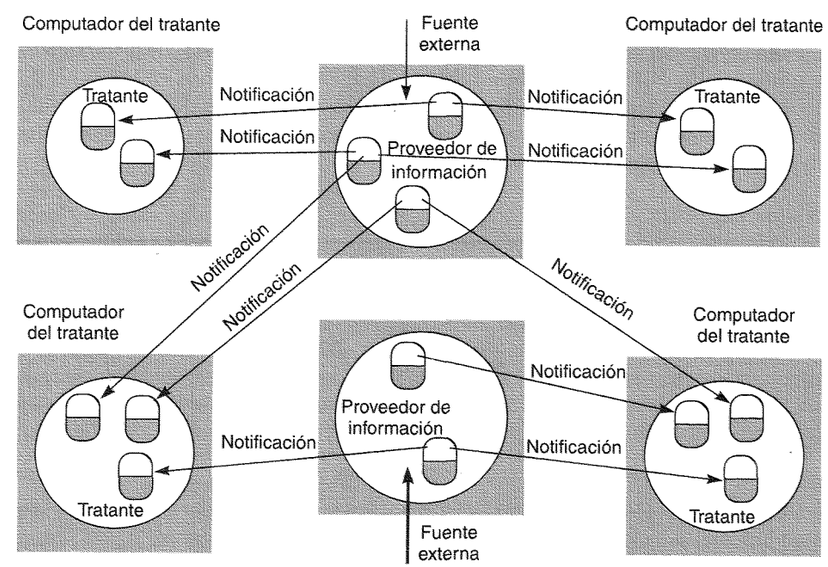


Figura 5.9. Sistema de sala de contratación.

o el identificador del evento que lo generó, la operación, sus parámetros y el tiempo (o un número de secuencia). Los tipos y los atributos se usan tanto para en la suscripción como en la notificación. Cuando nos suscribimos a un evento, se especifica el tipo del evento, a veces modificado con ciertos criterios sobre los valores de los atributos. Cuando quiera que aparezca un evento que concuerde con los atributos, se notificará a las partes interesadas. En el ejemplo de la sala de contratación, hay un tipo de evento (la llegada de una actualización de una mercancía), y los atributos podrían especificar el nombre de la mercancía, su precio actual, y la última elevación o caída de precio. Los tratantes podrán, por ejemplo, especificar si están interesados en todos los eventos relacionados con una mercancía con un nombre concreto.

5.4.1. LOS PARTICIPANTES EN UNA NOTIFICACIÓN DE EVENTOS DISTRIBUIDA

La Figura 5.10 muestra una arquitectura que especifica los papeles que juegan los objetos que participan en un sistema distribuido basado en eventos. Nuestra descripción proviene del papel sobre eventos y notificaciones en Internet de Rosenblum y Wolf [1997]. La arquitectura está diseñada para desacoplar los anunciantes de los suscriptores, permitiendo que los anunciantes sean desarrollados independientemente de sus suscriptores, y limitando hasta donde sea posible el trabajo que imponen los suscriptores sobre los anunciantes. El principal componente es un servicio de eventos que mantiene una base de datos de eventos publicados en el servicio de eventos. Los suscriptores informan al servicio de eventos de los tipos de eventos en los que están interesados. Cuando ocurre un evento en un objeto de interés se envía una notificación a los suscriptores de ese tipo de evento.

Los papeles de los objetos que participan son los siguientes:

El objeto de interés: éste es un objeto que experimenta cambios de estado, como resultado de las operaciones que se invocan sobre él. Sus cambios de estado podrían ser de interés para los otros objetos. Esta descripción permite eventos como el que una persona que lleva un distintivo identificador entre en una habitación, en cuyo caso la habitación es el objeto de interés y la operación consiste en añadir información sobre la nueva persona a su registro o quién está en la habitación. Se considera que el objeto de interés es parte del servicio de eventos si transmite notificaciones.

Evento: un evento aparece en un objeto de interés como resultado de la finalización de la ejecución de la ejecución de un método.

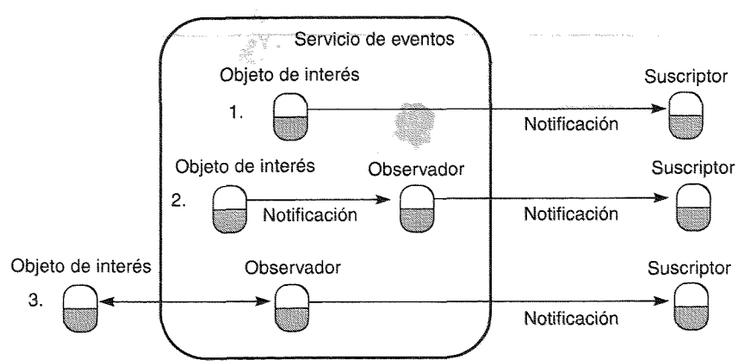


Figura 5.10. Arquitectura para la notificación distribuida de eventos.

Notificación: una notificación es un objeto que contiene información sobre un evento. Generalmente, contiene el tipo de evento y sus atributos, que generalmente incluyen la identidad del objeto de interés, el método que se invoca, y el momento en que ocurre, o un número de secuencia.

Suscriptor: un suscriptor es un objeto que se ha suscrito a algún tipo de evento en otro objeto. Recibirá notificaciones sobre tales eventos.

Objetos observadores: el principal objetivo de un observador es desacoplar un objeto de interés de sus suscriptores. Un objeto de interés puede tener muchos suscriptores diferentes con diferentes intereses. Por ejemplo, los suscriptores podrían diferir en el tipo de eventos en que están interesados, o aquellos que comparten los mismos requisitos sobre el tipo podrían divergir en los atributos sobre los que están interesados. Si hubiera que realizar, en el objeto de interés, todas las operaciones lógicas para distinguir las necesidades de sus suscriptores dicho objeto estaría complicado en exceso. Se pueden interponer uno o más observadores entre el objeto de interés y sus suscriptores. Los papeles de los observadores se discuten con más detalle en un párrafo posterior.

Anunciante: éste es un objeto que declara que generará notificaciones de tipos concretos de eventos. Un anunciante pudiera ser un objeto de interés o un observador.

La Figura 5.10 muestra tres casos:

1. Un objeto de interés en el interior de un servicio de eventos sin un observador. Envía notificaciones directamente a los suscriptores.
2. Un objeto de interés en el interior de un servicio de eventos con un observador. El objeto de interés envía notificaciones a los suscriptores vía el observador.
3. Un objeto de interés fuera del servicio de eventos. En este caso, un observador consulta al objeto de interés para descubrir cuando ocurre un evento. El observador envía notificaciones a los suscriptores.

◇ **Semántica de reparto.** Se puede proporcionar una variedad de garantías diferentes para las notificaciones; aquella que se elija dependerá de los requisitos de las aplicaciones. Por ejemplo, si se emplea multidifusión IP para enviar las notificaciones a un grupo de receptores, el modo de fallo estará relacionado con el que se describe para la multidifusión IP en la Sección 4.5.1 y no habrá garantías de que cualquier receptor particular reciba un mensaje de notificación concreto. Esto es apropiado en algunas aplicaciones, por ejemplo, para repartir el último estado de un jugador de un juego en Internet, porque es muy posible que se haga comprender en la próxima actualización.

Sin embargo, otras aplicaciones tienen requisitos más estrictos. Considere la aplicación de la sala de contratación: para ser justo con los tratantes interesados en una mercancía particular, requerimos que todos los tratantes para la misma mercancía reciban la misma información. Esto implica que habrá que usar un protocolo de multidifusión fiable.

En el sistema Mushroom mencionado anteriormente, las notificaciones sobre el cambio de estado de un objeto se envían de modo fiable a un servidor, cuya responsabilidad es mantener copias actualizadas de los objetos. Sin embargo, las notificaciones pueden enviarse a las réplicas de los objetos en los computadores de los usuarios mediante una multidifusión no fiable; el caso de que hubiera notificaciones perdidas, éstas pudieran recuperarse del estado del objeto desde el servidor. Cuando la aplicación lo requiera, se podrán ordenar las notificaciones y ser enviadas de modo fiable a las réplicas del objeto.

Algunas aplicaciones tienen restricciones de tiempo real. Éstas incluyen eventos de una planta de energía nuclear o un monitor de un paciente en un hospital. Es posible diseñar protocolos de multidifusión que proporcionen garantías de tiempo real así como fiabilidad y ordenamiento en un sistema que satisfaga las propiedades de un sistema distribuido síncrono.

◊ **Reglas para los observadores.** A pesar de que se pueden enviar las notificaciones directamente desde el objeto de interés al receptor, la tarea de procesar las notificaciones puede dividirse entre los procesos observadores que juegan una variedad de papeles diferentes. Describimos algunos ejemplos:

Encaminamiento: un observador encaminador puede llevar a cabo todo el trabajo de enviar las notificaciones a los suscriptores en representación de uno o más objetos de interés. Todo lo que necesita hacer un objeto de interés es enviar una notificación al observador de encaminamiento, y seguir realizando su tarea habitual. Para utilizar un observador de encaminamiento, un objeto de interés envía la información sobre los intereses de sus suscriptores al observador de encaminamiento.

Filtrado de notificaciones: un observador puede aplicar filtros para reducir el número de notificaciones recibidas según algún predicado sobre los contenidos de cada notificación. Por ejemplo, en el caso de que un evento trate de extracciones sobre la cuenta bancaria, pero el receptor sólo esté interesado en aquéllas de un monto mayor que 100 unidades.

Patrones de eventos: cuando un objeto se suscribe a eventos de un objeto de interés, puede especificar patrones de eventos sobre los que está interesado. Un patrón especifica una relación entre varios eventos. Por ejemplo, un suscriptor pudiera estar interesado en el caso en que haya tres extracciones de una cuenta bancaria sin un depósito entre medias. Tenemos un requisito similar cuando se correlacionan eventos en una variedad de objetos de interés; por ejemplo, notificar al suscriptor sólo cuando cierto número de ellos han generado eventos.

Buzones de notificaciones: en algunos casos, hay que retrasar las notificaciones hasta que cierto suscriptor particular está listo para recibirlas. Por ejemplo, si el suscriptor tiene conexiones defectuosas o un objeto puede haberse desactivado o activado de nuevo. Un observador podría tomar el papel de buzón de notificaciones, que recibirá las notificaciones en lugar del suscriptor, pasándolas al titular (en un solo envío) sólo cuando el suscriptor esté listo para recibirlas. El suscriptor podrá activar o desactivar el reparto según lo necesite. El suscriptor establece un buzón de notificación cuando se registra frente a un objeto de interés al especificar este buzón como el lugar donde se enviarán las notificaciones.

5.4.2. ESPECIFICACIÓN DE EVENTOS DISTRIBUIDOS DE JINI

La especificación de eventos distribuidos de Jini fue descrita por Arnold y otros [1999] y permite que un suscriptor potencial en una Máquina Virtual Java (JVM, *Java Virtual Machine*) se suscriba y reciba notificaciones de eventos de un objeto de interés en otra JVM, habitualmente en otro computador. Se podría insertar una cadena de observadores entre el objeto de interés y el suscriptor. Los principales objetos involucrados en la especificación de eventos distribuidos son:

Generadores de eventos: un generador de eventos es un objeto que permite que otros objetos se suscriban a sus eventos y generen notificaciones.

Oyentes de eventos remotos: un oyente de eventos remotos es un objeto que puede recibir notificaciones.

Eventos remotos: un evento remoto es un objeto que es pasado por valor a un oyente de eventos remotos. Un evento remoto es el equivalente de lo que hemos llamado una notificación.

Agentes terceros: se pueden interponer agentes terceros entre un objeto de interés y un suscriptor. Son los equivalentes a nuestros observadores.

Un objeto se suscribe a eventos del tipo de éstos al generador de eventos y al especificar un oyente de eventos remotos como destino de las notificaciones.

Para enviar notificaciones desde el generador de eventos al suscriptor se utiliza Java RMI, posiblemente mediante uno o más agentes terceros. Los diseñadores sostienen que los oyentes de eventos debieran replicar a las llamadas de notificación tan pronto como sea posible para evitar retrasos en los generadores de eventos. Éstos pueden procesar cada notificación tras la réplica. También se emplea Java RMI para suscribirse a los eventos. Los eventos Jini se proporcionan mediante las siguientes interfaces y clases:

RemoteEventListener: esta interfaz proporciona un método llamado *notify*. Los suscriptores y los agentes terceros implementan la interfaz *RemoteEventListener* de modo que puedan recibir las notificaciones cuando se invoca el método *notify*. Una instancia de la clase *RemoteEvent* representa una notificación y se pasa como argumento de cada método *notify*.

RemoteEvent: esta clase tiene variables de instancia que contienen:

- Una referencia a un generador de eventos en el que apareció el evento.
- Un identificador de eventos, que especifica el tipo de evento en ese generador de eventos.
- Un número de secuencia, que se aplica a los eventos de ese tipo. El número de secuencia debería incrementarse según ocurren los eventos en el tiempo. Puede usarse para permitir que los receptores ordenen los eventos de cada tipo provenientes de una fuente dada o para evitar aplicar el mismo evento dos veces.
- Un objeto empaquetado. Se aporta cuando el receptor se suscribe a ese tipo de evento y puede ser utilizada por un receptor, con cualquier finalidad. Generalmente, mantiene cualquier información que necesite el receptor para identificar el evento y reaccionar a su aparición. Por ejemplo, podría contener un procedimiento de finalización que haya que ejecutar cuando se notifica.

EventGenerator: esta interfaz proporciona un método llamado *register*. Los generadores de eventos implementan la interfaz *EventGenerator*, cuyo método *register* se emplea para suscribirse a eventos frente al generador de eventos. Los argumentos de *register* especifican:

- Un identificador de eventos, que especifica el tipo de evento.
- Un objeto empaquetado que se devolverá con cada notificación.
- Una referencia remota a un objeto oyente de eventos; el lugar donde enviar las notificaciones.
- El período de cesión que se solicita. El período de cesión especifica la duración de la concesión que requerida por el suscriptor, aunque la concesión real se devuelve con los resultados de *register*. Los tiempos límites sobre las suscripciones evitan el problema de que los generadores de eventos alojen suscripciones de eventos ya inútiles. Se pueden renovar las suscripciones en el caso de que expire el tiempo límite de las suscripciones.

La especificación de Jini indica que la interfaz *EventGenerator* es sólo un ejemplo del tipo de interfaz que podría ser usada por los suscriptores para registrar su interés en los eventos de un objeto de interés. Algunas aplicaciones podrían necesitar un tipo de interfaz diferente.

◊ **Agentes terceros.** Los agentes terceros que se intercalan entre un generador de eventos y un suscriptor pueden jugar una gran variedad de papeles útiles, incluyendo todos los descritos anteriormente.

En el caso más simple, un suscriptor registra su interés sobre un tipo particular de evento frente a un generador de eventos y se especifica a sí mismo como el oyente del evento remoto. Esto se corresponde con el caso 1 mostrado en la Figura 5.10.

Los agentes terceros pueden ser establecidos por un generador de eventos o por un suscriptor.

Un generador de eventos puede interponer uno o más agentes terceros entre el mismo y un suscriptor. Por ejemplo, los generadores de eventos de cada computador podrían utilizar un agente tercero compartido que fuera responsable del reparto fiable de las notificaciones.

Un suscriptor puede construir una cadena de agentes terceros para producir cualquier política de reparto que necesite. Entonces registra su interés frente a un generador de eventos, especificando el primero de la cadena de agentes terceros como el lugar destino de las notificaciones. Por ejemplo un suscriptor podría indicar que sus notificaciones se almacenaran en un agente tercero hasta el momento en que esté listo para recibirlo. El agente tercero puede tomar la responsabilidad de renovar las concesiones.

En la Sección 17.3.2 se discutirá el Servicio de Eventos de CORBA.

5.5. EL CASO DE ESTUDIO JAVA RMI

Java RMI extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en el lenguaje Java. En particular, permite que los objetos invoquen métodos sobre objetos remotos empleando la misma sintaxis que en las invocaciones locales. Además, la comprobación de tipos se aplica de modo igual en las invocaciones remotas como en las locales. Sin embargo, un objeto que realice una invocación remota conoce que su destino es remoto porque debe manejar *RemoteExceptions*; y el implementador de un objeto remoto conoce que es remoto porque debe implementar la interfaz *Remote*. A pesar de que el modelo de objetos distribuidos está integrado en Java de forma natural, la semántica de paso de parámetros difiere en que el objeto que invoca y el destino son remotos entre sí.

La programación de aplicaciones distribuidas en Java RMI debiera ser relativamente simple dado que es un sistema de un solo lenguaje; las interfaces remotas se definen en el lenguaje Java. Al emplear un lenguaje multi-sistema como CORBA, el programador debe conocer un IDL y comprender cómo se relaciona con el lenguaje de implementación. Sin embargo, incluso en un sistema de un solo lenguaje, el programador de un objeto remoto debe considerar su comportamiento en un entorno concurrente.

En el resto de esta introducción, damos un ejemplo de una interfaz remota, para después discutir la semántica de paso de parámetros con referencia al ejemplo. Finalmente discutiremos la descarga de clases y el enlazador. La segunda sección de este caso de estudio discute cómo construir programas cliente y servidor para la interfaz de ejemplo. La tercera sección trata del diseño e implementación de Java RMI. Para todos los detalles de Java RMI, véase el manual sobre invocación remota [java.sun.com I].

En este caso de estudio y en el caso de estudio CORBA del Capítulo 17, emplearemos el ejemplo de un *tablero* (*whiteboard*). Esta aplicación distribuida permite a un grupo de usuarios compartir la vista común de una superficie de dibujo que contiene objetos gráficos, tales como rectángulos, líneas y círculos, cada uno de los cuales ha sido dibujado por uno de los usuarios. El servidor mantiene el estado actual del dibujo proporcionando a los clientes una operación para informarles sobre la última figura dibujada por sus usuarios y guardando un registro de todas las figuras que hayan sido recibidas. El servidor también proporciona operaciones que permiten a los clientes recuperar las últimas formas dibujadas por otros usuarios mediante un servidor de escrutinio. El servidor tiene un número de versión (un entero) que se incrementa cada vez que llega una nueva forma y se añade a esta nueva forma. El servidor proporciona operaciones que permiten a los clientes preguntar por su número de versión y el número de versión de cada forma, de modo que puedan evitar el recuperar formas que ya poseen.

◇ **Interfaces remotas en Java RMI.** Las interfaces remotas se definen mediante la extensión de una interfaz denominada *Remote* que proporciona el paquete *java.rmi*. Los métodos deberán lanzar *RemoteException*, además de lanzar las excepciones específicas de la aplicación. La Figura 5.11 muestra un ejemplo de dos interfaces remotas denominadas *Forma* y *ListaForma*. En este ejemplo, *ObjetoGrafico* es una clase que aloja el estado del objeto gráfico, por ejemplo su tipo, su


```
import java.rmi.*;
import java.util.Vector;
public interface Forma extends Remote {
    int dameVersion() throws RemoteException;
    ObjetoGrafico dameTodoEstado() throws RemoteException;                    1
}
public interface ListaForma extends Remote {
    Forma nuevaForma(ObjetoGrafico g) throws RemoteException;                 2
    Vector todasFormas() throws RemoteException;
    int dameVersion() throws RemoteException;
}
```

Figura 5.11. Interfaces *Remote* en Java para *Forma* y *ListaForma*.

posición, el rectángulo que la encierra, el color de línea y el color de relleno, y proporciona operaciones para acceder y actualizar su estado. *ObjetoGrafico* debe implementar la interfaz *Serializable*. Considere primero la interfaz *Forma*: el método *dameVersion* devuelve un entero, mientras que *dameTodoEstado* devuelve una instancia de la clase *ObjetoGrafico*. Ahora considere la interfaz *ListaForma*: su método *nuevaForma* pasa una instancia de *ObjetoGrafico* como argumento pero devuelve un objeto con una interfaz remota (es decir, un objeto remoto) como resultado. Un detalle importante es advertir que tanto los objetos ordinarios como los objetos remotos pueden aparecer como argumentos y resultados de una interfaz remota. Estos últimos se denotan por el nombre de su interfaz remota. En el próximo párrafo, discutimos cómo se pasan como argumento y como resultado los objetos ordinarios y los objetos remotos.

◇ **Paso de parámetros y resultados.** En Java RMI, se supone que los parámetros de un método son parámetros de *entrada* y el resultado de un método es un único parámetro de *salida*. La Sección 4.3.2 describía la serialización en Java, que es empleada para empaquetar los argumentos y los resultados en Java RMI. Cualquier objeto que sea serializable, es decir, que implemente la interfaz *Serializable*, puede pasarse como argumento o resultado en Java RMI. Todos los tipos primitivos y los objetos remotos son serializables. El receptor puede descargar las clases para los argumentos y los valores resultantes cuando sea necesario mediante el sistema RMI.

Paso de objetos remotos: cuando el tipo del valor de parámetro o de resultado implementa una interfaz remota, el correspondiente argumento o resultado siempre se pasa como una referencia a un objeto remoto. Por ejemplo, en la Figura 5.2 (en la línea 2) el valor de retorno del método *nuevaForma* se define como *Forma*, una interfaz remota. Cuando se recibe una referencia a un objeto remoto, puede ser utilizado para hacer llamadas RMI sobre el objeto remoto al que se refiere.

Paso de objetos no remotos: todos los objetos no remotos serializables se copian y se pasan por valor. Por ejemplo, en la Figura 5.11 (en las líneas 2 y 1) el argumento de *nuevaForma* y el valor devuelto por *dameTodoEstado* son del tipo *ObjetoGrafico*, que es serializable y es pasado por valor. Cuando se pasa un objeto por valor, se crea un nuevo objeto en el proceso del receptor. Los métodos de este nuevo objeto pueden invocarse localmente, causando, posiblemente, que su estado difiera del estado del objeto original del proceso que lo envió.

Consiguientemente, en nuestro ejemplo, el cliente usa el método *nuevaForma* para pasar una instancia de *ObjetoGrafico* al servidor; el servidor crea un objeto remoto del tipo *Forma* que contiene el estado del *ObjetoGrafico* y devuelve una referencia remota a él. Los argumentos y valores devueltos de una invocación remota se serializan para encauzarlos empleando el método descrito en la Sección 4.3.2, con las siguientes modificaciones:

1. Cuando quiera que un objeto que implemente la interfaz *Remote* sea serializado, se reemplaza por su referencia a objeto remoto, que contiene el nombre de su clase (la del objeto remoto).
2. Cuando quiera que sea serializado un objeto, se anota su información de clase con la ubicación de la clase (como un URL), permitiendo la descarga de la clase por el receptor.

◇ **Descarga de las clases.** Java está diseñado para permitir la descarga de las clases desde una máquina virtual a otra. Esto es particularmente relevante para que los objetos remotos se comuniquen mediante la invocación remota. Hemos visto que los objetos no remotos se pasan por valor y los objetos remotos se pasan por referencia como argumentos y resultados de cada RMI. Si el receptor no posee ya la clase de un objeto que es pasado por valor, su código se descargará automáticamente. De igual modo, si el receptor de una referencia a un objeto remoto no posee ya la clase de un proxy, se descargará automáticamente su código. Esto permite dos ventajas:

1. No hay necesidad alguna de que cada usuario presente el mismo conjunto de clases en su entorno de trabajo.
2. Tanto los programas cliente como servidor pueden hacer un uso transparente de las instancias de las clases nuevas cuando quiera que éstas se añadan.

Como ejemplo, considere el programa de tablero y suponga que su implementación inicial de *ObjetoGrafico* no contempla el texto. Entonces un cliente con un objeto textual podrá implementar una subclase de *ObjetoGrafico* que trate con texto y pase una instancia al servidor como argumento del método *nuevaForma*. Tras ello, otros clientes podrán recuperar la instancia utilizando el método *dameTodaForma*. El código de la nueva clase se descargará automáticamente desde el primer cliente hacia el servidor y desde ahí a los otros clientes según lo vayan necesitando.

◇ **RMIregistry.** RMIregistry es el enlazador para Java RMI. En cada computador de servicio que aloje objetos remotos deberá haber un ejemplar de RMIregistry. Éste da soporte a una relación en forma de tabla textual, que contiene nombres al estilo URL y referencias a objetos remotos presentes en ese computador. Se accede a ella mediante métodos de la clase *Naming*, cuyos métodos toman como argumento una cadena formateada al estilo URL de la siguiente forma:

//nombreComputador:puerto/nombreObjeto

donde *nombreComputador* y *puerto* se refieren a la ubicación de RMIregistry. Si se omite, se presupone que es el computador local y el puerto por defecto. Su interfaz ofrece los métodos que se muestran en la Figura 5.12, en la que no se listan las excepciones (todos los métodos pueden lanzar *RemoteException*). Este servicio no es un sistema de enlace con amplitud de sistema. Los clientes deben dirigir sus consultas (*lookup*) a computadores concretos.

5.5.1. CONSTRUCCIÓN DE PROGRAMAS CLIENTES Y SERVIDORES

Esta sección bosqueja los pasos necesarios para producir programas clientes y servidores que empleen las interfaces *Remote*: *Forma* y *ListaForma* que se muestran en la Figura 5.11. El programa servidor es una versión simplificada de un servidor de tipo tablero que implementa las dos interfaces *Forma* y *ListaForma*. Describiremos un programa cliente simple de escrutinio y presentaremos la técnica de devolución de llamada que se emplea para evitar la necesidad de sondear al servidor. Las versiones completas de las clases que se ilustran en esta sección están disponibles en cdk3.net/rmi.

◇ **Programa servidor.** El servidor es un servidor de tablero: representa cada forma como un objeto remoto que implementa la interfaz *Forma* y mantiene el estado de un objeto gráfico así

void rebind(String nombre, Remote obj)
Este método es empleado por un servidor para registrar el identificador de un objeto remoto mediante su nombre, como se indica en la Figura 5.13, en la línea 3.

void bind(String nombre, Remote obj)
Este método puede ser empleado, alternativamente al anterior, por un servidor para registrar un objeto remoto mediante su nombre, pero si el nombre ya está ligado a una referencia a un objeto remoto se lanza una excepción.

void unbind(String nombre, Remote obj)
Este método destruye un enlace.

Remote lookup(String nombre)
Este método es usado por los clientes para buscar un objeto remoto mediante su nombre, según se indica en la Figura 5.15, en la línea 1. Devuelve una referencia a un objeto remoto.

String [ ] list( )
Este método devuelve una tira de *Strings* que contienen los nombres enlazados del registro.

Figura 5.12. La clase *Naming* de Java RMIregistry.

como su número de versión; representa su conjunto de formas mediante un objeto remoto que implementa la interfaz *ListaForma* y mantiene un conjunto de formas en un *Vector*.

El servidor consta de un método *main* y una clase sirviente para implementar cada una de sus interfaces remotas. El método *main* del servidor crea una instancia de *SirvienteListaForma* y la enlaza a un nombre en RMIregistry, como se muestra en la Figura 5.13 (en las líneas 1 y 2). Adviértase que el valor ligado al nombre es una referencia a un objeto remoto, y su tipo es el tipo de su interfaz remota (*ListaForma*). Las dos clases sirvientes son *SirvienteListaForma*, que implementa la interfaz *ListaForma*, y *SirvienteForma*, que implementa la interfaz *Forma*. La Figura 5.14 proporciona un bosquejo de la clase *SirvienteListaForma*. Observe que *SirvienteListaForma* (en la línea 1), como muchas clases sirvientes, extiende una clase denominada *UnicastRemoteObject*, que proporciona objetos remotos que duran tanto tiempo como el proceso en que son creadas.

Las implementaciones de los métodos de la interfaz remota en una clase sirviente son directas dado que pueden escribirse sin que importen los detalles de la comunicación. Observe que el método de *nuevaForma* de la Figura 5.14 (en la línea 2), puede considerarse un método factoría dado que permite al cliente la creación de un objeto remoto. Emplea el constructor *SirvienteForma*, que crea un nuevo objeto remoto que contiene el *ObjetoGrafico* y el número de versión, que se pasan como argumentos. El tipo del valor devuelto por *nuevaForma* es *Forma*; la interfaz que implementa el nuevo objeto remoto. Antes de devolver el control, el método *nuevaForma* añade la forma nueva a su vector, que ya contiene la lista de formas (véase la línea 3).

```
import java.rmi.*;
public class ServidorListaForma{
    public static void main(String args[ ]){
        System.setSecurityManager(new RMISecurityManager( ));
        try{
            ListaForma unaListaForma = new SirvienteListaForma( );          1
            Naming.rebind("Lista Forma", unaListaForma);                    2
            System.out.println("Servidor de ListaForma listo");
        }catch (Exception e) {
            System.out.println("main del servidor ListaForma:" + e.getMessage( ));}
    }
}
```

Figura 5.13. Clase Java *ServidorListaForma* con el método *main*.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class SirvienteListaForma extends UnicastRemoteObject implements ListaForma {
    private Vector laLista;          // contiene la lista de Formas                    1
    private int version;
    public SirvienteListaForma( ) throws RemoteException{...}
    public Forma nuevaForma(ObjetoGrafico g) throws RemoteException {                   2
        version ++;
        Forma s = new SirvienteForma(g, version);                                       3
        laLista.addElement(s);
        return s;
    }
    public Vector todasFormas( ) throws RemoteException{...}
    public int dameVersion( ) throws RemoteException { ... }
}
```

Figura 5.14. Clase Java *SirvienteListaForma* implementando la interfaz *ListaForma*.

El método *main* de un servidor necesita crear un gestor de seguridad para permitir que la seguridad de Java aplique la protección apropiada para un servidor RMI. Por defecto, se proporciona un gestor de seguridad denominado *RMISecuriyManager*. Éste protege los recursos locales para asegurar que las clases que se descargan desde lugares remotos no tengan efecto alguno sobre los recursos locales como los archivos, pero diverge en que permite al programa proporcionar su propio cargador de clases y el empleo de reflexión. Si el servidor RMI no establece gestor de seguridad alguno, sólo se puede cargar cada proxy y cada clase desde la ruta local de clases (*classpath*), con el fin de proteger al programa del código descargado como resultado de las invocaciones de métodos remotas.

◇ **Programa cliente.** En la Figura 5.15 se muestra un cliente simplificado para el servidor *ListaForma*. Cualquier programa cliente comienza su andadura dialogando con el enlazador para buscar las referencias a objetos remotos. Nuestro cliente establece un gestor de seguridad y posteriormente busca una referencia a un objeto remoto para un objeto remoto empleando la operación *lookup* (busca) del RMIregistry (véase la línea 1). Habiendo obtenido una referencia a un objeto remoto inicial, el cliente continúa lanzando varios RMI a un objeto remoto, o a otros descubiertos

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ClienteListaForma{
  public static void main(String args[ ]){
    System.setSecurityManager(new RMISecurityManager( ));
    ListaForma unaListaForma = null;
    try{
      unaListaForma = (ListaForma) Naming.lookup("//bruno.ListaForma");          1
      Vector sLista = unaListaForma.todasFormas( );                               2
    }catch(RemoteException e) {System.out.println(e.getMessage( ));
    }catch(Exception e) {System.out.println("Cliente:" + e.getMessage( ));}
  }
}
```

Figura 5.15. Cliente Java de *ListaForma*.

durante su ejecución, de acuerdo con las necesidades de la aplicación. En nuestro ejemplo, el cliente invoca el método *todasFormas* en el objeto remoto (véase la línea 2) y recibe un vector de referencias a objetos remotos para todas las formas almacenadas en ese momento en el servidor. Si el cliente implementara un visualizador para el tablero, utilizaría el método del servidor *dameTodoEstado* de la interfaz *Forma* para recuperar cada uno de los objetos gráficos en el vector y mostrarlos en una ventana. Cada vez que el usuario complete un objeto gráfico, invocará el método *nuevaForma* en el servidor, pasando como argumento el nuevo objeto gráfico. El cliente conservará un registro del último número de versión en el servidor, y de vez en cuando invocará el método *dameVersion* del servidor para conocer si otros usuarios han añadidos nuevas formas. Si es así, las recuperará y visualizará.

◇ **Devolución de llamada.** La idea básica de la devolución de llamada es que en lugar de que los clientes sondeen al servidor para buscar si ha ocurrido algún evento, el servidor debería informarles si ha ocurrido algún evento. El término *devolución de llamada* se emplea para referirse a la acción del servidor en que notifica a los clientes acerca de un evento. Las devoluciones de llamada pueden implementarse en RMI como sigue:

- El cliente crea un objeto remoto que implementa una interfaz que contiene un método para que lo invoque el servidor. A este objeto lo denominamos objeto de devolución de llamada o de retrollamada.
- El servidor proporciona una operación que permite a los clientes interesados informarle de sus referencias a objetos remotos de sus objetos de retrollamada. El servidor las almacena en una lista.
- Cuando quiera que aparezca un evento de interés, el servidor llamará a los clientes interesados. Por ejemplo, el servidor de tablero llamará a sus clientes cuando quiera que se añada un objeto gráfico.

El empleo de devoluciones de llamada evita la necesidad de que un cliente sondee los objetos de interés en el servidor y las desventajas de atenderlas:

- Las prestaciones del servidor pueden degradarse si se somete a una encuesta constante.
- Los clientes podrían no notificar de las actualizaciones a los usuarios, si se basa en un método de tiempo.

Sin embargo, las devoluciones de llamada tienen otros problemas: primero, el servidor necesita tener una lista actualizada de los objetos de devolución de llamada de sus clientes. Pero puede que los clientes no siempre informen al servidor antes de terminarse, con lo que dejarían al servidor con listas incorrectas. Podría utilizarse la técnica de *concesiones*, que se discutió en la Sección 5.2.6, para soslayar este problema. El segundo problema asociado con las devoluciones de llamada, es que el servidor necesita realizar una serie de RMI síncronas a los objetos de retrollamada de la lista. Véase la Sección 5.4.1 y el Ejercicio 5.18 con algunas ideas sobre cómo resolver el segundo problema.

Ilustraremos el empleo de devoluciones de llamada en el contexto de la aplicación de tablero. La interfaz *RetrollamadaTablero* podría definirse como sigue:

```
public interface RetrollamadaTablero implements Remote {
    void retrollamada(int version) throws RemoteException;
};
```

Esta interfaz viene implementada en forma de objeto remoto por parte del cliente, permitiendo que el servidor le envíe un número de versión cuando vea que se añada un objeto nuevo. Aunque previamente a que el servidor pueda hacerlo, el cliente necesita informar al servidor sobre su objeto

de retrollamada. Para que esto sea posible, la interfaz *ListaForma* requiere métodos adicionales como *registra* y *desregistra*, que se definen como sigue:

int registra(RetrolladamaTablero retrollamada) throws RemoteException;
void desregistra(int idRetrollamada) throws RemoteException;

Después de que el cliente haya obtenido una referencia al objeto remoto con interfaz *ListaForma* (por ejemplo como en la Figura 5.15, en la línea 1) y creado una instancia de su objeto de retrollamada, emplea el método *registra* de *ListaForma* para informar al servidor de que está interesado en recibir devoluciones de llamada. El método *registra* devuelve un entero (el *idRetrollamada*) referente al registro. Cuando el cliente finaliza, debiera llamar a *desregistra* para informar al servidor de que no requiere más devoluciones de llamada. El servidor es responsable de guardar una lista de clientes interesados y de notificar a todos ellos cada vez que el número de versión del tablero se incrementa.

5.5.2. DISEÑO E IMPLENTACIÓN DE JAVA RMI

El sistema original de Java RMI empleaba todos los componentes mostrados en la Figura 5.6. Pero en Java 1.2, se utilizaron las posibilidades de reflexión para construir un distribuidor genérico y evitar la necesidad de los esqueletos. Los clientes proxy se generan mediante un compilador denominado *rmic* desde las clases compiladas del servidor; y no desde las definiciones de las interfaces remotas.

◇ **Empleo de la reflexión.** La reflexión se utiliza para pasar información, en los mensajes de petición, sobre el método que se ha de invocar. Esto se obtiene con la ayuda de la clase *Method* en el paquete de reflexión. Cada instancia de *Method* representa las características de un método particular, incluyendo su clase, los tipos de sus argumentos, el valor retornado y las excepciones. La característica más interesante de esta clase es su método *invoke*. El método *invoke* requiere dos argumentos: el primero especifica el objeto que recibirá la invocación y el segundo es una cadena de *Object* que contiene los argumentos. El resultado devuelto tiene como tipo *Object*.

Volviendo al uso de la clase *Method* en RMI: el proxy tiene que desempaquetar información acerca de un método y sus argumentos en el mensaje de *petición*. Para el método, el proxy empaqueta un objeto de clase *Method*. Pone los argumentos en una cadena de *Objetos* y después empaqueta dicha cadena. El distribuidor desempaqueta el objeto *Method* y sus argumentos desde la cadena de *Objects* del mensaje de *petición*. Como es lógico, se habrá desempaquetado la referencia al objeto remoto destino y se habrá obtenido la referencia al objeto local correspondiente desde el módulo de referencias remotas. Entonces, el distribuidor llama al método *invoke* del objeto *Method*, aportando el destino y la cadena de valores como argumento. Cuando se haya ejecutado el método, el distribuidor empaquetará el resultado de cualquier excepción en el mensaje de *respuesta*.

De este modo el distribuidor es genérico; es decir, se puede emplear el mismo distribuidor para cualquier clase de objeto remoto, sin necesitar esqueletos.

◇ **Clases de Java que dan soporte a RMI.** La Figura 5.16 muestra la estructura hereditaria de las clases que dan soporte a los servidores RMI en Java. La única clase de la que tiene que estar pendiente el programador es *UnicastRemoteObject*, que debe ser extendida por cualquier simple clase sirviente. La clase *UnicastRemoteObject* extiende una clase abstracta llamada *RemoteServer*, que proporciona versiones abstractas de los métodos que requieren los servidores remotos. *UnicastRemoteObject* fue el primer ejemplo de *RemoteServer* que se proporcionó. Otra clase, denominada *Activatable* está ahora disponible para proporcionar objetos activables. Podrían proporcionarse

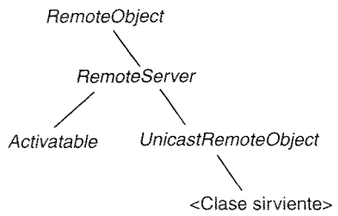


Figura 5.16. Clases de soporte de Java RMI.

varias alternativas más para objetos replicados. La clase *RemoteServer* es una subclase de *RemoteObject* que tiene una variable de instancia que aloja la referencia al objeto remoto y proporciona los siguientes métodos:

equals: este método compara referencias a objetos remotos;

toString: este método proporciona los contenidos de una referencia a un objeto remoto en forma de *String*;

readObject, *writeObject*: estos métodos deserializan/serializan objetos remotos.

Además, se puede utilizar el operador *instanceOf* para comprobar objetos remotos.

5.6. RESUMEN

Este capítulo ha discutido dos paradigmas de la programación distribuida: la invocación de métodos remotos y los sistemas basados en eventos. Ambos paradigmas contemplan los objetos como entidades independientes que pueden recibir invocaciones desde objetos remotos. En el primer caso, un método en la interfaz remota de un objeto particular es invocado síncronamente (el que invoca espera por la respuesta). En el segundo caso, se envían notificaciones asíncronamente a varios múltiples suscriptores cuando quiera que ocurra un evento publicado hacia un objeto interesado.

El modelo de objetos distribuidos es una extensión del modelo de objetos locales que se emplea en los lenguajes de programación basados en objetos. Los objetos encapsulados constituyen componentes útiles en un sistema distribuido, puesto que la encapsulación los hace enteramente responsables de gestionar su propio estado, y la invocación de métodos locales puede extenderse hacia la invocación remota. Cada objeto de un sistema distribuido tiene una referencia a un objeto remoto (un identificador global único) y una interfaz remota que especifica cuáles de sus operaciones pueden ser invocadas remotamente.

Una invocación a un método local da una semántica del tipo *exactamente una vez*, mientras que una invocación remota no puede dar las mismas garantías dado que los objetos participantes están en computadores diferentes, que pueden fallar independientemente y están ligados por una red, que también podría fallar. Lo mejor que podemos obtener es una semántica del tipo *como máximo una vez*. Debido a sus diferentes características de fallo y prestaciones, y a la posibilidad de acceso concurrente a estos objetos remotos, no parece una buena idea hacer que una invocación remota parezca idéntica a una invocación local.

Las implementaciones de middleware para RMI proporcionan componentes (que comprenden: proxy, esqueleto y distribuidor) que ocultan los detalles del empaquetado, el paso de mensajes y la localización de los objetos remotos desde los programas cliente y servidor. Estos componentes pueden generarse mediante un compilador de interfaces. Java RMI extiende la invocación local en invocación remota empleando la misma sintaxis, pero habrá que especificar las interfaces remotas

como una extensión de una interfaz denominada *Remote* y también cada método deberá lanzar la excepción *RemoteException*. Esto garantiza que los programadores son conscientes de que realizan llamadas remotas o implementan objetos remotos, y permitiéndoles gestionar los errores o diseñar objetos adecuados para su acceso en condiciones de concurrencia.

Los sistemas distribuidos basados en eventos pueden emplearse para permitir que grupos distribuidos de objetos heterogéneos se comuniquen unos con otros. A diferencia de RMI, los objetos no necesitan tener interfaces remotas para recibir mensajes; todo lo que necesitan es implementar una interfaz para recibir las notificaciones y para suscribirse a los eventos. Los objetos que generan eventos necesitan enviar notificaciones asíncronas. La simplicidad de las interfaces debiera hacer que el añadir eventos a los objetos fuera bastante sencillo. El trabajo adicional de procesar los eventos, por ejemplo filtrarlos o buscar patrones, puede ser llevado a cabo por observadores (objetos terceros que se añaden al sistema con este fin).

EJERCICIOS

5.1. La interfaz *Eleccion* proporciona dos métodos remotos:

vota: con dos parámetros mediante los que el cliente indica el nombre del candidato (una cadena de texto) y el *número del votante* (un entero que sirve para asegurar que cada usuario sólo vota una vez). Los números de votante se escogen de modo disperso del intervalo de los enteros para que sean difíciles de adivinar.

resultado: con dos parámetros mediante los cuales el servidor informa al cliente del número del candidato y el número de votos para ese candidato.

¿Cuáles de los parámetros de estos procedimientos son de *entrada* y cuáles son de *salida*?

5.2. Discuta la semántica de invocación que puede lograrse cuando se implementa el protocolo petición-respuesta sobre una conexión TCP/IP, que garantiza que los datos se entregan en el orden de su envío, sin pérdidas y sin duplicación. Tenga en cuenta todas las condiciones que pueden causar la ruptura de una conexión.

5.3. Defina la interfaz del servicio *Eleccion* en CORBA IDL y en Java RMI. Recuerde que CORBA IDL proporciona el tipo *long* para enteros de 32 bits. Compare los métodos de los dos lenguajes para especificar los argumentos de *entrada* y *salida*.

5.4. El servicio *Eleccion* debe asegurar que se registra cada voto cuando quiera que un usuario piense que ha emitido su voto.

Discuta el efecto de la semántica de llamada *pudiera ser* sobre el servicio *Eleccion*.

¿Sería aceptable la semántica de llamada *al menos una vez* para el servicio *Eleccion* o recomendaría usted una semántica de llamada del tipo *como máximo una vez*?

5.5. Un protocolo del tipo petición-respuesta está implementado sobre un servicio de comunicación con omisión de fallos para proveer una semántica de invocación RMI del tipo *al menos una vez*. En el primer caso el programador supone que está en un sistema distribuido asíncrono. En el segundo el programador supone que el tiempo máximo para la comunicación y ejecución de un método remoto es *T*. ¿En qué forma se simplifica la implementación con la segunda suposición?

5.6. Bosqueje una implementación del servicio *Eleccion* que garantice que sus registros permanecen consistentes en el caso de que se acceda a él concurrentemente desde varios clientes.

5.7. El servicio *Eleccion* debe garantizar que todos los votos se almacenan de modo seguro incluso cuando el proceso de servicio se malogra. Explique cómo puede conseguirse tomando como referencia el bosquejo de implementación de su respuesta al Ejercicio 5.6.

5.8. Muestre cómo utilizar la reflexión de Java para construir la clase proxy del cliente para la interfaz del servicio *Eleccion*. De los detalles de la implementación de uno de los métodos de esta clase, el cual debería llamar al método *hazOperacion* que posee la siguiente signatura:

byte [ ] hazOperacion (RemoteObjectRef o, Method m, byte [ ] argumentos);

Sugerencia: cualquier variable de instancia de la clase proxy debería alojar una referencia a un objeto remoto (véase el Ejercicio 4.12).

5.9. Muestre cómo generar una clase proxy del cliente empleando un lenguaje como C++ (sin soporte de reflexión), por ejemplo para la definición de interfaz en CORBA dada en su respuesta al Ejercicio 5.3. De los detalles de la implementación de uno de los métodos de esta clase, que debería llamar al método *hazOperacion* definido en la Figura 4.12.

5.10. Explique cómo emplear la reflexión de Java para construir un distribuidor genérico. Dé el código en Java para un distribuidor cuya signatura sea:

public void despacha(Object destino, Method unMetodo, byte [ ] argumentos);

Los argumentos proporcionan el objeto destino, el método que se ha de invocar y los argumentos para ese método en una cadena de bytes.

5.11. El Ejercicio 5.8 requería que el cliente convirtiera argumentos de tipo *Object* en una cadena de bytes antes de invocar *hazOperacion* y el Ejercicio 5.10 requería que el distribuidor convirtiera una cadena de bytes en una cadena de elementos de tipo *Object* antes de invocar el método. Discuta la implementación de una nueva versión de *hazOperacion* con la siguiente signatura:

Object [ ] hazOperacion(RemoteObjectRef o, Method m, Object [ ] argumentos);

Que emplea las clases *FlujoSalidaObjeto* y *FlujoEntradaObjeto* para encauzar los mensajes de petición y respuesta entre el cliente y el servidor sobre una conexión TCP. ¿Cómo afectarán estos cambios al diseño del distribuidor?

5.12. Un cliente realiza llamadas a procedimientos remotos de un servidor. El cliente invierte 5 milisegundos en calcular los argumentos para cada petición, y el servidor se toma 10 milisegundos para procesar cada respuesta. El tiempo de procesamiento del sistema operativo local, para cada envío y recepción, es de 0,5 milisegundos, y el tiempo de la red para transmitir cada petición o cada respuesta es de 3 milisegundos. Tanto el empaquetado como el desempaquetado llevan 0,5 milisegundos por cada mensaje.

Calcúlese el tiempo que se invierte en el cliente para generar y volver de dos peticiones:

(i) Si es de un solo hilo.
(ii) Si tiene dos hilos que pueden hacer peticiones concurrentes sobre un único procesador.

Pueden ignorarse los tiempos de cambio de contexto. ¿Es necesario que RPC sea asíncrona si los procesos cliente y servidor son multihilo?

5.13. Diseñe una tabla de objetos remotos que pueda soportar compactación automática de memoria distribuida así como la traducción entre las referencias a objetos locales y remotos. Dé un ejemplo que implique varios objetos remotos y varios proxy en varios lugares para

ilustrar el empleo de la tabla. Muestre los cambios en la tabla cuando se crea un nuevo proxy como consecuencia de una invocación remota. Asimismo, muestre los cambios en la tabla cuando uno de los proxy se vuelve inaccesible.

5.14. Una versión simplificada del algoritmo de compactación automática de memoria distribuido que se describe en la Sección 5.2.6 sólo invoca *añadeRef* en el lugar donde vive cada objeto cuando se cree un proxy y *eliminaRef* cuando se destruya un proxy. Bosqueje todos los posibles efectos de los fallos, de la comunicación y los procesos, sobre el algoritmo. Sugiera cómo sobrellevar cada uno de estos efectos, aunque sin usar concesiones.

5.15. Discuta cómo utilizar los eventos y las notificaciones, tal y como se describen en la especificación de eventos distribuidos de Jini, en el contexto de la aplicación de tablero compartido. La clase *RemoteEvent* se define como sigue (según Arnold y otros [1999]).

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object fuente, long idEvento,
        long numSec, MarshalledObject vuelta)
    public Object getSource( ) {...}
    public long getID( ) {...}
    public long getSequenceNumber( ) {...}
    public MarshalledObject getRegistrationObject( ) {...}
}
```

El primer argumento del constructor es un objeto remoto. Las notificaciones informan a los oyentes de que ha ocurrido un evento, aunque los oyentes son responsables del resto de los detalles.

5.16 Sugiera un diseño para un servicio de notificación de buzones, que se supone que almacena notificaciones en representación de múltiples suscriptores, y que permite a los suscriptores especificar cuándo desean que se les repartan las notificaciones. Explique cómo los suscriptores que no siempre están activos pueden hacer uso del servicio que usted describe. ¿Cómo se las arregla el servicio con los suscriptores que se malogren justo cuando esté activo el reparto?

5.17. Explique cómo se puede emplear un observador de redireccionamiento para mejorar la fiabilidad y las prestaciones de los objetos de interés en un servicio de eventos.

5.18. Sugiera formas en las que se puedan usar los observadores para mejorar la fiabilidad o las prestaciones de su solución al Ejercicio 5.13.

6

SOPORTE DEL SISTEMA OPERATIVO

6.1. Introducción
6.2. El nivel de sistema operativo
6.3. Protección
6.4. Procesos e hilos
6.5. Comunicación e invocación
6.6. Arquitectura del sistema operativo
6.7. Resumen

Este capítulo describe cómo los servicios del sistema operativo en los nodos de un sistema distribuido dan soporte al middleware. El sistema operativo facilita la encapsulación y protección de los recursos dentro de los servidores; además soporta los mecanismos de invocación requeridos para el acceso a dichos recursos, incluyendo comunicación y planificación.

Una cuestión importante en este capítulo es el papel del núcleo del sistema. Este capítulo aspira a proporcionar al lector la comprensión de las ventajas e inconvenientes de la división de la funcionalidad entre dominios de protección; en particular, la división de la funcionalidad entre el núcleo y el código de nivel de usuario. Se discute la relación entre los recursos del nivel de núcleo y los del nivel de usuario, incluyendo el enfrentamiento entre la eficiencia y la robustez.

El capítulo analiza el diseño e implementación del procesamiento multi-hilo y de los servicios de comunicación. Además, se exploran las principales arquitecturas de núcleo que han sido desarrolladas.

6.1. INTRODUCCIÓN

El Capítulo 2 introdujo los principales niveles de software que existen en un sistema distribuido. Hemos aprendido que un aspecto importante de los sistemas distribuidos es la compartición de recursos. Las aplicaciones cliente solicitan operaciones sobre recursos que, a menudo, residen en otro nodo o, como mínimo, en otro proceso. Las aplicaciones (en forma de clientes) y los servicios (en forma de gestores de recursos) utilizan el nivel de middleware para sus interacciones. El middleware permite invocaciones remotas entre objetos o procesos en los nodos de un sistema distribuido. En el Capítulo 5 se han explicado los principales tipos de invocaciones remotas que aparecen en el middleware, del tipo de Java RMI y CORBA. En este capítulo continuaremos centrándonos en las invocaciones remotas, sin tener en cuenta requisitos de tiempo real (el Capítulo 5 examinó el soporte para las comunicaciones multimedia, las cuales son de tipo tiempo real y orientadas a flujos de datos).

Por debajo del nivel de middleware se sitúa el nivel de sistema operativo (OS), el tema de este capítulo. Examinaremos la relación entre ambos, y en particular la forma en la que el sistema operativo puede resolver los requisitos del middleware. Entre estos requisitos se incluye el acceso robusto y eficiente a los recursos físicos junto con la flexibilidad para implementar múltiples políticas de gestión de recursos.

La tarea de un sistema operativo es la de proporcionar abstracciones, orientadas a un cierto problema, de los recursos físicos subyacentes, entre los que están el procesador, la memoria, las comunicaciones y los dispositivos de almacenamiento. Un sistema operativo de tipo UNIX o Windows NT [Custer 1998] proporciona al programador, por ejemplo, archivos en lugar de bloques de disco o sockets en lugar de acceso directo a la red. Se hace cargo de los recursos físicos en cada nodo y los gestiona para proporcionar las abstracciones de dichos recursos mediante una interfaz de llamadas al sistema.

Antes de comenzar con un estudio detallado de las tareas de soporte del middleware proporcionadas por el sistema operativo, resultará útil tener una cierta perspectiva histórica a través del estudio de dos nociones distintas de sistema operativo que han surgido durante el desarrollo de los sistemas distribuidos: sistemas operativos en red y sistemas operativos distribuidos. Las definiciones que se dan de ambos pueden variar, pero los conceptos fundamentales son los que se presentan a continuación.

Tanto UNIX como Windows NT son ejemplos de *sistemas operativos en red*. Tienen, incluida en ellos mismos, la capacidad de acceder a la red y por lo tanto pueden utilizarse para el acceso a recursos remotos. Este acceso se realiza de forma transparente a la red para algunos (aunque no todos) los tipos de recurso. Por ejemplo, mediante un sistema de archivos distribuido como NFS, los usuarios pueden acceder a los archivos de forma transparente a la red. Esto quiere decir que muchos de los archivos que utilizan los usuarios se almacenan de forma remota, en un servidor, y esto se realiza de forma completamente transparente a las aplicaciones.

Sin embargo, la principal característica diferenciadora consiste en que los nodos que ejecutan un sistema operativo en red mantienen la autonomía para la gestión de sus propios recursos de procesamiento. Es decir, existen múltiples imágenes del sistema, una en cada nodo. En un sistema operativo en red un usuario puede acceder de forma remota a otro computador mediante el uso de *rlogin* o *telnet*, y lanzar procesos en ese computador. Sin embargo, al contrario del control que un sistema operativo tiene sobre los procesos que se ejecutan en su propio nodo, no se pueden planificar los procesos entre diferentes nodos. El usuario deberá implicarse en esa tarea.

Por otra parte podemos concebir un sistema operativo sobre el cual los usuarios no deban preocuparse acerca del nodo en el que se lanzan sus programas, o de la ubicación de sus recursos. Existe una *única imagen del sistema*. El sistema operativo controla todos los nodos del sistema, y de forma transparente envía los procesos nuevos a cualquier nodo que cumpla su política de planificación. Por ejemplo, se puede crear un nuevo proceso en el nodo con menor carga de trabajo del sistema, para evitar que pueda haber nodos sobrecargados arbitrariamente.

Un sistema operativo que genera, según la forma descrita, una única imagen del sistema para todos los recursos en un sistema distribuido se llama *sistema operativo distribuido* [Tanenbaum y van Renesse 1985].

◇ **Middleware y sistemas operativos en red.** En realidad, no se emplea ampliamente ningún sistema operativo distribuido, sino sólo sistemas operativos en red del tipo de UNIX, MacOS y diferentes variantes de Windows. Esta situación tiende a mantenerse, principalmente por dos razones. La primera es que los usuarios ya han realizado grandes inversiones en su software de aplicación, que normalmente resuelve sus problemas; por lo tanto no cambiarán a un nuevo sistema operativo que es incapaz de ejecutar sus aplicaciones, por muchas ventajas que ofrezca. Se han realizado distintos esfuerzos para emular los núcleos de UNIX y otros sistemas operativos sobre otros núcleos, pero las prestaciones de estas emulaciones no han sido nunca satisfactorias. En cualquier caso, el mantener actualizadas las emulaciones de los principales sistemas operativos al tiempo que evolucionan sería una tarea incalculable.

La segunda razón en contra de la utilización de un sistema operativo distribuido es que los usuarios prefieren tener autonomía en la gestión de sus propias máquinas, incluso en organizaciones fuertemente cohesionadas. Esto es así, concretamente, para conseguir buenas prestaciones [Douglis y Ousterhout 1991]. Por ejemplo, Pérez necesita buena respuesta interactiva mientras escribe sus documentos y podría molestarse si los programas de González retrasaran su trabajo.
La combinación de middleware y sistemas operativos en red proporciona un equilibrio aceptable entre los requisitos de autonomía, por un lado, y la transparencia de red en el acceso a recursos, por el otro. Los sistemas operativos en red permiten a los usuarios ejecutar su procesador de textos preferido junto con otro tipo de aplicaciones de tipo independiente. El middleware les permite acceder a los servicios que aparecen como disponibles en el sistema distribuido.

En la siguiente sección se explica el funcionamiento del nivel de sistema operativo. En la Sección 6.2 se examinan los mecanismos de bajo nivel dedicados a la protección de recursos; éstos son necesarios para comprender la relación entre procesos e hilos y el papel del núcleo. La Sección 6.4 examina las abstracciones de proceso, espacio de direcciones e hilos. En este apartado los principales conceptos son la concurrencia, la gestión y protección de recursos locales y la planificación. La Sección 6.5 estudia las comunicaciones como parte de los mecanismos de invocación. La Sección 6.6 abarca las diferentes arquitecturas existentes de sistemas operativos, incluyendo los diseños monolíticos y de micronúcleo. En el Capítulo 18 se realizará un estudio del núcleo de Mach. El lector podrá encontrar estudios del caso para los sistemas operativos Amoeba, Chorus y Clouds en www.cdk3.net/oss.

6.2. EL NIVEL DE SISTEMA OPERATIVO

Los usuarios únicamente estarán satisfechos si la combinación entre el middleware y el sistema operativo proporciona buenas prestaciones. El middleware se ejecuta en múltiples combinaciones hardware/sistema operativo, es decir, en múltiples plataformas en los nodos de un sistema distribuido. El sistema operativo que se ejecuta en un cierto nodo (un núcleo junto con los servicios asociados de nivel de usuario, por ejemplo bibliotecas) proporciona dentro de ese nodo su propia imagen sobre las abstracciones de los recursos hardware locales de procesamiento, de almacenamiento y de comunicación. El middleware utiliza una combinación de esos recursos locales para implementar los mecanismos de invocación remota entre objetos o procesos en los nodos.

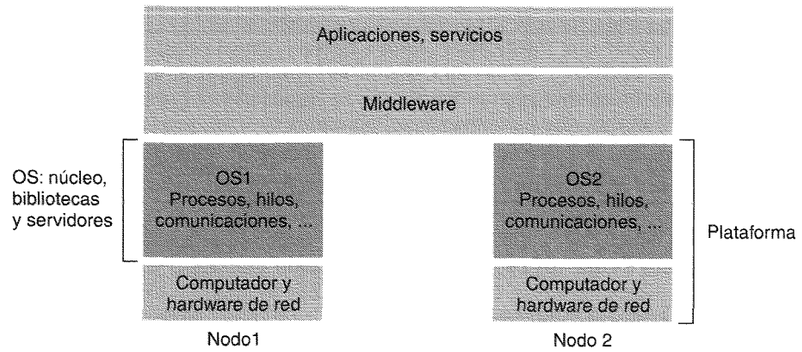


Figura 6.1. Niveles del sistema.

La Figura 6.1 muestra cómo el nivel de sistema operativo en cada uno de los dos nodos soporta un nivel de middleware común para proporcionar una infraestructura distribuida para aplicaciones y servicios.

Nuestro objetivo en este capítulo es examinar el impacto que los mecanismos concretos de los sistemas operativos tienen en la habilidad del middleware para proporcionar compartición de recursos distribuida a los usuarios. Los núcleos y los procesos cliente y servidor que se ejecutan entre ellos son los principales componentes de la arquitectura sobre los que trataremos. Los núcleos y el proceso servidor son los componentes que gestionan los recursos y los presentan a los clientes a través de una interfaz de recursos. Esta interfaz debe tener al menos las siguientes características:

Encapsulamiento: debe proporcionar un servicio de interfaz útil sobre los recursos, es decir, un conjunto de operaciones que cubra las necesidades de sus clientes. Los detalles del tipo gestión de la memoria o dispositivos utilizados para implementar los recursos deben ocultarse a los clientes.

Protección: los recursos deben protegerse de los accesos no permitidos; por ejemplo, los archivos se protegen contra la lectura por parte de usuarios que no tengan dicho permiso, así como los registros de los dispositivos se protegen de los procesos de usuario.

Procesamiento concurrente: los clientes pueden compartir múltiples recursos y acceder a ellos concurrentemente. Los gestores de dichos recursos son los responsables de conseguir transparencia en la concurrencia.

Los clientes acceden a los recursos mediante, por ejemplo, invocaciones a un método remoto de un objeto que reside en un cierto servidor o bien mediante llamadas al sistema en el núcleo. El mecanismo de acceso a un recurso encapsulado se denomina *mecanismo de invocación*, independientemente de cómo esté implementado. Una combinación de bibliotecas, núcleos y servidores puede utilizarse para realizar las siguientes tareas de invocación:

Comunicación: los parámetros de operación y los resultados deben pasarse hacia y desde los gestores de recursos, respectivamente, utilizando una red o bien dentro del computador.

Planificación: cuando se invoca una cierta operación, su procesamiento debe planificarse dentro del núcleo o del servidor.

La Figura 6.2 muestra la funcionalidad básica del sistema operativo la cual está relacionada con: la gestión de procesos e hilos, la gestión de la memoria, y la comunicación entre procesos en el mismo computador (las divisiones horizontales en la figura indican dependencias). El núcleo proporciona la mayor parte de esta funcionalidad; o bien la proporciona de forma completa en el caso de algunos sistemas operativos.

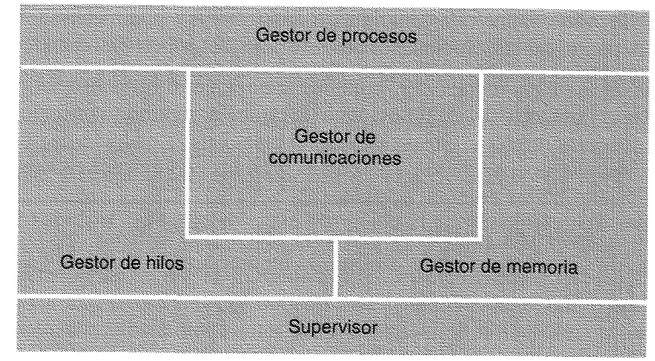


Figura 6.2. Funcionalidad básica del sistema operativo.

El software del sistema operativo se diseña para ser portable, en la mayor medida en que sea posible, entre las diferentes arquitecturas de computadores. Esto supone que en su mayor parte se codifica en un lenguaje de alto nivel, ya sea C, C++ o Modula-3, y que sus servicios se organizan por niveles de forma que los componentes dependientes de la máquina se reducen a un único nivel inferior. Algunos núcleos pueden ejecutarse en multiprocesadores de memoria compartida, cuestión que se discute en el siguiente recuadro.

◇ **Multiprocesadores de memoria compartida.** Los computadores multiprocesador de memoria compartida se componen de varios procesadores que comparten uno o más módulos de memoria (RAM). De forma añadida, los procesadores pueden tener también su propia memoria privada. Los computadores multiprocesador se pueden construir de diferentes formas [Stone 1993]. Los multiprocesadores más simples y menos caros están construidos incorporando una tarjeta que contiene unos pocos procesadores (2-8) en un computador personal.

En la *arquitectura de procesamiento simétrico* cada procesador ejecuta el mismo núcleo y los núcleos realizan, en su mayor parte, las mismas tareas en la gestión de los recursos hardware. Los núcleos comparten las estructuras de datos clave como la cola de hilos en estado de ejecución, siendo sin embargo privados parte de sus datos de trabajo. Cada procesador puede ejecutar un hilo de forma simultánea, el cual accede a datos de la memoria compartida, que pueden ser privados (protegidos mediante hardware) o compartidos con otros hilos.

Los multiprocesadores se pueden utilizar para múltiples tareas de computación de altas prestaciones. En los sistemas distribuidos son particularmente muy utilizados para la implementación de servidores de altas prestaciones ya que el servidor puede ejecutar un único programa con varios hilos los cuales gestionan varias solicitudes llegadas de diferentes clientes de forma simultánea, proporcionando, por ejemplo, el acceso a una base de datos compartida (véase la Sección 6.4).

Los principales componentes de un sistema operativo son los siguientes:

Gestor de procesos: gestiona la creación y las diferentes operaciones sobre procesos. Un proceso es una unidad de gestión de recursos que incluye un espacio de direcciones y uno o más hilos.

Gestor de hilos: incluye la creación, sincronización y planificación de hilos. Los hilos son actividades planificables asociadas a procesos. Se describen de forma completa en la Sección 6.4.

Gestor de las comunicaciones: comunicaciones entre hilos asociados a diferentes procesos en un mismo computador. Algunos núcleos también soportan comunicaciones entre hilos asociados a procesos remotos. Otros núcleos no manejan el concepto de computador remoto de forma que deben añadirse servicios adicionales para las comunicaciones externas. La Sección 6.5 trata del diseño de las comunicaciones.

Gestor de memoria: gestión de memoria física y virtual. Las Secciones 6.4 y 6.5 describen la utilización de las técnicas de gestión de memoria para la copia y la compartición de datos de forma eficiente.

Supervisor: resolución de interrupciones, interrupciones internas de llamada al sistema y otras excepciones; control de la unidad de gestión de memoria y de las antememorias (caché) hardware; manipulación de registros del procesador y de la unidad en punto flotante. A todo esto se le llama Nivel de Abstracción del Hardware en Windows NT. El lector puede acudir a Bacon [1998] y Tanenbaum [1992] para una descripción más completa de los aspectos del núcleo que son más dependientes del computador.

6.3. PROTECCIÓN

Como se dijo previamente, los recursos necesitan protección contra los accesos no permitidos. Hay que tener en cuenta que las amenazas a la integridad de un sistema no provienen únicamente de código ideado de forma maliciosa. También un código bien intencionado puede contener un error o puede tener un comportamiento no esperado, provocando que parte del sistema se comporte a su vez de forma incorrecta.

Para comprender lo que se quiere decir con «acceso ilegítimo» a un recurso, consideremos un archivo. Supongamos, a efectos de explicación, que sobre los archivos abiertos, únicamente se pueden realizar dos operaciones, *lectura* y *escritura*. La protección del archivo se puede descomponer en dos subproblemas. El primero consiste en asegurar que cada una de las dos operaciones sobre el archivo puede ser realizada únicamente por clientes con los derechos suficientes. Por ejemplo, González, que es el propietario del archivo, tiene los derechos de *lectura* y *escritura* sobre él. Pérez sólo necesita acceder a la operación de *lectura*. En este caso se produciría un acceso ilegítimo si Pérez intentara realizar una operación de *escritura* sobre el archivo. Una solución completa a este subproblema de protección de recursos en un sistema distribuido requeriría técnicas criptográficas que se estudiarán en el Capítulo 7.

El otro tipo de acceso ilegítimo, que es preciso tratar, ocurre cuando un proceso se comporta de forma errónea, esquivando las operaciones que se exportan sobre un recurso. En nuestro ejemplo esto ocurriría si González o Pérez, de alguna forma, ejecutaran una operación que no fuera *lectura* ni *escritura*. Supongamos, por ejemplo, que González tratara de acceder directamente a la variable de puntero al archivo. Para ello construiría la operación *setFilePointerRandomly*, la cual asignaría un valor arbitrario al puntero al archivo. Ésta es, desde luego, una operación sin sentido que podría perturbar la normal utilización del archivo y que nunca debiera exportarse como tal.

Podemos proteger, los recursos, de invocaciones ilegítimas del tipo de *setFilePointerRandomly*. Una posibilidad consiste en utilizar un lenguaje de programación con sistema de tipos seguro (*type-safe*), como Java o Modula-3. Estos lenguajes no permiten que un módulo acceda a otro módulo objeto a no ser que previamente obtenga una referencia a este último; él mismo no puede crear directamente un puntero que lo referencie, como se haría en C o C++. Además sólo podrá utilizar la referencia al módulo destino para realizar aquellas invocaciones (llamadas a método o a procedimiento) que el programador del módulo destino haya publicado. En otras palabras, no se pueden cambiar las variables del destino de forma arbitraria. Por el contrario, en C++ un programador puede realizar asignaciones a un puntero a voluntad, realizando de esta forma invocaciones de tipo inseguro.

También podemos utilizar soporte hardware para proteger los módulos entre sí en el nivel de invocaciones individuales, independientemente del lenguaje en que hayan sido escritas. Para operar con este esquema en un computador de propósito general se necesita un núcleo.

◇ **Núcleos y protección.** El núcleo es un programa cuya principal propiedad es que su código siempre se ejecuta con privilegios completos de acceso a los recursos físicos en el computador huésped. En concreto puede controlar la unidad de gestión de memoria y escribir en los registros del procesador de forma que se consiga impedir que cualquier otro código acceda a los recursos físicos de la máquina excepto en las formas aceptables.

La mayor parte de los procesadores tienen un registro de modo cuyo valor determina si pueden realizarse o no instrucciones privilegiadas; ejemplos de estas instrucciones son aquellas que determinan qué tablas de protección debe emplear la unidad de gestión de memoria. Un proceso del núcleo se ejecuta siempre con el procesador en modo *supervisor* (privilegiado); el núcleo hace que el resto de procesos se ejecuten en modo *usuario* (no privilegiado).

El núcleo también gestiona los *espacios de direcciones* para protegerse a sí mismo, y a otros procesos, de accesos desde un proceso anómalo, y para proporcionar a los procesos su espacio de memoria virtual. Un espacio de direcciones es un conjunto de rangos de posiciones de memoria virtual, existiendo en cada uno de ellos una combinación de derechos de acceso del tipo: acceso en sólo lectura o bien acceso en lectura-escritura. Un proceso no puede acceder a la memoria fuera de su espacio de direcciones. Los términos *proceso de usuario* o bien *proceso de nivel de usuario* se utilizan normalmente para describir aquellos procesos que se ejecutan en modo usuario y tienen un espacio de direcciones de nivel de usuario; es decir, un espacio de direcciones con derechos de acceso a memoria restringidos en comparación con el espacio de direcciones del núcleo.

Cuando un proceso ejecuta su código de aplicación lo hace en un espacio de direcciones de nivel de usuario único para esa aplicación; por el contrario cuando el mismo proceso ejecuta código del núcleo lo hace dentro del espacio de direcciones del núcleo. El proceso puede conmutar de forma segura entre el espacio de direcciones de nivel de usuario y el espacio de direcciones del núcleo por medio de una excepción, como una interrupción o bien una *interrupción interna de llamada al sistema*; este último mecanismo es el utilizado para acceder a los recursos gestionados por el núcleo. Una interrupción interna de llamada al sistema se implementa a través de una instrucción de lenguaje máquina llamada TRAP, la cual pone el procesador en modo supervisor y conmuta al espacio de direcciones del núcleo. Cuando se ejecuta la instrucción TRAP, de igual forma que ocurre en cualquier otro tipo de excepción, el hardware provoca que el procesador ejecute la rutina de manejo proporcionada por el núcleo para impedir que ningún proceso pueda acceder de forma ilegal al hardware.

Los programas deben pagar un precio por la protección. La conmutación entre espacios de direcciones consume varios ciclos de procesador y una interrupción interna de llamada al sistema es una operación más cara que una invocación a un procedimiento o a un método. Veremos a continuación cómo estas penalizaciones afectan a los costes de invocación.

6.4. PROCESOS E HILOS

El concepto tradicional en los sistemas operativos de que un proceso ejecuta una única actividad, ya se comprobó, en la década de los ochenta, que era poco adecuado para las especificaciones de los sistemas distribuidos; y también para ciertas aplicaciones sobre un solo computador que nece-

sitan concurrencia interna. El problema, como estudiaremos a continuación, es que los procesos tradicionales realizan la compartición entre actividades relacionadas de una forma difícil y cara.

La solución propuesta consistió en ampliar el concepto de proceso de forma que pudiera asociarse a múltiples actividades. Actualmente un proceso consiste en un entorno de ejecución formado por uno o más hilos. Un *hilo* es una abstracción del sistema operativo asociada a una actividad (este término deriva de la frase «hilo de ejecución»). Un *entorno de ejecución* equivale a una unidad de gestión de recursos: una colección de recursos locales gestionados por el núcleo sobre los que tienen acceso los hilos. Un entorno de ejecución consiste básicamente de los siguientes elementos:

- Un espacio de direcciones.
- Recursos de comunicación y sincronización de hilos, como semáforos e interfaces de comunicación (por ejemplo, sockets).
- Recursos de alto nivel, como archivos abiertos y ventanas.

Los entornos de ejecución son ciertamente caros en cuanto a su creación y gestión, aunque sin embargo pueden ser compartidos por más de un hilo, es decir, éstos pueden compartir todos los recursos accesibles dentro de cada entorno. En otras palabras, un entorno de ejecución representa el dominio de protección en el que se ejecutan los hilos.

Los hilos pueden crearse y destruirse de forma dinámica, según se necesite. El principal propósito para la existencia de múltiples hilos de ejecución es maximizar el grado de ejecución concurrente entre las diferentes operaciones, permitiendo así habilitar el solapamiento de la computación con la entrada-salida y el procesamiento concurrente en los multiprocesadores. Esto es particularmente útil en los servidores en los que el procesamiento concurrente de las solicitudes de los clientes pueden reducir la tendencia de los servidores a convertirse en cuellos de botella. Por ejemplo, un hilo puede procesar una solicitud de un cliente mientras que un segundo hilo que está sirviendo otra solicitud está esperando por la finalización de un acceso a disco para poder terminar.

Un entorno de ejecución proporciona protección contra los hilos externos de forma que los datos y otros recursos contenidos en él sean inaccesibles para los hilos residentes en otros entornos de ejecución. Sin embargo ciertos núcleos permiten la compartición controlada de recursos como la memoria física, entre entornos de ejecución diferentes residentes en el mismo computador.

Debido a que algunos sistemas operativos permiten un único hilo por proceso algunas veces usaremos el término *proceso multi-hilo* para resaltar la idea. Desconcertantemente, algunos modelos de programación y de sistemas operativos asocian el término proceso a lo que hemos llamado hilo. El lector encontrará en la literatura específica sobre el tema los términos *proceso de peso pesado* para indicar un entorno de ejecución y *proceso de peso ligero* en el caso contrario. En el recuadro siguiente se describe mediante una analogía la discusión entre hilos y entornos de ejecución.

6.4.1. ESPACIOS DE DIRECCIONES

Un espacio de direcciones, concepto introducido en la sección anterior, es una unidad de gestión de la memoria virtual de un proceso. Es grande (normalmente hasta 2^{32} bytes y a veces hasta 2^{64} bytes) y está formado por una o más *regiones*, separadas por áreas de memoria virtual inaccesibles. Una región (véase la Figura 6.3) es una zona de memoria virtual contigua accesible por los hilos del proceso propietario. Las regiones no se solapan. Hay que resaltar la distinción entre regiones y sus contenidos. Cada región se caracteriza por las siguientes propiedades:

- Su tamaño (su dirección virtual más baja y su tamaño).

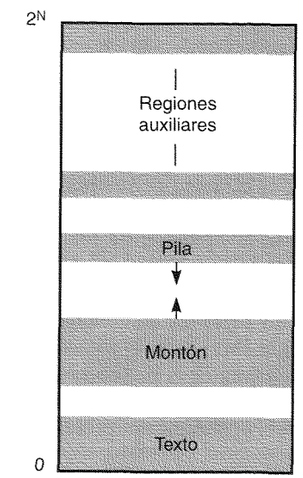


Figura 6.3. Espaciado de direcciones.

◇ **Semejanzas entre hilos y procesos.** La siguiente descripción de hilos y procesos, particularmente descollante aunque un poco desagradable, fue encontrada en el grupo de USENET *comp.os.mach* y se debe a Chris Lloyd. Un entorno de ejecución es como un tarro taponado junto con el aire y la comida dentro de él. Inicialmente hay una mosca (un hilo) en el tarro. Esta mosca puede producir otras moscas y matarlas. Cualquier mosca puede consumir cualquier recurso (aire o comida) del tarro. Estos insectos podrían programarse para hacer cola de forma ordenada en el acceso a los recursos. Si las moscas resultan ser indisciplinadas, chocarán entre ellas dentro del tarro y producirán resultados impredecibles cuando intenten consumir los mismos recursos de una forma descontrolada. Podrán comunicarse entre ellas (enviar mensajes), y con otras moscas de otros recipientes, pero ninguna podrá escapar de su vasija al igual que ninguna mosca de fuera podrá entrar. Desde este punto de vista, un proceso UNIX estándar es un único tarro con una sola mosca estéril dentro de él.

- Los permisos de lectura/escritura/ejecución para los hilos del proceso.
- Una indicación de crecimiento hacia arriba o hacia abajo.

Hay que destacar que este modelo está orientado a páginas, no a segmentos. Las regiones, al contrario que los segmentos, podrían eventualmente solaparse si su tamaño se extendiera. Entre las regiones se dejan huecos para permitir su crecimiento. Esta representación de un espacio de direcciones como un conjunto disperso de regiones disjuntas es una generalización del espacio de direcciones de UNIX, el cual está formado por tres regiones: una región de texto de tamaño fijo, y no modificable, que contiene el código del programa; un montón (*heap*), parte del cual se inicializa con los valores almacenados en el archivo binario con el programa, y que es ampliable hacia direcciones virtuales crecientes; y una pila, que crecerá hacia direcciones virtuales decrecientes.

La existencia de un número indeterminado de regiones tiene su razón de ser en varios factores. Uno de ellos es la necesidad de que haya pilas separadas para cada hilo. La existencia de regiones de pila distintas para cada hilo permite detectar los intentos de exceder el límite de la pila y controlar el crecimiento de cada una de ellas. La memoria virtual no utilizada se halla debajo de cada

región de pila y los intentos de acceder a esa región generarán excepciones (faltas de página). Una alternativa consiste en asignar la pila de cada hilo en el montón, sin embargo en este caso se hace difícil detectar cuándo un hilo excede el límite de su pila.

Otra motivación es la de permitir que un archivo (cualquiera de ellos no sólo las secciones de texto y datos de los archivos binarios) puedan ponerse en correspondencia con espacios de direcciones. Los *archivos indexados por posición* (*memory mapped*) permiten un acceso similar a un vector de bytes en memoria. El sistema de memoria virtual permite que los accesos realizados en memoria se transmitan al sistema de almacenamiento de archivos. La Sección 18.6 describe cómo el núcleo de Mach extiende la abstracción de memoria virtual de forma que las regiones puedan corresponderse de forma arbitraria con objetos en la memoria y no únicamente con archivos.

La necesidad de compartir memoria entre procesos, o entre procesos y el núcleo, es otro factor que deriva en la necesidad de regiones adicionales en el espacio de direcciones. Una *región de memoria compartida* (de forma abreviada *región compartida*) es aquella en la que se asocia la misma memoria física a una o más regiones pertenecientes a otros espacios de direcciones. Por lo tanto los procesos acceden a los mismos contenidos de memoria en las regiones compartidas mientras que sus regiones no compartidas permanecen protegidas. Algunos usos de las regiones compartidas son los siguientes:

Bibliotecas: el código de las bibliotecas puede llegar a ser muy grande y se desperdiciaría una cantidad considerable de memoria si se cargaran de forma separada en cada proceso que las utilizase. Por contra, varios procesos pueden compartir una sola copia del código de las bibliotecas empleando una región compartida del espacio de direcciones.

Núcleo: frecuentemente el código y los datos del núcleo se corresponden dentro del espacio de direcciones en las mismas posiciones. Cuando un proceso realice una llamada al sistema, o bien cuando se genere una excepción, no habrá necesidad de conmutar a un nuevo conjunto de direcciones.

Compartición de datos y comunicación: entre dos procesos, o bien entre un proceso y el núcleo, se puede necesitar compartir datos para cooperar en la realización de una cierta tarea. Es más eficiente utilizar regiones de direcciones compartidas en lugar de intercambiar los datos mediante mensajes. La utilización de regiones compartidas para la comunicación se describirá en la Sección 6.5.

6.4.2. CREACIÓN DE UN PROCESO NUEVO

Tradicionalmente la creación de un proceso nuevo es una operación indivisible realizada por el sistema operativo. Por ejemplo, la llamada al sistema *fork* de UNIX crea un proceso con un entorno de ejecución que es copia del entorno del proceso que la invoca (y sólo difiere en el valor de retorno de *fork*). La llamada al sistema *exec* de UNIX transforma el proceso que la invoca en otro, producto de ejecutar el código del programa que se le indica.

Para un sistema distribuido el diseño del mecanismo de creación de un proceso debe tener en cuenta la utilización de múltiples computadores; de esta forma la infraestructura de gestión de procesos se divide en servicios de sistema separados.

La creación de un proceso nuevo puede separarse en dos aspectos independientes:

- La elección del computador destino. Por ejemplo, el nodo puede elegirse entre varios de un grupo (*clúster*) de computadores que actúan como servidores de computación (véase el recuadro siguiente).
- La creación de un entorno de ejecución (y la de un hilo inicial en él).

◇ **Elección del nodo de proceso.** La elección del nodo en el que residirá el nuevo proceso (la decisión de la ubicación de procesos) es una cuestión de política. En general, las políticas de localización de procesos varían desde aquellas que siempre lanzan los procesos nuevos en la estación de trabajo originaria hasta aquellos que comparten la carga de procesamiento entre un conjunto de computadores. Eager y otros [1986] distinguen las siguientes categorías de políticas de compartición de la carga.

La *política de transferencia* decide si un proceso nuevo debe ubicarse en el nodo local o en un nodo remoto. Esta decisión dependerá, por ejemplo, de que el nodo local esté poco o muy cargado.

La *política de ubicación* determina qué nodo alojará un proceso nuevo seleccionado para transferirse. Esta decisión puede depender de las cargas relativas de los nodos, de su arquitectura y de algún recurso específico que posea. El sistema V [Cheriton 1984] y Sprite [Douglis y Ousterhout 1991] proporcionan mandatos de usuario para lanzar programas en estaciones de trabajo ociosas (a menudo hay varias al mismo tiempo) elegidas por el sistema operativo. En el sistema Amoeba [Tanenbaum 1990] el *servidor de ejecución* elige para cada proceso un nodo de entre un conjunto de procesadores compartidos. En todos los casos la elección del nodo destino es transparente tanto para el programador como para el usuario. Sin embargo, aquellos que programen para conseguir paralelismo explícito o tolerancia a fallos necesitan un medio para especificar la localización de cada proceso.

◇ **Clústeres.** Un grupo (*clúster*) no es más que un conjunto de computadores corrientes (desde decenas a varias centenas) conectados mediante una red de comunicaciones de alta velocidad, por ejemplo, Ethernet 100 megabit/segundo. Cada computador individualmente puede ser un PC usual, una estación de trabajo o varias tarjetas de procesador tipo PC montadas en grupo; cada nodo puede ser de tipo monoprocesador o multiprocesador. Una aplicación de los grupos es la de proporcionar capacidad de cómputo a los usuarios de computación basada en clientes ligeros (*thin-client*). Otra aplicación consiste en proporcionar servicios con alta disponibilidad y escalabilidad (como los motores de búsqueda en Internet) mediante la replicación o la partición del estado del servidor en el grupo de procesadores [Fox y otros 1997]. Los grupos también se utilizan para ejecutar programas paralelos [Anderson y otros 1995, now.cs.berkeley.edu, TFCC].

Las políticas de localización de procesos pueden ser *estáticas* o *adaptativas*. Las primeras operan sin tener en consideración el estado actual del sistema, aunque su diseño tiene en cuenta las características esperadas para el sistema a largo plazo. Se basan en análisis matemáticos orientados a la optimización de ciertos parámetros como la productividad global de procesos. Pueden ser deterministas (el nodo A debe siempre transferir procesos al nodo B) o probabilísticos (el nodo A debe transferir procesos a cualquiera de los nodos B-E de forma aleatoria). Por otra parte las políticas adaptativas aplican reglas heurísticas para tomar las decisiones de ubicación basándose en factores de tiempo de ejecución no predecibles, como la medida de la carga en cada nodo.

Los sistemas de compartición de carga pueden ser centralizados, jerárquicos o descentralizados. En el primer caso existe un *gestor de carga*, mientras que en el segundo hay varios, organizados en una estructura arborescente. Los gestores de carga consiguen información sobre los nodos y la usan para asignar nuevos procesos a esos nodos. En los sistemas jerárquicos, estos gestores toman las decisiones de localización de procesos sobre aquellos nodos con mayor profundidad posible en el árbol; sin embargo los gestores pueden intercambiarse procesos a través de un ancestro común, aunque siempre bajo determinadas condiciones de carga. En un sistema de compartición de carga descentralizado los nodos intercambian información entre ellos directamente para tomar las decisiones de localización. Por ejemplo, el sistema Spawn [Waldspurger y otros 1992] diferencia los nodos entre *compradores* y *vendedores* de recursos de computación y los organiza como una *economía de mercado* (descentralizada).

En los algoritmos de compartición de la carga *iniciados por el emisor*, el nodo que solicita la creación de un nuevo proceso es responsable de iniciar la decisión de transferencia. Normalmente inicia una transferencia cuando su propia carga traspasa un cierto umbral. Por el contrario en los algoritmos *iniciados por el receptor*, un nodo cuya carga está por debajo de un cierto umbral advierte de su existencia a otros nodos para que aquellos que tengan una carga relativamente alta inicien la transferencia de trabajo a dicho nodo.

Las políticas iniciadas por el receptor son más apropiadas cuando hay algún nodo que puede transferir un proceso que se está ejecutando en él. La transferencia de un proceso en ejecución entre dos nodos se llama *migración de procesos*. Milojicic y otros [1999] disponen de un conjunto de artículos sobre migración de procesos y otros tipos de movilidad. A pesar de que se han construido varios mecanismos de migración de procesos, no han tenido una repercusión relevante. Esto se debe, en su mayor parte, a que es una operación cara y a lo difícil que es obtener el estado completo de un proceso en su núcleo para conseguir moverlo a otro nodo.

Eager y otros [1988] analizaron tres aproximaciones diferentes al problema de la compartición de la carga y llegaron a la conclusión de que la simplicidad es una propiedad muy importante en cualquier esquema de compartición de carga. La razón es que las sobrecargas relativamente elevadas (por ejemplo, las sobrecargas para la recolección del estado) pueden eclipsar las ventajas de los esquemas más complejos.

◇ **Creación de un nuevo entorno de ejecución.** Una vez que ha sido seleccionado el computador, el nuevo proceso necesita un entorno de ejecución formado por un espacio de direcciones iniciado con una serie de contenidos (y probablemente otros recursos como archivos abiertos por defecto).

Existen dos aproximaciones en la definición e iniciación del espacio de direcciones de un proceso recientemente creado. La primera de ellas se utiliza cuando el espacio de direcciones tiene un formato definido estáticamente. Por ejemplo, puede contener únicamente la región de texto de un programa, el montón y la pila. En este caso las regiones del espacio de direcciones se crean utilizando una lista donde se especifica su tamaño. Las regiones del espacio de direcciones se inicializan utilizando el archivo ejecutable o bien con ceros, según corresponda.

Alternativamente el espacio de direcciones puede definirse respecto a un entorno de ejecución existente. Por ejemplo, para el caso de la semántica *fork* de UNIX, el proceso hijo que se acaba de crear comparte físicamente la región de texto del padre y tiene sus regiones montón y pila que son una copia de las correspondientes del padre en extensión (al igual que en sus contenidos iniciales). Este esquema se ha generalizado de forma que cada región del proceso padre puede ser heredada por (o bien omitida desde) el proceso hijo. Una región heredada puede, o bien ser compartida, o bien copiarse desde la región del padre. Cuando el padre y el hijo comparten una región, los marcos de página (unidades de memoria física que se corresponden con las páginas de memoria virtual) pertenecientes a la región del padre concuerdan simultáneamente con las correspondientes regiones del hijo.

Por ejemplo, Mach [Accetta y otros 1986] y Chorus [Rozier y otros 1988, 1990] utilizan una optimización llamada *copia en escritura* cuando una región heredada se copia desde el padre. La región es copiada pero por defecto no se realiza una copia física. Los marcos de página que componen la región heredada se comparten entre los dos espacios de direcciones. Una página de la región es copiada físicamente únicamente cuando uno de los dos procesos intente modificarla.

La copia en escritura es una técnica general y se utiliza, por ejemplo, en la copia de mensajes largos, por lo que se explicará su forma de operar a continuación. Sean dos regiones de ejemplo, *RA* y *RB*, cuya memoria es compartida a través de copia en escritura entre dos procesos, *A* y *B* (véase la Figura 6.4). Con el propósito de ser más concretos, supongamos que el proceso *A* configura su región *RA* para ser heredada en la copia por su hijo, el proceso *B*, y que la región *RB* fue, por lo tanto, creada en el proceso *B*.

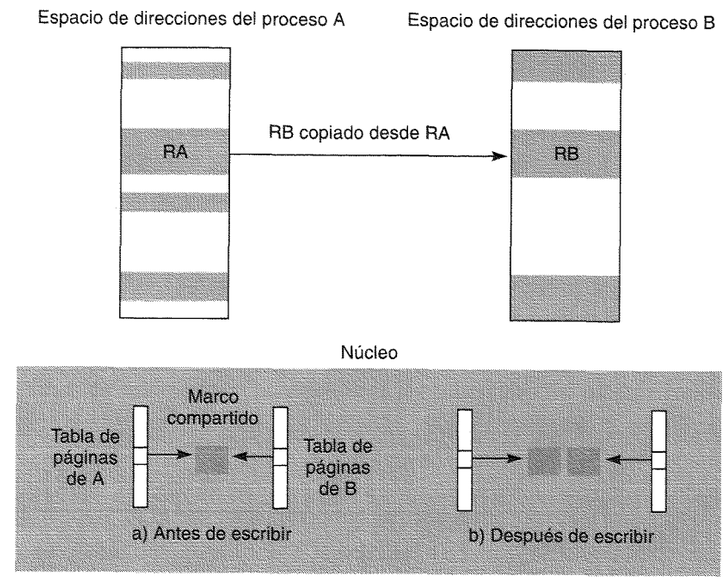


Figura 6.4. Copia en escritura.

Suponemos, con el ánimo de simplificar la explicación, que las páginas que pertenecen a la región *A* residen en memoria. Inicialmente todos los marcos de página asociados con las regiones se comparten mediante las tablas de páginas de los dos procesos. Las páginas están inicialmente protegidas contra escritura a nivel del hardware incluso si pertenecen a regiones que tienen privilegios lógicos de escritura. Si un hilo en cualquier proceso intenta modificar los datos se genera una excepción hardware llamada *fallo* o *falta de página*. Supongamos que el proceso *B* intentó la escritura. El gestor de faltas de páginas localiza un nuevo marco para el proceso *B* y copia los datos del marco original en él byte a byte. El número del marco antiguo es reemplazado por el número del nuevo marco en la tabla de páginas de un proceso (no importa de cual) y el número del marco antiguo se deja en la tabla de páginas del otro proceso. Las dos páginas equivalentes en los procesos *A* y *B* tienen ahora permisos de escritura al nivel de hardware. Tras las operaciones descritas, la instrucción de modificación del proceso *B* puede continuar.

6.4.3. HILOS

El siguiente aspecto importante de un proceso que debe considerarse en mayor detalle son sus hilos. En esta subsección se examinan las ventajas que supone la posibilidad de poseer más de un hilo para los procesos clientes y servidores. Se discute a continuación la programación con hilos, utilizando los hilos de Java a modo de ejemplo y se finaliza con diseños alternativos para implementar hilos.

Sea el servidor mostrado en la Figura 6.5 (en breve se analizará el cliente). El servidor tiene un conjunto formado por uno o más hilos cada uno de los cuales, y de forma repetitiva, toma una solicitud de la cola de solicitudes recibidas y la procesa. En este punto no interesa el instante en el que se reciben las solicitudes y se almacenan para ser procesadas por los hilos. Además se supone,

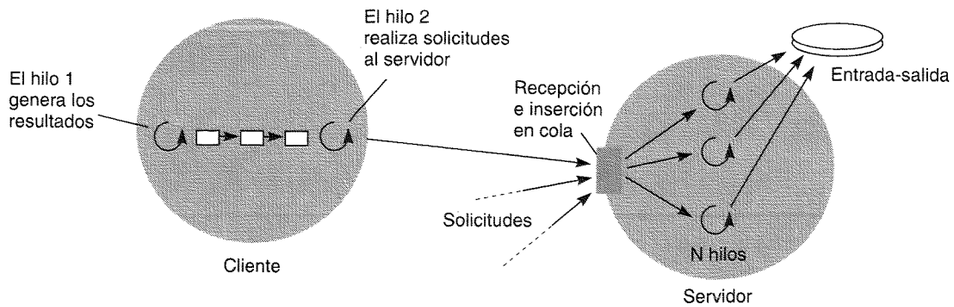


Figura 6.5. Clientes y servidores con hilos.

a efectos de simplificar el ejemplo, que cada hilo aplica el mismo procedimiento para procesar las solicitudes. Supongamos que cada solicitud necesita, por término medio, 2 milisegundos de procesamiento y 8 milisegundos de entrada-salida para que el servidor realice las lecturas desde el disco (no existe caché). Supongamos además temporalmente que el servidor se ejecuta en un computador con un único procesador.

Estudiemos la productividad *máxima* del servidor medida en solicitudes de cliente manejadas por segundo, en función del número de hilos. Si un único hilo debe realizar todo el procesamiento, el tiempo de retorno para manejar cualquier solicitud es de $2 + 8 = 10$ milisegundos, por término medio, de forma que este servidor podrá atender a 100 clientes por segundo. Cualquier mensaje de solicitud que llegue al servidor mientras está manejando una solicitud es insertado en la cola del puerto del servidor.

Ahora considérese qué ocurre cuando el servidor dispone de dos hilos. Se asume que los hilos se pueden planificar de forma independiente, es decir, un hilo puede planificarse cuando otro pasa a estado bloqueado por una operación de entrada-salida. El hilo número dos puede procesar una segunda solicitud mientras el hilo número uno está bloqueado, y viceversa. Esto incrementa la productividad del servidor. Desgraciadamente, en nuestro ejemplo los hilos se bloquean sobre una única unidad de disco. Si todas las solicitudes al disco se envían en serie y necesitan 8 milisegundos cada una, la productividad máxima resulta ser de $1.000/8 = 125$ solicitudes por segundo.

Supóngase ahora que existe una caché de bloques de disco. El servidor mantiene los datos leídos en búferes dentro de su espacio de direcciones; el hilo del servidor encargado de la obtención de los datos examina previamente la caché compartida evitándose el acceso al disco si los datos se encuentran allí. Si se consigue una tasa de aciertos del 75 %, el tiempo medio de entrada-salida por cada solicitud se reduce a $(0{,}75 \times 0 + 0{,}25 \times 8) = 2$ milisegundos, y la productividad máxima teórica se incrementa hasta 500 solicitudes por segundo. Sin embargo si el tiempo medio de *procesador* por cada solicitud se incrementa hasta 2,5 milisegundos por solicitud debido a la gestión de la caché (en cada operación es preciso gastar un cierto tiempo en la búsqueda de los datos en la caché) no se llega a la cantidad anterior. El servidor, limitado por el procesador, podrá ahora gestionar un máximo de $1.000/2{,}5 = 400$ solicitudes por segundo.

La productividad puede incrementarse mediante el uso de multiprocesadores de memoria compartida para resolver los cuellos de botella del procesador. Un proceso multi-hilo se adapta de forma natural a un multiprocesador de memoria compartida. El entorno de ejecución compartida puede implementarse en memoria compartida y los múltiples hilos pueden planificarse para su ejecución en múltiples procesadores. Considérese ahora el caso en el que el servidor de nuestro ejemplo se ejecuta en un multiprocesador con dos procesadores. Suponiendo que los hilos pueden planificarse independientemente sobre los diferentes procesadores entonces podrán procesar solicitudes en paralelo hasta un máximo de dos hilos. El lector deberá comprobar que dos hilos pueden procesar 444 solicitudes por segundo y tres o más hilos, limitados por el tiempo de entrada-salida, pueden procesar 500 solicitudes por segundo.

◇ **Arquitecturas para servidores multi-hilo.** Hemos descrito cómo el multi-hilo permite a los servidores maximizar su productividad, medida como el número de solicitudes procesadas por segundo. Para describir las diferentes formas que existen para vincular solicitudes a hilos dentro de un servidor resumimos el informe escrito por Schmidt [1998], en el que se describen las arquitecturas basadas en hilos de diferentes implementaciones del Agente de Solicitud de Objetos de CORBA (ORB, *Object Request Broker*). Cada ORB procesa solicitudes que llegan a través de un conjunto de sockets activos. Sus arquitecturas basadas en hilos son relevantes para múltiples tipos de servidores, independientemente del hecho que usen CORBA.

La Figura 6.5 muestra una de las posibles arquitecturas basadas en hilos, la *arquitectura de asociación de trabajadores*. En su forma más simple, el servidor, durante la inicialización, crea un conjunto fijo de hilos de *trabajadores* para procesar las solicitudes. El módulo marcado como *receptor y gestor de cola* en la Figura 6.5 se implementa normalmente como un hilo de *E/S*, el cual recibe solicitudes desde una colección de sockets o puertos y las sitúa en una cola de solicitudes compartidas para ser recuperadas por los trabajadores.

En ocasiones existe un requisito para tratar las solicitudes con prioridades cambiantes. Por ejemplo, un servidor web corporativo debiera procesar las solicitudes con diferentes prioridades en función del tipo de cliente del que se derive la solicitud [Bhatti y Friedrich 1999]. Se pueden gestionar prioridades de solicitud variables mediante múltiples colas en la arquitectura de asociación de trabajadores, de forma que los hilos del trabajador puedan examinar las colas en orden decreciente de prioridad. Una desventaja de esta arquitectura es su escasa flexibilidad: como se vio en el ejemplo previo, el número de hilos de trabajo en la asociación puede ser demasiado pequeño para manejar adecuadamente la tasa actual de llegada de solicitudes. Otra desventaja es el alto nivel de conmutación entre la E/S y los hilos de trabajo al manejar la cola compartida.

En la *arquitectura de hilo-por-solicitud* (véase la Figura 6.6a) el hilo de E/S genera un nuevo hilo de trabajo para cada solicitud, y ese trabajador se elimina a sí mismo cuando ha procesado la solicitud asociada a un objeto remoto. Esta arquitectura tiene la ventaja de que los hilos no compiten por el acceso a una cola compartida y la productividad se puede maximizar potencialmente dado que el hilo de E/S puede crear tantos trabajadores como solicitudes pendientes existan. Su desventaja es la sobrecarga debida a las operaciones de creación y destrucción de hilos.

La *arquitectura de hilo-por-conexión* (véase la Figura 6.6b) asocia un hilo a cada conexión. El servidor crea un nuevo hilo trabajador cuando un cliente realiza una conexión y lo destruye cuando el cliente cierra dicha conexión. En el intervalo, el cliente puede realizar múltiples solicitudes sobre la conexión, cuyo destino puede ser uno o más objetos remotos. La *arquitectura de hilo-por-objeto* (véase la Figura 6.6c) asocia un hilo a cada objeto remoto. Un hilo de E/S recibe las solici-

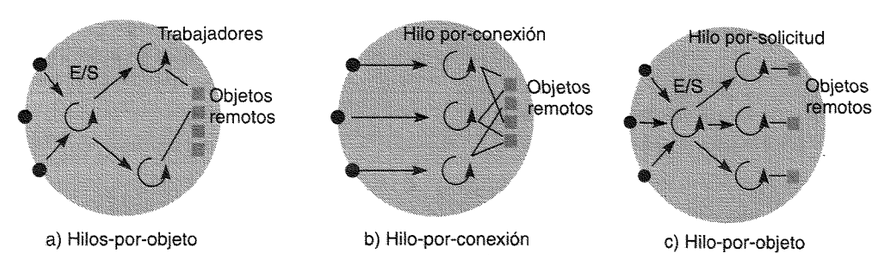


Figura 6.6. Arquitecturas alternativas de servidor basadas en hilos (véase también la Figura 6.5).

tudes y las inserta en colas para los trabajadores con la diferencia de que en este caso existe una cola por cada objeto.

En cada una de las dos últimas arquitecturas el servidor se beneficia de sobrecargas pequeñas en la gestión de los hilos en comparación con la arquitectura de hilo-por-solicitud. Su desventaja es que los clientes pueden sufrir retrasos si un hilo trabajador tiene varias solicitudes pendientes mientras que otro hilo puede estar parado sin trabajo que realizar.

Schmidt [1998] describe variantes de estas arquitecturas, así como algunas mezclas, y discute sus ventajas e inconvenientes en mayor detalle. En la Sección 6.5 se describe un modelo diferente basado en hilos para solicitudes en una única máquina, en el que los hilos cliente comparten el espacio de direcciones del servidor.

◇ **Hilos dentro de los clientes.** Los hilos pueden ser tan útiles para los clientes como para los servidores. En la Figura 6.5 se muestra un proceso cliente con dos hilos. El primer hilo genera resultados que se van a enviar al servidor mediante una invocación a método remoto, no necesitando, sin embargo, una respuesta. Las invocaciones a métodos remotos normalmente bloquean al que invoca, incluso aunque no sea estrictamente preciso esperar. Este proceso cliente puede añadir un segundo hilo, el cual realiza las invocaciones a los métodos remotos y se bloquea mientras el primer hilo puede continuar computando nuevos resultados. El primer hilo deposita sus resultados en búferes que son vaciados por el segundo hilo. Únicamente se bloquea cuando todos los búferes estén llenos.

La cuestión de clientes multihilo es evidente también en el ejemplo de los navegadores web. El usuario sufre retardos significativos mientras se cargan las páginas; es por lo tanto esencial para los navegadores la gestión de múltiples solicitudes concurrentes a páginas web.

◇ **Hilos frente a múltiples procesos.** De los ejemplos anteriores se puede deducir la utilidad de los hilos, que permiten que la computación se solape con la entrada-salida y, para el caso de un multiprocesador, con otros cómputos. El lector puede haber notado, sin embargo, que se puede obtener el mismo solapamiento mediante la utilización de múltiples procesos mono-hilo. Entonces, ¿por qué razón se prefiere el modelo de proceso multi-hilo? La respuesta es doble: la creación y gestión de los hilos es más barata que la de procesos y la compartición de recursos se puede conseguir de forma más eficiente entre hilos que entre procesos ya que los hilos comparten los entornos de ejecución.

En la Figura 6.7 se muestran algunos de los principales componentes del estado que deben gestionarse para entornos de ejecución e hilos, respectivamente. Un entorno de ejecución tiene un espacio de direcciones, interfaces de comunicación, como sockets, recursos de alto nivel del tipo de archivos abiertos y objetos para la sincronización de hilos como los semáforos; además debe mantener una relación de los hilos que se asocian a dicho entorno. Un hilo tiene una prioridad de planificación, un estado de ejecución (como *BLOQUEADO* o *PREPARADO*), los valores almacenados correspondientes a los registros del procesador cuando el hilo está *BLOQUEADO* y el estado relacionado con la gestión de *interrupciones software* del hilo. Una *interrupción software* es un evento que provoca la interrupción de un hilo (de forma análoga al caso de interrupción hardware). Si el hilo tiene asignado un procedimiento de gestión se le transfiere el control. Las señales en UNIX son ejemplos de interrupciones software.

Entorno de ejecución	*Hilo*
Tablas de espacio de direcciones	Registros del procesador salvados
Interfaces de comunicación, archivos abiertos	Prioridad y estado de ejecución (del tipo de *BLOQUEADO*)
Semáforos, otros objetos de sincronización	Información de gestión de interrupciones software
Lista de identificadores de hilos	Identificador de entorno de ejecución
Páginas del espacio de direcciones residentes en memoria; entradas de la caché hardware	

Figura 6.7. Estado asociado con los entornos de ejecución y los hilos.

La figura muestra que un entorno de ejecución y los hilos que pertenecen a él se asocian con páginas que pertenecen al espacio de direcciones mantenido en memoria principal, mientras que los datos e instrucciones se almacenan en las cachés hardware.

Podemos resumir la comparación realizada entre procesos e hilos de la siguiente forma:

- La creación de un nuevo hilo dentro de un proceso existente es más barata que la creación de un proceso.
- Más importante aún, la conmutación a un hilo diferente dentro del mismo proceso es más barata que la conmutación entre hilos que pertenecen a diferentes procesos.
- Los hilos dentro de un proceso pueden compartir datos y otros recursos de forma más adecuada y eficiente que con procesos separados.
- Sin embargo, del mismo modo, los hilos dentro de un proceso no están protegidos entre ellos.

Considérese el coste de creación de un nuevo hilo dentro de un entorno de ejecución existente. La principal tarea consiste en asignar una región para la pila y proporcionar los valores iniciales de los registros del procesador y de la prioridad y estado de ejecución del hilo (inicialmente puede estar *SUSPENDIDO* o *PREPARADO*). Debido a la existencia del entorno de ejecución, únicamente se debe colocar un identificador para él en el registro que describe al hilo (el cual contiene los datos necesarios para gestionar la ejecución del hilo). La sobrecarga asociada con la creación de un proceso es, por regla general, considerablemente mayor que la asociada a la creación de un hilo nuevo. Es preciso crear en primer lugar un nuevo entorno de ejecución, incluyendo las tablas del espacio de direcciones. Anderson y otros [1991] citan valores de alrededor de 11 milisegundos para la creación de un proceso UNIX nuevo y sobre 1 milisegundo para la creación de un hilo sobre la misma arquitectura de procesador CVAX ejecutando un núcleo Topaz; en cada caso el tiempo medido supone que la entidad creada realiza exclusivamente una invocación a un procedimiento vacío antes de terminar. Los valores anteriores se muestran exclusivamente a efectos de orientación.

Cuando la nueva entidad realiza alguna tarea significativa en lugar de la invocación a un procedimiento nulo, se producen costes a largo plazo que tienden a ser mayores para un nuevo proceso que para un nuevo hilo dentro de un proceso existente. En un núcleo que soporte memoria virtual, el nuevo proceso generará faltas de página según vaya accediendo por primera vez a datos e instrucciones; las cachés hardware no contendrán inicialmente valores de los datos del nuevo proceso y deberán adquirir las diferentes entradas de la caché según se vaya ejecutando. Por otro lado para el caso de la ejecución de un hilo las sobrecargas a largo plazo también pueden ocurrir, pero tienden a ser menores. Cuando el hilo accede al código y a los datos que han sido utilizados recientemente por otros hilos del mismo proceso, automáticamente se aprovecha del hecho de que dichos datos ya residen en cada caché hardware o bien en la memoria principal.

La segunda ventaja de prestaciones de los hilos se refiere a la *conmutación* entre los hilos, es decir, la sustitución de un hilo por otro en un cierto procesador. Éste es el coste más relevante porque puede generarse en muchas ocasiones durante la vida del hilo. La conmutación entre hilos que comparten el mismo entorno de ejecución es considerablemente más barata que la de hilos que pertenecen a diferentes procesos. Las sobrecargas asociadas con la conmutación de hilos corresponden a la planificación (la elección del siguiente hilo a ejecutar) y al cambio de contexto.

Un contexto de procesador está formado por los valores de los registros del procesador como el contador de programa y el dominio de protección actual del hardware: el espacio de direcciones y el modo de protección del procesador (supervisor o usuario). El *cambio de contexto* es la transición

entre contextos que ocurre cuando se conmuta entre hilos o cuando un único hilo realiza una llamada al sistema o genera otro tipo de excepción. Supone realizar las siguientes tareas:

- Almacenamiento del registro de estado original del procesador y carga del nuevo estado.
- En algunas situaciones se realiza una transferencia a un nuevo dominio de protección; a esto se le llama *transición de dominio*.

La conmutación entre hilos que comparten un mismo entorno de ejecución de nivel de usuario no supone la realización de transición de dominio y es relativamente barata. La conmutación al núcleo o a otro hilo que pertenece al mismo entorno de ejecución a través del núcleo, supone la realización de una transición de dominio. El coste es así mayor aunque, si el núcleo está vinculado al espacio de direcciones del cliente, es aún relativamente bajo. Sin embargo, cuando se conmuta entre hilos que pertenecen a diferentes entornos de ejecución se generan mayores sobrecargas. El recuadro siguiente explica las implicaciones de coste que supone la caché hardware en las transiciones de dominio. Los costes a largo plazo debidos a la ocupación de las entradas de la caché hardware y de las páginas de memoria principal tienden a aplicarse cuando ocurre la mencionada transición de dominio. Los valores presentados por Anderson y otros [1991] son de 1,8 milisegundos para realizar una conmutación entre procesos UNIX y de 0,4 milisegundos para la conmutación entre hilos pertenecientes al mismo entorno de ejecución realizada en el núcleo Topaz. Se han conseguido costes incluso menores (0,04 milisegundos) para hilos que conmutan a nivel de usuario. Damos estos valores únicamente a efectos de orientación, y no incluyen los costes a largo plazo debidos a la caché.

Como se vio en el anterior ejemplo del proceso cliente con dos hilos, el primer hilo genera los datos y se los pasa al segundo hilo el cual genera una invocación a un método remoto o bien una llamada a procedimiento remoto. Debido a que los hilos comparten el espacio de direcciones, no hay necesidad de utilizar sistemas de paso de mensajes para enviar los datos. Ambos hilos pueden acceder a los datos a través de variables compartidas. Precisamente en este punto residen tanto las ventajas como el principal peligro de utilizar procesos multihilo. La ventaja estriba en la comodidad y eficiencia de la compartición de datos. Esto es particularmente cierto para los servidores, como se mostró en el ejemplo basado en caché de datos de archivos. Sin embargo, los hilos que comparten un espacio de direcciones y que no han sido codificados en un lenguaje de tipos seguro no están protegidos entre ellos. Un hilo erróneo puede modificar de forma arbitraria los datos de otro hilo, provocando que éste falle. Si se requiere protección, o bien se utiliza un lenguaje de tipos seguro o bien es preferible utilizar múltiples procesos en lugar de múltiples hilos.

◇ **Programación de hilos.** La programación de hilos equivale a la programación concurrente, tal y como ha sido estudiada tradicionalmente, por ejemplo, en el ámbito de los sistemas operativos. En esta sección se abarcan los siguientes conceptos de programación concurrente, los cuales han sido explicados de forma completa por Bacon [1998]: *condiciones de carrera*, *secciones críticas* (Bacon las llama *regiones críticas*), *monitores*, *variables de condición*, *semáforos*.

La mayor parte de la programación de hilos se realiza en lenguajes de programación convencionales como C aumentados con bibliotecas de hilos. Un ejemplo es el paquete de Hilos para C (*C Threads*) desarrollado para el sistema operativo Mach [Cooper 1998]. Más recientemente, el estándar de hilos IEEE POSIX 1003.4a, conocido como *pthreads*, está siendo ampliamente aceptado. Boykin y otros [1993] describen tanto los Hilos para C como pthreads en el contexto de Mach.

Algunos lenguajes proporcionan un soporte directo para hilos, como Ada95 [Burns y Wellings 1998], Modula-3 [Harbison 1992] y, de forma más reciente, Java [Oaks y Wong 1999]. A continuación explicaremos someramente los hilos de Java.

De la misma forma que en cualquier implementación de hilos, Java proporciona métodos para la creación, destrucción y sincronización de hilos. La clase de Java *Thread* incluye el constructor y los métodos de gestión mostrados en la Figura 6.8. Los métodos de sincronización *Thread* y *Object* están en la Figura 6.9.

Thread(ThreadGroup grupo, Runnable destino, String nombre)
Crea un nuevo hilo en estado *SUSPENDIDO*, perteneciente a *grupo* e identificado con *nombre*; el hilo ejecutará el método *run()* de *destino*.

setPriority(int nuevaPrioridad), getPriority()
Cambia y devuelve la prioridad del hilo.

run()
Un hilo ejectua el método *run()* de su objeto destino, en el caso de que lo tenga, o bien su propio método *run()* en caso de que no lo tenga (los hilos de C implementan la función *Runnable*).

start()
Cambia el estado de un hilo desde *SUSPENDIDO* a *PREPARADO*.

sleep(int milisegundos)
Provoca que el hilo pase al estado *SUSPENDIDO* durante el tiempo que se especifica.

yield()
Pasa a estado READY e invoca a continuación al planificador.

destroy()
Elimina el hilo.

Figura 6.8. Métodos constructor y de gestión de hilos Java.

◇ **El problema de la suplantación.** Las unidades de gestión de memoria normalmente incluyen una caché hardware para mejorar la velocidad de traducción entre direcciones virtuales y físicas, llamado *búfer de apoyo de traducción* (*translation lookaside buffer*, TLB). Tanto un TLB como una caché de datos e instrucciones accedidas mediante direcciones virtuales sufren del *problema de suplantación* (*aliasing*). La misma dirección virtual puede ser válida sobre dos espacios de direcciones diferentes, refiriéndose, sin embargo, a datos físicos diferentes en dos espacios. A no ser que sus entradas estén etiquetadas con un identificador de contexto, tanto un TLB como una caché direccionada virtualmente no tienen en cuenta lo anterior y por lo tanto pueden contener datos incorrectos. Es por ello por lo que deben vaciarse tanto un TLB como los contenidos de una caché cuando se conmuta a un espacio de direcciones diferente. Una caché direccionada físicamente no tiene el problema de la suplantación; sin embargo la utilización de direcciones virtuales para acceder a una caché es algo común, en gran medida porque permite que las búsquedas se solapen con la traducción de direcciones.

◇ **Tiempos de vida de los hilos.** Cada nuevo hilo se crea en la misma máquina virtual Java (JVM, *Java virtual machine*) que su creador y en estado *SUSPENDIDO*. Una vez que ha pasado a estado *PREPARADO* con el método *start*(), ejecuta el método *run*() de un objeto designado en su constructor. JVM y los hilos que hay sobre él se ejecutan en un proceso sobre el sistema operativo

thread.join(int milisegundos)
Bloquea el hilo invocador durante el tiempo especificado hasta que el *hilo* haya terminado.

thread.interrupt()
Interrumpe el *hilo*: le obliga a volver desde una llamada a método bloqueante como *sleep()*.

object.wait(long milisegundos, int nanosegundos)
Bloquea el hilo invocador hasta que una llamada realizada a *notify()* o *notifyAll()* en el objeto despierte el hilo, o bien el hilo sea interrumpido, o bien el tiempo especificado se haya cumplido.

object.notify(), object.notifyAll()
Despierta, respectivamente, uno o todos los hilos que han invocado a *wait()* en el objeto.

Figura 6.9. Llamadas de sincronización de hilos en Java.

subyacente. Los hilos pueden tener asignada una prioridad de forma que las implementaciones de Java que soportan prioridades ejecutarán cada hilo de forma preferente al resto de hilos con menor prioridad. Un hilo finaliza su vida cuando vuelve del método *run*(), o bien cuando se invoca el método *destroy*().

Los programas pueden gestionar los hilos en grupos. Cada hilo pertenece a un grupo, al que se le asigna en el momento de su creación. Los grupos de hilos son útiles cuando varias aplicaciones coexisten en la misma JVM. Un ejemplo de su uso es la seguridad: por defecto un hilo en un cierto grupo no puede gestionar operaciones sobre un hilo en otro grupo. De esta forma un hilo de una cierta aplicación no puede, de forma maliciosa, interrumpir un hilo del entorno de ventanas del sistema (AWT).

Los grupos de hilos también facilitan el control de las prioridades relativas de los hilos (en las implementaciones de Java que soporten prioridades). Esto es útil para visualizadores que ejecuten applets y para los servidores web que ejecuten los programas llamados *servlets* [Hunter y Crawford 1998], los cuales crean páginas web de forma dinámica. Un hilo no privilegiado dentro de un applet o servlet únicamente puede crear un nuevo hilo que pertenezca a su propio grupo, o a un grupo de menor prioridad creado dentro del suyo; las restricciones exactas dependen del administrador de seguridad (*SecurityManager*) que esté activo. Los visualizadores y los servidores pueden asignar hilos pertenecientes a diferentes applets o servlets a diferentes grupos y establecer la prioridad máxima de cada grupo en su conjunto (incluyendo los grupos descendientes). Para un hilo de applet o de servlet no existe forma de ignorar las prioridades de grupo impuestas por el gestor de hilos, ya que no pueden ser modificadas mediante llamadas a *setPriority*().

◇ **Sincronización de hilos.** La programación de un proceso multi-hilo debe realizarse de forma cuidadosa. La principal dificultad estriba en la compartición de objetos y en las técnicas utilizadas para la coordinación y cooperación de los hilos. Las variables locales de cada hilo en sus métodos son privadas del hilo, es decir, los hilos tienen pilas privadas. Sin embargo, los hilos no disponen de copias privadas de las variables (clases) estáticas o de las variables instancia de objetos.

Suponga, por ejemplo, las colas compartidas descritas previamente en esta sección, las cuales tienen hilos de E/S e hilos de trabajo utilizados para transferir las solicitudes en arquitecturas de servidores basadas en hilos. Pueden aparecer condiciones de competencia (*race conditions*) cuando los hilos manipulan concurrentemente estructuras de datos del tipo de colas. Las solicitudes que residen en las colas pueden perderse o duplicarse, a no ser que los punteros de manipulación de los hilos hayan sido cuidadosamente coordinados.

Java proporciona a los programadores la palabra clave *synchronized* para designar un monitor, que es un recurso de sincronización de hilos muy conocido. Los programadores designan métodos completos o bien bloques de código de tamaño arbitrario como pertenecientes a un monitor asociado a un objeto individual. La garantía de un monitor viene del hecho de que como máximo un hilo puede ejecutarse dentro del monitor en cualquier instante. Podríamos seriar las acciones de los hilos de E/S y trabajadores en nuestro ejemplo designando los métodos *añadirA*() y *eliminarDe*() en la clase *Cola* como métodos *synchronized*. Todos los accesos a variables dentro de dichos métodos se realizan con exclusión mutua respecto a las invocaciones de esos métodos.

Java permite bloquear y despertar hilos a través de objetos arbitrarios que actúan como variables de condición. Un hilo que necesita bloquearse en espera de una cierta condición invoca un método *wait*(). Todos los objetos implementan este método, ya que pertenece a la clase raíz de Java llamada Object. Otro hilo invocará *notify*() para desbloquear como máximo un hilo, o bien invocará a *notifyAll*() para desbloquear todos los hilos en espera sobre el mismo objeto. Ambos métodos de notificación pertenecen también a la clase *Object*.

Como ejemplo, cuando un hilo trabajador descubre que ya no hay más solicitudes para procesar, invoca a *wait*() en la instancia de *Cola*. Cuando posteriormente el hilo de E/S añada una solicitud a la cola, invocará al método *notify*() de gestión de colas, para despertar al trabajador.

Los métodos de sincronización de Java se muestran en la Figura 6.9. Además de las primitivas de sincronización mencionadas, el método *join*() bloquea al que invoca hasta la terminación del hilo destino. El método *interrupt*() es útil para despertar prematuramente un hilo en espera. Todas las primitivas estándar de sincronización, como los semáforos, pueden implementarse en Java. Sin embargo es preciso hacerlo cuidadosamente ya que la garantía del monitor de Java se aplica únicamente a código de objetos declarado como *synchronized*; una clase tendrá una mezcla de métodos *synchronized* y no-*synchronized*. Además el monitor implementado en un objeto Java tiene una única variable de condición implícita mientras que en general un monitor puede tener varias variables de condición.

◇ **Planificación de hilos.** Una distinción importante entre modelos de planificación de hilos es si es apropiativa o no apropiativa. En la *planificación apropiativa* un hilo puede suspenderse en cualquier punto para dejar paso a otro hilo, incluso aunque el hilo pudiera seguir en ejecución. En la *planificación no apropiativa* (a veces llamada *planificación de corrutinas*), un hilo se ejecuta hasta que él mismo realiza una invocación al sistema de gestión de hilos (por ejemplo, una llamada al sistema), siendo en ese momento cuando el sistema puede desalojarle para planificar otro hilo.

La ventaja de la planificación no apropiativa es que cualquier sección de código que no contenga una llamada al sistema de gestión de hilos es automáticamente una sección crítica. Así se pueden evitar cómodamente las condiciones de competencia. Por otro lado, los hilos planificados de forma no apropiativa no pueden aprovechar los sistemas multiprocesador, ya que se ejecutan de forma exclusiva. Es preciso tener cuidado con las secciones de código grandes que no contengan llamadas al sistema de gestión de hilos. El programador puede necesitar insertar invocaciones a *yield*() cuya función es la de permitir que otros hilos puedan planificarse y progresar en su trabajo. Los hilos planificados de forma no apropiativa no son aptos para su uso en aplicaciones en tiempo real, ya que en éstas los eventos llevan asociados tiempos absolutos en los que deben procesarse.

Java no soporta, por defecto, el procesamiento en tiempo real, a pesar de que existen implementaciones de tiempo real [www.rtj.org]. Por ejemplo, las aplicaciones multimedia que procesan datos del tipo de voz y vídeo tienen requisitos en tiempo real tanto para la comunicación como para el procesamiento (un ejemplo es el filtrado y la compresión) [Govindan y Anderson 1991]. El Capítulo 15 examinará los requisitos de planificación de hilos en tiempo real. El control de procesos es otro ejemplo del dominio del tiempo real. En general cada dominio de tiempo real tiene sus propios requisitos de planificación de hilos. Es por lo tanto deseable que a veces las aplicaciones implementen sus propias políticas de planificación. Para examinar esto estudiaremos ahora la implementación de los hilos.

◇ **Implementación de los hilos.** Muchos núcleos dan soporte para procesos multi-hilo de forma nativa, incluyendo Windows NT, Solaris, Mach y Chorus. Estos núcleos proporcionan las llamadas al sistema de creación y gestión de hilos, y planifican de forma individual los hilos. Otros núcleos disponen únicamente de la abstracción de procesos mono-hilo. Los procesos multi-hilo deben entonces implementarse en una biblioteca de procedimientos enlazada a los programas de aplicación. En estos casos el núcleo no conoce estos hilos de nivel de usuario y por lo tanto no puede planificarlos independientemente. Una biblioteca en tiempo de ejecución de hilos organiza su planificación. Un hilo podría bloquear el proceso, y por lo tanto todos los hilos dentro de él, si realiza una llamada bloqueante al sistema, de forma que debe usarse la entrada-salida asíncrona (no bloqueante) del núcleo subyacente. Análogamente la implementación puede utilizar los temporizadores proporcionados por el núcleo y las posibilidades de interrupciones software para realizar una compartición del tiempo entre hilos.

Cuando el núcleo no tiene soporte para procesos multi-hilo, las implementaciones al nivel de usuario de los hilos adolecen de estos problemas:

- Los hilos de un cierto proceso no pueden aprovecharse de la existencia de un multiprocesador.

- Un hilo que genera una falta de página bloquea el proceso completo y todos los hilos dentro de él.
- Los hilos de diferentes procesos no pueden planificarse de acuerdo a un único criterio de prioridad relativa.

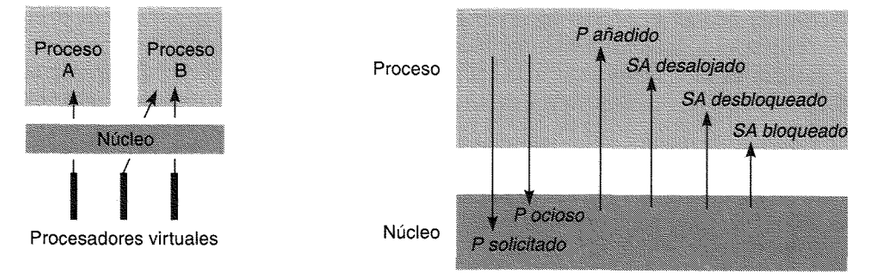
Sin embargo las implementaciones de hilos del nivel de usuario tienen ventajas significativas sobre las implementaciones al nivel de núcleo:

- Algunas operaciones sobre hilos son bastante más baratas. Por ejemplo la conmutación entre hilos pertenecientes al mismo proceso no supone necesariamente una llamada al sistema, es decir, una interrupción interna que es relativamente cara.
- Debido a que el módulo de planificación de hilos se implementa fuera del núcleo, puede personalizarse de forma que se adapte a los requisitos particulares de una cierta aplicación. Las variaciones en los requisitos de planificación se dan en gran medida por las consideraciones específicas de la aplicación, por ejemplo la naturaleza en tiempo real del procesamiento multimedia.
- Pueden soportarse muchos más hilos de nivel de usuario de los que el núcleo puede proporcionar por defecto.

Es posible combinar las ventajas de las implementaciones de hilos de nivel de usuario y de nivel de núcleo. Una posible aproximación, aplicada en el núcleo de Mach [Black 1990], es la de permitir al código de nivel de usuario proporcionar indicaciones de planificación al planificador de hilos del núcleo. Otra posibilidad, utilizada en el sistema operativo Solaris 2, es un tipo de planificación jerárquica. Cada proceso crea uno o más hilos de nivel de núcleo, conocidos en Solaris como *procesos de peso ligero*. También soporta hilos de nivel de usuario. Un planificador de nivel de usuario asigna cada hilo de nivel de usuario a un hilo de nivel de núcleo. Este esquema puede explotar los multiprocesadores y se beneficia del hecho de que algunas operaciones de creación y conmutación de hilos se realizan al nivel de usuario. La desventaja de este esquema es su falta de flexibilidad: si un hilo se bloquea en el núcleo entonces a todos los hilos de nivel de usuario asignados a él se les impide la ejecución, independientemente de si tienen o no la posibilidad de ejecutarse.

Varios proyectos de investigación han desarrollado esquemas de planificación jerárquica más avanzados para obtener mayor eficiencia y flexibilidad. Entre ellos está el trabajo llamado de activación del planificador [Anderson y otros 1991] en el trabajo multimedia de Govindan y Anderson [1991], el sistema operativo multiprocesador Psyche [Marsh y otros 1991], el núcleo Nemesis [Leslie y otros 1996] y el núcleo SPIN [Bershad y otros 1995]. La idea conductora de estos diseños es que un planificador de nivel de usuario no sólo necesita del núcleo simplemente un conjunto de hilos soportados por el núcleo sobre los que se puedan vincular los hilos de nivel de usuario. El planificador de nivel de usuario también necesita que el núcleo le notifique los *eventos* relevantes en sus decisiones de planificación. Describimos el diseño de activaciones del planificador para aclarar este concepto.

El paquete FastThreads de Anderson y otros [1991] es una implementación de un sistema de planificación jerárquico basado en eventos. Considera que los principales componentes del sistema son un núcleo ejecutándose en un computador con uno o más procesadores y un conjunto de programas de aplicación ejecutándose sobre él. Cada proceso de aplicación contiene un planificador de nivel de usuario que se encarga de gestionar los hilos dentro del proceso. El núcleo es responsable de la asignación de *procesadores virtuales* a procesos. El número de procesadores virtuales asignados a un proceso depende de diferentes factores como los requisitos de la aplicación, sus prioridades relativas y la demanda total en los procesadores. En la Figura 6.10a se muestra un ejemplo de una máquina con tres procesadores en la que el núcleo asigna un procesador virtual al proceso *A*, ejecutando un trabajo de prioridad relativamente baja, y dos procesadores virtuales al proceso *B*. Se llaman procesadores *virtuales* porque el núcleo puede asignar procesadores físicos



A. Asignación de procesadores virtuales a procesos

B. Eventos entre el planificador de nivel de usuario y el núcleo. Leyenda: P = procesador; SA = activación del planificador

Figura 6.10. Activaciones del Planificador.

diferentes a cada proceso a lo largo del tiempo, mientras se garantice el número de procesadores asignados.

El número de procesadores virtuales asignados a un proceso puede variar. Los procesos pueden devolver un procesador virtual que han dejado de necesitar; además pueden solicitar procesadores virtuales extra. Por ejemplo, si el proceso *A* ha solicitado un procesador virtual extra y *B* termina, entonces el núcleo asigna uno a *A*.

En la Figura 6.10b se muestra que un proceso notifica al núcleo la ocurrencia de uno de estos dos tipos de eventos: cuando un procesador virtual está ocioso y no será necesitado en el futuro o cuando se necesita un procesador virtual extra.

Además la Figura 6.10b muestra que el núcleo notifica al proceso cuándo ocurre un evento de entre cuatro posibles. Una *activación del planificador* (SA, *scheduler activation*) es una llamada desde el núcleo a un proceso que sirve para notificar al planificador del proceso la ocurrencia de un evento. La ejecución de esta manera de código desde un nivel inferior (el núcleo) es también conocida como *retrollamada* (*upcall*). El núcleo crea un SA cargando los registros físicos del procesador con un contexto que provoca la ejecución de código en el proceso, en una dirección de procedimiento designada por el planificador de nivel de usuario. Una SA puede de esta forma considerarse como una unidad de asignación de tiempo en un procesador virtual. El planificador de nivel de usuario tiene la tarea de asignar sus hilos *PREPARADOS* al conjunto de SA que en ese momento se están ejecutando en él. El número de SA es como máximo el número de procesadores virtuales que el núcleo ha asignado al proceso.

Los cuatro tipos de eventos que el núcleo notifica al planificador de nivel de usuario (al que nos referiremos en el futuro simplemente como *el planificador*) son los siguientes:

Procesador virtual asignado: el núcleo ha asignado un nuevo procesador virtual al proceso y éste es el primer intervalo temporal sobre él; el planificador puede cargar el SA con el contexto de un hilo *PREPARADO* el cual puede recomenzar su ejecución.

SA bloqueado: un SA se ha bloqueado en el núcleo y éste está utilizando un nuevo SA para notificarlo al planificador; el planificador pone el estado del hilo correspondiente a *BLOQUEADO* y puede asignar un hilo *PREPARADO* al SA de notificación.

SA desbloqueado: un SA que ha estado bloqueado en el núcleo se ha desbloqueado y está preparado para ejecutarse de nuevo al nivel de usuario; el planificador puede ahora devolver el hilo a la lista de *PREPARADOS*. Para la creación del SA de notificación, el núcleo puede asignar un procesador virtual nuevo al proceso o bien apropiarse de otro SA en el mismo proceso. En el segundo caso también debe comunicar el evento de apropiación al planificador, el cual recalculará la asignación de hilos a SA.

SA apropiado: el núcleo ha arrebatado el SA especificado al proceso (a pesar de que deberá realizar esta tarea asignando un procesador al nuevo SA en el mismo proceso); el planificador asigna el hilo desalojado a la lista de *PREPARADOS* y recalcula la asignación de hilos a los SA.

El esquema de planificación jerárquica es flexible ya que el planificador de procesos de nivel de usuario puede asignar hilos a SA mediante cualquier política que pueda construirse sobre el conjunto de eventos de bajo nivel. El núcleo se comporta siempre de la misma forma. No tiene ninguna influencia en el comportamiento del planificador de nivel de usuario, sin embargo ayuda al planificador mediante la notificación de eventos y proporcionando el estado de los registros de los hilos bloqueados y desalojados. El esquema es potencialmente eficiente ya que ningún hilo de nivel de usuario permanecerá en estado *PREPARADO* mientras exista un procesador virtual sobre el que ejecutarle.

6.5. COMUNICACIÓN E INVOCACIÓN

En este apartado nos concentramos en la comunicación como parte de la implementación de lo que hemos llamado una invocación, una construcción del tipo de las invocaciones a métodos remotos, llamadas a procedimientos remotos o notificación de eventos, cuyo propósito es el de efectuar una operación sobre un recurso en un espacio de direcciones diferente.

Trataremos cuestiones de diseño y conceptos de sistemas operativos mediante el planteamiento de las siguientes cuestiones acerca de los sistemas operativos:

- ¿Qué primitivas de comunicación debe proporcionar?
- ¿Qué protocolos debe soportar y cómo de abierta es la implementación de la comunicación?
- ¿Qué pasos han sido dados para hacer que la comunicación sea todo lo eficiente posible?
- ¿Qué soporte se proporciona para las operaciones de latencia elevada y para las desconectadas?

◇ **Primitivas de sincronización.** Algunos núcleos diseñados para sistemas distribuidos han proporcionado primitivas de comunicación similares a los tipos de invocación descritos en el Capítulo 5. Por ejemplo Amoeba [Tanenbaum y otros 1990] proporciona *hazOperación*, *tomarPetición* y *enviaRespuesta* como primitivas. Amoeba, el sistema V y Chorus proporcionan primitivas de comunicación en grupo. La inserción de funcionalidad de comunicación de relativamente alto nivel en el núcleo tiene la ventaja de la eficiencia. Si, por ejemplo, el middleware proporciona RMI sobre sockets (TCP) conectados en UNIX, entonces un cliente debe realizar dos llamadas al sistema de comunicaciones (*write* y *read* sobre un socket) para cada invocación remota. Sobre Amoeba solamente se necesitaría una única llamada a *hazOperación*. El ahorro en la sobrecarga de las llamadas al sistema tiende a ser mayor con la comunicación en grupo.

En la práctica, es el middleware y no el núcleo el que proporciona la mayor parte de los recursos de comunicación de alto nivel que aparecen en los sistemas actuales, incluyendo RPC/RMI, notificación de eventos y comunicación de grupos. El desarrollo de este software tan complejo al nivel de usuario es mucho más sencillo que desarrollarlo para el núcleo. Los programadores normalmente implementan el middleware sobre los sockets proporcionando acceso a los protocolos estándar de Internet, a menudo mediante sockets TCP orientados a conexión pero a veces también con sockets UDP no orientados a conexión. Las principales razones para la utilización de sockets son la portabilidad e interoperabilidad: el middleware debe operar sobre la mayor parte de los sistemas operativos ampliamente utilizados y todos los sistemas operativos comunes del tipo de UNIX y de la familia Windows proporcionan un API de sockets similar proporcionando acceso a los protocolos TCP y UDP.

A pesar del amplio uso de los sockets TCP y UDP proporcionados por los núcleos más comunes, se continúa investigando para conseguir primitivas de comunicación con menores costes sobre núcleos experimentales. Examinaremos las cuestiones de prestaciones en la Sección 6.5.1.

◇ **Protocolos y apertura.** Uno de los principales requisitos de los sistemas operativos es el de proporcionar protocolos estándar que permitan la intercomunicación entre implementaciones middleware sobre diferentes plataformas. Algunos núcleos de investigación en la década de los ochenta incorporaban sus propios protocolos de red ajustados para las interacciones con RPC, siendo ejemplos notables Amoeba RPC [van Renesse y otros 1989], VMTP [Cheriton 1986] y Sprite RPC [Ousterhout y otros 1988]. Sin embargo, estos protocolos no fueron ampliamente usados fuera de sus entornos de investigación nativos. Por el contrario, los diseñadores de los núcleos Mach 3.0 y Chorus (al igual que algunos núcleos de la década de los noventa, como L4 [Härtig y otros 1997]) decidieron dejar la elección de los protocolos de red como una cuestión abierta. Estos núcleos proporcionan un sistema de paso de mensajes únicamente entre procesos locales y dejan el procesamiento del protocolo de red a un servidor que se ejecuta sobre el núcleo.

Dados los requisitos diarios de acceso a Internet, es necesario que los sistemas operativos sean compatibles al nivel de TCP y UDP para todo, excepto para los dispositivos de red más pequeños. Además el sistema operativo debe habilitar al middleware para que pueda sacar partido de los nuevos protocolos de bajo nivel. Por ejemplo, los usuarios quieren beneficiarse de las tecnologías inalámbricas como las transmisiones por infrarrojos y por radiofrecuencia (RF), preferiblemente sin tener que actualizar sus aplicaciones. Esto supone que puedan integrarse los protocolos correspondientes, como IrDA para redes de infrarrojos y BlueTooth o HomeRF para las redes RF.

Los protocolos se organizan normalmente en una *pila* de niveles (véase el Capítulo 3). Muchos sistemas operativos permiten la integración estática de nuevos niveles, mediante la inclusión de un nivel del tipo de IrDa como *manejador* (*driver*) de protocolo instalado permanentemente. Por el contrario la *composición dinámica de protocolos* es una técnica en la que puede componerse dinámicamente una pila de protocolo para ajustarse a los requisitos de una aplicación particular y puede utilizar cualquier nivel físico disponible dada la conectividad actual de la plataforma. Por ejemplo un navegador web ejecutándose en un computador portátil debe ser capaz de sacar partido de un enlace de área global inalámbrico mientras el usuario está en carretera, mientras que usará una conexión Ethernet de mayor velocidad cuando el usuario esté de vuelta en la oficina.

Otro ejemplo de composición dinámica de protocolos es el uso de un protocolo de solicitudes-respuestas a medida sobre un nivel de red inalámbrica para reducir las latencias de ida y vuelta. Las implementaciones estándar TCP tienen prestaciones mediocres sobre redes inalámbricas [Balakrishnan y otros 1996] donde tienen mayores tasas de paquetes perdidos que en las redes cableadas. Inicialmente, un protocolo de solicitud-respuesta del tipo de HTTP podría modificarse para trabajar de forma más eficiente sobre nodos conectados de modo inalámbrico mediante la utilización directa del nivel de transporte inalámbrico, en lugar de utilizar un nivel intermedio TCP.

El soporte para la composición de protocolos apareció en el diseño de los Streams de UNIX [Ritchie 1984]. De forma más reciente, Horus [van Renesse y otros 1995] y x-kernel [Hutchinson y Peterson 1991] tienen la capacidad de composición dinámica de protocolos.

6.5.1. PRESTACIONES DE LA INVOCACIÓN

Las prestaciones de la invocación son un factor crítico en el diseño de sistemas distribuidos. Cuanto más separan los diseñadores de la funcionalidad entre espacios de direcciones, más invocaciones remotas se necesitan. Los clientes y servidores pueden realizar muchos millones de operaciones asociadas a invocaciones a lo largo de su vida, de forma que pequeñas fracciones de milisegundos son relevantes en los costes de invocación. Las tecnologías de red continúan mejorando pero los tiempos de invocación no han decrecido en proporción con el incremento del ancho de banda de

las redes. En esta sección se explicará cómo las sobrecargas del software predominan a menudo sobre las sobrecargas de la red en los tiempos de invocación (al menos para el caso de LAN o intranets). Esto no ocurre en una invocación remota sobre Internet, por ejemplo para solicitar un recurso web. En Internet la latencia de red es muy variable pero por término medio muy elevada, el ancho de banda es a menudo bajo y la carga del servidor muchas veces predomina sobre los costes de procesamiento por cada solicitud.

Las implementaciones de RPC y RMI han sido un tema de estudio debido a la amplia aceptación de estos mecanismos para el procesamiento cliente-servidor de propósito general. Mucha de esta investigación ha tenido como objetivo el estudio de las invocaciones en la red, y particularmente sobre cómo los mecanismos de invocación pueden aprovecharse de las redes de altas prestaciones [Hutchinson y otros 1989, van Renesse y otros 1989, Schroeder y Burrows 1990, Johnson y Zwaenepoel 1993, von Eicken y otros 1995, Gokhale y Schmidt 1996]. Existe también, como veremos, el caso importante de RPC entre procesos residentes en el mismo computador [Bershad y otros 1990, Bershad y otros 1991].

◇ **Costes de invocación.** La llamada y la invocación a un procedimiento convencional y a un método convencional, respectivamente, la realización de una llamada al sistema, el envío de un mensaje, la invocación a un procedimiento remoto y la invocación a un método remoto son ejemplos de mecanismos de invocación. Cada mecanismo provoca que se ejecute código fuera del ámbito del procedimiento u objeto que invoca. Cada invocación supone, en general, la comunicación de argumentos a este código y la devolución de datos al que invoca. Los mecanismos de invocación pueden ser síncronos, por ejemplo para el caso de llamadas a procedimientos convencionales o remotos, o asíncronos.

Los aspectos que afectan en mayor medida a las prestaciones de los mecanismos de invocación, aparte de si son o no síncronos, son si éstos suponen una transición de dominio (es decir, si se cambia de espacio de direcciones), si implican comunicaciones a través de la red y si suponen planificación y conmutación de hilos. En la Figura 6.11 se muestran los casos particulares de una llamada al sistema, una invocación remota entre procesos residentes en el mismo computador y una invocación remota entre procesos en diferentes nodos de un sistema distribuido.

◇ **Invocación utilizando la red.** Un *RPC nulo* (de igual modo, un *RMI nulo*) se define como un RPC sin parámetros que ejecuta un procedimiento nulo y no devuelve valores. Su ejecución supone un intercambio de mensajes que lleva muy pocos datos de sistema y ningún dato de usuario. Actualmente el tiempo necesario para un RPC nulo entre dos procesos de usuario ejecutándose en un PC de 500 MHz sobre una LAN a 100 megabits/segundo es del orden de decenas de milisegundos. A efectos de comparación, una llamada a un procedimiento convencional nulo puede suponer una pequeña fracción de microsegundo. En un RPC nulo se envía alrededor de un total de 100 bytes. Con un ancho de banda de 100 megabits/segundo, el tiempo total de transferencia sobre la red es de alrededor de 0,01 milisegundos para esa cantidad de datos. Es evidente que la mayor parte del *retardo* observado (el tiempo total de llamada a un RPC observado por un cliente) debe apuntarse a las tareas del núcleo del sistema operativo y del código en tiempo de ejecución del RPC de nivel de usuario.

Los costes de las invocaciones nulas (RPC, RMI) tienen mucha importancia ya que miden una sobrecarga fija, la *latencia*. Los costes de invocación se incrementan con los tamaños de los argumentos y resultados, pero en muchos casos la latencia es significativa en comparación con el resto del retardo.

Considérese un RPC que solicita una cierta cantidad de datos desde un servidor. Tiene un argumento entero de entrada que especifica el tamaño de los datos solicitados. Tiene dos argumentos de salida, un entero que indica éxito o fallo (el cliente puede haber proporcionado un tamaño inválido) y, cuando la invocación ha tenido éxito, una serie de bytes desde el servidor.

En la Figura 6.12 se muestra, de forma esquemática, el retardo del cliente en comparación con el tamaño de los datos solicitados. El retardo es, aproximadamente, proporcional al tamaño, hasta

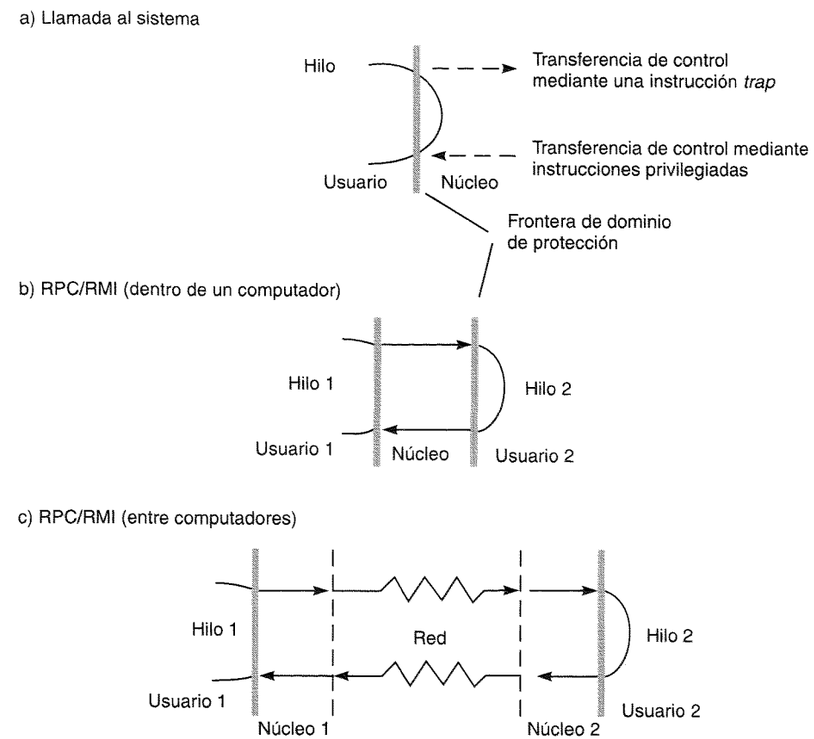


Figura 6.11. Invocaciones entre espacios de direcciones.

que este tamaño alcanza un umbral del tamaño del paquete de red. Por encima de este umbral debe enviarse al menos un paquete adicional para llevar los datos extra. En función del protocolo, puede usarse otro paquete más para reconocer la llegada de este paquete extra. Los saltos en el gráfico ocurren cada vez que se incrementa el número de paquetes.

El retardo no es el único valor de interés en una implementación de RPC: también está implicado el ancho de banda de RPC (o *productividad*) cuando hay que transferir datos en masa. Se trata de la tasa de transferencia de datos entre computadores en un único RPC. Si examinamos la Figura 6.12 se puede observar que el ancho de banda es relativamente bajo para pequeñas cantidades de datos, cuando las sobrecargas fijas de procesamiento predominan. Según se incrementa la cantidad de datos, el ancho de banda también se incrementa ya que las sobrecargas pasan a ser menos significativas. Gokhale y Schmidt [1996] calcularon una productividad de alrededor de 80 megabits/segundo en la transferencia de 64 kilobytes en un único RPC entre estaciones de trabajo sobre una red ATM con un ancho de banda nominal de 155 megabits/segundo. Para transferir 64 kilobytes se necesita alrededor de 0,8 milisegundos lo cual está en el mismo orden de magnitud que el tiempo calculado anteriormente para un RPC nulo sobre una red Ethernet a 100 megabits/segundo.

Recordemos que los pasos en un RPC son los siguientes (RMI realiza pasos similares):

- El resguardo del procedimiento del cliente empaqueta los argumentos de llamada dentro de un mensaje, envía el mensaje de solicitud y recibe y desempaqueta la respuesta.

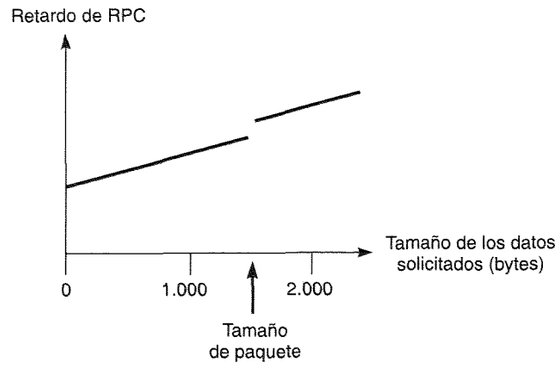


Figura 6.12. Retardo del RPC en función del tamaño del parámetro.

- En el servidor, un hilo de trabajo recibe la solicitud entrante, o bien un hilo de E/S recibe la solicitud y se la envía a un hilo de trabajo; en cualquier caso, el trabajador invoca el resguardo apropiado del procedimiento del servidor.
- El resguardo del servidor desempaqueta el mensaje de solicitud, invoca el procedimiento designado y empaqueta y envía la respuesta.

Los siguientes son los principales componentes significativos en el retardo de invocación remota, aparte del tiempo de transmisión sobre la red:

Empaquetamiento: el empaquetamiento y desempaquetamiento, que llevan asociados la copia y conversión de datos, suponen una sobrecarga significativa al incrementarse la cantidad de datos.

Copia de datos: potencialmente, incluso después del empaquetamiento, los datos del mensaje son copiados varias veces durante un RPC:

1. En la frontera entre el usuario y el núcleo, es decir, entre el espacio de direcciones del cliente o del servidor y los búffers del núcleo.
2. Sobre cada nivel de protocolo (por ejemplo, RPC/UDP/IP/Ethernet).
3. Entre el interfaz de red y los búffers del núcleo.

Las transferencias entre la interfaz de red y la memoria principal son gestionadas normalmente por el acceso directo a memoria (DMA). El procesador gestiona el resto de copias.

Inicialización de paquetes: supone la inicialización de las cabeceras y terminaciones del protocolo, incluyendo los *checksums*. El coste es por lo tanto proporcional, en parte, a la cantidad de datos enviados.

Planificación de hilos y conmutación de contexto: puede ocurrir de la siguiente forma:

1. En un RPC se realizan varias llamadas al sistema (es decir, cambios de contexto) al invocar los resguardos las operaciones de comunicación del núcleo.
2. Se planifican uno o más hilos en el servidor.
3. Si el sistema operativo utiliza un proceso independiente de gestión de red, entonces cada *Envía* supone un cambio de contexto a uno de sus hilos.

Espera por reconocimientos: la elección del protocolo RPC puede influir en los retardos y en particular cuando se envían grandes cantidades de datos.

Un diseño cuidadoso del sistema operativo puede ayudar a reducir algunos de estos costes. El caso de estudio del diseño de Firefly RPC, disponible en www.cdk3.net/oss, muestra alguna de esas posibilidades en detalle, al mismo tiempo que muestra técnicas aplicables en la implementación del middleware. Ya hemos mostrado cómo un soporte correcto de hilos por parte del sistema operativo puede ayudar a reducir las sobrecargas debidas al multi-hilo. El sistema operativo también puede influir significativamente en la reducción de las sobrecargas de copia en memoria mediante recursos de compartición de memoria.

◇ **Compartición de memoria.** Las regiones compartidas (presentadas en la Sección 6.4) pueden utilizarse para una comunicación rápida entre un proceso de usuario y el núcleo, o bien entre procesos de usuario. Los datos se comunican mediante la escritura y la lectura en la región compartida. De esta forma los datos son enviados eficientemente, sin necesidad de copiarlos hacia o desde el espacio de direcciones del núcleo. Sin embargo, se pueden necesitar llamadas al sistema e interrupciones software para la sincronización, por ejemplo cuando el proceso de usuario ha escrito datos que deben ser transmitidos o cuando el núcleo ha escrito datos que deben ser consumidos por el proceso de usuario. Por supuesto una región compartida sólo se justifica cuando se utiliza suficientemente, de forma que se amortiza el coste de inicialización.

Incluso si se utilizan regiones compartidas, el núcleo debe copiar los datos desde cada búfer a las interfaces de red. La arquitectura U-Net [von Eicken y otros 1995] permite al código de nivel de usuario acceder directamente a la interfaz de red propia de forma que el código de nivel de usuario puede transferir datos a la red sin necesidad de copiarlos.

◇ **Elección del protocolo.** El retardo que sufre un cliente durante las interacciones solicitud-respuesta sobre TCP no es necesariamente peor que sobre UDP y en ocasiones es menor, por ejemplo en los mensajes largos. Sin embargo, cuando se implementan interacciones de tipo solicitud-respuesta sobre TCP es preciso hacerlo de forma cuidadosa, ya que no fue diseñado específicamente para este propósito. En particular el comportamiento del almacenamiento en TCP puede impedir la obtención de buenas prestaciones, y sus sobrecargas de conexión son una desventaja en comparación con UDP, a no ser que se envíe un número suficiente de mensajes sobre una única conexión para hacer despreciable la sobrecarga por cada solicitud.

Las sobrecargas de conexión de TCP son particularmente evidentes en las invocaciones web, debido a que HTTP 1.0 realiza una conexión TCP diferente en cada invocación. Los navegadores cliente se retrasan hasta que se realice la conexión. Además en muchos casos el algoritmo de arranque lento de TCP afecta retrasando la transferencia de datos HTTP de forma innecesaria. El algoritmo de arranque lento opera de forma pesimista en prevención de una posible congestión en la red enviando en primer lugar únicamente una pequeña ventana de datos, de forma previa a la recepción de un reconocimiento. Nielson y otros [1997] discuten cómo HTTP 1.1 utiliza las llamadas *conexiones persistentes*, que permanecen a lo largo de varias invocaciones. Por lo tanto los costes iniciales de conexión son amortizados siempre que se realicen varias invocaciones al mismo servidor web. Esto es razonable, ya que los usuarios a menudo solicitan varias páginas del mismo sitio, cada una de ellas conteniendo varias imágenes.

Nielson y otros también encontraron que la desactivación del almacenamiento por defecto proporcionado por el sistema operativo podría tener un impacto significativo en el retardo de invocación. A menudo es beneficioso almacenar varios mensajes pequeños para enviarlos a continuación todos juntos, en lugar de enviarlos en paquetes separados, debido a la latencia por paquete que se describió previamente. Por esta razón el sistema operativo no envía necesariamente los datos sobre la red inmediatamente después de la llamada de sockets *write*(). El comportamiento por defecto del sistema operativo es el de esperar hasta que el búfer esté lleno o bien, con la esperanza de que puedan llegar más datos, utilizar un temporizador que indique cuándo se deben despachar los datos hacia la red.

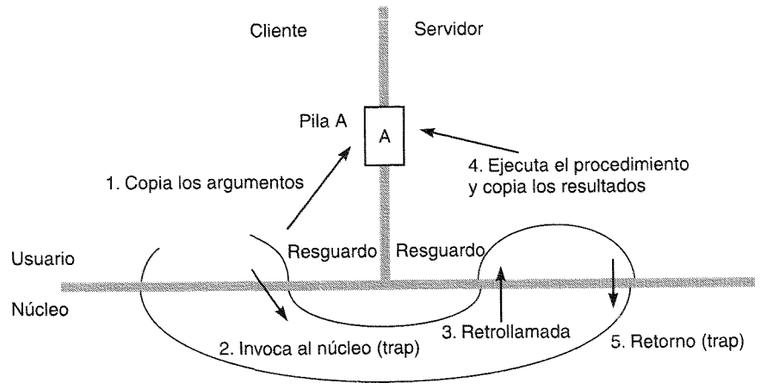


Figura 6.13. Una invocación a procedimiento remoto de peso ligero.

Nielson y otros llegaron a la conclusión de que para el caso de HTTP 1.1 el comportamiento del almacenamiento por defecto del sistema operativo podría causar significativos retardos innecesarios debidos a los tiempos límite. Para eliminar estos retardos alteraron la configuración del TCP en el núcleo para forzar el envío a la red de cada solicitud HTTP de forma individual. Éste es un buen ejemplo de cómo un sistema operativo puede ayudar o entorpecer al middleware en función de las políticas que implemente.

◇ **Invocación dentro de un computador.** Bershad y otros [1990] realizaron un estudio que mostraba que, en la instalación examinada, la mayor parte de las invocaciones entre espacios de direcciones distintos se realizaban dentro de un computador y no, como es de esperar en una instalación cliente-servidor, entre diferentes computadores. La tendencia de situar la funcionalidad del servicio dentro de servidores de nivel de usuario implica que cada vez más invocaciones se realizarán a un proceso local. Esto se acentúa con políticas de caché más agresivas cuando sea factible que los datos requeridos por un cliente estén almacenados en un servidor local. El coste de un RPC dentro de un computador está tomando cada vez más importancia como parámetro de prestaciones del sistema. Estas consideraciones sugieren que el caso local debería optimizarse.

La Figura 6.11 sugiere que una invocación a un espacio de direcciones diferente se implementa dentro de un mismo computador exactamente de la misma forma que entre computadores diferentes, excepto en que el sistema de paso de mensaje subyacente actúa de forma local. Por lo tanto, éste ha sido a menudo el modelo implementado. Bershad y otros [1990] desarrollaron un mecanismo de invocación más eficiente, llamado RPC *de peso ligero* (LRPC, *light weight* RPC), para el caso de dos procesos en la misma máquina. El diseño de LRPC se basa en optimizaciones de la copia de datos y de la planificación de los hilos.

En primer lugar se dieron cuenta que debería ser más eficiente la utilización de regiones de memoria compartida para la comunicación cliente-servidor, con una región (privada) diferente entre el servidor y cada uno de sus clientes. Esta región contiene una o más *pilas A* (de Argumento; véase la Figura 6.13). En lugar de parámetros RPC copiándose entre el espacio de direcciones del núcleo y del usuario involucrado, el cliente y el servidor pueden pasarse argumentos y devolver valores directamente a través de una pila A. Los resguardos del cliente y del servidor utilizan la misma pila. En LRPC, los argumentos se copian una sola vez: cuando son empaquetados en la pila A. En un RPC equivalente, son copiados cuatro veces: desde la pila del resguardo del cliente hacia un mensaje; desde el mensaje al búfer del núcleo; desde el búfer del núcleo al mensaje del servidor; desde el mensaje del servidor a la pila del resguardo del servidor. Puede haber varias pilas A en una región compartida, debido a que varios hilos en el mismo cliente pueden invocar al servidor al mismo tiempo.

Bershad y otros también consideraron el coste de la planificación de hilos. Compárese el modelo de llamada al sistema y llamadas a procedimientos remotos de la Figura 6.11. Cuando se genera una llamada al sistema la mayor parte de los núcleos no planifican un nuevo hilo para manejar la llamada realizando en su lugar un cambio de contexto en el hilo que invoca de forma que él mismo maneja la llamada al sistema. En un RPC debe existir un procedimiento remoto en un computador diferente del computador donde se ejecuta el hilo del cliente, de forma que debe planificarse un hilo diferente para ejecutarlo. Sin embargo, para el caso local, será más eficiente para el hilo del cliente (el cual en otro caso estará *BLOQUEADO*) llamar al procedimiento invocado en el espacio de direcciones del servidor.

En este caso, debe programarse un servidor propio de modo diferente en la forma en la que hemos descrito los servidores previamente. En lugar de inicializar uno o más hilos que escuchan en los puertos esperando solicitudes de invocación, el servidor exporta un conjunto de procedimientos que están preparados para ser llamados. Los hilos en los procesos locales pueden entrar en el entorno de ejecución del servidor mientras se inicializan mediante la llamada a uno de los procedimientos exportados por el servidor. Un cliente que necesita invocar una operación del servidor debe en primer lugar enlazar con la interfaz del servidor (no mostrado en la figura). Esto lo realiza a través del núcleo, que a su vez notifica al servidor; cuando el servidor ha respondido al núcleo con una lista de direcciones de procedimiento permitidas, el núcleo responde al cliente con una capacidad para invocar las operaciones del servidor.

En la Figura 6.13 se muestra una invocación. Un hilo cliente entra en el entorno de ejecución del servidor realizando inicialmente una interrupción software hacia el núcleo y a continuación presentando al núcleo una capacidad. El núcleo lo comprueba y únicamente permite un cambio de contexto hacia un procedimiento válido del servidor; si es válido, el núcleo conmuta el contexto del hilo para llamar al procedimiento en el entorno de ejecución del servidor. Cuando el procedimiento en el servidor termina, el hilo vuelve al núcleo, el cual conmuta el hilo hacia atrás, es decir, hacia el entorno de ejecución del cliente. Hay que resaltar que tanto los clientes como el servidor emplean procedimientos resguardo para ocultar a los programadores de aplicaciones los detalles descritos.

◇ **Discusión de LRPC.** Hay pocas dudas acerca de que LRPC es más eficiente que RPC en el caso local, mientras se realicen suficientes invocaciones para amortizar los costes de gestión de la memoria. Bershad y otros obtuvieron valores de retardo para LRPC menores en un factor de tres que los valores de los RPC ejecutados de forma local.

La transparencia de ubicación no se sacrifica en la implementación de Bershad. Un resguardo del cliente examina un bit actualizado durante el enlace que indica si el servidor es local o remoto, y utiliza LRPC o RPC respectivamente. La aplicación ignora cuál se usa. Sin embargo, la transparencia frente a migración puede ser difícil de conseguir cuando se transfiere un recurso desde un servidor local a un servidor remoto, o viceversa, debido a la necesidad de cambiar los mecanismos de invocación.

En trabajos posteriores, Bershad y otros [1991] describieron varias mejoras de prestaciones, que se dirigen especialmente a la operación en multiprocesadores. Las mejoras se refieren en su mayor parte a la eliminación de interrupciones software hacia el núcleo y a la planificación de procesadores de forma que se eliminen transiciones de dominio innecesarias. Por ejemplo si un procesador está ocioso en el contexto de gestión de memoria del servidor en el momento en que un hilo cliente intenta invocar a un procedimiento del servidor, entonces el hilo debe transferirse a ese procesador. Esto elimina una transición de dominio; al mismo tiempo el procesador del cliente puede ser usado por otro hilo en el cliente. Estas mejoras implican una implementación a dos niveles (usuario y núcleo) de la planificación de hilos, como fue descrito en la Sección 6.4.

6.5.2. OPERACIÓN ASÍNCRONA

Hemos discutido cómo el sistema operativo puede ayudar al nivel de middleware a proporcionar mecanismos de invocación remota eficientes. Sin embargo, hemos observado que en el entorno de Internet los efectos de las elevadas latencias, bajos anchos de banda y elevadas cargas del servidor pueden superar cualquier beneficio que el sistema operativo pueda proporcionar. A esto se le puede añadir el fenómeno de desconexión y reconexión de la red, que puede considerarse como el causante de latencias de comunicación extremadamente elevadas. Los usuarios de computación móvil no están conectados a la red todo el tiempo. Incluso si tienen acceso a una red de área global sin hilos (por ejemplo mediante el uso de GSM) estarán obligatoriamente desconectados cuando, por ejemplo, su tren entre en un túnel.

Una técnica muy utilizada para vencer las latencias muy elevadas es la operación asíncrona, la cual aparece en dos modelos de programación: invocaciones concurrentes e invocaciones asíncronas. Estos modelos aparecen sobre todo en el dominio del middleware en lugar de en el diseño del núcleo del sistema operativo, sin embargo es útil tratarlos aquí, ya que estamos estudiando el tema de las prestaciones de las invocaciones.

◇ **Realizando invocaciones concurrentemente.** En el primer modelo el middleware proporciona únicamente invocaciones bloqueantes, sin embargo la aplicación crea múltiples hilos para realizar estas invocaciones bloqueantes de forma concurrente.

Un buen ejemplo de este tipo de aplicaciones son los navegadores web. Una página web normalmente contiene varias imágenes. El navegador debe pedir cada una de esas imágenes en solicitudes individuales de tipo HTTP GET (ya que los servidores web HTTP 1.0 estándar únicamente soportan solicitudes para recursos individuales). El navegador no necesita obtener las imágenes en una secuencia particular, de forma que realiza solicitudes concurrentes, normalmente hasta cuatro a la vez. De esta forma el tiempo empleado para completar todas las solicitudes de imágenes es normalmente menor que el retardo de hacer las mismas solicitudes en serie. Pero no sólo se reduce el retardo de comunicación total ya que, en general, el navegador puede solapar las actividades de computación del tipo de representación de imágenes con la comunicación.

En la Figura 6.14 se muestran los beneficios potenciales del entrelazado de las invocaciones (del tipo de solicitudes HTTP) entre un cliente y un único servidor en una máquina monoprocesador. En el caso serializado, el cliente empaqueta los argumentos, invoca la operación *Envía* y espera hasta que la respuesta del servidor llegue; cuando eso ocurra ejecuta *Recibe*, desempaqueta y finalmente procesa los resultados. Después de todo esto, puede realizar una segunda invocación.

En el caso concurrente, el primer hilo cliente empaqueta los argumentos y llama a la operación *Envío*. El segundo hilo inmediatamente realiza la segunda invocación. Cada hilo espera la recepción de sus resultados. El tiempo total necesario tiende a ser menor que en el caso serializado, como se muestra en la figura. Beneficios similares se obtienen si los hilos cliente realizan solicitudes concurrentes a diferentes servidores; y si el cliente se ejecuta en un multiprocesador entonces se puede conseguir potencialmente una mayor productividad ya que el procesamiento de los dos hilos se puede realizar de forma solapada.

Volviendo al caso particular de HTTP, el estudio de Nielson y otros [1997] al que nos hemos referido previamente también mide los efectos de invocaciones a HTTP 1.1 entrelazadas concurrentemente (llamados *pipelining*, *entubadas*) sobre conexiones persistentes. Encontraron que el pipelining reduce el tráfico de red y puede derivar en mejores prestaciones para los clientes, siempre que el sistema operativo proporcione un interfaz accesible para vaciar búferes, de forma que se ignore el comportamiento por defecto de TCP.

◇ **Invocaciones asíncronas.** Una *invocación asíncrona* es aquella que se realiza de forma asíncrona respecto al invocador. Es decir, se realiza con una llamada no bloqueante, la cual retorna

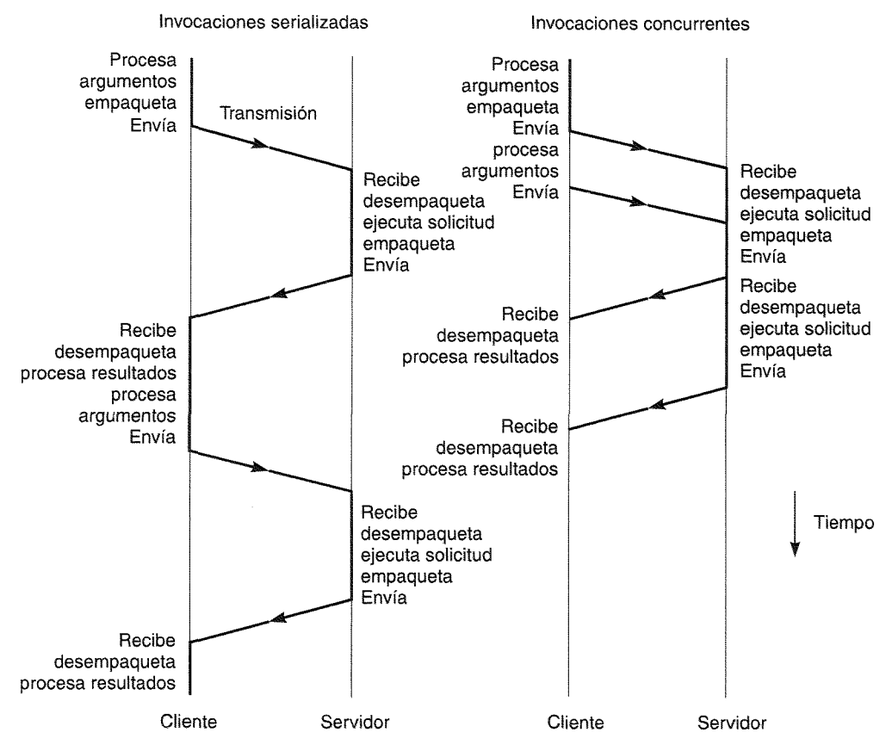


Figura 6.14. Temporización para invocaciones serializadas y concurrentes.

tan pronto como el mensaje de solicitud de invocación ha sido creado y está preparado para ser enviado.

Algunas veces el cliente no necesita respuesta (excepto posiblemente una indicación de fallo si el nodo destino no puede ser alcanzado). Por ejemplo, las invocaciones *de una dirección* de CORBA tienen semántica *pudiera*. En caso contrario el cliente utiliza una llamada separada para obtener los resultados de la invocación. Por ejemplo, el sistema de comunicación Mercury [Liskov y Shira 1988] soporta invocaciones asíncronas. Una operación asíncrona devuelve un objeto llamado *promesa*. Finalmente, cuando la invocación tiene éxito o bien se considera que ha fallado, el sistema Mercury pone el estado y cualquier valor de retorno en la promesa. El que invoca utiliza la operación *reclamación* para obtener los resultados desde la promesa. La operación reclamación se bloquea hasta que la promesa está preparada, con lo que devuelve los resultados o las excepciones desde la llamada. La operación *preparado* está disponible para comprobar una promesa sin necesidad de bloquearse; devuelve *cierto* o *falso* en función de si la promesa está preparada o bloqueada.

◇ **Invocaciones asíncronas persistentes.** Los mecanismos de invocación asíncrona tradicionales como las invocaciones Mercury y las invocaciones *de una dirección* de CORBA son implementadas mediante streams TCP y fallan si un stream se rompe, es decir, si el enlace de red está fuera de servicio o el nodo destino deja de funcionar.

Sin embargo, un modelo de invocación asíncrona más desarrollado, al que llamaremos *invocación asíncrona persistente*, está resultando cada vez más relevante debido a la operación desconec-

tada. Este modelo es similar a Mercury respecto a las operaciones de programación que proporciona, pero la diferencia estriba en la semántica de fallos. Un mecanismo de invocación convencional (síncrono o asíncrono) está diseñado para fallar después de que han ocurrido un cierto número de timeouts. Sin embargo, a menudo estos timeouts a corto plazo no son apropiados cuando se generan desconexiones o latencias muy elevadas.

Un sistema de invocación asíncrona persistente intenta realizar la invocación indefinidamente, hasta que puede determinar que ha tenido éxito o ha fracasado, o hasta que la aplicación cancela la invocación. Un ejemplo es QRPC (RPC en colas) en el conjunto de herramientas Rover para acceso a información móvil [Joseph y otros 1997].

Tal y como el nombre sugiere, QRPC inserta en colas, realizando un registro estable, las solicitudes de invocaciones salientes mientras no exista conexión de red y planifica las operaciones que son enviadas a los servidores a través de la red cuando existe conexión. De forma similar inserta en colas en el servidor los resultados de las invocaciones en lo que se puede considerar como el buzón de invocaciones del cliente, hasta que el cliente vuelva a conectarse y los obtenga. Las solicitudes y los resultados pueden comprimirse al insertarse en las colas, antes de ser transmitidos sobre una red de pequeño ancho de banda.

QRPC se puede aprovechar de los diferentes enlaces de comunicación para el envío de solicitudes de invocación y para la recepción de las respuestas. Por ejemplo, puede enviarse una solicitud sobre un enlace GSM mientras que el usuario esté en la carretera y la respuesta recibirse posteriormente sobre un enlace Ethernet cuando el usuario conecte su dispositivo a la intranet corporativa. En principio, el sistema de invocación puede incluso almacenar los resultados de la invocación cerca del siguiente punto de conexión esperado para el usuario.

El planificador de red del cliente opera según diferentes criterios y no envía necesariamente las invocaciones en orden FIFO. Las aplicaciones pueden asignar prioridades a las invocaciones individuales. Cuando una conexión pasa a estar disponible, QRPC evalúa su ancho de banda y los gastos de su utilización. En primer lugar envía solicitudes de invocación de alta prioridad, no enviándolas todas si el enlace es lento y caro (como las conexiones sin hilos de área global) asumiendo que un enlace más rápido y barato como Ethernet estará finalmente disponible. De forma similar, QRPC utiliza prioridades cuando solicita los resultados de las invocaciones desde el buzón sobre un enlace de bajo ancho de banda.

La programación con un sistema de invocación asíncrona (persistente o de otro tipo) suscita la cuestión de cómo los usuarios pueden continuar usando las aplicaciones en sus dispositivos cliente si los resultados de las invocaciones no son todavía conocidos. Por ejemplo, el usuario se puede preguntar si ha tenido éxito actualizando un párrafo sobre un documento compartido, o bien algún otro usuario puede haber realizado una actualización conflictiva como el borrado de un párrafo. En el Capítulo 14 se examina esta cuestión.

6.6. ARQUITECTURA DEL SISTEMA OPERATIVO

En esta sección se examina la arquitectura de un núcleo válido para un sistema distribuido. Adoptamos una aproximación de principios básicos comenzando con el requisito de apertura y examinando las principales arquitecturas de núcleo propuestas con este requisito en mente.

Un sistema distribuido abierto debería posibilitar lo siguiente:

- Ejecutar en cada computador únicamente el software de sistema que sea necesario para jugar su papel particular en la arquitectura del sistema; los requisitos del software de sistema pueden variar entre, por ejemplo, asistentes digitales personales y servidores de cálculo dedicados. La carga de módulos redundantes desperdicia los recursos de memoria.
- Permitir al software (y al computador) implementar la posibilidad de que cualquier servicio particular sea cambiado independientemente de otros recursos.

- Permitir que se puedan proporcionar diferentes alternativas para el mismo servicio, cuando ello sea requerido por los diferentes usuarios o aplicaciones.
- Introducir nuevos servicios sin dañar la integridad de los ya existentes.

La separación entre los *mecanismos* de gestión de recursos fijos y las *políticas* de gestión de recursos, las cuales varían entre diferentes aplicaciones y entre diferentes servicios, ha sido durante mucho tiempo un principio guía en el diseño de los sistemas operativos [Wulf y otros 1974]. Por ejemplo, decimos que un sistema de planificación ideal debería proporcionar mecanismos para permitir a una aplicación multimedia, como una aplicación de videoconferencia, conseguir sus necesidades de tiempo real mientras coexiste con aplicaciones que no son en tiempo real, como un navegador web.

De forma ideal, el núcleo debería proporcionar únicamente los mecanismos básicos sobre los que se implementan en un cierto nodo las tareas de gestión de recursos generales. Los módulos del servidor deberían cargarse de forma dinámica bajo solicitud, para implementar las políticas de gestión de recursos que solicitan las aplicaciones actualmente en ejecución.

◇ **Núcleos monolíticos y micronúcleos.** Existen dos ejemplos clave en el diseño de los núcleos: la aproximación *monolítica* y la aproximación *micronúcleo*. Estos diseños difieren básicamente en la decisión sobre qué funcionalidad pertenece al núcleo y qué se deja al proceso servidor que puede ser cargado dinámicamente para ejecutarse sobre él. A pesar de que los micronúcleos no están ampliamente extendidos, es instructivo entender sus ventajas e inconvenientes comparándolos con los núcleos típicos existentes actualmente.

Se dice que el núcleo del sistema operativo UNIX es *monolítico* (véase la definición en la caja). Este término intenta sugerir el hecho de que es *masivo*: realiza todas las funciones básicas del sistema operativo necesitando para ello del orden de megabytes de código y datos, y en su composición es *indistinguible*: está codificado de una forma no modular. El resultado es que en gran medida resulta *intratable*: es difícil alterar cualquier componente software individual para adaptarlo a los requisitos cambiantes. Otro ejemplo de un núcleo monolítico es el del sistema operativo en red Sprite [Ousterhout y otros 1988]. Un núcleo monolítico puede contener algunos procesos servidores que se ejecutan dentro de su espacio de direcciones, incluyendo servidores de archivos y de red. El código que ejecutan estos procesos es parte de la configuración estándar del núcleo (véase la Figura 6.15).

◇ **Monolítico.** El Diccionario Chambers del Siglo XX proporciona la siguiente definición de *monolito* y *monolítico*. «**monolito**», *n.* un pilar o una columna o una simple piedra: cualquier cosa que recuerde la uniformidad de un monolito, carácter masivo e inflexibilidad. *-adj.* relacionado con monolítico o parecido a un monolito: de un estado, una organización, etc., masivo y en todo sin diferenciar: por esta razón intratable.

Por el contrario, para el caso de un diseño micronúcleo, el núcleo proporciona únicamente las abstracciones más básicas, principalmente espacios de direcciones, hilos y comunicación *local* entre procesos; el resto de servicios del sistema vienen dados por servidores que se cargan dinámicamente, precisamente en aquellos computadores del sistema distribuido que los requieran (véase la Figura 6.15). Los clientes acceden a esos servicios del sistema utilizando los mecanismos de invocación basados en mensajes proporcionados por el núcleo.

Hemos dicho previamente que los usuarios tienden a rechazar sistemas operativos que no ejecutan sus aplicaciones. Además de la extensibilidad, los diseñadores de micronúcleos tienen otro objetivo: la emulación binaria de sistemas operativos estándar como el UNIX [Armand y otros 1989, Golub y otros 1990, Hartig y otros 1997].

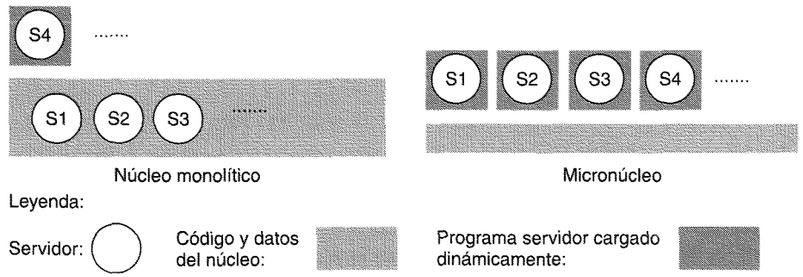


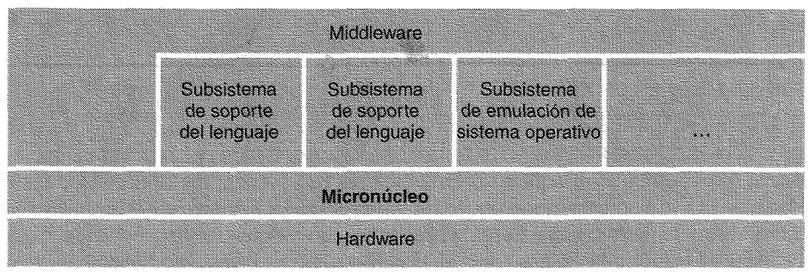
Figura 6.15. Núcleo monolítico y micronúcleo.

La situación del micronúcleo (en su forma más general) dentro del diseño del sistema distribuido en su conjunto se muestra en la Figura 6.16. El micronúcleo aparece como un nivel entre el de hardware y el que forman los principales componentes del sistema, llamados *subsistemas*. Si las prestaciones son el principal objetivo, en lugar de la portabilidad, entonces el middleware puede utilizar directamente los servicios del micronúcleo. En otro caso, utiliza un lenguaje con un subsistema de soporte en tiempo de ejecución, o bien una interfaz de sistema operativo de alto nivel proporcionado por un subsistema de emulación de sistema operativo. Cada uno de ellos se implementa mediante una combinación de procedimientos de biblioteca enlazados dentro de las aplicaciones y un conjunto de servidores ejecutándose sobre el micronúcleo.

Puede existir más de una interfaz de llamadas al sistema (más de un *sistema operativo*) a disposición del programador en la misma plataforma. Esta situación es una reminiscencia de la arquitectura del IBM 370, cuyo sistema operativo VM puede presentar varias máquinas virtuales completas a diferentes programas que se ejecutan en el mismo computador (monoprocesador). Un ejemplo para el caso de sistemas distribuidos es la implementación de UNIX y OS/2 sobre el núcleo del sistema operativo distribuido MACH.

◇ **Comparación.** Las principales ventajas de un sistema operativo basado en micronúcleo son su extensibilidad y su capacidad para hacer cumplir la modularidad por encima de las fronteras de la protección de memoria. Además es más probable que un núcleo relativamente pequeño esté libre de errores que uno mayor y más complejo.

La ventaja de un diseño monolítico es la eficiencia relativa con la que pueden invocarse las operaciones. Las llamadas al sistema pueden ser más caras que los procedimientos convencionales,



El micronúcleo soporta el middleware mediante subsistemas

Figura 6.16. El papel del micronúcleo.

pero incluso utilizando las técnicas que hemos estudiado en la sección anterior, una invocación a un espacio de direcciones de nivel de usuario separado en el mismo nodo es todavía más costosa.

La falta de estructura en los diseños monolíticos puede evitarse utilizando técnicas de ingeniería de software como el diseño por niveles (utilizado en MULTICS [Organick 1972]) o el diseño orientado a objetos, utilizado por ejemplo en Choices [Campbell y otros 1993]. Windows NT utiliza una combinación de ambos [Custer 1998]. Sin embargo, Windows NT sigue siendo *masivo*, y la mayor parte de su funcionalidad no está diseñada para ser reemplazada de forma rutinaria. Incluso un núcleo grande modularizado puede ser difícil de mantener y proporciona un soporte muy limitado para un sistema distribuido abierto. Mientras los módulos se ejecuten dentro del mismo espacio de direcciones utilizando un lenguaje del tipo C o C++, que genera código con el objetivo de la eficiencia pero permite accesos a datos de forma arbitraria, es posible que la modularidad sea abandonada por parte de programadores que buscan implementaciones eficientes, y que un error en un módulo corrompa los datos de otro.

◇ **Algunas aproximaciones híbridas.** Dos de los micronúcleos originales, Mach [Acetta y otros 1986] y Chorus [Rozier y otros 1990], comenzaron su evolución ejecutando los servidores únicamente como procesos de usuario. Por lo tanto la modularidad está asegurada por el hardware mediante los espacios de direcciones. Donde los servidores necesiten acceso directo al hardware, pueden proporcionarse llamadas especiales al sistema para esos procesos privilegiados, las cuales vincularán registros de dispositivo y búferes sobre sus espacios de direcciones. El núcleo convierte interrupciones en mensajes que permitirán a los servidores de nivel de usuario el manejo de interrupciones.

Debido a problemas en las prestaciones, los diseños del micronúcleo de Chorus y Mach finalmente cambiaron para permitir que los servidores pudieran cargarse dinámicamente sobre el espacio de direcciones del núcleo o bien sobre el espacio de direcciones de nivel de usuario. En cualquier caso los clientes interactúan con los servidores utilizando las mismas llamadas de comunicación entre procesos. Un programador puede por lo tanto depurar un servidor al nivel de usuario y, cuando el desarrollo esté aparentemente completo, permitir al servidor ejecutarse dentro del espacio de direcciones del núcleo para optimizar las prestaciones del sistema. Sin embargo, un servidor de este tipo amenaza la integridad del sistema ya que puede llegar a contener errores.

El diseño del sistema operativo SPIN [Bershad y otros 1995] afronta de forma astuta el problema de ceder eficiencia por protección mediante el empleo de servicios de protección en el lenguaje. El núcleo y todos los módulos cargados de forma dinámica e insertados en el núcleo se ejecutan en un único espacio de direcciones. Sin embargo, todos ellos son escritos en un lenguaje de tipos seguro (Modula-3), de forma que pueden estar mutuamente protegidos. Los dominios de protección dentro del espacio de direcciones del núcleo se establecen utilizando espacios de nombres protegidos. Ningún módulo insertado en el núcleo puede acceder a un recurso a no ser que haya gestionado una referencia a él; y Modula-3 fuerza el cumplimiento de la regla que dice que una referencia puede utilizarse únicamente para realizar operaciones permitidas por el programador.

En un intento de minimizar las dependencias entre los módulos del sistema, los diseñadores de SPIN eligen un modelo basado en eventos como mecanismo para la interacción entre módulos insertados en el espacio de direcciones del núcleo (véase la Sección 5.4 para una descripción de la programación basada en eventos). El sistema define un conjunto de eventos centrales, como la llegada de un paquete de red, interrupciones del temporizador, faltas de página y cambios de estado de los hilos. Los componentes del sistema operan registrándose a ellos mismos como manejadores de los eventos que los afectan. Por ejemplo, un planificador se registrará a sí mismo como gestor de eventos similares a aquellos estudiados en el sistema de activaciones del planificador; véase la Sección 6.4.

Los sistemas operativos del tipo de Nemesis [Leslie y otros 1996] explotan el hecho de que, incluso a nivel del hardware, un espacio de direcciones no es necesariamente un único dominio de

protección. El núcleo coexiste en un único espacio de direcciones con todos los módulos del sistema cargados dinámicamente y con todas las aplicaciones. Cuando el núcleo carga una aplicación, sitúa el código de la aplicación y los datos en regiones seleccionadas entre aquéllas disponibles en tiempo de ejecución. La llegada de procesadores con direcciones de 64-bits ha provocado que los sistemas operativos de espacio de direcciones único pasen a ser especialmente atractivos, ya que soportan espacios de direcciones muy grandes que pueden albergar muchas aplicaciones.

El núcleo de un sistema operativo de espacio de direcciones único configura los atributos de protección sobre regiones individuales dentro del espacio de direcciones para restringir el acceso al código de nivel de usuario. El código de nivel de usuario se ejecuta todavía con el procesador en un contexto de protección particular (determinado por la configuración en el procesador y en la unidad de gestión de memoria), lo que proporciona acceso completo a sus propias regiones y únicamente acceso compartido de forma selectiva a otras regiones. El ahorro conseguido por el espacio de direcciones único, comparado con la utilización de múltiples espacios de direcciones, es que el núcleo nunca vacía cualquier caché cuando implementa una transición de dominio.

Algunos diseños de núcleos más recientes, del tipo de L4 [Hartig y otros 1997] y Exokernel [Kaashoek y otros 1997], se basan en la idea de lo que hemos llamado *micronúcleos* todavía contienen excesiva política en relación con mecanismos. L4 es un diseño de micronúcleo de *segunda generación* que obliga a los módulos del sistema cargados dinámicamente a ejecutarse en espacios de direcciones de nivel de usuario, pero optimiza la comunicación entre procesos para compensar los costes de ese tipo de operación. Descarga la mayor parte de la complejidad del núcleo cediendo la gestión de los espacios de direcciones a los servidores de nivel de usuario. Exokernel utiliza una aproximación completamente diferente, basada en el empleo de bibliotecas de nivel de usuario en lugar de servidores de nivel de usuario para proporcionar las extensiones funcionales. Proporciona asignación protegida de recursos de muy bajo nivel, como bloques de disco, y espera que el resto de la funcionalidad de gestión de recursos (incluso un sistema de archivos) sea realizada mediante el enlace de bibliotecas en las aplicaciones.

En palabras de un diseñador de micronúcleos [Liedtke 1996], *la historia de los micronúcleos está llena de buenas ideas y de callejones sin salida*. La cuestión todavía pendiente estriba en cómo diseñar una arquitectura de sistema operativo que sea lo suficientemente extensible y que proporcione buenas prestaciones en comparación con los diseños monolíticos.

6.7. RESUMEN

Este capítulo ha descrito cómo el sistema operativo soporta el nivel de middleware proporcionando invocaciones sobre recursos compartidos. El sistema operativo proporciona una colección de mecanismos sobre los que se pueden implementar múltiples políticas de gestión de recursos, con el objeto de satisfacer los requisitos locales y para aprovechar las mejoras tecnológicas. Permite a los servidores encapsular y proteger los recursos, mientras que permite a los clientes compartirlos concurrentemente. Proporciona los mecanismos necesarios para que los clientes invoquen operaciones sobre los recursos.

Un proceso está formado por un entorno de ejecución e hilos: un entorno de ejecución consiste en un espacio de direcciones, interfaces de comunicación y otros recursos locales como semáforos; un hilo es una abstracción de la actividad que se ejecuta dentro de un entorno de ejecución. Los espacios de direcciones deben ser grandes y dispersos para soportar la compartición y el acceso mediante correspondencias de objetos como los archivos. Se pueden crear espacios de direcciones nuevos cuyas regiones sean heredadas de los procesos padre. Una técnica importante para copiar regiones es la llamada copia en escritura. Los procesos pueden tener múltiples hilos, los cuales comparten el entorno de ejecución. Los procesos multi-hilo permiten conseguir concurrencia relativamente barata y aprovechar el paralelismo de los multiprocesadores. Son útiles tanto para los clientes como para los servidores. Implementaciones recientes de hilos permiten una planificación a dos niveles: el núcleo proporciona acceso a múltiples procesadores, mientras que el código de nivel de usuario maneja los detalles de la política de planificación.

El sistema operativo proporciona primitivas básicas de paso de mensajes y mecanismos de comunicación a través de memoria compartida. La mayor parte de los núcleos incluyen la comunicación sobre red como un servicio básico; otros proporcionan únicamente comunicación local y dejan la comunicación sobre la red a los servidores, que pueden implementar múltiples protocolos de comunicación. Se trata de una pugna de las prestaciones contra la flexibilidad.

Discutimos las invocaciones remotas y contabilizamos las diferencias entre las sobrecargas debidas directamente al hardware de red y las debidas a la ejecución de código de sistema operativo. Encontramos que la proporción del tiempo total debida al software es relativamente grande para una invocación nula pero disminuye proporcionalmente al tamaño de los argumentos de la invocación. Las principales sobrecargas relacionadas con una invocación que son susceptibles de ser optimizadas son el empaquetamiento, la copia de datos, la inicialización de los paquetes, la planificación de los hilos y el cambio de contexto y el flujo de control del protocolo usado. La invocación entre diferentes espacios de direcciones dentro del mismo computador es un caso especialmente importante, por lo que hemos descrito las técnicas de gestión de hilos y de paso de parámetros utilizadas en los RPC de peso ligero.

Existen principalmente dos aproximaciones a la arquitectura del núcleo: núcleos monolíticos y micronúcleos. La principal diferencia entre ellos reside en dónde se coloca la línea separadora entre la gestión de recursos del núcleo y la gestión de recursos realizada por servidores cargados dinámicamente (y normalmente de nivel de usuario). Un micronúcleo debe soportar al menos la noción de proceso y la comunicación entre procesos. Soporta subsistemas de emulación de sistemas operativos así como soporte de lenguaje y otros subsistemas, como los de procesamiento en tiempo real.

EJERCICIOS

6.1. Discuta cada una de las tareas de encapsulamiento, procesamiento concurrente, protección, resolución de nombres, comunicación de parámetros y resultados y planificación para el caso del servicio de archivos de UNIX (o de cualquier otro núcleo con el que esté familiarizado)

6.2. ¿Por qué algunas interfaces de sistema están implementadas con llamadas al sistema dedicadas (o al núcleo), mientras que otras lo están sobre llamadas al sistema basadas en mensajes?

6.3. González decide que cada hilo de sus procesos debe tener su propia pila *protegida* mientras que el resto de regiones de cada proceso podrá estar completamente compartida. ¿Tiene esto sentido?

6.4. ¿Deben los gestores de señales (interrupciones software) pertenecer a un proceso o a un hilo?

6.5. Discuta la cuestión de los nombres aplicada a las regiones de memoria compartidas.

6.6. Sugiera un esquema para equilibrar la carga en un conjunto de computadores. Debe discutir:

i) Qué requisitos de usuario o sistema debe cumplir el esquema.
ii) A qué categoría de aplicaciones está dirigido.

iii) Cómo se mide la carga y con qué precisión.
iv) Cómo se monitoriza la carga y se selecciona la localización de un nuevo proceso. Suponga que los procesos no pueden migrar.

¿Cómo puede su diseño verse afectado si los procesos pueden migrar entre computadores? ¿Espera que la migración de procesos tenga un coste significativo?

6.7. Explique las ventajas de la copia en escritura de regiones en UNIX, en la que una invocación a *fork* está normalmente seguida de una invocación a *exec*. ¿Qué ocurrirá si una región que ha sido copiada utilizando copia en escritura se copia a sí misma?

6.8. Un servidor de archivos utiliza caché y consigue una tasa de acierto del 80 %. Las operaciones sobre archivos en el servidor suponen 5 milisegundos de tiempo de CPU cuando el servidor encuentra en la caché el bloque solicitado, y suponen otros 15 milisegundos adicionales de E/S en disco si no lo encuentra. Explicando cualquier suposición que realice, estime la productividad del servidor (valor medio de solicitudes/segundo) si éste es:

i) Mono-hilo.
ii) Dos hilos ejecutándose en un único procesador.
iii) Dos hilos ejecutándose en un computador con dos procesadores.

6.9. Compare la arquitectura multi-hilo de asociación de trabajadores con la arquitectura de hilo por solicitud.

6.10. ¿Qué operaciones sobre hilos son más costosas?

6.11. Un *spin lock* (véase Bacon [1998]) es una variable booleana accedida a través de una instrucción atómica *test-and-set*, que se utiliza para conseguir exclusión mutua. ¿Se podría utilizar un *spin lock* para obtener exclusión mutua entre hilos en un computador con un único procesador?

6.12. Explique qué debe proporcionar el núcleo para una implementación de hilos a nivel de usuario, como en Java o en UNIX.

6.13. ¿Las faltas de página son un problema para la implementación de hilos de nivel de usuario?

6.14. Explique los factores que motivan la planificación híbrida del diseño de *activaciones de planificación* (en lugar de la planificación pura de nivel de usuario o de nivel de núcleo).

6.15. ¿Por qué debería estar interesado un paquete de hilos en los eventos que provocan que un hilo pase a bloqueado o a desbloqueado? ¿Por qué debería estar interesado en el evento de desalojo inminente de un procesador virtual? (Nota: otros procesadores virtuales pueden ser asignados.)

6.16. El tiempo de transmisión de red supone un 20 % en un RPC nulo y un 80 % en un RPC que transmite 1.024 bytes de usuario (menor que el tamaño de un paquete de red). ¿En qué porcentaje de tiempo mejorarán esas dos operaciones si la red es actualizada desde 10 megabits/segundo a 100 megabits/segundo?

6.17. Un RMI nulo sin parámetros que invoca un procedimiento vacío y que no devuelve ningún valor retarda al invocador 2,0 milisegundos. Explique qué contribuye a conseguir ese valor. En el mismo sistema RMI, cada 1 K de datos de usuario añade 1,5 milisegundos extra. Un cliente necesita solicitar 32 K de datos desde un servidor de archivos. ¿Debe usar un RMI de 32 K o bien 32 RMI de 1 K?

6.18. ¿Qué factores identificados en el coste de una invocación remota también afectan a un sistema de paso de mensajes?

6.19. Explique cómo puede utilizarse una región compartida para que un proceso lea datos escritos por el núcleo. Incluya en su explicación lo necesario para la sincronización.

6.20. i) ¿Puede un servidor que es invocado por una llamada a procedimiento de peso ligero controlar el grado de concurrencia dentro de él?
ii) Explique por qué y cómo un cliente que utiliza RPC's de peso ligero no puede realizar invocaciones a código arbitrario dentro de un servidor.
iii) ¿Expone LRPC a los clientes y servidores a mayores riesgos de interferencia mutua que los RPC convencionales (dada la compartición de memoria)?

6.21. Un cliente realiza RMI sobre un servidor. El cliente necesita 5 milisegundos para computar los argumentos de cada solicitud y el servidor necesita 10 milisegundos para procesar cada solicitud. El tiempo de procesamiento local del sistema operativo para cada operación *envío* o *recepción* es de 0,5 milisegundos y el tiempo necesario de red para transmitir cada solicitud o respuesta es de 3 milisegundos. El empaquetamiento y el desempaquetamiento necesitan 0,5 milisegundos por cada mensaje.
Estime el tiempo que necesita un cliente para generar y obtener resultados de 2 solicitudes (i) si es mono-hilo, y (ii) si tiene dos hilos que pueden realizar solicitudes de forma concurrente sobre un único procesador. ¿Hay necesidad de que RMI sea asíncrono si los procesos son multi-hilo?

6.22. Explique qué se entiende por política de seguridad y cuáles son los mecanismos correspondientes para el caso de un sistema operativo multiusuario del tipo de UNIX.

6.23. Explique los requisitos de enlazado de programas que deben cumplirse si un servidor es cargado dinámicamente dentro del espacio de direcciones del núcleo, y cómo estos requisitos difieren para el caso de ejecutar el servidor al nivel de usuario.

6.24. ¿Cómo puede comunicarse una interrupción a un servidor de nivel de usuario?

6.25. Cierto computador se estima que, independientemente de que lo ejecute o no el sistema operativo, la planificación de hilos cuesta alrededor de 50 microsegundos, una invocación a un procedimiento nulo 1 milisegundo, un cambio de contexto hacia el núcleo cuesta 20 microsegundos y una transición de dominio 40 microsegundos. Para Mach y SPIN, estime el coste de un cliente que invoca a un procedimiento nulo cargado dinámicamente.

7

SEGURIDAD

7.1. Introducción
7.2. Visión general de las técnicas de seguridad
7.3. Algoritmos criptográficos
7.4. Firmas digitales
7.5. Prágmática de la criptografía
7.6. Casos de estudio: Needham-Schroeder, Kerberos, SSL y Millicent
7.7. Resumen

Hay una necesidad apremiante de medidas para garantizar la privacidad, integridad y disponibilidad de recursos en los sistemas distribuidos. Los ataques contra la seguridad toman las formas de escuchas, suplantación, modificación y denegación de servicio. Los diseñadores de sistemas distribuidos seguros deben tratar con interfaces de servicio desprotegidos y redes inseguras en un entorno donde se supone que los atacantes tienen conocimiento sobre los algoritmos empleados para desplegar los recursos computacionales.

La criptografía proporciona la base de la autenticación de mensajes así como del secreto y la integridad; y para explotarla es preciso utilizar protocolos de seguridad diseñados cuidadosamente. La selección de algoritmos criptográficos y la administración de claves son puntos críticos para la efectividad, prestaciones y usabilidad de los mecanismos de seguridad. La criptografía de clave pública facilita la distribución de claves criptográficas pero sus prestaciones son inadecuadas para la encriptación de datos masivos. La criptografía de clave secreta es más adecuada para tareas de encriptación masiva. Los protocolos híbridos como SSL (*Secure Sockets Layer*) establecen canales seguros empleando criptografía de clave pública para intercambiar claves secretas que se usarán en subsiguientes intercambios de datos.

La información digital puede venir firmada, ofreciéndonos certificados digitales. Estos certificados permitirán establecer relaciones de confianza entre los usuarios y las organizaciones.

7.1. INTRODUCCIÓN

En todos aquellos sistemas de computadores, donde haya objetivos potenciales para ataques maliciosos o con fines de diversión habrá que incluir medidas de seguridad. Esto es especialmente cierto para sistemas que traten con transacciones financieras, confidenciales, clasificadas u otra información cuyo secreto e integridad sea crítica. En la Figura 7.1, se resume la evolución de las necesidades de seguridad en los sistemas de computadores desde su aparición con el advenimiento de la compartición de datos en los sistemas de tiempo compartido multiusuario de las décadas de los años sesenta y setenta. Hoy en día, con el advenimiento de los sistemas distribuidos abiertos de carácter global se nos presenta una amplia lista de cuestiones sobre seguridad.

La necesidad de proteger la integridad y la privacidad de la información, y otros recursos que pertenecen a individuos y organizaciones, conjuga ambos mundos: el físico y el digital. Nace, como es lógico, de la necesidad de compartir recursos. En el mundo físico, las organizaciones adoptan *políticas de seguridad* para poder compartir recursos dentro de unos límites especificados. Por ejemplo, una compañía podrá permitir la entrada a sus dependencias a ciertos empleados y visitantes autorizados. Una política de seguridad para documentos especificará grupos de empleados que podrán acceder a clases de documentos, o incluso se especificarán usuarios concretos para ciertos documentos importantes.

Las políticas de seguridad se hacen cumplir con la ayuda de los *mecanismos de seguridad*. Por ejemplo, un recepcionista controlará el acceso a un edificio, extenderá identificaciones a los visitantes acreditados, y esto se hará cumplir por un guardia de seguridad o mediante cerraduras electrónicas. El acceso a los documentos impresos se puede controlar mediante una custodia y una distribución rigurosa.

En el mundo electrónico, la distinción entre políticas de seguridad y los mecanismos también es importante; sin ella, sería difícil determinar si un sistema particular es seguro. Las políticas de seguridad son independientes de la tecnología empleada, así como el instalar una cerradura en una puerta no garantiza la seguridad del edificio a menos que haya una política de uso (por ejemplo, que la puerta esté cerrada cuando no esté vigilada). Los mecanismos de seguridad que describiremos no garantizan, por sí mismos, la seguridad de un sistema. En la Sección 7.1.2, se bosquejarán los requisitos de seguridad en distintos escenarios simplificados de comercio electrónico, ilustrando la necesidad de las políticas de seguridad en ese contexto. Como ejemplo de partida, considere la seguridad de un servidor de archivos en red cuya interfaz sea accesible a los clientes. Para garantizar que se mantiene el control de acceso a los archivos, deberá existir una política que asegure que todas las peticiones deban incluir una identidad de usuario autenticada.

	1965-75	*1975-89*	*1990-99*	*Actualmente*
Plataformas	Computadores multiusuario de tiempo compartido	Sistemas distribuidos basados en redes locales	Internet, servicios de área extensa	Internet + dispositivos móviles
Recursos compartidos	Memoria, archivos	Servicios locales (p. ej.: NFS), redes locales	e-mail, lugares web, comercio Internet	Objetos distribuidos, código móvil
Requisitos de seguridad	Identificación y autenticación de usuario	Protección de servicios	Seguridad robusta para transacciones comerciales	Control de acceso para objetos individuales, código móvil seguro
Entorno de gestión de la seguridad	Autoridad única, base de datos de autorización única (p. ej.: /etc/passwd)	Autoridad única, delegación, bases de datos de autorización replicadas (p. ej.: NIS)	Muchas autoridades, sin autoridad en la red, en general	Autoridades por actividad, grupos con responsabilidades compartidas

Figura 7.1. Contexto histórico: evolución de las necesidades de seguridad.

Cómo proporcionar mecanismos de seguridad para proteger datos (y otros recursos computacionales) y para asegurar las transacciones en la red es el tema de este capítulo. Describiremos los mecanismos que permiten hacer cumplir las políticas de seguridad en los sistemas distribuidos. Los mecanismos que veremos son suficientemente fuertes para resistir los ataques más determinados.

La distinción entre políticas de seguridad y mecanismos de seguridad es de utilidad cuando se diseñan sistemas seguros, pero no es fácil estar seguro de que cierto conjunto de mecanismos de seguridad implementan completamente las políticas de seguridad deseadas. En la Sección 2.3.3, presentamos un modelo de seguridad diseñado para ayudar en el análisis de las amenazas de seguridad potenciales en un sistema distribuido. Podemos resumir el modelo de seguridad del Capítulo 2 como sigue:

- Los procesos encapsulan recursos (tales como objetos de un lenguaje de programación de alto nivel y otros recursos definidos en el sistema) y acceden a comunicarse con los clientes a través de sus interfaces. Los principales (usuarios u otros procesos) pueden estar autorizados explícitamente para operar sobre los recursos. Los recursos deben estar protegidos contra accesos no autorizados.
- Los procesos interactúan en la red, que está siendo compartida por muchos usuarios. Los enemigos (atacantes) pueden acceder a la red. Así, podrán copiar o intentar leer cualquier mensaje que se transmita por este medio así como introducir mensajes arbitrarios, dirigidos hacia cualquier destino y simular que provienen de cualquier otro lugar de la red.

Este modelo de seguridad identifica las características de los sistemas de seguridad expuestos a ataques. En este capítulo detallaremos estos ataques y las técnicas de seguridad disponibles para derrotarlos.

◇ **La emergencia de la criptografía en el dominio público.** La criptografía proporciona la base para la mayoría de los sistemas de seguridad de los computadores. La criptografía tiene una larga y fascinante historia. La necesidad de comunicaciones militares seguras y la correspondiente necesidad del enemigo de interceptarlas y desencriptarlas ha fomentado la inversión de mucho esfuerzo intelectual, por parte de los mejores cerebros matemáticos de cada época. Los lectores interesados en explorar su historia encontrarán sobre el tema libros de absorbente lectura por David Kahn [1967, 1983, 1991] y Simon Singh [1999]. Whitfield Diffie, uno de los inventores de la criptografía de clave pública, ha relatado experiencias de primera mano sobre la historia y la política reciente [Diffie 1998, Diffie y Landau 1998] y también en el prefacio al libro de Schneier [1996].

Pero sólo en tiempos recientes la criptografía emerge de la trastienda en la que fue puesta por la clase dirigente de políticos y militares que solían controlar su desarrollo y su aplicación. Hoy en día es un tema de investigación abierto y con una comunidad de investigadores amplia y muy activa, donde los resultados se publican en muchos libros, revistas y conferencias. La publicación del libro de Schneier, *Applied Cryptography* [1996], fue la piedra de toque de la apertura de este campo del conocimiento. Fue el primer libro en que se publicaron muchos algoritmos importantes, incluyendo código fuente; una valiente aportación, dado que cuando apareció su primera edición en 1994 la situación legal de tal publicación no estaba nada clara. Schneier sigue siendo la referencia definitiva en la mayoría de los aspectos de la criptografía moderna. Menezes y otros [1997] también proporcionan un buen manual práctico con una interesante base teórica.

La reciente apertura es, en su mayor medida, resultado del importante crecimiento del interés en las aplicaciones no militares de la criptografía y los requisitos de seguridad de los sistemas de computadores distribuidos. Esto desembocó en la existencia, por primera vez, de una comunidad autosuficiente de criptógrafos aparte del entorno militar.

Irónicamente, esta apertura al público de la criptografía ha traído consigo un mayor avance de las técnicas criptográficas, su resistencia a los ataques criptoanalíticos y la comodidad con la que se despliegan las medidas criptográficas. La criptografía de clave pública es fruto de esta apertura. Un ejemplo más, el algoritmo de encriptación estándar DES fue inicialmente un secreto militar. Su

Alice	Primer participante
Bob	Segundo participante
Carol	Otro participante en los protocolos a tres o cuatro bandas
Dave	Participante en protocolos a cuatro bandas
Eve	Fisgón
Mallory	Atacante malevolente
Sara	Un servidor

Figura 7.2. Nombres familiares (del mundo anglosajón) para los protagonistas de los protocolos de seguridad.

eventual publicación y los esfuerzos exitosos para romperlo han traído consigo el desarrollo de algoritmos de encriptación de clave secreta mucho más resistentes.

Otro producto secundario útil ha sido el desarrollo de una terminología y aproximación común. Un ejemplo de esto último es la adopción de un conjunto de nombres familiares para los protagonistas (principales) involucrados en las transacciones que hay que asegurar. El uso de nombres familiares para los principales y los atacantes ayuda a aclarar y acercar al mundo las descripciones de los protocolos de seguridad y los potenciales ataques sobre ellos, lo que supone un paso importante hacia la identificación de sus debilidades. Los nombres mostrados en la Figura 7.2 se emplean ampliamente en literatura internacional sobre seguridad y aquí los emplearemos libremente. No hemos sido capaces de descubrir sus orígenes; la aparición más temprana de la que tenemos noticia está en el artículo original de criptografía de clave pública RSA [Rivest y otros 1978]. En Gordon [1984] podemos encontrar un divertido comentario sobre su utilización.

7.1.1. AMENAZAS Y ATAQUES

Algunos ataques son obvios; por ejemplo, en la mayoría de los tipos de redes locales es fácil construir y lanzar un programa sobre un computador conectado para que obtenga copias de los mensajes transmitidos entre otros computadores. Otras amenazas son más sutiles; si los clientes fallan en autenticar los servidores, un programa podría situarse a sí mismo en lugar del auténtico servidor de archivos y así obtener copias de información confidencial que los clientes, inconscientemente, envían para su almacenamiento.

Además del peligro de pérdida o daño de información o de recursos por violaciones directas, también pueden aparecer reclamaciones fraudulentas contra el propietario de un sistema que no sea demostrablemente seguro. Para evitar tales reclamaciones, el propietario debe estar en situación de desacreditar la reclamación mostrando que el sistema es seguro contra tales violaciones, o también produciendo un registro histórico de todas las transacciones durante el período en cuestión. Un ejemplo habitual es el problema de *débito fantasma* en los dispensadores automáticos de dinero en efectivo (cajeros automáticos). La mejor respuesta que un banco puede aportar a tal reclamación es proporcionar un registro de la transacción firmado digitalmente por el titular de la cuenta, de manera que no pueda ser falsificado por un tercero.

La principal meta de la seguridad es restringir el acceso a la información y los recursos de modo que sólo tengan acceso aquellos principales autorizados. Las amenazas de seguridad caen en tres amplias clases:

Fuga — la adquisición de información por receptores no autorizados.

Alteración — la modificación no autorizada de información.

Vandalismo — interferencia con el modo de operación adecuado de un sistema, sin ganancia alguna para el perpetrador.

Los ataques en los sistemas distribuidos dependen de la obtención de acceso a los canales de comunicación existentes o del establecimiento de canales nuevos que se suplantan a las conexiones. (Empleamos el término *canal* para hacer alusión a cualquier mecanismo de comunicación entre procesos.) Los métodos de ataque pueden clasificarse más aún en función del modo en que se abusa del canal:

Fisgar — obtener copias de mensajes sin autoridad.

Suplantar — enviar o recibir mensajes utilizando la identidad de otro principal sin su autorización.

Alterar mensajes — interceptar mensajes y alterar sus contenidos antes de pasarlos al receptor pretendido. El *ataque del 'hombre entre medias'* es una forma de *alteración de mensaje* en la cual un atacante intercepta el primer mensaje en un intercambio de claves de encriptación (al establecer un canal seguro). El atacante sustituye claves comprometidas que le permitirán desencriptar los subsiguientes mensajes antes de encriptarlos de nuevo, con las claves correctas, y dejarlos pasar hasta su destino.

Reenviar — almacenar mensajes interceptados y enviarlos más tarde. Este ataque pudiera ser efectivo incluso con mensajes encriptados autenticados.

Denegación de servicio — desbordar un canal u otro recurso con mensajes con el fin de impedir que otros accedan a él.

En teoría, éstos son los peligros, pero, ¿cómo se llevan a la práctica estos ataques? Los ataques victoriosos dependen del descubrimiento de agujeros en la seguridad de los sistemas. Desgraciadamente, estos problemas son demasiado comunes en los sistemas de hoy en día, y no son necesariamente complicados. Cheswick y Bellovin [1994] identificaron cuarenta y dos debilidades que pudieran poner en serios riesgos para sistemas y componentes de Internet ampliamente usados. Empezando desde la adivinación de claves de acceso hasta ataques en los programas que implementan el protocolo de tiempo de red, o manejan la transmisión de correo. Algunos de ellos han desembocado en famosos ataques [Stoll 1898, Spafford 1989], y muchos de ellos han sido explotados con fines lúdicos o criminales.

Cuando se diseñó Internet y los sistemas conectados a ella, la seguridad no era una prioridad. Los diseñadores, probablemente, no tenían un concepto adecuado de la escala en que crecería Internet, aparte de que el diseño básico de sistemas como UNIX es previo al advenimiento de las redes de computadores. Como ya veremos, la incorporación de medidas de seguridad requiere ser cuidadoso con la etapa de diseño, y se pretende que el contenido de este capítulo proporcione las bases de tal idea.

Nos hemos concentrado en los ataques a los sistemas distribuidos que nacen de la exposición de sus canales de comunicación y sus interfaces. Para muchos sistemas, éstos son los únicos ataques que debemos considerar y que no se derivan de los errores humanos; los mecanismos de seguridad no pueden protegernos contra una clave de acceso mal elegida o mal custodiada. Pero para sistemas que incluyan programas móviles y sistemas cuya seguridad sea particularmente sensible a la fuga de información, hay aún más ataques.

◇ **Ataques desde código móvil.** Varios lenguajes de programación, recientemente desarrollados, han sido diseñados para permitir que la descarga de programas desde servidores remotos y así lanzar procesos que se ejecutan localmente. En este caso, las interfaces internas y los objetos del interior de un proceso en ejecución pueden quedar expuestos a un ataque por código móvil.

Java es el lenguaje de este tipo más conocido, y los diseñadores pusieron considerable atención en el diseño y la construcción del lenguaje y los mecanismos para la descarga remota, en un esfuerzo para restringir el riesgo (el modelo de *caja de arena —sandbox—* de protección contra código móvil).

La Máquina Virtual Java (JVM, *Java Virtual Machine*) se diseñó con el código móvil en mente. A cada aplicación se le da su propio entorno de ejecución. Cada entorno tiene un gestor de

seguridad que determina qué recursos están disponibles para la aplicación. Por ejemplo, el gestor de seguridad podría detener una aplicación que lee y escribe archivos, o darle acceso limitado a las conexiones de red. Una vez lanzado un gestor de seguridad, no puede remplazarse. Cuando un usuario ejecuta un programa como un visualizador web que descarga código móvil para ser utilizado localmente en su nombre, no hay ninguna buena razón para confiar en que éste se comporte de una forma responsable. De hecho, existe el peligro de descargar y ejecutar código malicioso que borra archivos o accede a información privada. Para proteger a los usuarios contra código no fiable la mayoría de los navegadores especifican que los *applets* no puedan acceder a archivos locales, impresoras o *sockets* de red. Algunas aplicaciones de código móvil son capaces de asumir diversos niveles de confianza en el código descargado. En este caso, los gestores de seguridad se configuran para proporcionar más acceso a los recursos locales.

JVM toma dos medidas a mayores para proteger el entorno local:

1. Las clases descargadas se almacenan separadas de las clases locales, previniendo que puedan reemplazar a estas últimas con versiones espurias.
2. Se comprueba la validez de los *bytecodes*. El *bytecode* Java válido se compone de instrucciones de máquina virtual Java de un conjunto especificado. Las instrucciones también se comprueban para asegurar que no producen ciertos errores cuando se ejecuta el programa, tales como el acceso ilegal a ciertas posiciones de memoria.

La seguridad de Java ha sido el tema de mucha investigación posterior, en el curso de la cual ha surgido a la luz que los mecanismos originales no estaban libres de agujeros [McGraw y Felden 1999]. Los agujeros identificados se corrigieron y el sistema de protección de Java se refinó para permitir que el código móvil acceda a recursos locales cuando se le autorice a ello [Java.sun.com V].

A pesar de la inclusión de mecanismos de comprobación de tipo y de validación de código, los mecanismos de seguridad incorporados a los sistemas de código móvil aún no dan el mismo nivel de confianza en su efectividad que los usados para proteger los canales e interfaces de comunicación. Esto es así porque la construcción de un entorno para la ejecución de programas ofrece muchas oportunidades de error, y es difícil tener la certeza de que se han evitado todos. Volpano y Smith [1999] apuntan que una solución apropiada pasaría por el empleo de pruebas sobre la solidez del comportamiento del código móvil.

◇ **Fugas de información.** Si pudiera observarse la sucesión de mensajes en la comunicación entre dos procesos, sería posible vislumbrar información importante aún de su sola existencia; por ejemplo, un flujo de mensajes hacia un vendedor acerca de una mercancía en particular podría indicar un alto nivel de comercio en esa mercancía. Hay muchas formas sutiles de fugas de información, algunas maliciosas y otras que son consecuencia de errores inadvertidos. El potencial de las fugas aparece cuando se pueden observar los resultados de un cómputo. Desde los años 70 se ha trabajado en la prevención de este ataque a la seguridad [Denning y Denning 1977]. La aproximación empleada es la asignación de niveles de seguridad a la información y los canales, y analizar el flujo de información hacia los canales con el objetivo de asegurar que la información de alto nivel no fluya hacia los canales de bajo nivel. Un método para el control seguro de los flujos de información fue descrito primeramente por Bell y LaPadua [1975]. La extensión de esta aproximación a los sistemas distribuidos con desconfianza mutua entre sus componentes es un tema reciente de investigación [Myers y Liskov 1997].

7.1.2. SEGURIDAD DE LAS TRANSACCIONES ELECTRÓNICAS

Muchas aplicaciones de Internet en la industria, el comercio y demás implican transacciones que dependen crucialmente de la seguridad. Por ejemplo:

Email: aunque los sistemas originales de correo electrónico no incluyeran soporte de seguridad, hay muchos usos del correo en que los contenidos de los mensajes deben mantenerse confidenciales (por ejemplo, al enviar un número de tarjeta de crédito) o los contenidos y el emisor del mensaje deben estar autenticados (por ejemplo cuando se remite una puja a una subasta por email). La seguridad criptográfica basada en las técnicas que se describen en este capítulo se incluye actualmente en muchos clientes de correo.

Compra de bienes y servicios: tales transacciones son, hoy en día, algo usual. Los compradores seleccionan bienes y pagan por ellos empleando el Web, más tarde le son enviados por el mecanismo de reparto más apropiado. El software y otros productos digitales (como grabaciones y vídeo) se pueden enviar mediante el procedimiento de descarga desde Internet. Los bienes tangibles como libros, discos compactos y casi cualquier otro tipo de producto puede venderse desde comercios en Internet; éstos se envían mediante un servicio de reparto.

Transacciones bancarias: los bancos electrónicos de hoy en día ofrecen virtualmente a los usuarios todos los servicios que proporcionan los bancos convencionales. Éstos proporcionan la comprobación de los cargos y balance de cuentas, transferencias de efectivo entre cuentas, el establecimiento de cargos regulares automáticos y demás desde la cuenta.

Micro-transacciones: Internet se presta a proporcionar pequeñas cantidades de información y otros servicios hacia sus clientes. Por ejemplo, el acceso a la mayoría de las páginas web no exige ningún pago, pero el desarrollo del Web como un medio de publicación de alta calidad seguramente depende de hasta qué punto los proveedores de información puedan obtener beneficio de los clientes de esta información. El uso de Internet para voz y videoconferencia nos da otro ejemplo de servicio que se proporciona sólo cuando se anticipa el pago por los usuarios finales. El precio de tales servicios puede suponer sólo una fracción de un céntimo, y el pago de los gastos generales deberá ser correspondientemente bajo. En general los esquemas de pago basados en transferencia bancaria y servicio de tarjeta de crédito para cada transacción no son los más adecuados.

Las transacciones como éstas sólo se pueden realizar de modo seguro cuando se encuentran protegidas contra la revelación de los códigos de crédito (números de tarjeta) durante la transmisión, y contra un vendedor fraudulento que obtenga un pago sin intención de proveer bien alguno. Los vendedores deben recibir el pago antes de enviar los bienes, y para los productos que pueden descargarse deberá asegurarse de que sólo el comprador recibe los datos de forma utilizable. La protección necesaria debe obtenerse a un coste razonable comparado con el valor de la transacción.

Una política de seguridad sensata para vendedores y compradores de Internet exige los siguientes requisitos par asegurar las compras web:

1. Autenticación del vendedor al comprador, de modo que el comprador pueda confiar en que está en contacto con un servidor operado por el vendedor con el que se supone que tiene que tratar.
2. Mantenimiento del número de tarjeta de crédito y otros detalles del comprador bajo secreto, para evitar que caigan en manos de terceras partes y asegurar que se transmiten de forma inalterada del comprador al vendedor.
3. Si los bienes se encuentran en una forma útil para su descarga, asegurar que su contenido llega al comprador sin alteración y sin ser desvelados a terceras partes.

La identidad del comprador no suele ser un requisito del vendedor (excepto por necesidades de envío de los bienes en el caso en que no puedan descargarse electrónicamente). El vendedor deseará comprobar que el comprador tiene los fondos necesarios para pagar la compra, aunque esto se realiza usualmente requiriendo el pago al banco del comprador antes de enviar los bienes.

Las necesidades de seguridad de las transacciones bancarias que emplean una red abierta son similares a las de las transacciones de compra, con el titular de la cuenta y el banco como vendedor, aunque aquí ciertamente *hay* necesidad de:

4. Autenticar la identidad del titular de la cuenta hacia el banco antes de darle acceso a su cuenta.

Observe que en esta situación es importante para el banco estar seguro de que el titular de la cuenta no pueda negar haber participado en una transacción. A este requisito se le da el nombre de *no repudio*.

Además de los requisitos anteriores, dictados por políticas de seguridad, hay otros requisitos de sistema. Éstos surgen de la gran escala de Internet, que hace impracticable el requerir que los compradores inicien algún tipo de relación especial con los vendedores (por ejemplo, registrando claves de encriptación para un posterior uso, etc.). Debiera ser posible para un comprador completar una transacción segura con un vendedor incluso si no ha habido contacto previo entre el comprador y el vendedor, y sin la participación de una tercera parte. Técnicas tales como el uso de «*cookies*» (registros de transacciones previas almacenadas en el host del cliente del usuario) tienen ciertas debilidades obvias; los dispositivos de sobremesa y los host móviles suelen estar situados en entornos físicos inseguros.

A causa de la importancia de la seguridad para el comercio en Internet y el rápido crecimiento de éste, hemos elegido, para ilustrar el uso de las técnicas de seguridad criptográfica que se describirán en la Sección 7.6, el protocolo de seguridad estándar *de facto* empleado en la mayoría del comercio electrónico (*Secure Sockets Layer*, SSL), y Millicent, un protocolo diseñado específicamente para micro-transacciones.

El comercio en Internet es una aplicación importante de las técnicas de seguridad, pero no es ciertamente la única. Es una necesidad cuando quiera que dos computadores sean utilizados por individuos u organizaciones para almacenar y comunicar información importante. La utilización de correo electrónico encriptado para la comunicación privada entre individuos es un caso concreto que ha sido objeto de considerable discusión política. Nos referiremos a este debate en la Sección 7.5.2.

7.1.3. DISEÑO DE SISTEMAS SEGUROS

En los últimos años se han dado pasos inmensos en el desarrollo y la aplicación de las técnicas criptográficas, aunque el diseño de sistemas seguros siga siendo una tarea inherentemente difícil. En el corazón de este dilema está el hecho de que el objetivo del diseñador es excluir *todos* los posibles ataques y agujeros. La situación es análoga a la del programador cuyo principal objetivo es excluir todos los errores de su programa. En ningún caso existe un método concreto para asegurar las metas durante el diseño. Cada uno diseña con los mejores estándares disponibles y aplica un análisis informal y comprobaciones. Una vez que un diseño está completo, una opción es la validación formal. El trabajo en la validación formal de protocolos de seguridad ha producido algunos resultados importantes [Lampson y otros 1992, Schneider 1996, Abadi y Gordon 1999]. Una descripción de uno de los primeros pasos en esta dirección, la lógica de autenticación BAN [Burrows y otros 1990] y su aplicación puede encontrarse en ww.cdk3.net/security.

La seguridad trata de evitar los desastres y minimizar los contratiempos. Cuando se diseña para seguridad es necesario pensar siempre en lo peor. El recuadro de las páginas 243 y 244 muestra un conjunto de premisas y guías de diseño útiles. Estas premisas penetran las ideas subyacentes a cada técnica que se describe en este capítulo.

Para demostrar la validez de los mecanismos de seguridad empleados en un sistema, los diseñadores deben construir, en primer lugar, una lista de amenazas (métodos por los cuales se puede violar una política de seguridad) y probar que cada una de ellas se puede prevenir mediante los mecanismos empleados. Esta demostración puede tomar la forma de un argumento informal, o mejor aún, de una demostración lógica.

Es lógico pensar que ninguna lista de amenazas sea completamente exhaustiva, de modo que en el caso de aplicaciones sensibles a la seguridad deberán emplearse métodos de auditoría. Su aplicación es directamente implementable bajo la forma de un histórico de aquellas acciones del sistema sensibles a la seguridad, guardando siempre los detalles relativos a los usuarios, y la autoridad de que disponen, que realizan tales acciones.

Un histórico de seguridad contendrá una secuencia de registros fechados de las acciones de los usuarios. Como mínimo, los registros incluirán la identidad del principal, la operación realizada (por ejemplo, borrar un archivo, actualizar un registro de contabilidad), la identidad del objeto sobre el que se opera y la fecha y hora. Donde se sospeche que pudiera haber violaciones concretas, los registros pueden contener información a mayores para incluir la utilización de los recursos físicos (ancho de banda de red, periféricos), o disparar un procedimiento histórico especial de operaciones sobre objetos concretos. Posteriormente se puede efectuar un análisis de carácter estadístico o bien basado en búsquedas. Incluso aunque no se sospeche de alguna violación, las técnicas estadísticas permitirán comparar registros a lo largo del tiempo para descubrir tendencias o cualquier suceso inusual.

El diseño de sistemas seguros es un ejercicio de balance entre los costes y las amenazas. El abanico de técnicas útiles para desplegar procesos de protección y asegurar la comunicación entre procesos es suficientemente potente como para resistir cualquier ataque, aunque su utilización trae consigo ciertos costes e inconveniencias:

- Su uso acarrea un coste (en esfuerzo computacional y uso de la red). Los costes deben compensar la amenaza.
- Unas especificaciones de medidas de seguridad inapropiadas podrían impedir a los usuarios legítimos el realizar ciertas acciones necesarias.

Tales inconvenientes son difíciles de identificar sin comprometer la seguridad y puede parecer que entran en conflicto con el consejo del primer párrafo de esta subsección, pero podemos cuantificar la fuerza de las técnicas de seguridad y seleccionarlas basándonos en una estimación del coste de un ataque. El coste relativamente bajo de las técnicas empleadas en el protocolo Millicent para pequeñas transacciones comerciales, descrito en la Sección 7.6.4, proporciona un ejemplo de ello.

Premisas del peor caso posible y guías de diseño

Las interfaces están desprotegidas: Los sistemas distribuidos se componen de procesos que ofrecen servicios que comparten información. Sus interfaces de comunicación son necesariamente abiertas (para permitir el acceso a nuevos clientes); un atacante puede enviar un mensaje a cualquier interfaz.

Las redes son inseguras: Por ejemplo, puede falsificarse la fuente de cualquier mensaje; pudiera parecer que ciertos mensajes provienen de Alice cuando en realidad fueron generados por Mallory. Se puede suplantar la dirección de cualquier host; Mallory puede conectarse a la red con la misma dirección de Alice y recibir copias de mensajes pretendidamente enviados para ésta.

Limítese el tiempo de vida y el alcance de cada secreto: Cuando se genera por primera vez una clave secreta podemos confiar en que no se encuentra comprometida. Cuando más la usemos y más ampliamente se conozca, mayor será el riesgo. La utilización de secretos como las contraseñas y las claves secretas compartidas deberían tener una caducidad, y un alcance de uso restringido.

Los algoritmos y el código de los programas están disponibles para los atacantes: Cuanto mayor y más ampliamente se difunde un secreto, mayor es el riesgo de que se descubra. Los algoritmos de encriptación secreta son totalmente inadecuados para los entornos de red de gran escala de hoy en día. La mejor práctica es publicar los algoritmos empleados para la encriptación y la autentificación, confiándose solamente en el secreto de las claves criptográficas. Abriendo los algoritmos al público el escrutinio de terceras partes nos ayuda a garantizar el que éstos sean suficientemente fuertes.

Los atacantes tienen acceso a suficientes recursos: El coste de la potencia de cálculo decrece con rapidez. Deberíamos presuponer que los atacantes tendrán acceso a los computadores más grandes y más potentes que se puedan proyectar durante la vida útil de un sistema, y aun así añadir unos cuantos órdenes de magnitud más para contemplar desarrollos inesperados.

Minimícese la base de confianza: Las porciones de un sistema responsable de la implementación de su seguridad, *y todos los componentes hardware y software sobre los que descanse*, deben ser confiables; a esto se le denomina *base de computación confiable*. Cualquier defecto o error de programación en esta base de confianza puede producir debilidad en el sistema, de modo que debiéramos tender a minimizar su tamaño. Por ejemplo, los programas de aplicación de usuario no debieran ser dignos de confianza para proteger los datos de sus usuarios.

7.2. VISIÓN GENERAL DE LAS TÉCNICAS DE SEGURIDAD

El objetivo de esta sección es presentar al lector algunas de las técnicas y mecanismos más importantes para asegurar sistemas y aplicaciones distribuidas. Aquí se las describe informalmente, reservando otra descripción más rigurosa para las Secciones 7.3 y 7.4. Emplearemos los nombres ya familiares para los principales que se presentaron en la Figura 7.2 y la notación para los elementos encriptados y firmados que se muestra en la Figura 7.3.

7.2.1. CRIPTOGRAFÍA

La encriptación es el proceso de codificación de un mensaje de forma que queden ocultos sus contenidos. La criptografía moderna incluye algunos algoritmos seguros de encriptación y desencriptación de mensajes. Todos ellos se basan en el uso de ciertos secretos llamados *claves*. Una clave criptográfica es un parámetro empleado en un algoritmo de encriptación de forma que la encriptación no sea reversible sin el conocimiento de una clave.

Hay dos clases principales de algoritmos de encriptación de uso general. La primera emplea *claves secretas compartidas*; el emisor y el receptor deben compartir el conocimiento de una clave y ésta no debe ser revelada a ningún otro. La segunda clase de algoritmos de encriptación emplea

K_A	Clave secreta de Alice
K_B	Clave secreta de Bob
K_{AB}	Clave secreta compartida por Alice y Bob
K_{Apriv}	Clave privada de Alice (sólo conocida por Alice)
K_{Apub}	Clave pública de Alice (publicada por Alice para lectura de cualquiera)
$\{M\}_K$	Mensaje *M* encriptado con la clave *K*
$[M]_K$	Mensaje *M* firmado con la clave *K*

Figura 7.3. Notación criptográfica.

pares de claves pública/privada; el emisor de un mensaje emplea una *clave pública*, difundida previamente por el receptor, para encriptar el mensaje. El receptor emplea la *clave privada* correspondiente para desencriptar el mensaje. A pesar de que una multitud de principales pudiera examinar la clave pública, solamente el receptor puede desencriptar el mensaje, gracias a su clave privada.

Ambas clases de algoritmo de encriptación son extremadamente útiles y ampliamente utilizados para la construcción de sistemas distribuidos seguros. Los algoritmos de encriptación de clave pública requieren usualmente de 100 a 1.000 veces más potencia de procesamiento que los algoritmos de clave secreta, aunque hay situaciones en las que su conveniencia compensa esta desventaja.

7.2.2. USOS DE LA CRIPTOGRAFÍA

La criptografía juega tres papeles principales en la implementación de los sistemas seguros. Aquí hacemos un bosquejo de ellas a través de escenarios simplificados. En las secciones posteriores de este capítulo, las describimos con mayor detalle junto a otros protocolos, desglosando también algunos problemas aún no resueltos que aquí solamente se reseñan.

En todos nuestros escenarios que veremos más adelante, podemos presuponer que Alice, Bob y cualquier otro participante se han puesto previamente de acuerdo sobre los algoritmos de encriptación que desean utilizar y además tienen implementaciones de ellos. También partimos de que cualquier clave de que dispongan, ya sea secreta o privada, puede almacenarse de forma segura para prevenir que los atacantes las obtengan.

◇ **Secreto e integridad.** La criptografía se emplea para mantener el secreto y la integridad de la información dondequiera que pueda estar expuesta a ataques potenciales, por ejemplo durante la transmisión a través de redes vulnerables a la indiscreción y a la manipulación de mensajes. Este uso de la criptografía se corresponde con su papel tradicional en las actividades militares y de inteligencia. Se explota el hecho de que un mensaje encriptado con una clave de encriptación particular sólo puede ser desencriptado por un receptor que conozca la correspondiente clave de desencriptación. Así se conseguirá el secreto del mensaje encriptado en la medida de que la clave de desencriptado no esté *comprometida* (desvelada a los que no intervienen en la comunicación) y a condición de que el algoritmo de encriptación sea suficientemente fuerte como para derrotar cualquier posible intento de romperlo. La encriptación también preserva la integridad de la información encriptada, supuesto que se incluya algo de información redundante, de la misma forma en que se incluyen y comprueban las sumas de chequeo (*checksums*).

Escenario 1. Comunicación secreta con clave secreta compartida: Alice desea enviar cierta información a Bob de modo secreto. Alice y Bob comparten una clave secreta K_{AB}.

1. Alice emplea K_{AB} y acuerda una función de encriptación $E(K_{AB}, M)$ para encriptar y enviar cualquier cantidad de mensajes $\{M_i\}_{K_{AB}}$ para Bob. Alice puede continuar utilizando K_{AB} tanto como sea seguro suponer que K_{AB} no ha sido comprometida.
2. Bob lee los mensajes encriptados con la correspondiente función de desencriptado $D(K_{AB}, M)$.

Bob podrá ahora leer el mensaje original *M*. Si el mensaje tiene sentido cuando es desencriptado por Bob, o mejor, si incluye algún valor acordado entre Alice y Bob, tal como una suma de comprobación del mensaje, entonces Bob conoce con certeza que el mensaje proviene de Alice y no ha sido saboteado. Pero aún hay otros problemas:

Problema 1: ¿Cómo puede enviar Alice a Bob una clave compartida K_{AB} de modo seguro?

Problema 2: ¿Cómo sabe Bob que cualquier $\{M_i\}$ no es nunca copia de un anterior mensaje encriptado de Alice, y que fue capturado por Mallory para ser reenviado más tarde? Mallory no necesita tener la clave K_{AB} para llevar a cabo este ataque; no hay más que hacer una simple

copia del patrón de bits que representa el mensaje y enviarlo a Bob más tarde. Por ejemplo, si el mensaje es una solicitud para hacer cierto pago a alguien, Mallory podría engañar a Bob para que pagara dos veces.

Más adelante, en este capítulo, veremos cómo se pueden resolver estos problemas.

◇ **Autenticación.** La criptografía se emplea como base para los mecanismos para autenticar la comunicación entre pares de principales. Un principal que desencripta un mensaje con éxito empleando una clave particular puede presuponer que el mensaje es auténtico si contiene una suma de chequeo correcta o, si se emplea el modo de encriptación de encadenamiento de bloques, que se describe en la Sección 7.3, cualquier otro valor esperado. Éstos podrán inferir que el emisor del mensaje estaba en posesión de la clave de encriptación correspondiente y entonces deducir la identidad del emisor, siempre que la clave sólo sea conocida por las dos partes. En resumen, si las claves se mantienen en secreto, una desencriptación con éxito da fe de que el mensaje desencriptado proviene de un emisor concreto.

Escenario 2. Comunicación autenticada con un servidor: Alice desea acceder a los archivos guardados por Bob, en un servidor de archivos de la red local de la organización donde ella trabaja. Sara es un servidor de autenticación administrado de modo seguro. Sara administra usuarios con su clave de acceso y almacena las claves secretas actuales de todos los principales en el sistema que sirve (generadas aplicando alguna transformación a la clave de acceso del usuario). Por ejemplo, conoce la clave K_A de Alice y la clave K_B de Bob. En nuestro escenario hacemos uso de *tickets*. Un ticket es un elemento encriptado emitido por un servidor de autenticación, que contiene la identidad del principal a quien se dirige y una clave compartida que ha sido generada por la actual sesión de comunicación.

1. Alice envía un mensaje (no encriptado) a Sara indicando su identidad y reclamando un ticket para acceder a Bob.
2. Sara responde a Alice con un mensaje encriptado con K_A que consta de un ticket (para enviar a Bob con cada petición de acceso a archivos) encriptado con K_B y una nueva clave secreta K_{AB} para su uso en la comunicación con Bob. Así la respuesta que recibe Alice se parece a ésta: $\{\{Ticket\}_{K_B}, K_{AB}\}_{K_A}$.
3. Alice desencripta la respuesta utilizando K_A (que genera con su palabra de acceso empleando la misma transformación; la palabra clave no se transmite en la red y una vez que se ha empleado, se borra del almacenamiento local para evitar ponerla en peligro). Si Alice tiene la clave correcta derivada de la palabra de acceso, K_A, obtiene un ticket válido para utilizar el servicio de Bob, más una nueva clave de encriptación para comunicarse con Bob. Alice no puede desencriptar o manipular el ticket, porque está encriptado con K_B. Si el receptor no es Alice entonces no conocerá la clave de Alice, de modo que no será capaz de desencriptar el mensaje.
4. Alice envía el ticket a Bob junto con su identidad y una solicitud R para acceder a un archivo: $\{Ticket\}_{K_B}$, *Alice*, R.
5. El ticket, originalmente creado por Sara, es en realidad $\{K_{AB}, Alice\}_{K_B}$. Bob desencripta el ticket empleando su clave K_B. Así Bob obtiene la identidad auténtica de Alice (basada en el conocimiento compartido entre Alice y Sara acerca de la clave de Alice) y una nueva clave secreta compartida K_{AB} que le permite interaccionar con Alice. (A ésta se le denomina *clave de sesión* porque puede utilizarse de modo seguro entre Alice y Bob para una serie de interacciones.)

Este escenario es una versión simplificada del protocolo de autenticación desarrollado originalmente por Roger Needham y Michael Schroeder [1978] y utilizado posteriormente en el sistema Kerberos desarrollado y utilizado en el MIT [Steiner y otros, 1988], y que se describirá en la Sección 7.6.2. En nuestra descripción simplificada de este protocolo no existe protección contra la repetición de mensajes de autenticación antiguos. Ésta y alguna otra debilidad se tratan en nuestra descripción del protocolo Needham-Schroeder completo, que se describirá en la Sección 7.6.1.

El protocolo de autenticación que hemos descrito depende del conocimiento previo por el servidor de autenticación Sara de las claves de Alice y Bob, K_A y K_B. Esto es realizable en una organización única donde Sara emplea un computador físicamente seguro y está administrado por un principal fiable que genera valores iniciales para las claves y las transmite a los usuarios mediante un canal seguro independiente. Pero este esquema no es apropiado para el comercio electrónico u otras aplicaciones de área extensa, donde el emplear un canal independiente es extremadamente inconveniente y los requisitos para una tercera parte digna de crédito es poco realista. La criptografía de clave pública nos rescata de este dilema.

Utilidad de los desafíos: Un aspecto importante del avance de 1978 de Needham y Schroeder fue el darse cuenta de que no hay por qué enviar una contraseña de usuario a un servicio de autenticación (y ser expuesta a la red) cada vez que se auténtica. En su lugar, presentaron el concepto de *desafío* criptográfico. Como se puede ver en el paso 2, donde el servidor Sara emite un ticket a Alice *encriptado en la clave secreta de Alice*, K_A. Esto conforma un desafío porque Alice no podrá utilizar el ticket a menos que pueda desencriptarlo, y sólo puede hacerlo si puede conseguir K_A, que se deriva de la clave de acceso de Alice. Un impostor que afirmara ser Alice sería vencido en este momento.

Escenario 3. Comunicación autenticada con claves públicas: Supongamos que Bob ha generado un par de claves pública/privada, así el siguiente diálogo les permite a Bob y Alice establecer una clave secreta compartida K_{AB}:

1. Alice accede a un servicio de distribución de claves para obtener un *certificado de clave pública*, dada la clave pública de Bob. Se denomina certificado porque viene firmado por una autoridad en quien se puede confiar; una persona u organización de la que se sabe bien que es de fiar. Tras comprobar la firma, ella lee la clave pública de Bob K_{Bpub} a partir del certificado. (Discutiremos la construcción y utilización de los certificados de clave pública en la Sección 7.2.3.)
2. Alice crea una nueva clave compartida K_{AB} y la encripta usando K_{Bpub} con un algoritmo de clave pública. Entonces envía el resultado a Bob, junto con un nombre clave que identifica de modo único un par de claves pública/privada (dado que Bob puede tener varias). Así Alice envía a Bob: *nombreClave*, $\{K_{AB}\}_{K_{Bpub}}$.
3. Bob selecciona la clave privada correspondiente K_{Bpriv} de su almacén de claves privadas y la emplea para desencriptar K_{AB}. Observe que el mensaje de Alice a Bob pudiera haber sido corrompido o saboteado mientras estaba en tránsito. La consecuencia seguramente sería que Bob y Alice no compartirían la misma clave K_{AB}. Si esto fuera un problema, podría soslayarse añadiendo un valor acordado o un texto al mensaje, tales como los nombres o direcciones de correo de Bob y Alice, con lo cual Bob podría comprobarlo tras desencriptarlo.

El escenario anterior muestra el uso de la criptografía de clave privada para distribuir una clave compartida. Esta técnica se conoce como *protocolo criptográfico híbrido* y se utiliza ampliamente, dado que explota características útiles de ambos algoritmos de encriptación, de clave pública y de clave privada.

Problema: este intercambio de claves es vulnerable a ataque del *hombre entre medias*. Mallory podría interceptar la petición inicial de Alice al servicio de distribución de claves buscando el certificado de clave pública de Bob y enviar una respuesta que contenga su propia clave pública. Él puede, en ese momento, interceptar todos los mensajes subsiguientes. En nuestra descripción de arriba, se impide este ataque obligando que el certificado de Bob esté firmado por una autoridad bien conocida. Para protegerse de este ataque, Alice debe asegurarse de que el certifi-

1.	*Tipo de certificado*:	Número de cuenta
2.	*Nombre*:	Alice
3.	*Cuenta*:	6262626
4.	*Autoridad certificadora*:	Banco de Bob
5.	*Firma*:	$\{Resumen(campo\ 2 + campo\ 3)\}_{K_{Bpriv}}$

Figura 7.4. Certificado de cuenta bancaria de Alice.

cado de clave pública está firmado con una clave pública (como se describe más arriba) que haya sido recibido por ella de forma totalmente segura.

◇ **Firmas digitales.** La criptografía se emplea para implementar un mecanismo conocido como *firma digital*. Ésta emula el papel de las firmas convencionales, verificando a una tercera parte que un mensaje o un documento es una copia inalterada producida por el firmante.

Las técnicas de firmas digitales se basan en una modificación irreversible sobre el mensaje o el documento de un secreto que únicamente conoce el firmante. Esto se puede lograr encriptando el mensaje; o mejor, una forma comprimida del mensaje denominada *resumen* (*digest*), empleando una clave sólo conocida por el firmante. Un resumen es un valor de longitud fija calculado mediante la aplicación de una *función de resumen segura*. Un resumen seguro es similar a una función de chequeo, con la diferencia de que es muy difícil que dos mensajes diferentes produzcan el mismo resumen. El resumen resultante encriptado actúa como una firma que acompaña el mensaje. La firma se suele encriptar mediante un método de clave pública: el firmante genera una firma con su clave privada; la firma puede ser desencriptada por el receptor utilizando la correspondiente clave pública. Hay un requerimiento adicional: el verificador de la firma debe asegurarse de que la clave pública que emplea es la que realmente pretende el firmante; esta última se distribuye mediante el uso de certificados de clave pública, descritos en la Sección 7.2.3.

Escenario 4. Firmas digitales con una función de resumen segura: Alice quiere firmar un documento M de modo que cualquier receptor pueda verificar que ella es la creadora. Así cuando Bob accede posteriormente a este documento firmado, tras recibirlo por cualquier medio (por ejemplo, podría ser enviado en un mesaje o recuperado desde una base de datos), él puede verificar que su creadora es Alice.

1. Alice calcula un resumen de longitud fija a partir del documento: $Resumen(M)$.
2. Alice encripta el resumen con su clave privada, añadiéndole a M y haciendo el resultado M, $\{Resumen(M)\}_{K_{Apriv}}$ público a los usuarios finales.
3. Bob obtiene el mensaje firmado, extrae M y calcula $Resumen(M)$.
4. Bob desencripta $\{Resumen(M)\}_{K_{Apriv}}$ utilizando la clave pública de Alice K_{Apub} y compara el resultado con el suyo $Resumen(M)$. Si concuerdan, la firma es válida.

7.2.3. CERTIFICADOS

Un certificado digital es un documento que contiene una sentencia (generalmente corta) firmada por un principal. Ilustremos este concepto con un escenario.

Escenario 5. Empleo de certificados: Bob es un banco. Cuando sus clientes establecen contacto con él necesitan estar seguros de estar hablando con Bob el banco, incluso si nunca han contactado con él antes. Bob necesita autenticar sus clientes antes de darles acceso a sus cuentas.

Por ejemplo, Alice podría encontrar útil obtener un certificado de su banco atestiguando su número de cuenta bancario (véase la Figura 7.4). Alice podría utilizar este certificado cuando compra

1.	*Tipo de certificado*:	Clave pública
2.	*Nombre*:	Banco de Bob
3.	*Cuenta*:	K_{Bpub}
4.	*Autoridad certificadora*:	Fred, la Federación de Banqueros
5.	*Firma*:	$\{Resumen(campo\ 2 + campo\ 3)\}_{K_{Fpriv}}$

Figura 7.5. Certificado de clave pública para el Banco de Bob.

para certificar que tiene una cuenta con el Banco de Bob. El certificado se firma con la clave privada de Bob K_{Bpriv}. Un vendedor Carol puede aceptar tal certificado para cargar compras a la cuenta de Alice siempre que ella pueda validar la firma del campo 5. Para ello, Carol necesita la clave pública de Bob y también estar segura de su autenticidad para estar a cubierto de la posibilidad de que Alice pueda firmar un certificado falso asociando su nombre con la cuenta de algún otro. Para cometer este ataque, Alice simplemente generaría un nuevo par de claves $K_{B'pub}$, $K_{B'priv}$ y los utilizaría para generar un certificado falso pretendiendo que proviene del Banco de Bob.

Lo que Carol necesita es un certificado donde figure la clave pública de Bob, firmado por una autoridad bien conocida y de fiar. Supongamos que Fred representa a la Federación de Banqueros, uno de cuyos papeles es certificar las claves públicas de los bancos. Entonces Fred emitiría un *certificado de clave pública* para Bob (véase la Figura 7.5).

Como es lógico, este certificado depende de la autenticidad de la clave pública de Fred K_{Fpub}, de modo que tenemos un problema recursivo de autenticidad; Carol sólo puede fiarse de este certificado si está segura de que conoce la auténtica clave pública de Fred K_{Fpub}. Podemos romper esta recursión asegurándonos que Carol obtiene K_{Fpub} mediante alguna forma en que ella confíe; esta clave podría haberse entregado en mano por un representante de Fred o haber recibido una copia firmada por éste de alguien en quien ella confíe y que le asegure que lo recibió directamente de Fred. Nuestro ejemplo muestra una cadena de certificados; en este caso, una con dos enlaces.

Ya hemos hecho alusión a uno de los problemas que aparecen con los certificados: la dificultad al elegir una autoridad fiable de donde pueda arrancar una cadena de autenticaciones. La confianza raramente es absoluta, de modo que la elección de una autoridad dependerá del objetivo para el que se necesite el certificado. Otros problemas nacen del riesgo de comprometer (desvelar) las claves privadas y de la longitud permisible de una cadena de certificación; cuanto más larga, mayor es el riesgo de un eslabón débil.

Supuesto que se haya tenido cuidado con estas cuestiones, ocurre que las cadenas de certificados son una piedra angular importante para el comercio electrónico y de otro tipo de transacciones del mundo real. Ayudan a abordar el problema de la escala: hay seis mil millones de personas en el mundo, de modo que cómo podemos construir un entorno electrónico en el que podamos establecer las credenciales de cualquiera de ellas.

Los certificados pueden emplearse para establecer la autenticidad de muchos tipos de enunciados. Por ejemplo, los miembros de un grupo o asociación desean mantener una lista de correo abierta tan sólo para los miembros del grupo. Una buena forma que el administrador de socios (Bob) pueda llevarlo a cabo es emitir un certificado de miembro (S, Bob, $\{Resumen(S)\}_{K_{Bpriv}}$) para cada miembro, donde S es un enunciado de la forma Alice es un miembro de la Sociedad de Amigos y K_{Bpriv} es la clave privada de Bob. Un miembro (Alice) que quiera pertenecer a la lista de correo de la Sociedad de Amigos tendrá que aportar una copia de su certificado al sistema de administración de la lista, que comprobará el certificado antes de permitir que Alice se una a la lista.

Para que los certificados sean útiles se requieren dos cosas:

- Un formato estándar y una representación para ellos de modo que los emisores de certificados y los usuarios de certificados puedan construirlos e interpretarlos.

- Un acuerdo sobre la forma en que se construyen las cadenas de certificados, y en particular la noción de autoridad fiable.

Volveremos a estos requisitos en la Sección 7.4.4.

A veces se requiere revocar un certificado; por ejemplo, Alice podría interrumpir su asociación con la Sociedad de Amigos, pero ella y otros pudieran seguir almacenando copias de su certificado de pertenencia. Sería caro o incluso imposible seguir la pista y eliminar todos esos certificados, y no es fácil invalidar un certificado, podría ser necesario notificar a todos los posibles receptores de un certificado revocado. La solución habitual a este problema es incluir una fecha de expiración en el certificado. Cualquiera que reciba un certificado expirado debería rechazarlo, y el sujeto del certificado debería pedir su renovación. Si se requiere una revocación más rápida, habrá que utilizar alguno de los engorrosos mecanismos anteriormente citados.

7.2.4. CONTROL DE ACCESO

Aquí se delinean los conceptos sobre los que se basa el control de acceso a los recursos en los sistemas distribuidos y las técnicas mediante las que se implementa. La base conceptual para la protección y el control de acceso fue establecida en términos muy precisos en un clásico artículo de Lampson [1971], y los detalles de las implementaciones no distribuidas pueden encontrarse en muchos libros sobre sistemas operativos [Stallings 1998b].

Históricamente, la protección de los recursos de los sistemas distribuidos se implanta mayormente en el servicio concreto que queremos proteger. Los servidores reciben mensajes con peticiones de la forma <*op*, *principal*, *recurso*>, donde *op* es la operación solicitada, *principal* es una identidad o un conjunto de credenciales del principal que realiza la petición y *recurso* identifica el recurso sobre el que se aplica la operación. El servidor debe, en primer lugar, comprobar la autenticidad del mensaje de petición y las credenciales del principal y después aplicar el control de acceso, rehusando cualquier petición para la cual el principal solicitante no tenga los derechos de acceso pertinentes para realizar la operación requerida sobre el recurso especificado.

En sistemas distribuidos orientados al objeto puede haber muchos tipos de objeto a los cuales habrá que aplicar el control de acceso, y las decisiones son siempre específicas para cada aplicación. Por ejemplo, puede que a Alice sólo se le permita extender un cheque de su cuenta cada día, mientras que a Bob se le admitan tres. Las decisiones de control de acceso se dejan usualmente al código de nivel de aplicación, pero se proporciona un soporte genérico para gran parte de la mecánica que soporta las decisiones. Esto incluye la autenticación de los principales, la firma y autenticación de las peticiones, y la administración de credenciales y datos de derechos de acceso.

◇ **Dominios de protección.** Un dominio de protección es un entorno de ejecución compartido por un conjunto de procesos: contiene un conjunto de pares <*recurso*, *derechos*>, que listan los recursos que pueden ser accedidos por todos los procesos en ejecución dentro del dominio y especificando las operaciones permitidas sobre cada recurso. Un dominio de protección se asocia generalmente con un principal dado; cuando un usuario entra en el sistema, se comprueba su identidad y se crea un dominio de protección para todos los derechos de acceso que posee el principal. Conceptualmente, el dominio incluye todos los derechos que posea el principal, incluyendo cualquier derecho que ella adquiera por el hecho de pertenecer a varios grupos. Por ejemplo, en UNIX, el dominio de protección de un proceso se determina mediante los identificadores de usuario y grupo ligados al proceso en el momento de entrada al sistema. Los derechos se especifican en términos de operaciones permitidas. Por ejemplo, un archivo podría ser legible y sobre-escribible por un proceso y sólo legible por otro.

Un dominio de protección es sólo una abstracción. En los sistemas distribuidos se emplean usualmente dos implementaciones alternativas.

Habilitaciones (*capabilities*): Cada proceso, según el dominio en que esté ubicado, aloja un conjunto de habilitaciones. Una habilitación es un valor binario que actúa como una clave de acceso permitiendo al que lo posee acceder a ciertas operaciones sobre un recurso especificado. Para su uso en los sistemas distribuidos, donde las habilitaciones deben ser infalsificables, toman la forma de:

Identificador del recurso	Un identificador único para el recurso objeto
Operaciones	Una lista de operaciones permitidas sobre el recurso
Código de autenticación	Una firma digital que hace la habilitación infalsificable

Los servicios sólo proporcionan habilitaciones a los clientes cuando se les ha autentificado como pertenecientes al dominio de protección pretendido. La lista de operaciones en cada habilitación es un subconjunto de operaciones definidas para el recurso objetivo y siempre se codifica como un mapa de bits. Para las diferentes combinaciones de derechos de acceso al mismo recurso se emplean habilitaciones diferentes.

Cuando se emplean habilitaciones, las peticiones de los clientes tienen la forma <*op*, *idUsuario*, *habilitación*>. Esto es, se incluye en ellas una habilitación para acceder al recurso en lugar de un simple identificador, lo que da al servidor una prueba inmediata de que el cliente está autorizado a acceder al recurso identificado por la habilitación con las operaciones especificadas por ésta. Una comprobación de control de acceso sobre una petición acompañada por una habilitación involucra solo la validación de la habilitación y una comprobación de que la operación solicitada está en el conjunto permitido por la habilitación. Esta característica es la mayor ventaja de las habilitaciones, que constituyen una clave de acceso autocontenida, tal y como una llave física para una cerradura es una clave para acceder al edificio protegido por la cerradura.

Las habilitaciones comparten dos inconvenientes de las llaves de una cerradura física:

Robo de llaves: cualquiera que posea la llave de un edificio puede utilizarla para tener acceso a él, sea o no un poseedor autorizado de la llave; ésta pudiera haber sido robada u obtenida de alguna forma fraudulenta.

El problema de la revocación: la cualificación para poseer una llave cambia con el tiempo. Por ejemplo, el poseedor pudiera dejar de ser un empleado o el propietario del edificio, pero podría seguir reteniendo la llave, o una copia de ella, y utilizarla de manera no autorizada.

Las únicas soluciones para estos problemas para las llaves físicas son (1) encerrar en la cárcel al poseedor ilícito de la llave (no siempre posible en el intervalo de tiempo que pudiera prevenir que efectuara algún daño), o (2) cambiar la cerradura y reexpedir llaves a todos los portadores de la llave (una operación torpe y onerosa).

Quedan claros los problemas análogos para las habilitaciones:

- Las habilitaciones podrían, debido a un descuido o como resultado de un ataque a la confidencialidad, caer en manos de otros principales que no son para quienes fueron emitidas. Si ocurre esto, los servidores quedan a merced de ser utilizados ilícitamente.
- Resulta difícil cancelar las habilitaciones. El estado del portador pudiera cambiar y sus derechos de acceso debieran cambiar conforme a ello, pero aún pueden utilizar la habilitación antigua.

Se han propuesto y desarrollado soluciones a ambos problemas, basadas tanto en la inclusión de información que identificaría al portador, como en tiempos de caducidad más listas de habilitaciones revocadas [Gong 1989, Hayton y otros 1998]. A pesar de que añaden complejidad, a un concepto bastante simple de otro modo, las habilitaciones siguen siendo una técnica importante, por ejemplo, pueden usarse en conjunción con las listas de control de acceso para optimizar el control

de acceso en accesos repetidos al mismo recurso, y además proporcionan el mecanismo más limpio para la implementación de la delegación (véase la Sección 7.2.5).

Es interesante observar la similitud entre las habilitaciones y los certificados. Considere el certificado de propiedad de Alice de su cuenta bancaria, presentado en la Sección 7.2.3. Difiere de las habilitaciones, tal y como aquí se describen, sólo en que no hay una lista de operaciones permitidas y en que se identifica al emisor. Los certificados y las habilitaciones pueden ser conceptos intercambiables bajo ciertas circunstancias. El certificado de Alice puede contemplarse como una clave de acceso para realizar todas las operaciones sobre la cuenta bancaria de Alice que se permiten a los titulares de las cuentas, dando por sentado que pueda probarse que el solicitante sea Alice.

Listas de control de acceso: Se almacena una lista con cada recurso, con una entrada de la forma <*dominio*, *operaciones*> para cada dominio que tenga acceso al recurso y especificando las operaciones permitidas al dominio. Un dominio puede especificarse mediante un identificador de un principal o puede ser una expresión susceptible de ser utilizada para determinar la pertenencia del principal al dominio. Por ejemplo, *el propietario de este archivo* es una expresión que puede evaluarse comparando la identidad del principal solicitante con la identidad del propietario almacenada con el archivo.

Éste es el esquema adoptado en la mayoría de los sistemas de archivos, incluyendo UNIX y Windows NT, donde se asocia a cada archivo un conjunto de bits de permisos de acceso, y los dominios a los cuales se ceden los permisos se definen por medio de una referencia a la información de pertenencia almacenada con cada archivo.

Las peticiones a los servidores son de la forma <*op*, *principal*, *recurso*>. Para cada petición, el servidor identifica el principal y comprueba si la operación solicitada está incluida en la entrada del principal en la lista de control de acceso del recurso relevante.

◇ **Implementación.** Las firmas digitales, las credenciales y los certificados de clave pública proporcionan las bases criptográficas para el control de acceso seguro. Los canales seguros ofrecen beneficios de prestaciones, posibilitando el manejo de múltiples peticiones sin necesidad de comprobar los principales y las credenciales reiteradamente [Wobber y otros 1994].

Ambos CORBA y Java ofrecen un API de seguridad. Uno de sus principales objetivos es dar soporte para el control de acceso. Java proporciona soporte para que los objetos distribuidos administren su propio control de acceso mediante las clases Principal, Signer y ACL, así como métodos de autenticación por defecto y soporte para certificados, validación de firmas y comprobaciones de control de acceso. También están soportadas la criptografía de clave secreta y pública. Farley [1998] presenta una buena introducción a estas características de Java. La protección de los programas Java que incluyen código móvil se basa en el concepto de dominio de protección; el código local y el código descargado tienen diferentes dominios de protección en que ejecutarse. Puede haber un dominio de protección para cada código fuente descargado, con derechos de acceso para diferentes conjuntos de recursos locales dependiendo del nivel de confianza, el cual se indica en el código descargado.

CORBA ofrece una especificación de Servicio de Seguridad [Blakley 1999, OMG 1998b] con un modelo de ORB para proporcionar comunicación, autenticación, control de acceso con credenciales, ACL y auditoría; éstas se describirán con más detalle en la Sección 17.3.4.

7.2.5. CREDENCIALES

Las credenciales son un conjunto de evidencias presentadas por un principal cuando pide acceso a un recurso. En el caso más simple, un certificado de una autoridad relevante afirmando la identidad del principal es suficiente, y ésta podrá emplearse para comprobar los permisos del principal en una lista de control de acceso (véase la Sección 7.2.4). A menudo esto es todo lo que se necesita o se proporciona, pero el concepto puede generalizarse para tratar con muchos requisitos más sutiles.

No es conveniente pedir que los usuarios contacten con el sistema y se autentiquen ellos mismos cada vez que se requiere su autoridad para realizar una operación sobre un recurso protegido. En su lugar, se introduce la noción de credencial que *habla por* un principal. Así un certificado de clave pública de un usuario habla por ese usuario; cualquier proceso que reciba una petición autenticada con la clave privada del usuario puede asumir que la petición fue enviada por ese usuario.

La idea de *hablar por* puede llevarse mucho más adelante. Por ejemplo, en una tarea cooperativa, se puede pedir que ciertas acciones sensibles sólo se realicen con la autoridad de dos miembros del equipo; en tal caso, el principal que solicita la acción remitiría su credencial de identificación y una credencial de respaldo de otro miembro del equipo, junto con una indicación de que se empleen simultáneamente cuando se comprueben las credenciales.

De modo similar, para votar en una elección, cada emisión de voto vendría acompañada por un certificado de elector así como un certificado de identificación. Un certificado de delegación permite al principal actuar en lugar de otro, etcétera. En general, una comprobación de control de acceso involucra la evaluación de una fórmula lógica que combina los certificados aportados. Lampson y otros [1992] han desarrollado una completa lógica de autenticación para su empleo en la evaluación de la autoridad de tipo *hablar por* asociada a un conjunto de credenciales. Wobber y otros [1994] describen un sistema que soporta esta aproximación tan general. En [Rowley 1998] se puede encontrar más material sobre formas útiles de credenciales para su aplicación en tareas cooperativas en el mundo real.

Las credenciales basadas en funciones parecen especialmente útiles en el diseño de esquemas de control de acceso prácticos [Sandhu y otros 1996]. Los conjuntos de credenciales basadas en funciones se definen para las organizaciones o para tareas cooperativas, y los derechos de acceso de nivel de aplicación se construyen con referencia a ellas. Se pueden asignar funciones o roles a principales específicos mediante la generación de un certificado de rol que asocia un principal con un rol determinado en una tarea u organización específica [Coulouris y otros 1998].

◇ **Delegación.** Una forma particularmente útil de credencial es aquella que permite a un principal, o proceso actuando para un principal, realizar una acción con la autoridad de otro principal. Una necesidad de delegación puede aparecer en cualquier situación donde un servicio necesite acceder a un recurso protegido para completar una acción en representación de su cliente. Considere el ejemplo de un servidor de impresora que acepta solicitudes de impresión de archivos. Sería un gasto inútil de recursos copiar el archivo, de modo que únicamente se pasa el nombre del archivo al servidor de impresión y éste es accedido por el servidor en representación del usuario que realiza la petición. Si el archivo está protegido contra lectura, no funcionará a menos que el servidor de impresión pueda tomar temporalmente los derechos de lectura del archivo. La delegación es un mecanismo diseñado para resolver problemas como éste.

Se puede conseguir la delegación utilizando un certificado de delegación o una habilitación. El certificado está firmado por el principal solicitante y autoriza a otro principal (el servidor de impresión en nuestro ejemplo) para acceder a un recurso con nombre (el archivo que hay que imprimir). En los sistemas que lo soportan, las habilitaciones pueden lograr el mismo resultado sin necesidad de identificar a los principales; se puede pasar una habilitación para acceder a un recurso en la petición al servidor. La habilitación es un conjunto codificado e infalsificable de derechos de acceso al recurso.

Cuando se delegan derechos, es común restringirse a un subconjunto de los derechos que posee el principal que lo emite, de este modo el principal delegado no podrá abusar de ellos. En nuestro ejemplo, el certificado podría estar limitado en el tiempo para reducir el riesgo de que el código del servidor de impresión esté comprometido posteriormente y se desvele el archivo a terceras par-

tes. El Servicio de Seguridad de CORBA incluye un mecanismo para la delegación de derechos basado en certificados con soporte para la restricción de los derechos que conlleva.

7.2.6. CORTAFUEGOS

Se presentaron los cortafuegos en la Sección 3.4.8. Con ellos se protege una intranet, se realizan acciones de filtrado en las comunicaciones entrantes y salientes. Aquí se discuten sus ventajas y desventajas como mecanismos de seguridad.

En un mundo ideal, la comunicación debiera ocurrir siempre entre procesos que confíen mutuamente y transcurrir siempre entre canales seguros. Hay muchas razones por las que este ideal es irrealizable, algunas se pueden resolver, pero otras no como consecuencia de la naturaleza abierta de ciertos sistemas distribuidos o debido a los errores presentes en la mayoría del software. La facilidad con la que se pueden enviar mensajes hacia cualquier servidor, en cualquier lugar, y el hecho de que muchos servidores no están diseñados para resistir ataques maliciosos de los hackers o por errores accidentales, convierten en algo sencillo la fuga de información confidencial de los servidores de una organización. Ciertos elementos indeseables pueden, también, penetrar en la red de una organización, permitiendo que entren programas de gusano y virus en los computadores. Véase [web.mit.edu II] para una crítica más profunda de los cortafuegos.

Los cortafuegos producen un entorno de comunicación local en el que se intercepta toda comunicación externa. Los mensajes se reenvían al recipiente local final sólo para las comunicaciones que estén autorizadas explícitamente.

Se puede controlar el acceso a las redes internas mediante cortafuegos, pero el acceso a los servicios públicos a Internet no puede restringirse porque su objetivo es ofrecer éstos a un amplio conjunto de usuarios. El empleo de cortafuegos no ofrece protección contra los ataques desde el interior de una organización, y es ciertamente tosco en el control del acceso externo. En consecuencia existe una necesidad de mecanismos de seguridad de un grano más fino, que permitan a los usuarios individuales compartir información con otros usuarios seleccionados sin comprometer la privacidad y la integridad. Abadi y otros [1998] describen una aproximación a la provisión de acceso a datos web privados para usuarios externos basada en un mecanismo de *tunel web* que pueda ser integrado con un cortafuegos. Éste ofrece acceso a los usuarios fiables y autenticados hacia los servidores web mediante un proxy seguro basado en el protocolo HTTPS (HTTP sobre SSL).

Los cortafuegos no son particularmente útiles contra ataques de denegación de servicios tales como el que se describe en la Sección 3.4.2, basado en la suplantación de direcciones IP. El problema es que el flujo de mensajes generado por tales ataques sobrepasa las capacidades de cualquier elemento de defensa singular como un cortafuegos. Cualquier remedio contra las riadas de mensajes debiera aplicarse corriente arriba del objetivo. Los remedios basados en el empleo de mecanismos de calidad de servicio para restringir el flujo de mensajes desde la red hasta un nivel que el objetivo pueda manejar parece ser que prometen.

7.3. ALGORITMOS CRIPTOGRÁFICOS

Un mensaje puede encriptarse mediante la aplicación por el emisor de alguna regla que transforme el *texto en claro* del mensaje (cualquier secuencia de bits) a un *texto cifrado* o *criptograma* (una secuencia diferente de bits). El receptor debe conocer la regla inversa para transformar el texto cifrado en el texto original. El resto de los principales deben ser incapaces de descifrar el mensaje

a menos que conozcan esta regla inversa. La transformación de encriptación se define mediante dos elementos, una *función E* y una *clave K*. El mensaje resultante encriptado se escribe $\{M\}_K$.

$$E(K, M) = \{M\}_K$$

La función de encriptación E define un algoritmo que transforma los datos de texto en datos encriptados al combinarlos con la clave y transformándolos de un modo que depende fuertemente del valor de la clave. Podemos pensar en un algoritmo de encriptación como en una especificación de una familia grande de funciones, de la que seleccionamos un miembro particular mediante una clave dada. La desencriptación se lleva a cabo empleando una función inversa D, que también toma como parámetro una clave. Para la encriptación de clave secreta, la clave de desencriptado es la misma que la de encriptado:

$$D(K, E(K, M)) = M$$

Debido este uso simétrico de las claves, a menudo se habla de la criptografía de clave secreta como *criptografía simétrica*, mientras que la criptografía de clave pública se denomina *asimétrica* debido a que las claves empleadas para el encriptado y el desencriptado son diferentes, como veremos más adelante. En la próxima sección, describiremos varias funciones de encriptación de ambos tipos, ampliamente utilizadas.

◇ **Algoritmos simétricos.** Si eliminamos de la consideración el parámetro de la clave y definimos $F_K([M]) = E(K, M)$, una propiedad de las funciones de encriptación robustas es que $F_K([M])$ sea relativamente fácil de calcular, mientras que la inversa, $F_K^{-1}([M])$ sea tan difícil de calcular que no sea factible. Tales funciones se las conoce como funciones de un solo sentido. La efectividad de cualquier método para encriptar información depende del uso de una función de encriptación F_K que posea esta propiedad de irreversibilidad. Es esta propiedad la que protege a M de ser descubierto dado $\{M\}_K$.

Para los algoritmos simétricos bien diseñados como los que se describen en la próxima sección, su fortaleza contra los intentos de descubrir K dado un texto en claro M y el correspondiente texto cifrado $\{M\}_K$ depende del tamaño de K. La causa de esto se encuentra en que la forma más general de ataque efectivo es la más rudimentaria, conocida como *ataque por fuerza bruta*. La aproximación de fuerza bruta consiste en recorrer todos los valores posibles de K, calculando $E(K, M)$ hasta que el resultado concuerda con el valor de $\{M\}_K$ conocido. Si K tiene N bits entonces tal ataque requiere 2^{N-1} iteraciones de media, y un máximo de 2^N iteraciones, para encontrar K. Entonces el tiempo para romper K es exponencial con el número de bits en K.

◇ **Algoritmos asimétricos.** Cuando se emplea un par de claves pública/privada, las funciones de un solo sentido se explotan de otra forma. La factibilidad de un esquema de clave pública fue propuesta por primera vez por Diffie y Hellman [1976] como un método criptográfico que eliminaba la necesidad de confianza entre dos partes que se comunican. La base de todos los esquemas es la existencia de *funciones de puerta falsa*. Una función de puerta falsa es una función de un solo sentido con una salida secreta: es fácil calcularla en una dirección pero impracticable calcular su inversa a menos que se conozca un secreto. Fue la posibilidad de encontrar tales funciones y utilizarlas en la práctica criptográfica lo que Diffie y Hellman propusieron por primera vez. Desde entonces, se han propuesto y desarrollado varios esquemas prácticos de clave pública. Todos ellos dependen del empleo de funciones de números grandes como funciones de puerta falsa.

El par de claves necesario para los algoritmos asimétricos se deriva de una raíz común. Para el algoritmo RSA, que se describirá en la Sección 7.3.2, la raíz es un par de números primos muy grandes que se eligen arbitrariamente. La derivación del par de claves desde la raíz es una función de un solo sentido. En el caso del algoritmo RSA, los números primos grandes se multiplican uno con otro; un cálculo que precisa unos pocos segundos, incluso para números primos verdadera-

mente grandes. El producto resultante, N, es por supuesto mucho más grande que los multiplicandos. Este uso de la multiplicación es una función de un solo sentido en cuanto a que es impracticable obtener los multiplicandos originales a partir del producto (esto es, factorizar el producto).

Un miembro del par de claves se emplea para encriptar. Para RSA, la función de encriptación oscurece el texto en claro tratando cada bloque de bits como un número binario y elevándolo a la potencia de la clave, módulo N. El número resultante es el bloque criptografiado correspondiente.

El tamaño de N y al menos un miembro del par de claves es mucho mayor que el tamaño de clave seguro para claves simétricas con el fin de asegurar que N no sea factorizable. Por esta razón el potencial de los ataques por fuerza bruta sobre RSA es pequeño; su resistencia a los ataques depende de la imposibilidad de factorizar N. Discutiremos los tamaños seguros para N en la Sección 7.3.2.

◇ **Cifradores de bloque.** La mayoría de los algoritmos de encriptación operan sobre bloques de datos de tamaño fijado; 64 bits es un tamaño de bloque popular. Cada mensaje se subdivide en bloques, el último bloque se rellena hasta la longitud estándar si fuera necesario y cada bloque se encripta independientemente. El primer bloque está listo para ser transmitido tan pronto como haya sido encriptado.

Para un simple cifrador de bloque, el valor de cada bloque del criptograma no depende de los bloques precedentes. Esto constituye una debilidad, dado que un atacante puede reconocer patrones repetidos e inferir su relación con el texto en claro. Tampoco se garantiza la integridad de los mensajes a menos que se añada una suma de chequeo o se emplee un mecanismo de resumen seguro. La mayoría de los algoritmos de cifrado de bloque emplean un encadenamiento de bloques de cifrado (*cipher block chaining*, CBC) para soslayar estas debilidades.

Encadenamiento de bloques de cifrado: En el modo de encadenamiento de bloques de cifrado, cada bloque de texto en claro se combina con el bloque de criptograma precedente empleando la operación O-exclusivo (XOR) antes de encriptarlo (véase la Figura 7.6). En el desencriptado, el bloque se desencripta y se efectúa una operación XOR con el bloque encriptado precedente (que habrá sido almacenado con este fin) obteniendo el nuevo bloque de texto en claro. Esto es posible porque la operación XOR es *idempotente* (dos aplicaciones sucesivas producen el mismo valor).

Con el CBC se pretende impedir que dos porciones idénticas de texto en claro deriven en dos porciones de código encriptado idénticas. Aunque persista una debilidad al comienzo de cada secuencia de bloques: si abrimos dos conexiones encriptadas hacia dos destinatarios y enviamos el mismo mensaje, las secuencias de bloques encriptadas serán la misma, y un fisgón podría obtener algo de información útil de ello. Para prevenir esto, necesitamos insertar un trozo de texto en claro diferente a la cabeza de cada mensaje. Tal texto se denomina *vector de iniciación*. Un número conteniendo la fecha y la hora del mensaje sirve bien como vector de iniciación, y fuerza a que cada mensaje comience con un bloque de texto en claro diferente. Esto, combinado con la operación CBC, deviene en criptogramas diferentes, incluso con dos textos en claro idénticos.

El uso del modo CBC se restringe a la encriptación de datos que se transmiten a lo largo de una conexión fiable. El desencriptado fallará si se pierde cualquiera de los bloques del criptograma, dado que el proceso de desencriptado será incapaz de desencriptar bloques a partir de ese momento. En consecuencia, es inadecuado para ser empleado en aplicaciones como las que se describen

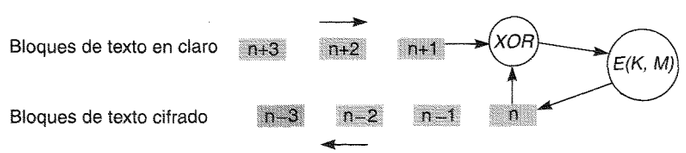


Figura 7.6. Encadenamiento de bloques de cifrado.

en el Capítulo 15, en las que se puede tolerar alguna pérdida de datos. En tales circunstancias habrá que utilizar un cifrador de flujos de datos.

◇ **Cifradores de flujo.** Para algunas aplicaciones, tales como el encriptado de conversaciones telefónicas, la encriptación en bloques es inapropiada porque los flujos de datos se producen en tiempo real en pequeños fragmentos. Las muestras de datos pueden ser tan pequeñas como 8 bits o incluso de 1 bit, y sería un desperdicio rellenar el resto de los 64 bits antes de encriptar y transmitirlos. Los cifradores de caudal son algoritmos de encriptado que pueden realizar la encriptación incrementalmente, convirtiendo el texto en claro en texto criptografiado bit a bit.

Esto parece difícil de lograr, pero de hecho resulta muy simple convertir un algoritmo de cifrado de bloque en un cifrador de caudal. El truco está en construir un *generador del flujo de clave*. Un caudal de clave es una secuencia de bits de tamaño arbitrario que puede emplearse para oscurecer los contenidos de un caudal de datos combinando el caudal de clave con el caudal de datos mediante la función XOR (véase la Figura 7.7). Si el caudal de clave es seguro, el caudal de datos encriptados también lo será.

La idea es análoga a la técnica empleada en la comunidad de espionaje e inteligencia para frustrar las posibles escuchas en la que se emite ruido blanco para ocultar la conversación en una habitación aunque se esté grabando la conversación. Si el ruido ambiente y el ruido blanco se graban por separado, la conversación puede limpiarse sustrayendo el ruido blanco de la grabación del ruido ambiente.

Se puede construir un generador del caudal de clave iterando una función matemática sobre un rango de valores de entrada para producir un caudal continuo de valores de salida. Los valores de salida se concatenan entonces para construir bloques de texto en claro, y los bloques se encriptan empleando una clave compartida por el emisor y el receptor. El caudal de clave puede además disfrazarse aplicando CBC. Los bloques encriptados resultantes se emplean como el caudal de clave. Una iteración de casi cualquier función que devuelva uno de un rango de valores no enteros diferentes funcionará como material fuente, aunque se suele emplear un generador de números aleatorios con un valor de comienzo acordado entre el emisor y receptor. Para conservar la calidad de servicio del caudal de datos, los bloques del flujo de clave deberían producirse con un poco de antelación sobre el momento en que vayan a ser empleados, además el proceso que los produce no debiera exigir demasiado esfuerzo de procesamiento como para retrasar el caudal de datos.

Así en principio, los flujos de datos en tiempo real pueden encriptarse de modo tan seguro como los datos por lotes, siempre que haya disponible suficiente potencia de cálculo para encriptar el caudal de clave en tiempo real. Obviamente, algunos dispositivos que podrían beneficiarse de la encriptación en tiempo real, como los teléfonos móviles, no están equipados con procesadores muy potentes y en este caso sería necesario reducir la seguridad del algoritmo de caudal de clave.

◇ **Diseño de algoritmos criptográficos.** Existen muchos algoritmos criptográficos bien diseñados tales que $E(K, M) = \{M\}_K$ oculta el valor de M y resulta prácticamente imposible recuperar K con mejor éxito que el que proporciona la fuerza bruta. Todos los algoritmos de encriptación se apoyan en manipulaciones que preservan la información de M y emplean principios basados en

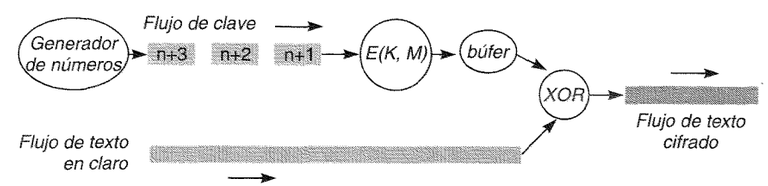


Figura 7.7. Cifrador de caudal.

la teoría de la información [Shannon 1949]. Schneier [1996] describe los principios de *confusión* y *difusión* de Shannon para el ocultamiento del contenido del bloque de criptograma M, combinándolo con una clave K de tamaño suficiente para ponerlo a prueba de ataques por fuerza bruta.

Confusión: Las operaciones de carácter no destructivo como XOR y el desplazamiento circular se emplean para combinar cada bloque de texto en claro con la clave, produciendo un nuevo patrón de bits que oscurece la relación entre los bloques de M y los de $\{M\}_K$. Si los bloques son mayores de unos pocos caracteres se podrá vencer el análisis basado en el conocimiento de la frecuencia de los caracteres en un idioma. (La máquina alemana Enigma empleada en la Segunda Guerra Mundial empleaba bloques de una letra encadenados, y esto podría derrotarse mediante análisis estadístico.)

Difusión: En el texto en claro es normal que aparezcan repeticiones y redundancias. La difusión disipa la existencia de patrones regulares mediante la transposición de partes de cada bloque de texto en claro. Si se emplea CBC, la redundancia también se distribuye a lo largo de un texto suficientemente grande. Los cifradores de caudal no pueden emplear difusión dado que no hay bloques.

En las próximas dos secciones, describiremos el diseño de varios algoritmos prácticos importantes. Todos ellos han sido diseñados a la luz de los principios anteriores y han estado sujetos a un análisis riguroso, y se piensa que son seguros contra cualquier ataque conocido con un margen de seguridad considerable. Excepto el algoritmo TEA, que se muestra con fines ilustrativos, los algoritmos que se muestran aquí se encuentran entre los más empleados en aplicaciones donde se requiera una fuerte seguridad. En alguno de ellos perviven algunas debilidades o cautelas menores; cuestiones de espacio no nos permiten describir aquí todas aquellas precauciones debidas, y se remite al lector a Schneier [1996] para más información. En la Sección 7.5.1 se resumen y comparan la seguridad y prestaciones de los algoritmos.

Aquellos lectores que no precisen comprender la operación de los algoritmos criptográficos pueden omitir las Secciones 7.3.1 y 7.3.2.

7.3.1. ALGORITMOS DE CLAVE SECRETA (SIMÉTRICOS)

Recientemente se han desarrollado y publicado muchos algoritmos criptográficos. Schneier [1996] describe más de 25 algoritmos simétricos, muchos de los cuales él identifica como seguros contra ataques conocidos. Aquí tenemos espacio para describir solamente tres de ellos. El primero, TEA, se ha escogido por la simplicidad de su diseño e implementación y lo empleamos para proporcionar un ejemplo concreto de la naturaleza de tales algoritmos. Se continúa la discusión con DES e IDEA aunque en menor detalle. DES ha sido un estándar nacional de los EE.UU. durante mucho tiempo, aunque su interés ahora es mayormente histórico dado que sus claves de 56 bits son demasiado reducidas para resistir un ataque por fuerza bruta con el hardware actual. IDEA emplea una clave de 128 bits y es, probablemente, el algoritmo de encriptación simétrico de bloques más efectivo y una elección muy acertada para el encriptado en masa.

En 1997, el Instituto Nacional para los Estándares y la Tecnología de los EE.UU. (NIST) emitió una invitación para admitir propuestas de un algoritmo que sería adoptado como nuevo Estándar de Encriptación Avanzada (*Advanced Encription Standard*, AES). Más adelante se proporciona alguna información sobre el progreso de esta actividad.

◇ **TEA.** Los principios de diseño para los algoritmos simétricos que se delinearon anteriormente se ven bien ilustrados en *Tiny Encryption Algorithm* (TEA, pequeño algoritmo de encriptación) desarrollado en la Universidad de Cambridge [Wheeler y Needham 1994]. La función de encriptación, programada en C, se muestra completamente en la Figura 7.8.


```
void encripta(unsigned long k[ ], unsigned long texto[ ] {
    unsigned long y = texto[0], z = texto[1];                                   1
    unsigned long delta = 0x9e3779b9, suma = 0; int n;                          2
    for (n = 0; n < 32; n++) {                                                  3
        suma += delta;                                                          4
        y += ((z << 4) + k[0]) ^ (z+suma) ^ ((z >> 5) + (k[1]);                 5
        z += ((y << 4) + k[2]) ^ (y+suma) ^ ((y >> 5) + (k[3]);                 6
    }
    texto[0] = y; texto[1] = z;                                                 7
}
```

Figura 7.8. Función de encriptación de TEA.

El algoritmo TEA emplea vueltas de sumas enteras, XOR (el operador ^) y desplazamientos lógicos de bits (<< y >>) para obtener la difusión y la confusión de los patrones de bits en el texto en claro. El texto en claro es un bloque de 64 bits representado como dos enteros de 32 bits en el vector *texto[ ]*. La clave tiene 128 bits, representada como cuatro enteros de 32 bits.

En cada una de las 32 etapas, se combinan repetidamente las dos mitades del texto con porciones desplazadas de la clave y entre sí en las líneas 5 y 6. El empleo de XOR sobre porciones del texto desplazadas introduce confusión, y el desplazamiento e intercambio de las dos porciones del texto introduce difusión. La constante *delta* que se combina con cada porción del texto en cada ciclo para oscurecer la clave en caso de que fuera revelada por una sección de texto que no variara. La función de desencriptado es la inversa de la función de encriptación y se muestra en la Figura 7.9.

Este pequeño programa proporciona una encriptación de clave secreta rápida y razonablemente segura. Se han medido sus prestaciones y ronda el triple de velocidad que el algoritmo DES, a la vez que lo conciso del programa se presta a la optimización e implementación por hardware. La clave de 128 bits es segura contra los ataques de fuerza bruta. Los estudios realizados por sus autores y otros han revelado sólo dos pequeñas debilidades que han sido tenidas en cuenta en una nota subsiguiente [Wheeler y Needham 1997].

Para ilustrar su utilización, la Figura 7.10 muestra un simple procedimiento que emplea la encriptación y desencriptación entre un par de archivos abiertos previamente (empleando la librería estándar C *stdio*).

◇ **DES.** El Estándar de Encriptación de Datos (*Data Encryption Standard*, DES) [National Bureau of Standars 1997] fue desarrollado por IBM y adoptado subsiguientemente como un estándar nacional de los EE.UU. para aplicaciones gubernamentales y de negocios. En este estándar, la función de encriptación proyecta un texto en claro de 64 bits en una salida encriptada de 64 bits usando una clave de 56 bits. El algoritmo tiene 16 etapas dependientes de clave conocidas como *vueltas*,

```
void desencripta(unsigned long k[ ], unsigned long texto[ ] {
    unsigned long y = texto[0], z = texto[1];
    unsigned long delta = 0x9e3779b9, suma = delta << 5; int n;
    for (n = 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y+suma) ^ ((y >> 5) + (k[3]);
        y -= ((z << 4) + k[0]) ^ (z+suma) ^ ((z >> 5) + (k[1]);
        suma -= delta;
    }
    texto[0] = y; texto[1] = z;
}
```

Figura 7.9. Función de desencriptación de TEA.

```
void tea(char modo, FILE *archivoEntrada, FILE *archivoSalida, unsigned long k[ ]) {
/* el modo es 'e' para encriptar, 'd' para desencriptar, k[ ] es la clave. */
        char car, Texto[8]; int i;

        while(!feof(archivoEntrada)) {
            i = fread(Texto, 1, 8, archivoEntrada);     /* leer 8 bytes de archivoEntrada sobre Texto[ ] */
            if (i < = 0) break;
            while (i < 8) Texto[i+ +] = '';}            /* rellena el último bloque con espacios */
            switch (modo) {
            case 'e':
                encripta(k, (unsigned long *) Texto); break;
            case 'd':
                desencripta(k, (unsigned long *) Texto); break;
            }
            fwrite(Texto, 1, 8, archivoSalida);         /* escribe 8 bytes desde Texto sobre archivoSalida */
        }
}
```

Figura 7.10. Tea en la práctica.

en las que el dato a encriptar se rota bit a bit un número de veces que depende de la clave y tres transposiciones no dependientes de la clave. Sobre los computadores de los años setenta y ochenta, el algoritmo suponía un coste computacional no despreciable, y consecuentemente fue implementado en hardware VLSI con su consiguiente aplicación en interfaces de red y demás dispositivos de comunicación.

En junio de 1997, fue derrotado con éxito en un ataque por fuerza bruta al que se dio amplia publicidad. El ataque fue realizado en el contexto de una competición con el fin de demostrar la falta de seguridad de los algoritmos de encriptación con claves menores que 128 bits [www.rsasecurity.com I]. Un consorcio de usuarios de Internet, inicialmente 1.000 hasta posteriormente 14.000, superó el reto mediante una aplicación que corría un programa cliente en varios computadores personales (PC y otras estaciones) [Curtin y Dolske 1998].

El programa cliente trataba de romper la clave particular utilizada en una muestra conocida de texto en claro y texto criptografiado, para después desencriptar cierto mensaje secreto de desafío. Los clientes interactuaban con un único servidor, que coordinaba el trabajo, que enviaba a cada computador cliente un rango de claves a comprobar y posteriormente procesaba los informes del trabajo. El computador cliente típico, usualmente un procesador Pentium a 200 MHz, lanzaba el cliente como un proceso en trasfondo. Se encontró la clave en doce semanas, habiendo comprobado aproximadamente el 25 % de los posibles 2^{56}, ó 6×10^{16} valores. En 1998 una máquina desarrollada por la fundación Electronic Frontier Foundation [EFF 1998] podía resolver con éxito claves DES en, aproximadamente, tres días.

A pesar de que aún se emplea en muchas aplicaciones comerciales y de otro tipo, DES, en su forma básica, debe considerarse obsoleta para la protección de aquello que no sea información de bajo interés. En la práctica se utiliza un sistema conocido como *triple-DES* (o *3DES*) [ANSI 1985, Schneier 1996]. Este método conlleva la aplicación repetida de DES con dos claves K_1 y K_2:

$$E_{3DES}(K_1, K_2, M) = E_{DES}(K_1, D_{DES}(K_2, E_{DES}(K_1, M)))$$

Esto proporciona una resistencia contra los ataques por fuerza bruta equivalente a una longitud de clave de 112 bits, proporcionando resistencia suficiente para un futuro previsible a corto plazo, aunque tiene el problema menor de mayores requerimientos de cálculo, como resultado de la aplicación repetida de un algoritmo que aisladamente ya es lento para los estándares modernos.

◇ **IDEA.** El Algoritmo de Encriptación de Datos Internacional (*International Data Encription Algorithm*, IDEA) se desarrolló a comienzos de los años noventa [Lai y Massey 1990, Lai 1992] como sucesor de DES. Como TEA, emplea una clave de 128 bits para encriptar bloques de 64 bits. El algoritmo se basa en el álgebra de grupos y tiene ocho vueltas XOR, suma módulo 2^{16} y multiplicación. Tanto DES como IDEA, emplean una misma función para la encriptación y desencriptación: una propiedad útil para los algoritmos que han de implementarse en hardware.

La resistencia de IDEA ha sido analizada extensamente, y no se han encontrado debilidades significativas. Realiza la encriptación y desencriptación a una velocidad tres veces superior que la de DES.

◇ **AES.** En 1997, el Instituto Nacional para los Estándares y la Tecnología (NIST) publicó una invitación para remitir propuestas de un algoritmo seguro y eficiente que sería adoptado como nuevo Estándar de Encriptación Avanzada (Advanced Encryption Standard, AES) [NIST 1999]. La evaluación concluyó en mayo del año 2000, cuando se seleccionó un estándar preliminar; el estándar final se publicará en el año 2001[1].

La comunidad de investigación sobre criptografía remitió quince algoritmos, como respuesta a la invitación inicial para el AES. Tras una intensa inspección técnica se seleccionaron cinco de ellos para la siguiente fase de evaluación. Todos los candidatos soportaban claves de 128, 192 y 256 bits, siendo todos ellos de altas prestaciones. Es de esperar que, una vez anunciado, el AES se convierta en el algoritmo de encriptación simétrica más ampliamente utilizado.

7.3.2. ALGORITMOS DE CLAVE PÚBLICA (ASIMÉTRICOS)

Hasta la fecha sólo se han desarrollado unos pocos esquemas prácticos de clave pública. Dependen del uso de funciones de puerta falsa de números grandes para producir las claves. Las claves K_e y K_d son un par de números muy grandes, y la función de encriptación realiza una operación, con una exponenciación de M, usando una de ellas. La desencriptación es una función similar usando la otra clave. Si la exponenciación usa una aritmética modular, puede demostrarse que el resultado es el mismo que el valor original de M; es decir:

$$D(K_d, E(K_e, M)) = M$$

Un principal que desee participar en una comunicación segura con otros confecciona un par de claves K_e y K_d y guarda en secreto la clave de desencriptación K_d. La clave de encriptación K_e puede publicarse para cualquiera que quiera comunicar. La clave de encriptación K_e puede verse como parte de una función de encriptación de un sentido E, y la clave de desencriptación K_d es el trozo de conocimiento secreto que permite al principal p invertir la encriptación. Cualquiera que posea K_e (disponible públicamente) puede encriptar mensajes $\{M\}_{K_e}$, pero solamente el principal que posea el secreto K_d puede operar la puerta falsa.

El empleo de números grandes conlleva costes de procesamiento importantes en el cálculo de las funciones E y D. Veremos más tarde que éste es un problema que hay que abordar de forma que el uso de las claves públicas se restrinja a las etapas iniciales de las sesiones de comunicación segura. El algoritmo RSA es ciertamente el algoritmo de clave pública más conocido y lo describiremos con cierto detalle. Otro tipo de algoritmos se basa en funciones derivadas del comporta-

[1] En octubre de 2000, el NIST seleccionó a uno de los candidatos, Rijndael, como propuesta firme de AES. Tras un período de tres meses de revisión pública, en febrero de 2001 la Secretaría de Comercio anunció la disponibilidad al público del borrador de Estándar Federal para el Procesamiento de Información (FIPS, *Federal Information Processing Standard*), para su revisión. Según los planes, las pruebas de validación para el estándar estarán disponibles hacia el verano de 2001. *(N. del T.)*

miento de las curvas elípticas en un plano. Estos algoritmos ofrecen la posibilidad de menores costes en las funciones de encriptación y desencriptación, con el mismo nivel de seguridad, pero su aplicación práctica se encuentra menos avanzada y las trataremos sólo brevemente.

◇ **RSA.** El diseño de Rivest, Shamir y Adelman (RSA) para el encriptador de clave pública [Rivest y otros 1978] se basa en el uso del producto de dos números primos muy grandes (mayores que 10^{100}), empleando el hecho de que la determinación de los factores primos de números tan grandes es computacionalmente tan difícil como para considerarlo imposible de calcular.

A pesar de ser objeto de extensas investigaciones, no se han encontrado flaquezas en él, y hoy en día se usa ampliamente. A continuación se muestra un bosquejo del método. Para encontrar un par de claves e, d:

1. Elíjanse dos números primos grandes, P y Q (mayores que 10^{100}), y fórmese

   $N = P \times Q$

   $Z = (P - 1) \times (Q - 1)$

2. Para d elíjase cualquier número primo con relación a Z (esto es, que d no tenga factores en común con Z).

Ilustremos los cálculos correspondientes empleando valores pequeños de P y Q:

$P = 13,\ Q = 17 \rightarrow N = 221,\ Z = 192$

$d = 5$

3. Para encontrar e resuélvase la ecuación:

   $e \times d = 1 \ mod\ Z$

Esto es, $e \times d$ es el número más pequeño divisible por d en la serie $Z + 1$, $2Z + 1$, $3Z + 1$, ...

$e \times d = 1 \ mod\ 192 = 1, 193, 385, \ldots$

385 es divisible por d

$e = 385/5 = 77$

Para encriptar texto usando el método RSA, el texto en claro se divide en bloques iguales de longitud k bits donde $2^k < N$ (es decir, el valor numérico de un bloque es siempre menor que N; en aplicaciones prácticas, k está generalmente en el rango de 512 a 1.024).

$k = 7$, ya que $2^7 = 128$

La función de encriptación de un bloque de texto en claro M es:

$$E'(e, N, M) = M^e \ mod\ N$$

Para un mensaje M, el criptograma es $M^{77}\ mod\ 221$

La función para desencriptar un bloque de texto encriptado c para producir el bloque de texto en claro original es:

$$D'(d,N,c) = c^d \ mod\ N$$

Rivest, Shamir y Adelman probaron que E' y D' son inversas mutuas (esto es, que $E'(D'(x)) = D'(E'(x)) = x$) para cualquier valor P en el rango $0 \leqslant P \leqslant N$.

Los dos parámetros e, N pueden considerarse como clave para la función de encriptación, e igualmente d, N representan una clave para función de desencriptación. Así podemos escribir $K_e = <e, N>$ y $K_d = <d, N>$, y obtener las funciones de encriptación $E(K_e, M) = \{M\}_K$ (esta notación indica que el mensaje encriptado puede ser encriptado sólo por el portador de la clave *privada* K_d) y $D(K_d, \{M\}_K) = M$.

Merece la pena fijarse en una potencial debilidad de todos los algoritmos de clave pública: dado que la clave pública está disponible para los atacantes, éstos pueden generar fácilmente mensajes encriptados. Así, se podría intentar desencriptar un mensaje desconocido encriptando exhaustivamente secuencias arbitrarias de bits hasta que concuerden con el mensaje objetivo. Este ataque se conoce como *ataque de texto en claro escogido*, y se vence asegurando que todos los mensajes son más largos que la longitud de la clave, de forma que el ataque por fuerza bruta sea menos factible que un ataque directo sobre la clave.

Un aspirante a receptor de información secreta debe publicar o distribuir de cualquier forma el par $<e, N>$ manteniendo d secreto. La publicación de $<e, N>$ no compromete el secreto de d, porque cualquier intento de determinar d requiere el conocimiento de los números primos P y Q originales, y esto sólo se puede obtener mediante la factorización de N. La factorización de números primos grandes (recordamos que P y Q se eligen $> 10^{100}$, y entonces $N > 10^{200}$) es extremadamente costoso en tiempo, incluso con computadores de altas prestaciones. En 1978, Rivest y otros calcularon que factorizar un número del tamaño de 10^{200} tomaría más de cuatro mil millones de años con el mejor algoritmo conocido sobre un computador que efectuara un millón de instrucciones por segundo.

Desde la afirmación de Rivest en 1978 se han desarrollado computadores mucho más rápidos y métodos de factorización mejores. RSA Corporation publicó una serie de retos para realizar factorizaciones de números de más de 100 dígitos decimales [www.rsasecurity.com II]. Se han logrado factorizar números de hasta 155 dígitos decimales de longitud mediante un consorcio distribuido de usuarios de Internet trabajando de un modo similar al esfuerzo de romper DES que se describe en la sección anterior. En el momento de escribir esto, el hecho es que se han factorizado con éxito números de 155 dígitos (o aproximadamente 500 dígitos binarios), lo que recorta la confianza en la seguridad de claves de 512 bits. El esfuerzo necesario para factorizar no ha sido publicado, pero un número de 140 dígitos requirió del orden de 2.000 MIPS al año en febrero de 1999. Un MIPS al año es el esfuerzo de procesamiento generado por un procesador con una velocidad de un millón de instrucciones por segundo calculando durante un año. Los procesadores de un solo chip actuales tienen velocidades del orden de 200 MIPS, lo que les permite completar esta tarea en cerca de diez años, pero las aplicaciones distribuidas de factorización y los procesadores paralelos podrían reducir esta magnitud a días. RSA Corporation (propietarios de las patentes del algoritmo RSA) recomiendan una clave de al menos 768 bits, es decir 230 dígitos decimales, para la seguridad a largo plazo (cerca de 20 años). En algunas aplicaciones se emplean claves de 2.048 bits.

Todos los cálculos de resistencia anteriores, presuponen de que los algoritmos conocidos actualmente son los mejores disponibles. RSA, y otras formas de criptografía asimétrica que emplean multiplicación de números primos como función de un sentido, serían vulnerables en el caso de que se descubriera un algoritmo de factorización más rápido.

◇ **Algoritmos de curvas elípticas.** Se ha desarrollado y probado un algoritmo para generar pares de claves pública/privada basándose en las propiedades de las curvas elípticas. Los detalles

completos pueden encontrarse en el libro de Menezes dedicado al tema [Menezes 1993]. Las claves de derivan de una rama diferente de las matemáticas, y a diferencia de RSA su seguridad no depende de la dificultad de la factorización de números grandes. Las claves cortas son seguras, y los requisitos de procesamiento para la encriptación y la desencriptación son menores que los de RSA. Parece que los algoritmos de encriptación de curvas elípticas serán adoptados más ampliamente en el futuro, especialmente en sistemas como los que incorporan los dispositivos móviles, con recursos de procesamiento limitados. Las matemáticas relevantes involucran algunas propiedades bastante complejas de las curvas elípticas y se encuentran fuera del alcance de este libro.

7.3.3. PROTOCOLOS CRIPTOGRÁFICOS HÍBRIDOS

La criptografía de clave pública es apropiada para el comercio electrónico porque no hay necesidad de un mecanismo de distribución segura de claves. (Existe una necesidad de autentificar claves públicas pero esto es mucho menos oneroso, y sólo se requiere enviar con la clave un certificado de clave pública). Pero los costes de procesamiento de la criptografía de clave pública son demasiado altos para la encriptación incluso de mensajes de tamaño medio como los que se encuentran habitualmente en el comercio electrónico. La solución adoptada en la mayoría de sistemas distribuidos de gran escala es el empleo de un esquema de encriptación híbrido en el que se emplea criptografía de clave pública para autentificar cada parte y para encriptar un intercambio de claves secretas, que se emplearán para toda la comunicación subsiguiente. En la Sección 7.6.3 describiremos la implementación de un protocolo híbrido en el caso de estudio SSL.

7.4. FIRMAS DIGITALES

Una firma digital robusta es un requisito esencial para los sistemas seguros. Se las necesita para certificar ciertos trozos de información, por ejemplo, para proporcionar enunciados dignos de confianza que relacionan identidades de usuarios con sus claves públicas, derechos de acceso o roles.

La necesidad de firmas en muchos tipos de negocios y transacciones personales está más allá de discusión. Las firmas manuscritas se han usado como medio de verificación de documentos desde que existen éstos. Las firmas manuscritas se ajustan a las necesidades de los receptores de un documento y que necesitan verificar que éste es:

Auténtico: convence al receptor de que el firmante firmó deliberadamente el documento y que la firma no ha sido alterada por nadie más.

Infalsificable: aporta la prueba de que el firmante, y nadie más, firmó deliberadamente el documento. La firma no puede ser copiada y colocada en otro documento.

No repudiable: el firmante no puede negar de forma creíble que el documento fue firmado por él.

En la práctica, ninguna de estas propiedades deseables se logra completamente mediante las firmas convencionales; las falsificaciones y las copias son difíciles de detectar, los documentos pueden alterarse ya firmados y los firmantes a veces afirman haber sido engañados para firmar un documento involuntariamente o de mala fe. Aun así preferimos sobrellevar estas imperfecciones dada la dificultad de hacer trampas y el riesgo de que sean detectadas. Del mismo modo que las firmas manuscritas, se cuenta con que en las firmas digitales se establece un vínculo entre un atributo único y secreto del firmante y el documento. En el caso de las firmas manuscritas el secreto está en el patrón de escritura del firmante.

Las propiedades de los documentos digitales almacenados en archivos o mensajes son completamente diferentes de las de los documentos en papel. Los documentos digitales son trivialmente fáciles de generar, copiar y alterar. La simple adición de la identidad del emisor en forma de cadena de texto, una fotografía o una imagen manuscrita, no tiene valor alguno como medio de verificación.

Lo que se requiere es un medio de enlazar una identidad de un firmante a la secuencia completa de bits que representa un documento. Con esto cumplimos el primer requisito anterior, de autenticidad. Como con las firmas manuscritas, la fecha del documento no está garantizada por la sola firma. El destinatario de un documento firmado sólo sabe que el documento fue firmado con anterioridad a la recepción.

Con respecto al no repudio, existe un problema que no aparece con las firmas manuscritas. ¿Qué pasa si el firmante revela deliberadamente su clave privada y posteriormente niega haber firmado, argumentando que otros podrían haberlo hecho también, puesto que la clave no era privada? Se han desarrollado algunos protocolos para resolver este problema con el nombre de *firmas digitales innegables* [Schneier 1996], pero añaden una complejidad considerable al procedimiento.

Se puede considerar que un documento con firma digital es más robusto frente a la falsificación que uno escrito a mano. Pero la palabra «original» carece de significado con respecto a los documentos digitales. Como veremos en nuestro discurso sobre las necesidades del comercio electrónico, las firmas digitales aisladamente no pueden prevenir, por ejemplo, el gasto doble de dinero electrónico, y se requieren otras medidas para prevenirlo. Aquí describiremos dos técnicas para firmar documentos digitalmente, vinculando la identidad de un principal al documento. Ambas dependen del uso de la criptografía.

◇ **Firmado digital.** Un documento o mensaje electrónico M puede ser firmado por un principal A encriptando una copia de M con una clave K_A y añadiéndola a una copia del texto en claro M y un identificador de A. El documento firmado consta de: M, A, $[M]_{K_A}$. La firma puede ser verificada por un principal que reciba después el documento para comprobar si fue originada por A y que sus contenidos, M, no han sido alterados posteriormente.

Si se emplea una clave secreta para encriptar el documento, sólo los principales que comparten el secreto pueden verificar la firma. Pero si se emplea criptografía de clave pública, el firmante utiliza su clave privada y cualquiera que disponga de la clave pública correspondiente puede verificar la firma. Ésta proporciona una mejor analogía de una firma convencional y posee un rango de aplicación mayor para el usuario. La verificación de firmas digitales sigue caminos diferentes dependiendo de si se emplea criptografía de clave pública o privada para producir la firma. Describimos ambos casos en las Secciones 7.4.1 y 7.4.2.

◇ **Funciones resumen.** Las funciones resumen se denominan también *funciones de dispersión seguras* y se denotan con $H(M)$. Éstas deben diseñarse cuidadosamente para asegurar que $H(M)$ sea diferente de $H(M')$ para todos los probables pares de mensajes M y M'. Si hubiera cualquier par de mensajes similares M y M' de modo que $H(M) = H(M')$, un principal mentiroso podría enviar una copia firmada de M, pero al someterse a una validación argumentaría que envió M', y que el original fue alterado en el tránsito. Discutiremos algunas funciones de dispersión seguras en la Sección 7.4.3.

7.4.1. FIRMAS DIGITALES CON CLAVES PÚBLICAS

La criptografía de clave pública se adapta particularmente bien a la generación de firmas digitales dado que es relativamente simple y no requiere ninguna comunicación entre el destinatario de un documento firmado y el firmante o cualquier otro.

El método para que A firme un mensaje M y B lo verifique es como sigue (según se ilustra gráficamente en la Figura 7.11):

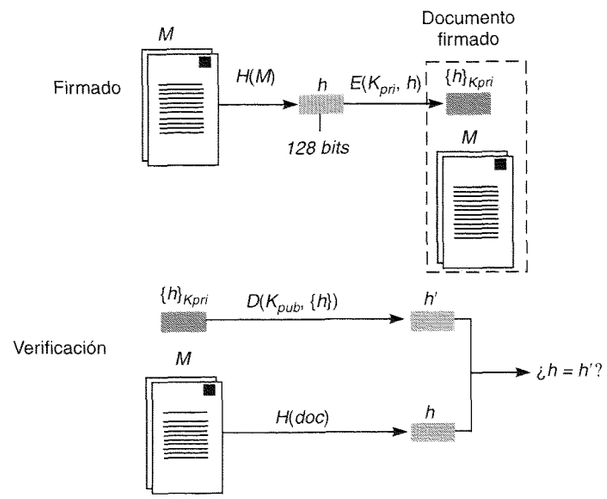


Figura 7.11. Firmas digitales con claves públicas.

1. A genera un par de claves K_{pub} y K_{priv}, y publica la clave pública K_{pub} situándola en una ubicación bien conocida.
2. A calcula el resumen de M, $H(M)$ empleando una función de dispersión segura H acordada y la encripta utilizando la clave privada K_{priv} para producir la firma $S = \{H(M)\}_{K_{priv}}$.
3. A envía el mensaje firmado $[M]_K = M, S$ a B.
4. B desencripta S empleando K_{pub} y calcula el resumen de M, $H(M)$. Si concuerdan, la firma es válida.

El algoritmo RSA es bastante adecuado para su uso en la construcción de firmas digitales. Advierta que la clave *privada* del firmante se usa para encriptar la firma, en contraste al uso de la clave *pública* del receptor para la encriptación cuando se emplea para transmitir información en secreto. La explicación de esta diferencia es inmediata: una firma se crea utilizando un secreto conocido sólo para el firmante y la verificación debe ser accesible para todos.

7.4.2. FIRMAS DIGITALES CON CLAVES SECRETAS, MAC

No hay ninguna razón técnica por la que un algoritmo de encriptación de clave secreta no pueda usarse para encriptar una firma digital, pero para verificar tales firmas habrá que descubrir la clave, y esto origina algunos problemas:

- El firmante debe conseguir que el verificador reciba la clave secreta empleada para firmar de modo seguro.
- Debe ser necesario poder verificar una firma en varios contextos diferentes y en momentos diferentes; en el momento de la firma el firmante puede no conocer la identidad de los verificadores. Para resolver esto, la verificación debe delegarse a una tercera parte de confianza que posea las claves secretas de todos los firmantes, sin embargo esto añade complejidad al modelo de seguridad y requiere una comunicación segura con la tercera parte digna de confianza.

- El descubrimiento de una clave secreta empleada para una firma es poco deseable puesto que debilita la seguridad de las firmas realizadas con ella; podría falsificarse una firma por alguien que posea la clave y que no sea su propietario.

Por todas estas razones, el método de clave pública para generar y verificar firmas ofrece la solución más conveniente en la mayoría de las situaciones.

Existe una excepción cuando se utiliza un canal seguro para transmitir los mensajes desencriptados pero subsiste la necesidad de verificar la autenticidad de los mensajes. Dado que un canal seguro proporciona comunicaciones seguras entre pares de procesos, se puede establecer una clave secreta empleando el método híbrido indicado en la Sección 7.3.3 y así producir firmas de coste bajo. Estas firmas se denominan *códigos de autenticación de mensajes* (*message autentication codes*, MAC) para reflejar su objetivo más limitado: autentican la comunicación entre pares de principales basándose en un secreto compartido.

En la Figura 7.12 se ilustra una técnica de firmado de bajo coste basada en claves secretas que da una seguridad suficiente para muchos propósitos, y que se bosqueja a continuación. El método depende de la existencia de un canal seguro por el que se puede distribuir la clave compartida:

1. A genera una clave aleatoria K para firmar, y la distribuye utilizando canales seguros a uno o más principales que necesiten autenticar los mensajes recibidos de A. Se *confía* en que los principales no descubran la clave compartida.
2. Para cualquier documento M que A desee firmar, A concatena M con K, calcula el resumen del resultado: $h = H(M + K)$ y envía el documento firmado $[M]_K = M, h$ a cualquiera que desee verificar la firma (el resumen h es un MAC). K no queda comprometida por la publicación de h, dado que la función de dispersión oscurece totalmente su valor.
3. El receptor, B, concatena la clave secreta K con el documento recibido M y calcula el resumen $h' = H(M + K)$. La firma es válida si $h = h'$.

Aun cuando este método sufre de las desventajas indicadas anteriormente, tiene la ventaja de sus buenas prestaciones porque no involucra encriptación alguna. (La dispersión segura es aproximadamente de 3 a 10 veces más rápida que la encriptación simétrica, véase la Sección 7.6.3.) El protocolo de canal seguro SSL descrito en la Sección 7.6.3 soporta el uso de una amplia variedad

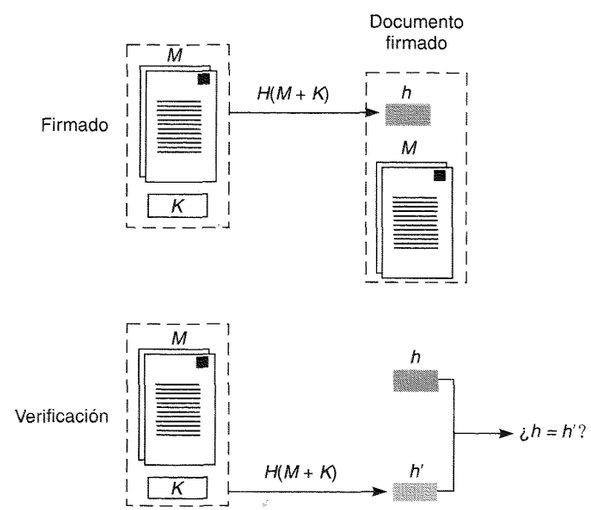


Figura 7.12. Firmas de pequeño coste con una clave secreta compartida.

de MAC incluyendo el esquema que aquí se describe. Este método también se emplea en el protocolo de dinero electrónico Millicent que se describe en la Sección 7.6.4, donde es muy importante mantener bajo el coste de procesamiento para transacciones de bajo valor.

7.4.3. FUNCIONES DE RESUMEN SEGURO

Hay muchas formas de producir un patrón de bits de longitud fija que caracterice un mensaje o documento de longitud arbitraria. Quizás la más simple es utilizar repetidamente la operación XOR para combinar trozos de tamaño fijo del documento fuente. Tal función se emplea a menudo en protocolos de comunicación para producir un patrón de dispersión de longitud fija para caracterizar un mensaje con fines de detección de errores; pero este sistema es inadecuado como base de un esquema de firma digital. Una función de resumen segura $h = H(M)$ debería poseer las siguientes propiedades:

1. Dado M, debe ser fácil calcular h.
2. Dado h, debe ser extremadamente difícil calcular M.
3. Dado M, debe ser extremadamente difícil encontrar otro mensaje M', tal que $H(M) = H(M')$.

Tales funciones se denominan también *funciones de dispersión de un solo sentido*. La razón de este nombre es evidente por sí misma si vemos las dos primeras propiedades. La propiedad 3 requiere una característica adicional: incluso aunque conozcamos que la aplicación de la función de dispersión no dé un valor único (porque el resumen es una transformación de reducción de información), necesitamos asegurarnos que un atacante, dado un mensaje M que produce un valor h, no pueda descubrir otro mensaje M' que también produzca h. Si un atacante *pudiera* hacer esto, podría falsificar un documento firmado M' sin tener que conocer la clave de la firma, simplemente copiando la firma del documento firmado M y añadiéndosela a M'.

Desde luego, el conjunto de mensajes que se aplican sobre el mismo valor está restringido y el atacante lo tendría difícil para producir una falsificación con sentido, pero con paciencia podría hacerse, de modo que hay que prevenirlo. La posibilidad de conseguirlo aumenta considerablemente si se emplea el denominado *ataque del cumpleaños* (*birdthday attack*):

1. Alice prepara dos versiones M y M' de un contrato para Bob. M es favorable a Bob y M' no.
2. Alice consigue varias versiones sutilmente diferentes de M y M' que aparenten ser visualmente la misma, añadiendo espacios al final de las líneas, etc. Ella compara los valores de dispersión de todos los M con la de todos los M'. Si encuentra dos que sean las mismas, ella puede seguir adelante; si no, sigue produciendo versiones visualmente indistinguibles hasta que encuentra dos adecuadas.
3. Cuando ya tiene un par de documentos M y M' que proyectan el mismo valor, le da el documento favorable M a Bob para que lo firme con una firma digital utilizando la clave privada de él. Cuando lo devuelve, ella intercambia el contenido por la versión M', manteniendo la firma de M.

Si nuestros valores de dispersión tienen un tamaño de 64 bits, sólo necesitamos 2^{32} versiones de M y M' por término medio. Es demasiado poco para quedar conforme. Al menos necesitamos que nuestros valores de dispersión tengan 128 bits para alejar el peligro de este ataque.

El ataque descansa sobre la paradoja estadística conocida como la paradoja del cumpleaños: la probabilidad de encontrar un par idéntico en un conjunto dado es mucho mayor que la de encontrar la pareja para un individuo dado. Stallings [1999] da una derivación estadística para la probabilidad de que haya dos personas con la misma fecha de nacimiento al considerar un conjunto de n personas. El resultado es que para un conjunto de sólo 23 personas la probabilidad es del 50 por ciento, mientras que para que la probabilidad de que una persona haya nacido cierto día sea también del 50 por ciento se requieren 253 personas.

Para satisfacer las propiedades indicadas arriba, hay que diseñar la función de resumen segura cuidadosamente. Las operaciones al nivel de bit empleadas y su secuenciación son similares a las que se encuentran en la criptografía simétrica, pero en este caso las operaciones no tienen por qué preservar la información, dado que la función no tiene por qué ser reversible en absoluto. De modo que una función de resumen segura puede emplear cualquier operación aritmética y lógica al nivel de bit. La longitud del texto fuente se incluye habitualmente entre los datos resumidos.

Dos funciones de resumen ampliamente utilizadas son el algoritmo MD5 (llamado así porque es el quinto de una serie de algoritmos de resumen desarrollados por Ron Rivest) y el SHA (*Secure Hash Algoritm*) adoptado para su estandarización por el NIST. Ambos han sido probados y analizados cuidadosamente, y puede considerarse que son apropiadamente seguros durante el futuro previsible, al mismo tiempo que sus implementaciones son razonablemente eficientes. Las describimos brevemente aquí. Schneier [1996] y Mitchell y otros [1992] informan en profundidad de las técnicas de firma digital y las funciones de resumen de mensajes.

◇ **MD5.** El algoritmo MD5 [Rivest 1992] emplea cuatro vueltas, cada una aplicando una de cuatro funciones no lineales a cada uno de los dieciséis segmentos de 32 bits de un bloque de texto fuente de 512 bits. El resultado es un resumen de 128 bits. MD5 es uno de los algoritmos más eficientes de que se dispone hoy en día.

◇ **SHA.** SHA [NIST 1995] es un algoritmo que produce un resumen de 160 bits. Se basa en el algoritmo de Rivest MD4 (similar a MD5) con algunas operaciones adicionales. Es sustancialmente más lento que MD5, pero el resumen de 160 bits ofrece una mayor seguridad contra los ataques por fuerza bruta y del cumpleaños.

◇ **Empleo de un algoritmo de encriptación para obtener un resumen.** Es posible utilizar un algoritmo de encriptación simétrico tales como los detallados en la Sección 7.3.1 para producir un resumen seguro. En este caso, la clave debería ser pública de modo que el algoritmo pudiera ser aplicado por cualquiera que deseara verificar una firma digital. El algoritmo de encriptado se emplea en modo CBC, y el resumen es el resultado de combinar el penúltimo valor CBC con el bloque final encriptado.

7.4.4. ESTÁNDARES DE CERTIFICACIÓN Y AUTORIDADES DE CERTIFICACIÓN

X.509 es el formato estándar de certificación más ampliamente usado [CCITT 1988b]. A pesar de que el formato de certificación X.509 es parte del estándar X.500 para la construcción de directorios globales de nombres y atributos [CCITT 1988a], se emplea habitualmente en el mundo criptográfico como una definición de formato para certificados aislados. Describiremos el estándar de nombres X.500 en el Capítulo 9.

La estructura y contenido de un certificado X.509 se muestra en la Figura 7.13. Como puede verse, se enlaza una clave pública a una entidad con nombre denominada *sujeto*. El enlace está en la firma, que es emitida por otra entidad denominada *emisora*. El certificado tiene un *período de validez*, que es definido mediante dos fechas. Los contenidos etiquetados como <Nombre Diferenciado> se refieren al nombre de una persona, organización o cualquier otra entidad junto con suficiente información contextual para darlo por único. En una implementación completa X.500 esta información contextual podría obtenerse de una jerarquía de directorio en la que aparezca la entidad nombrada, pero en ausencia de una implementación X.500 completa sólo puede usarse como una cadena descriptiva.

Este formato se incluye en el protocolo SSL para comercio electrónico y de amplia implantación en la autenticación de claves públicas de servicios y sus clientes. De hecho ciertas compañías

Sujeto	Nombre Diferenciado, Clave pública
Emisor	Nombre Diferenciado, Firma
Período de validez	Fecha inicio, Fecha expiración
Información administrativa	Versión, Número de Serie
Información añadida	

Figura 7.13. Formato de Certificado X.509.

y organizaciones de relieve en Internet se han establecido ellas mismas como *autoridades de certificación* (por ejemplo Verisign [www.verisign.com], CREN [www.cren.net]), y otras compañías e individuos pueden obtener certificados de clave pública X.509 de los anteriores remitiéndoles alguna evidencia satisfactoria de su identidad. Esto nos conduce a un procedimiento de verificación de dos pasos para cualquier certificado X.509:

1. Obtener el certificado de clave pública del emisor (una autoridad de certificación) desde una fuente fiable.
2. Validar la firma.

◇ **Aproximación SPKI.** La aproximación X.509 se basa en la unicidad global de los nombres diferenciados. Ya se ha indicado que esta meta es inalcanzable dado que no refleja la realidad actual legal y de la práctica comercial [Ellison 1996] en la que no se presupone la unicidad de la identidad de los individuos si no se hace referencia a otras personas y organizaciones. Esto se pone en evidencia, por ejemplo, en el uso de un permiso de conducción o una carta de un banco para autenticar el nombre y dirección de un individuo (un nombre a secas puede estar repetido si consideramos la población mundial). Esto nos lleva a cadenas de verificación más largas, puesto que hay muchos emisores posibles de certificados de clave pública, y sus firmas deben validarse a través de una cadena de verificación que nos lleve a alguien conocido en quien confíe el principal que realiza la verificación. A pesar de ello la verificación resultante suena más convincente, y muchos de los pasos de esta cadena pueden guardarse para acortar el proceso en otras ocasiones.

Los argumentos de arriba son la base de las propuestas recientemente desarrolladas para la Infraestructura de Clave Pública Simple (*Simple Public-key Infrastructure*, SPKI) (véase RFC 2693 [Ellison y otros 1999]). Éste es un esquema para la creación y administración de conjuntos de certificados públicos. Posibilita el procesamiento de cadenas de certificaciones empleando inferencia lógica para producir certificados derivados. Por ejemplo, «Bob cree que la clave pública de Alice es K_{Apub}» y «Carol confía en la clave de Alice que tiene Bob» implica que «Carol cree que la clave pública de Alice es K_{Apub}».

7.5. PRAGMÁTICA DE LA CRIPTOGRAFÍA

En la Sección 7.5.1, compararemos las prestaciones de la encriptación y los algoritmos de dispersión seguros descritos o mencionados anteriormente. Consideramos los algoritmos de encriptación al mismo tiempo que las funciones de dispersión seguras porque la encriptación puede usarse también como un método de firma digital.

En la Sección 7.5.2, discutiremos algunas cuestiones no técnicas que rodean al uso de la criptografía. No hay espacio suficiente que haga justicia a la enorme cantidad de discusión política que ha tenido lugar sobre este tema desde que aparecieron en el dominio público los primeros algoritmos criptográficos robustos, aunque los debates no hayan alcanzado muchas conclusiones definitivas. Nuestro objetivo es, meramente, hacer tomar conciencia al lector de este debate en curso.

	Tamaño de clave/tamaño de dispersión (bits)	*Velocidad extrapolada (kbytes/sec)*	*PRB optimizado (kbytes/s)*
TEA	128	700	—
DES	56	350	7.746
Triple-DES	112	120	2.842
IDEA	128	700	4.469
RSA	512	7	—
RSA	2.048	1	—
MD5	128	1.740	62.425
SHA	160	750	25.162

Figura 7.14. Prestaciones de los algoritmos de encriptación y resúmenes seguros.

7.5.1. PRESTACIONES DE LOS ALGORITMOS CRIPTOGRÁFICOS

La Figura 7.14 muestra el tamaño de la clave (o el tamaño del resultado en el caso de funciones de resumen seguras) y la velocidad de los algoritmos de encriptación y las funciones de resumen seguras que hemos discutido en este capítulo. Donde sea posible, damos dos medidas de velocidad; las que aparecen bajo el título «velocidad extrapolada» están basadas en cifras de Schneier (excepto para TEA, que se basan en [Wheeler y Needham 1994]). Bajo el título «PRB optimizado» se basan en las publicadas en [Preneel y otros 1998]. En ambos casos hemos ajustado las cifras para tener en cuenta los avances en las prestaciones de los procesadores entre esa fecha y últimos del año 1999; los números pueden tomarse como estimaciones aproximadas de las prestaciones de los algoritmos sobre un procesador Pentium II a 330 MHz. Las longitudes de las claves dan una indicación aproximada de la fortaleza de los algoritmos; en realidad sólo proporciona una indicación del coste computacional de un ataque por fuerza bruta sobre la clave; la auténtica resistencia de los algoritmos criptográficos es mucho más difícil de evaluar y descansa sobre razonamientos acerca del éxito de cada algoritmo para oscurecer el texto en claro. La gran diferencia entre las dos columnas de prestaciones proviene del hecho de que las cifras dadas por Schneier están basadas en el código C publicado en [Schneier 1996], sin ningún intento de optimizar el código, mientras que las cifras del PRB son el resultado de un sustancial esfuerzo para producir implementaciones propietarias, optimizadas, de los algoritmos en lenguaje ensamblador. Preneel y otros [1998] presentan una discusión útil sobre la resistencia y prestaciones de los principales algoritmos simétricos.

Sólo hemos presentado cifras de prestaciones para el software. En la práctica se realizan esfuerzos considerables para producir implementaciones de altas prestaciones y bajo coste (un solo chip) y de ellas cabe esperar prestaciones mucho mayores.

7.5.2. APLICACIONES DE LA CRIPTOGRAFÍA Y OBSTÁCULOS POLÍTICOS

Todos los algoritmos anteriormente descritos emergen durante los años ochenta y noventa, cuando comienzan a utilizarse las redes de computadores con objetivos comerciales y se hizo evidente que las necesidades de seguridad eran una cuestión importante. Tal y como mencionamos en la introducción de este capítulo, la emergencia del software criptográfico tuvo una fuerte resistencia del gobierno de los EE.UU. La oposición tuvo dos orígenes, la Agencia de Seguridad Nacional (*National Security Agency*, NSA) que urgía una política para restringir la robustez de la criptografía disponible a otras naciones a un nivel en el que el NSA pudiera desencriptar cualquier comunicación

secreta por objetivos de inteligencia militar y la Oficina Federal de Investigación (*Federal Bureau of Investigation*, FBI) cuyo objetivo era asegurar que sus agentes pudieran tener acceso privilegiado a las claves criptográficas empleadas por todas las organizaciones privadas e individuos de los EE.UU. con el objetivo de poder ejercer la ley.

El software criptográfico se clasificó en los EE.UU. como armamento y quedaba sujeto a estrictas restricciones de exportación. Otros países, especialmente aliados de los EE.UU., aplicaron restricciones similares, cuando no más estrictas. El problema se mezclaba con la ignorancia general entre los políticos y el público en general de lo que era el software criptográfico y sus potenciales aplicaciones no militares. Algunas compañías de software de los EE.UU. protestaron que las restricciones inhibían la exportación de cierto software como los visualizadores, y se reformaron las restricciones de exportación de modo que se permitía la exportación de código que utilizara claves de no más de 40 bits; ¡Una criptografía depauperada!

Las restricciones de exportación podrían haber impedido el crecimiento del comercio electrónico, pero no fueron particularmente eficaces en la prevención del despliegue del conocimiento sobre criptografía ni tampoco para mantener el software criptográfico fuera de las manos de los usuarios de otros países, dado que muchos programadores de dentro y fuera de los EE.UU. estaban deseosos y eran capaces de implementar y distribuir código criptográfico. La situación actual es que el software que implementa la mayoría de los algoritmos criptográficos ha estado disponible por todo el mundo durante años, impreso [Schneier 1996] y en línea, en versiones comerciales y gratuitas [www.rsasecurity.com, cryptography.org, privacy.nb.ca, www.openssl.org].

Un ejemplo es el programa llamado PGP (*Pretty Good Privacy*) [Garfinkel 1994, Zimmermann 1995], desarrollado originalmente por Philip Zimmermann y desarrollado por él mismo y otros. Éste es parte de una campaña técnica y política para asegurar que la disponibilidad de métodos criptográficos no se encuentre controlada por el gobierno de los EE.UU. PGP ha sido desarrollado y distribuido con el objetivo de permitir que todos los usuarios de computadores del mundo disfruten el nivel de privacidad e integridad que permite el uso de la criptografía de clave pública para sus comunicaciones. PGP genera y administra claves públicas y secretas en representación de un usuario. Emplea encriptación de clave pública RSA para la autenticación y para transmitir claves secretas al otro participante en la comunicación y emplea los algoritmos de clave privada IDEA o 3DES para encriptar mensajes de correo u otros documentos. (En el momento en que se desarrolló PGP, el uso del algoritmo DES estaba controlado por el gobierno de los EE.UU.) PGP está disponible ampliamente tanto en versiones libres como comerciales. Con el fin de evitar (aunque de forma legal) las regulaciones de exportación de los EE.UU., se distribuye desde lugares diferentes ya sea para los usuarios de los EE.UU. [www.pgp.com] o bien para los de otros países del mundo [International PGP].

El gobierno de los EE.UU. reconoció, eventualmente, la futilidad de la postura del NSA y el daño que estaba causando a la industria de computadores de los EE.UU. (que era incapaz de vender al resto del mundo versiones seguras de sus visualizadores web, sistemas operativos distribuidos y muchos otros productos). En enero del año 2000 el gobierno de los EE.UU. introdujo una nueva política [www.bxa.doc.gov] con la intención de permitir a los vendedores de software de los EE.UU. exportar software que incorporara encriptación fuerte, y es de esperar que conduzca a una simplificación del mercado internacional del software criptográfico. Las nuevas regulaciones permiten la exportación de productos software que incorporen claves de encriptación de hasta 64 bits y hasta 1.024 en el caso de claves públicas para su uso en firmas e intercambios de claves. La exportación de este software requiere el «visto bueno» del gobierno, aunque también se permite la exportación sin restricciones de código fuente criptográfico, aparentemente sin restricciones en la longitud de las claves.

Otras iniciativas políticas han tratado de mantener el control sobre el uso de la criptografía presentando leyes que insisten en la inclusión de agujeros o puertas falsas a disposición, únicamente, de las agencias de investigación criminal y de seguridad. Tales propuestas emanan de la percepción de que los canales de comunicación secretos pueden ser de mucha utilidad a criminales de todo tipo. Históricamente, los gobiernos han tenido los medios de interceptar y analizar las comunicaciones públicas. La criptografía fuerte altera radicalmente esa situación. Pero estas propuestas para legislar con el fin de prevenir el uso de criptografía fuerte no comprometida han tenido una fuerte oposición desde los grupos de ciudadanos para las libertades civiles, preocupados por su impacto en los derechos a la privacidad de los ciudadanos. Hasta el momento, no se ha adoptado ninguna de estas propuestas legislativas, pero los esfuerzos políticos continúan y la eventual introducción de un marco legal para el uso de la criptografía parece algo inevitable.

7.6. CASOS DE ESTUDIO: NEEDHAM-SCHROEDER, KERBEROS, SSL Y MILLICENT

Los protocolos de autenticación publicados originalmente por Needham y Schroeder [1978] son el núcleo de muchas técnicas de seguridad. Se presentan con detalle en la Sección 7.6.1. Una de las aplicaciones más importantes de su protocolo de autenticación de clave secreta es el sistema Kerberos [Steiner y otros 1988], que es sujeto de nuestro segundo caso de estudio (Sección 7.6.2). Kerberos se diseñó para proporcionar autenticación entre clientes y servidores en redes que constituían un único dominio de administración (intranets).

También presentamos dos casos de estudio que describen protocolos de seguridad en el nivel de aplicación y que son importantes para el comercio electrónico. El primer caso de estudio del nivel de aplicación trata con el protocolo de Capa de Sockets Segura (*Secure Sockets Layer*, SSL). Se diseñó específicamente para cumplimentar la necesidad de transacciones seguras. Hoy en día está soportada por la mayoría de los visualizadores y servidores web, y se emplea en la mayoría de las transacciones comerciales que tienen lugar vía Web. Nuestro segundo caso de estudio describe el protocolo Millicent, diseñado específicamente para las necesidades de un método de pago para micro-transacciones.

7.6.1. EL PROTOCOLO DE AUTENTICACIÓN DE NEEDHAM Y SCHROEDER

Los protocolos aquí descritos se desarrollaron en respuesta a la necesidad de medios seguros para tratar con claves (y passwords) en una red. En el momento de su publicación [Needham y Schroeder 1978], los servicios de archivos en red estaban justo empezando y había una necesidad urgente de mejores formas de administrar la seguridad de las redes locales.

En redes integradas con fines de administración, esta necesidad se cumplimenta mediante el uso de un servicio de claves seguras que envía claves de sesión bajo la forma de desafíos (véase la Sección 7.2.2). Éste es el objetivo del protocolo de clave secreta desarrollado por Needham y Schroeder. En la misma publicación, Needham y Schroeder también confeccionan un protocolo basado en el uso de claves públicas para autenticación y distribución de claves y que no depende de la existencia de servidores de claves seguros, y de este modo es más adecuado para su empleo en redes con muchos dominios de administración independientes, como Internet. No describimos aquí la versión de clave pública, pero el protocolo SSL que se describe en la Sección 7.6.3 es una variación de éste.

Needham y Schroeder propusieron una solución a la autenticación y la distribución de claves basada en un *servidor de autenticación* que proporciona claves secretas a sus clientes. El trabajo del servidor de autenticación es proporcionar una forma segura por la que pares de procesos obtengan claves compartidas. Para hacer esto, debe comunicarse con sus clientes usando mensajes encriptados.

◇ **Needham y Schroeder con claves secretas.** En su modelo, un proceso que represente a un principal A que desee iniciar una comunicación segura con otro proceso que represente al principal B puede obtener una clave con este fin. El protocolo se describe para dos procesos arbitrarios A y B, pero en sistemas cliente-servidor, A es cualquier cliente que inicie una secuencia de solicitudes hacia algún servidor B. La clave le viene dada a A de dos formas, una que A puede usar para encriptar los mensajes que envíe a B y otra que pueda transmitir de modo seguro a B. (La última se encuentra encriptada en una clave que es conocida por B pero no por A, de modo que B puede desencriptarla y no se compromete durante la transmisión.)

El servidor de autenticación S mantiene una tabla que contiene un nombre y una clave secreta para cada principal conocido por el sistema. La clave secreta se emplea sólo para autenticar los procesos del cliente frente al servidor de autenticación y para transmitir mensajes de modo seguro entre procesos cliente y el servidor de autenticación. Nunca se muestra a terceras partes y se transmite por la red como máximo una sola vez, cuando se genera. (Idealmente, sólo se debería transmitir por otros mecanismos, como en papel o de forma verbal, evitando cualquier exposición a la red.) Una clave secreta sería el equivalente de la contraseña (*password*) que se emplea para autenticar usuarios en los sistemas centralizados. Para los principales humanos, el nombre conocido por el servicio de autenticación es su «nombre de usuario» y la clave secreta es su password. El proceso cliente que actuará como representante del usuario le pedirá ambos.

El protocolo se basa en la generación y transmisión de tickets por el servidor de autenticación. Un ticket es un mensaje encriptado que contiene una clave secreta para su uso en la comunicación entre A y B. En la Figura 7.15 tabulamos los mensajes del protocolo de clave secreta de Needham y Schroeder. El servidor de autenticación es S.

N_A y N_B son *ocasiones*. Una ocasión es un valor entero que se añade a un mensaje para demostrar su frescura. Las ocasiones se utilizan una sola vez y se generan bajo demanda. Por ejemplo, se pueden generar ocasiones mediante una secuencia de valores enteros o leyendo el reloj de la máquina emisora.

Si el protocolo se completa con éxito, tanto A como B pueden estar seguros de que cualquier mensaje encriptado con K_{AB} que se recibe proviene del otro, y que cualquier mensaje encriptado con K_{AB} que se envía puede ser entendido por el otro y sólo por éste (además de S, que se supone que es de fiar). Esto es así porque los únicos mensajes que han sido enviados conteniendo K_{AB} fueron encriptados en la clave secreta de A o en la de B.

Encabezado	*Mensaje*	*Anotaciones*
1. A → S:	A, B, N_A	A solicita a S una clave para comunicarse con B.
2. S → A:	$\{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_B}\}_{K_A}$	S retorna un mensaje encriptado en la clave secreta de A, que contiene una clave nueva K_{AB}, y un «ticket» encriptado en la clave secreta de B. La ocasión N_A demuestra que el mensaje fue enviado en respuesta a la anterior. A cree que S envió el mensaje porque sólo S conoce la clave secreta de A.
3. A → B:	$\{K_{AB}, A\}_{K_B}$	A envía el «ticket» a B.
4. B → A:	$\{N_B\}_{K_{AB}}$	B desencripta el ticket y usa la nueva clave K_{AB} para encriptar otra ocasión N_B.
5. A → B:	$\{N_B - 1\}_{K_{AB}}$	A demuestra a B que fue el emisor del mensaje anterior devolviendo una transformación acordada sobre N_B.

Figura 7.15. Protocolo de autenticación de clave secreta de Needham-Schroeder.

Existe una debilidad en este protocolo en la que B no encuentre razones para creer que el mensaje 3 sea reciente. Un intruso que se las apañe para obtener la clave K_{AB} y que haga una copia del ticket y del autenticador $\{K_{AB}, A\}_{K_B}$ (cualquiera de los cuales podría haberse dejado en un área de almacenamiento desprotegida debido a un descuido o a un programa de cliente fallido que se ejecutara bajo la autoridad de A), podría utilizarlos para iniciar un intercambio posterior con B, suplantando a A. Para que ocurra este ataque debe comprometerse un antiguo valor de K_{AB}; en terminología de hoy en día, Needham y Schroeder no incluyeron esta posibilidad en su lista de amenazas, y la opinión consensuada es que uno debería hacerlo. La debilidad puede remediarse añadiendo una ocasión o una marca temporal mensaje 3, de modo que se convierte en: $\{K_{AB}, A, t\}_{K_{Bpub}}$. B desencripta este mensaje y comprueba que t es reciente. Ésta es la solución adoptada en Kerberos.

7.6.2. KERBEROS

Kerberos se desarrolló en el MIT en los años ochenta [Steiner y otros 1988] para proporcionar un catálogo de medios de autenticación y seguridad para su uso en la red de computación del campus en el MIT y otras intranets. Ha soportado varias revisiones y mejoras a la luz de la experiencia y la realimentación de las organizaciones de usuarios. La versión 5 de Kerberos [Neumann y T'so 1994], que describimos aquí, está en vías de ser un estándar Internet (véase el RFC 1510 [Kohl y Neuman 1993]) y es usada hoy en día por muchas compañías y universidades. El código fuente de la implementación de Kerberos está disponible desde el MIT [web.mit.edu I]; se incluye en el Entorno de Computación Distribuida de OSF (*Distributed Computing Environment*, DCE) [OSF 1997] y en el sistema operativo Windows 2000 como servicio de autenticación por defecto [www.microsoft.com II]. Se ha propuesto su extensión para incorporar el uso de certificados de clave pública para la autenticación inicial de principales (paso A de la Figura 7.16) [Neuman y otros 1999].

La Figura 7.16 muestra la arquitectura del proceso. Kerberos trata con tres clases de objetos de seguridad:

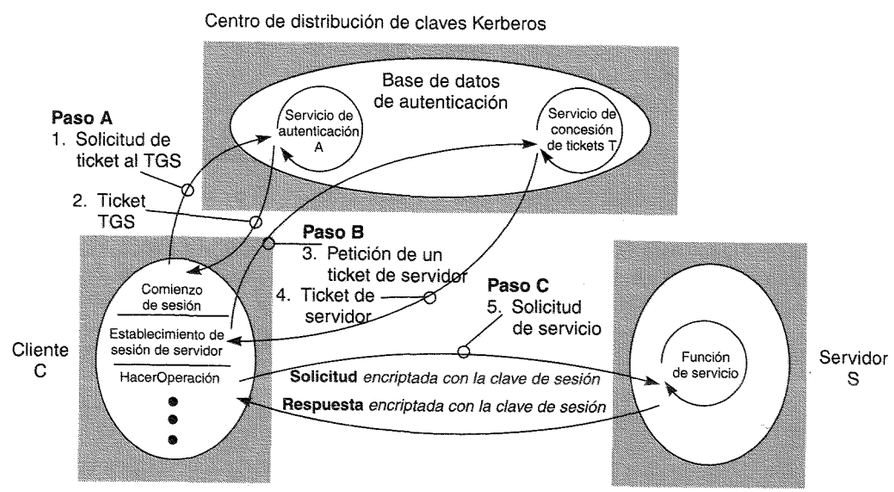


Figura 7.16. Arquitectura del sistema Kerberos.

Ticket: una palabra enviada a un cliente por el servicio de concesión de tickets para su presentación a un servidor particular, y que verifica que el emisor se ha autenticado recientemente frente a Kerberos. Los tickets incluyen un tiempo de expiración y una clave de sesión generada en este momento para su uso por el cliente y el servidor.

Autenticación: una palabra construida por un cliente y enviada al servidor para demostrar la identidad del usuario y la actualidad de cualquier comunicación con un servidor. Un autenticador sólo puede usarse una vez. Éste contiene el nombre del cliente y una marca temporal y se encripta con la clave de sesión apropiada.

Clave de sesión: una clave secreta generada aleatoriamente por Kerberos y enviada a un cliente para su uso en la comunicación con un servidor particular. La encriptación no es obligatoria para cualquier comunicación con los servidores; la clave de sesión se emplea para encriptar la comunicación con aquellos servidores que la pidan y para encriptar todos los autenticadores (véase más arriba).

Los procesos clientes deben poseer un ticket y una clave de sesión para cada servidor que empleen. Sería impracticable proporcionar un ticket y una clave nuevos para cada interacción cliente-servidor, de modo que la mayoría de los tickets se conceden a los clientes con un período de vida de varias horas, de modo que puedan utilizarse para interactuar con un servidor particular hasta que expiran.

Un servidor Kerberos es conocido como Centro de Distribución de Claves (*Key Distribution Centre*, KDC). Cada KDC ofrece un Servicio de Autenticación (*Authentication Service*, AS) y un Servicio de Concesión de Tickets (*Ticket-Granting Service*, TGS). Al presentarse al sistema, los usuarios son autenticados por el AS utilizando una variación segura frente a la red del método de clave de acceso, y el proceso del cliente que actúa en representación del usuario recibe un *ticket de concesión de tickets* y una clave de sesión para comunicarse con el TGS. Posteriormente, el proceso cliente original y sus descendientes pueden usar el ticket de concesión de tickets y las claves de sesión para los servicios especificados desde el TGS.

En Kerberos se sigue bastante fielmente el protocolo de Needham y Schroeder, donde los valores de tiempo (enteros que representan una fecha y hora) se usan como ocasiones. Esto sirve a dos objetivos:

- Protegerse de la repetición de mensajes antiguos interceptados en la red o rechazar los viejos tickets encontrados en algún lugar de la memoria de máquinas donde el usuario autorizado ha cerrado la sesión (en Needham y Schroeder se utilizaban ocasiones para lograr esto).
- Aplicar un tiempo de vida a los tickets, permitiendo que el sistema revoque los derechos de acceso de los usuarios cuando, por ejemplo, dejan de ser usuarios autorizados del sistema.

A continuación describimos los protocolos de Kerberos en detalle, utilizando la notación definida al final de la página. Primero, describimos el protocolo por el que el cliente obtiene un ticket y una clave de sesión para acceder al TGS.

Un ticket de Kerberos tiene un período de validez fijo que comienza en el instante t_1 y acaba en el instante t_2. Un ticket para que un cliente C acceda al servidor S toma la forma:

$\{C, S, t_1, K_{CS}\}_{K_S}$, que denotaremos como $\{ticket(C, S)\}_{K_S}$

El nombre del cliente se incluye en el ticket para evitar un posible uso por impostores, como veremos más adelante. El número de paso y de mensaje de la Figura 7.16 se corresponde con la descripción tabulada que sigue. Tenga en cuenta que el mensaje 1 no está encriptado y no incluye la clave de acceso de C. Contiene una ocasión que se emplea para comprobar la validez de la respuesta.

A. Obtención de una clave de sesión Kerberos y un ticket TGS, una vez por inicio de sesión

Encabezado	*Mensaje*	*Anotaciones*
1. C → A: Solicitud de un ticket TGS	C, T, n	El cliente C pide al servidor de autenticación Kerberos A que proporcione un ticket para comunicarse con el servicio de concesión de tickets T.
2. A → C: Clave de sesión TGS y ticket	$\{K_{CT}, n\}_{K_C}$, $\{ticket(C, T)\}_{K_T}$ conteniendo C, T, t_1, t_2, K_{CT}	A retorna un mensaje que contiene un ticket encriptado en su clave secreta y una clave de sesión para que C la use con T. La inclusión de la ocasión n encriptada en K_{CS} muestra que el mensaje proviene del receptor del mensaje 1, que debe conocer K_C.

Notación:

A	Nombre del servicio de autenticación Kerberos.	n	Una ocasión.
T	Nombre del servicio de concesión de tickets Kerberos.	t	Una marca temporal.
C	Nombre del cliente.	t_1	Instante de comienzo de la validez de un ticket.
		t_2	Instante de finalización de la validez de un ticket.

Al mensaje 2 se le denomina también «desafío» porque se le presenta al peticionario información que sólo tiene utilidad si conoce la clave secreta de C, K_C. Un impostor que intentara suplantar a C enviando el mensaje 1 no iría muy lejos, porque no puede desencriptar el mensaje 2. Para los principales que sean usuarios, K_C es una versión confusa de la clave de acceso del usuario. El proceso cliente pedirá que el usuario introduzca su clave de acceso e intentará desencriptar el mensaje 2 con ella. Si el usuario da la contraseña correcta, el proceso cliente obtendrá la clave de sesión K_{CT} y un ticket válido para el servicio de concesión de tickets; si no, obtendrá un texto incoherente. Los servidores tienen claves secretas de su propiedad, solamente conocidas por el proceso servidor relevante y el servidor de autenticación.

Cuando se ha obtenido un ticket válido del servicio de autenticación, el cliente C podrá usarlo para comunicarse con el servicio de concesión de tickets para obtener tickets para otros servidores cuantas veces quiera mientras no expire el ticket. Así para obtener un ticket para cualquier servidor S, C construye un autenticador encriptado en K_{CT} de la forma:

$\{C, t\}_{K_{CT}}$, que denotaremos como $\{aut(C)\}_{K_{CT}}$ y envía la solicitud a T:

B. Obtener un ticket para un servidor S, una vez por sesión cliente-servidor

3. C → T: Pedir ticket para el servicio S	$\{auth(C)\}_{K_{CT}}$, $\{ticket(C, T)_{K_T}, S, n$	C solicita al servidor de concesión de tickets T un ticket para comunicarse con otro servidor S.
4. T → C: Servicio del ticket	$\{K_{CS}, n\}_{K_{CT}}$, $\{ticket(C, S)\}_{K_S}$	T comprueba el ticket. Si es válido, T genera una nueva clave aleatoria de sesión K_{CS} y la devuelve junto a un ticket para S (encriptado en la clave secreta del servidor K_S).

C está listo entonces para enviar mensajes de solicitud al servidor, S:

C. Enviar una petición al servidor con un ticket		
5. C → S: Solicitud de servicio	$\{auth(C)\}_{K_{CS}}$, $\{ticket(C, S)\}_{K_S}$, *request*, *n*	C envía el ticket a S con un nuevo autenticador generado para C y una petición. La petición podría estar encriptada con K_{CS} si se requiere confidencialidad.

Para que el cliente esté seguro de la autenticidad del servidor, S debiera devolver la ocasión *n* a C. (Para reducir el número de mensajes requerido, se podría incluir en los mensajes que contienen la respuesta del servidor a la solicitud):

D. Autenticar el servidor (opcionalmente)		
6. S → C: Autenticación del servidor	$\{n\}_{K_{CS}}$	(Opcional): S envía la ocasión a C, encriptada con K_{CS}.

◇ **Aplicación de Kerberos.** Kerberos se desarrolló para su uso en el Proyecto Athena en el MIT; una infraestructura de computación en red por todo el campus, para las titulaciones de licenciatura e ingeniería, con muchas estaciones de trabajo y servidores que dan servicio a más de 5.000 usuarios. El entorno es tal que no es posible presuponer ni la confianza en los clientes ni la seguridad de la red y las máquinas que ofrecen servicios de red; por ejemplo, las estaciones de trabajo no están protegidas contra la instalación de software de sistema desarrollado por los usuarios, y las máquinas servidoras (diferentes al servidor Kerberos) no están aseguradas necesariamente contra una interferencia física con su configuración software.

Kerberos proporciona virtualmente toda la seguridad del sistema Athena. Se emplea para autenticar usuarios y otros principales. La mayoría de los servidores que operan en la red han sido extendidos para solicitar un ticket de cada cliente al comienzo de cada interacción cliente-servidor. En éstos se incluyen el almacenamiento de archivos (NFS y el sistema de archivos Andrew), correo electrónico, entrada remota en el sistema e impresión. Las claves de acceso de los usuarios son conocidas sólo por cada uno de ellos y por el servicio de autenticación Kerberos. Los servicios tienen claves secretas conocidas sólo para Kerberos y los servidores que proporcionan el servicio.

Ahora, describiremos la forma en que se aplica Kerberos a la autenticación de los usuarios al entrar en el sistema. Su empleo para asegurar el servicio de archivos NFS se describirá en el Capítulo 8.

◇ **Entrada en el sistema con Kerberos.** Cuando un usuario entra en una estación de trabajo, el programa de bienvenida envía el nombre del usuario al servicio de autenticación de Kerberos. Si el usuario es conocido al servicio de autenticación éste responde con una clave de sesión y una ocasión encriptada en su clave de acceso y un ticket para el TGS. El programa de bienvenida, ahora, intenta desencriptar la clave de sesión y la ocasión empleando la clave de acceso que deberá introducir el usuario en respuesta a una petición. Si la clave de acceso es correcta, el programa de bienvenida obtiene la clave de sesión y la ocasión. Entonces comprueba la ocasión y almacena la clave de sesión junto al ticket para ser usado posteriormente cuando se comunique con el TGS. En este momento, el programa de bienvenida puede borrar la clave de acceso del usuario de su memoria, dado que el ticket ya sirve para autenticar al usuario. En este momento se inicia la sesión del usuario en la estación del sistema. Observe que la clave del usuario nunca se expone al escrutinio en la red; se retiene en la estación de trabajo y se borra de la memoria tan pronto como se ha empleado.

◇ **Acceso a los servidores en Kerberos.** Cuando un programa en ejecución en una estación de trabajo solicita el acceso a un nuevo servicio, pide un ticket para este servicio al servicio de concesión de tickets. Por ejemplo, cuando un usuario UNIX desea acceder a un computador remoto, el mandato *rlogin* de la estación del usuario obtendrá un ticket del servicio de concesión de tickets de Kerberos para acceder al servicio de red *rlogind*. El programa del mandato *rlogin* envía el ticket, junto con un nuevo autenticador, en la petición al proceso *rlogind* del computador donde el usuario desea acceder. El programa *rlogind* desencripta el ticket con la clave secreta del servicio *rlogin* y comprueba la validez del ticket (esto es, que no haya expirado la validez del ticket). Las máquinas servidoras deben tener cuidado de almacenar sus claves secretas en un almacén que no sea accesible a los intrusos.

El programa *rlogind* usa entonces la clave de sesión incluida en el ticket para desencriptar el autenticador y comprueba que el autenticador es reciente (los autenticadores sólo pueden usarse una vez). Una vez que el programa *rlogind* se da por satisfecho, ticket y autenticador válidos, no hay necesidad de comprobar el nombre y la clave de acceso del usuario, dado que la identidad del usuario es conocida por el programa *rlogind* y consecuentemente se inicia la sesión del usuario remoto.

◇ **Implementación de Kerberos.** Kerberos se implementa como un servidor que se ejecuta sobre una máquina remota. Se proporciona un conjunto de bibliotecas para uso de las aplicaciones clientes y servicios. Se emplea el algoritmo de encriptación DES, pero se implementa en un módulo separado que puede reemplazarse fácilmente.

El servicio Kerberos es escalable; el mundo se divide en dominios de autoridades de autenticación separados, denominados *esferas* (*realms*), cada una con su propio servidor Kerberos. La mayoría de los principales están registrados en una sola esfera, pero los servidores de concesión de tickets de Kerberos están registrados en todas las esferas. Los principales pueden autenticarse a sí mismos frente a los servidores en otros dominios a través del servidor de concesión de tickets local.

Dentro de un dominio, puede haber varios servidores de autenticación, cada uno de los cuales tiene copias de la misma base de datos de autenticación. La base de datos de autenticación se replica mediante una simple técnica maestro-esclavo. Las actualizaciones se aplican a la copia maestra por un único servicio de Administración de Base de Datos Kerberos (*Kerberos Database Management*, KDBM) que se ejecuta sólo en la máquina maestra. El KDBM maneja las solicitudes de cambio de claves de acceso de los usuarios y las peticiones de los administradores para añadir o borrar principales y para cambiar sus claves de acceso.

Para hacer transparente este esquema a los usuarios, el tiempo de vida de los tickets TGS debería ser tan largo como la sesión de usuario más larga posible, dado que el uso de un ticket expirado trae consigo el rechazo de las solicitudes de servicio, y el único remedio es que el usuario re-autentifique su entrada y pida nuevos tickets de servidor para todos los servicios en uso. En la práctica, el período de vida de un ticket suele ser de 12 horas.

◇ **Críticas a Kerberos.** El protocolo de la versión 5 de Kerberos que se ha descrito contiene varias mejoras diseñadas para resolver las críticas de versiones anteriores [Bellovin y Merritt 1990, Burrows y otros 1990]. La crítica más importante de la versión 4 consiste en que las ocasiones usadas en los autenticadores se implementan como marcas temporales, y la protección contra el re-envío de autenticadores depende de una cierta, aunque relajada, sincronización del reloj de los clientes y los servidores. En consecuencia si se empleara un protocolo de sincronización para acoplar ligeramente la sincronía de los relojes de clientes y servidores; este mismo protocolo debería asegurarse contra ataques de seguridad. Véase el Capítulo 10 para información sobre protocolos de sincronización.

La definición de protocolo de la versión 5 permite que las ocasiones de los autenticadores se implementen como marcas temporales o como una secuencia de números. En ambos casos, se re-

quiere que sean únicas, y que los servidores mantengan una lista de ocasiones recientemente recibidas de cada cliente para comprobar que no se repiten. Éste es un requisito de implementación poco conveniente y difícil de garantizar en caso de fallos. Kehne y otros [1992] han publicado una propuesta de mejora al protocolo Kerberos que no emplea relojes sincronizados.

La seguridad de Kerberos depende de la limitación de los tiempos de vida de sesión; el período de validez de los tickets TGS se limita habitualmente a unas pocas horas; debe escogerse un período suficientemente largo como para evitar las inconvenientes interrupciones del servicio pero suficientemente corto como para asegurar que los usuarios desregistrados o degradados de categoría no puedan usar los recursos durante más tiempo. Esto puede ocasionar dificultades en algunos entornos comerciales, dado que el consiguiente requerimiento al usuario para que vuelva a introducir los detalles de autenticación en un punto arbitrario de su interacción con el sistema podría tener consecuencias en la aplicación.

7.6.3. SEGURIDAD DE TRANSACCIONES ELECTRÓNICAS CON SOCKETS SEGUROS

El protocolo de Capa de Socket Segura (*Secure Socket Layer*, SSL) fue desarrollado originalmente por Netscape Corporation [Netscape 1996] y se propuso como un estándar diseñado específicamente para resolver las necesidades descritas anteriormente. Bajo el nombre de protocolo de Seguridad de Capa de Transporte (*Transport Layer Security*, TLS), descrito en el RFC 2246 [Dierk y Allen 1999], podemos hallar una versión extendida de SSL adoptada como estándar Internet. SSL está soportada en la mayoría de los visualizadores y se usa ampliamente en el comercio en Internet. Sus características más importantes son:

◇ **Encriptación y algoritmos de encriptación negociables.** En una red abierta no deberíamos presuponer que todos los interlocutores emplean el mismo software de cliente o que todo el software de cliente y servidor incluyen un algoritmo de encriptación concreto. De hecho, las leyes de algunos países intentan restringir el uso de ciertos algoritmos de encriptación a ciertos países. SSL se ha diseñado para que puedan negociarse los algoritmos de encriptación y autenticación entre los procesos al principio y al final de la conexión durante el saludo (*handsake*) inicial. Pudiera ocurrir que no tuvieran suficientes algoritmos en común, y en este caso el intento de conexión fallaría.

◇ **Autoarranque de la comunicación segura.** Para cumplir con la necesidad de comunicación segura sin una negociación previa y sin ayuda de tercera partes, el canal seguro se establece usando un protocolo similar al esquema híbrido descrito con anterioridad. Se emplea una comunicación no encriptada en los intercambios iniciales, después criptografía de clave pública y finalmente se conmuta a criptografía de clave privada una vez que se ha establecido una clave de secreta compartida. El cambio a cada etapa es opcional y viene precedido por una negociación.

Consiguientemente el canal seguro es completamente configurable, permitiendo que la comunicación en ambas direcciones esté encriptada y autenticada, aunque no es imprescindible, de modo que los recursos computacionales requeridos no se consumen en realizar encriptaciones innecesarias.

Los detalles del protocolo SSL están publicados y estandarizados, además de haber diversas bibliotecas software y herramientas disponibles para apoyar la programación [Hirsch 1997, www.openssl.org], algunas de ellas de dominio público. Ha sido incorporado en un amplio rango de software de aplicación, y su seguridad ha sido verificada por revisores independientes.

SSL consta de dos capas (véase la Figura 7.17), una capa de Protocolo de Registro SSL, que implementa un canal seguro, encriptando y autenticando mensajes transmitidos a través de cualquier protocolo orientado a conexión; y una capa de handsake, conteniendo el protocolo de

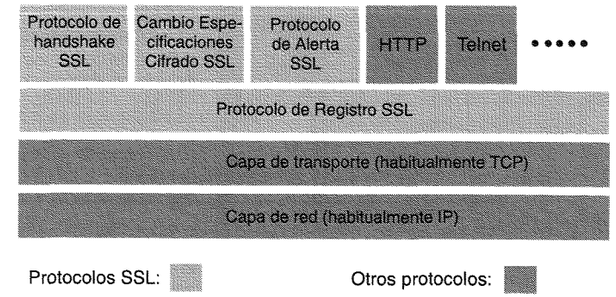


Figura 7.17. Pila de protocolos de SSL.
(Las figuras en esta sección están basadas sobre los diagramas en Hirsch [1997] y han sido publicadas con el permiso de Frederick Hirsch.)

handsake y otros dos protocolos relacionados que establecen y mantienen una sesión SSL (esto es, un canal seguro) entre un cliente y un servidor. Ambos están implementados habitualmente mediante bibliotecas software en el nivel de aplicación en el cliente y en el servidor. El protocolo de registro de SSL es una capa de nivel de sesión; puede emplearse para transportar datos del nivel de aplicación de modo transparente entre un par de procesos mientras garantiza su privacidad, integridad y autenticidad. Éstas son exactamente las propiedades que especificábamos para los canales seguros en nuestro modelo de seguridad (véase la Sección 2.3.3), pero en SSL hay opciones para que los participantes de la comunicación escojan si desplegar o no la desencriptación y autenticación de los mensajes en cada dirección. A cada sesión segura se le da un identificador y cada participante puede almacenar los identificadores de sesión en una caché para su uso subsiguiente, evitando la sobrecarga de establecer una nueva sesión cuando se requiere otra sesión segura con el mismo compañero.

SSL se usa ampliamente para añadir una capa de comunicación segura a los protocolos del nivel de aplicación existentes. El protocolo se publicó por primera vez en 1994, al que se añadieron revisiones hasta producir dos versiones posteriores. SSL 3.0 es el tema de una propuesta de estandarización [Netscape 1996]. Su uso más extendido es, probablemente, la seguridad de las interacciones HTTP en el comercio en Internet, aunque también se usa en otras aplicaciones sensibles a la seguridad. Se encuentra implementado en virtualmente todos los visualizadores y servidores web: el uso del prefijo de protocolo *https*: en un URL inicia el establecimiento de un canal seguro SSL entre un navegador y un servidor web. También se ha empleado para proporcionar implementaciones seguras de Telnet, FTP y muchos otros protocolos de aplicación. SSL es el estándar *de facto* para su uso en aplicaciones que requieran canales seguros, y existe un amplio catálogo de implementaciones disponibles, tanto comerciales como de dominio público, con interfaces de programación para CORBA y Java.

El protocolo de handshake para SSL se muestra en la Figura 7.18. El handshake se realiza sobre una conexión existente. Comienza con texto en claro y establece una sesión SSL intercambiando las opciones y parámetros acordados que se necesitan para realizar la encriptación y la autenticación. La secuencia de handshake varía dependiendo de si se requiere, o no, autenticación de cliente y servidor. El protocolo de handshake puede invocarse también más tarde para cambiar la especificación de un canal seguro, por ejemplo, la comunicación podría comenzar con un mensaje de autenticación empleando sólo códigos de autenticación de mensajes. En un punto posterior se puede añadir encriptación. Esto se obtiene efectuando el protocolo de handshake de nuevo para negociar una nueva especificación de encriptado usando el canal existente.

El handshake inicial de SSL es potencialmente vulnerable a los ataques del *hombre entre medias*, como se describe en el escenario 3 de la Sección 7.2.2. Para protegerse de él, la clave pública

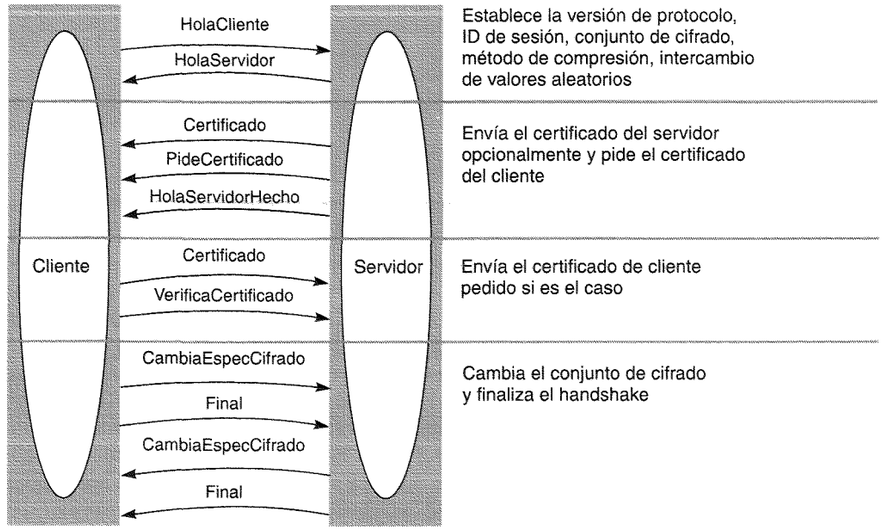


Figura 7.18. Protocolo de handshake de SSL.

usada para verificar el primer certificado recibido podría recibirse por un canal separado; por ejemplo, los navegadores y otros programas de Internet instalados desde CD-ROM pueden incluir un conjunto de claves públicas para algunas autoridades de certificado bien conocidas. Otra defensa para los clientes de servicios bien conocidos se basa en la inclusión del nombre de dominio del servicio en sus certificados de clave pública; los clientes sólo debieran tratar con servicios en los que su dirección IP se corresponda con su nombre de dominio.

SSL soporta una variedad de opciones de funciones criptográficas. El nombre colectivo que recibe es *catálogo de cifrado*. Un catálogo de cifrado incluye una elección única para cada una de las características que se muestran en la Figura 7.19.

Se suelen encontrar ya cargadas cierta variedad usual de catálogos de cifrado, con identificadores estándar tanto en el cliente como en el servidor. Durante el handshake, el servidor ofrece al cliente una lista de los identificadores de catálogos de cifrado disponibles, y el cliente responde seleccionando uno de ellos (o dando una indicación de error si no coincide con ninguno). En esta etapa también deben acordar (opcionalmente) un método de compresión y un valor de arranque aleatorio para las funciones de encriptación de bloque CBC (véase la Sección 7.3).

A continuación, los participantes deben autenticarse opcionalmente uno a otro intercambiando certificados de clave pública en el formato X.509. Estos certificados pueden obtenerse de una auto-

Componente	*Descripción*	*Ejemplo*
Método de intercambio de clave	El método que se usará para intercambiar una clave de sesión	RSA con certificados de clave pública
Cifrado para la transferencia de datos	El cifrador de bloque o de caudal que se usará para los datos	IDEA
Función de resumen de mensajes	Para crear los códigos de autenticación de mensajes (MAC)	SHA

Figura 7.19. Opciones de configuración del handshake de SSL.

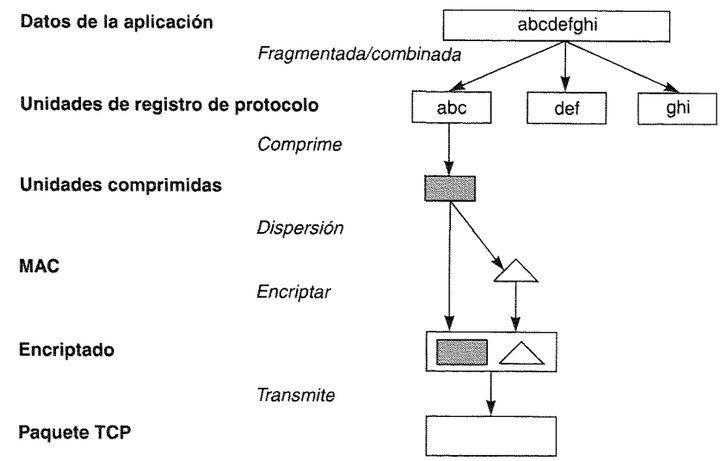


Figura 7.20. Protocolo de registro de seguridad de SSL.

ridad de clave pública o pueden generarse simplemente de modo temporal con este fin. De cualquier manera, debe haber al menos una clave pública para ser usada en la siguiente etapa del handshake.

Un participante genera entonces un *secreto pre-maestro* y lo envía encriptado con la clave pública al otro participante. Un secreto pre-maestro es un valor aleatorio grande con el que ambos participantes generan las dos claves de sesión (llamadas claves de *escritura*) para encriptar los datos en cada dirección y los secretos de autenticación de mensajes que habrá que usar para la autenticación de los mensajes. Cuando se ha hecho esto comienza una sesión segura. Ésta se inicia mediante los mensajes *CambiaEspecCifrado* que se intercambian entre los participantes. A los cuales siguen los mensajes *Final*. Una vez que se han intercambiado los mensajes *Final*, toda comunicación posterior se encripta y se firma con las claves acordadas y según el catálogo de cifrado elegido.

La Figura 7.20 muestra la operación del protocolo de registro. Un mensaje a transmitir se fragmenta inicialmente en bloques de un tamaño manejable, y opcionalmente después se comprimen. La compresión no es estrictamente una característica de la comunicación segura, pero se incorpora aquí para sacar provecho del procesamiento masivo de datos que se genera en los algoritmos de encriptación y firma digital. En otras palabras, dentro de la capa SSL de registro se puede establecer conjuntamente un flujo de transformaciones sobre los datos mucho más eficiente que si se realizaran individualmente.

Las transformaciones de encriptado y autenticación de mensajes (MAC) lanzan los algoritmos especificados en el catálogo de cifrado acordado, tal y como se describe en las Secciones 7.3.1 y 7.4.2. Finalmente, el bloque firmado y encriptado se transmite al destinatario mediante la conexión TCP asociada, donde se invierten las transformaciones para producir el bloque de datos originales.

◇ **Resumen.** SSL proporciona una implementación práctica de un esquema de encriptación híbrido con autenticación e intercambio de claves basado en claves públicas. Dado que los cifradores se negocian en la etapa de handshake, no depende de la disponibilidad de ningún algoritmo en particular. Tampoco depende de servicios de seguridad en el momento de establecer la sesión. El único requisito es que los certificados de clave pública enviados por una autoridad sean reconocidos por ambas partes.

Ya que se hicieron públicos tanto el protocolo como una implementación de referencia [Netscape 1996], fue tema de revisión y debate. Se hicieron algunas enmiendas a los primeros diseños, y fue apoyado ampliamente como un valioso estándar. SSL se integra, hoy en día, en la mayoría de los visualizadores y servidores web, así como en otras aplicaciones como Telnet y FTP seguros. Hay disponibles implementaciones comerciales y de dominio público [www.rsasecurity.com, Hirsch 1997, www.openssl.org] en forma de bibliotecas y conectores para navegadores web.

7.6.4. TRANSACCIONES ELECTRÓNICAS DE PEQUEÑO IMPORTE: EL PROTOCOLO MILLICENT

Un requisito de los sistemas de pago seguro en Internet es que sea económico y conveniente para la compra de servicios de importe pequeño y el acceso a recursos electrónicos con precios en el rango de 0,1 a 100 céntimos, por ejemplo para cobrar el acceso a páginas web, transmisiones de correo electrónico o llamadas de teléfono por Internet. La mayoría de los esquemas de pago tales como transacciones de tarjeta de crédito son demasiado caros para estos propósitos.

Aquí describimos la solución que ofrece el esquema Millicent [Glassman y otros 1995]. El esquema emplea la forma simple de una firma digital basada en clave secreta para reducir su coste computacional. Sólo se empleará la encriptación cuando se requiera privacidad.

◇ **Sistemas de pago actual y sus desventajas.** Los autores del protocolo Millicent han resumido las desventajas de los métodos de pago disponibles actualmente como sigue. Todos ellos necesitan seguridad criptográfica, sus costes de comunicación y de cálculo varían pero en cualquier caso son no despreciables.

Tarjetas de crédito: El problema del empleo de tarjetas de crédito para pequeñas transacciones es el alto coste de la transacción que resulta de la necesidad de interaccionar con el sistema central de la compañía, lo cual se agrava por los costes adicionales de las transacciones electrónicas seguras y otras características como la preparación de informes para los clientes.

Los clientes mantienen cuentas con los vendedores: Los clientes necesitan establecer una cuenta con un vendedor antes de la primera transacción. Los costes de la transacción son entonces bajos, pero la sobrecarga inicial tiende a desanimar las transacciones casuales. Un vendedor debe mantener una entrada de cliente en una base de datos de cuentas durante un período bastante largo.

Acumulación de transacciones: Cuando un cliente realiza varias cuentas, el vendedor puede guardar los registros de ellas y facturarle al final del período. Esto es parecido al mantenimiento de cuentas aunque puede evitar ciertos costes iniciales. El vendedor debe mantener registros incluso de los clientes que han realizado una sola compra, y el coste de ésta podría exceder el beneficio.

Dinero digital: Al igual que el dinero convencional, el dinero digital (es decir testigos digitales de un valor, emitidos por un banco o un agente de bolsa) ofrecería un medio eficiente de pago para transacciones pequeñas, pero los desarrolladores del dinero digital deben resolver el problema del *doble gasto*. El doble gasto es una consecuencia del hecho de que el portador de un testigo digital puede realizar cualquier número de copias indetectables. En consecuencia, los testigos deben ser identificables únicamente y deben ser validados como no gastados en el momento del uso. El reto está en disponer un esquema de validación que sea escalable, fiable y económico bajo cualquier circunstancia.

◇ **El esquema Millicent.** El proyecto Millicent ha desarrollado un esquema para la distribución segura y el uso de *vales* (*scrip*): una forma especializada de dinero digital diseñada para su uso en pequeñas transacciones. El interés de Millicent para nuestro caso viene de que emplea varias técnicas de seguridad que se describen en este capítulo con un resultado que, sorprendentemente, difiere de los que hemos visto sobre SSL y otros sistemas de seguridad.

El vale es una forma de dinero digital con valor sólo para un vendedor concreto. Se diseña para ofrecer las siguientes características:

- Tiene un valor sólo para un vendedor específico.
- Sólo se puede gastar una vez.
- Es resistente a las modificaciones y difícil de falsificar.
- Sólo puede gastarlo su auténtico propietario.
- Puede producirse y validarse eficientemente.

El diseño del esquema Millicent es escalable porque cada servidor del vendedor es responsable solamente de validar el vale que ha emitido. Los clientes pueden adquirir vales directamente del vendedor, o de un intermediario que posea vales de muchos vendedores, mediante una transacción comercial.

El vale (*scrip*), la moneda específica de un vendedor presentada por Millicent, se representa mediante fichas digitales con el siguiente formato:

Vendedor	Valor	ID vale	ID cliente	Fecha de expiración	Propiedades	Certificado

El campo de *propiedades* está disponible para los usos que determine el vendedor; por ejemplo, podría incluir el país o el estado donde reside el cliente, de forma que pueda aplicarse la tasa de impuestos apropiada. El *certificado* es una firma digital que protege todos los campos del vale contra su modificación. La firma se produce mediante un método MAC como el que se describe en la Sección 7.4.2. El objetivo del resto de los campos lo describiremos más adelante.

El vale lo generan y distribuyen *brokers*; los brokers, o intermediarios, son servidores que tratan con los vales de modo masivo, relevando a los clientes y a los servidores de algunas de las sobrecargas que implica el uso de vales. Los intermediarios intercambian vales por dinero real, comprando vales (o el derecho de generarlos) a los vendedores, con cierto descuento, y vendiéndolos a los clientes con cargo a una tarjeta de crédito u otro medio de pago. Los clientes pueden comprar vales de varios vendedores desde un solo intermediario, acumulando los cargos y el pago al final de un período.

◇ **Escenario.** Un proceso de transacción de compra electrónica con vales es como sigue: una cliente, Alice, está interesada en comprar un pequeño producto o servicio (por ejemplo una llamada telefónica o una página web) del vendedor Venetia. Alice podría poseer vales adecuados para las transacciones con Venetia que le pudieran haber sobrado de transacciones anteriores; si no, Venetia le daría un URL de un intermediario, que comercia con vales de Venetia (llamémoslos *vales-V*). Alice compra algunos vales-V de Bob.

Alice envía a Venetia una solicitud de compra, agregando un vale-V con un valor suficiente para cubrir la compra que ella desea hacer. Venetia valida el vale; comprobando que su ID-vale no está en su lista de vales gastados y que el ID-cliente del vale es el de Alice. Si el valor es mayor que el pago requerido, Venetia hace un nuevo vale para cubrir la diferencia y se lo envía de vuelta a Alice.

Los intermediarios pueden obtener vales del vendedor de muchas formas. En el caso más simple, Venetia crea vales y los vende a Bob con cierto descuento por ser una transacción masiva. Alternativamente, Venetia otorga una licencia a Bob para manufacturar vales en su lugar. Según este modelo, Venetia no es consciente del vale hasta que lo recibe como pago desde un cliente. De vez en cuando ella contacta con Bob y le envía los identificadores de los vales que ha recibido de los clientes, Bob le paga por ellos con una cierta rebaja.

◇ **Metas.** Millicent se diseñó como un sistema de dinero electrónico que evita las sobrecargas de comunicación y los retrasos asociados a los sistemas centralizados, mientras que proporciona

una seguridad adecuada para prevenir el uso fraudulento. La seguridad de los vales se basa en las firmas digitales; se emplea encriptación para mantener la privacidad sólo cuando sea necesario. Se emplea criptografía de bajo coste en comparación con el coste de las transacciones que se realizan con los vales, de modo que no merezca la pena el beneficio de criptoanalizarlo.

El esquema Millicent funciona aisladamente. Puede implementarse sin apoyo de otros servicios, como un servicio de nombres seguro, dado que no depende de la autenticación externa de la identidad de los clientes y vendedores. En su lugar, Millicent genera y distribuye claves secretas únicas para su uso por las partes de las transacciones con vales.

◇ **Implementación.** La Figura 7.21 muestra una visión general de la arquitectura. Hemos mencionado que los vales vienen firmados para protegerlos de cualquier alteración o falsificación. La firma contenida en el certificado, anexa a cada elemento del vale, se produce cuando se genera el vale, usando un *secreto de vale maestro* como clave de firmado. El método de firmado es exactamente el mismo que el método de firma MAC descrito en la Sección 7.4.2. La clave se añade a los otros campos del vale, y se usa una función de resumen segura tal como MD5 o SHA para producir un resumen de 128 bits, que se convierte en el certificado.

Antes de generar cualquier vale, un vendedor (o un intermediario con licencia de un vendedor) generará un vector de secretos maestros de vale de 64 bits. Cada secreto de esta clase sería suficiente, pero la disponibilidad de varios permite seleccionar uno nuevo cada vez para evitar su descubrimiento por accidente o en caso de un ataque con éxito.

Cada elemento del vale tiene un identificador único. Su valor incluye un *índice de secretos maestros* de 8 bits, que se emplea para seleccionar un secreto maestro del vector de secretos maestros.

Los secretos maestros son retenidos por el vendedor y se emplea uno de ellos para producir un certificado por cada elemento del vale que se genera.

Se han sugerido diversas variaciones para el protocolo Millicent, para ofrecer niveles de seguridad diferentes. En todos ellos el vendedor debe validar cada vale remitido por los clientes, que se describe a continuación.

Para validar elemento de un vale:

1. El vendedor comprueba que no ha sido falsificado o modificado por medio de una comprobación de firma para lo que emplea el secreto maestro relevante (usando el índice de secretos maestros a partir del ID-vale para seleccionarlo del vector de secretos maestros) y lo compara con la firma del certificado.

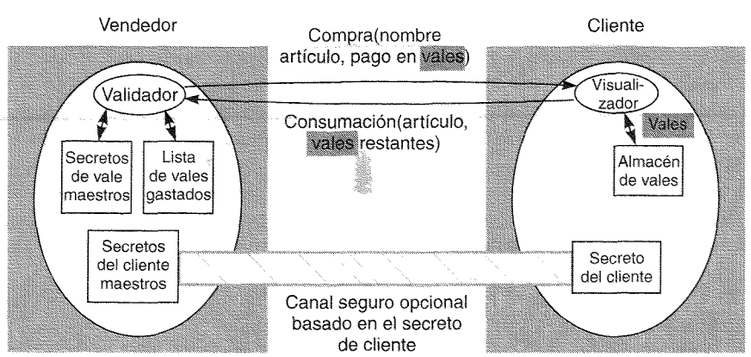


Figura 7.21. Arquitectura de Millicent.

2. El vendedor comprueba que el vale no ha sido ya gastado. Para hacer esto, ella mantiene una lista de los ID y fechas de expiración de todos los vales que ha emitido, con una indicación de si ha sido ya gastado. El campo de la fecha de expiración permite al vendedor borrar el vale caducado de la lista, previniendo así que crezca continuamente. Los clientes deben intercambiar vales viejos por los nuevos antes de que caduquen.

Las transacciones como las que se describen en el escenario anterior se pueden realizar de modo seguro basándonos sólo en la validación. Esto protege al vendedor de la falsificación y del doble gasto, pero no protege al cliente del robo de vales, ni proporciona confidencialidad entre el vendedor y el cliente.

La protección contra el robo requiere que el vendedor o el intermediario que vende vales mantengan un vector de *secretos maestros del cliente* (seleccionados utilizando una parte del ID-cliente) y enviar a cada cliente un *secreto de cliente*. El secreto del cliente se construye añadiendo el secreto maestro del cliente al ID-cliente y aplicando una función de dispersión segura como MD5 al resultado. El secreto del cliente debe transmitirse desde el vendedor al cliente mediante un canal seguro. Se recomienda emplear SSL en cada compra inicial de vales.

Una vez transmitido al cliente, el secreto del cliente puede usarse como una clave secreta compartida entre el cliente y el vendedor. El cliente puede usarla para firmar transacciones y tanto el cliente como el vendedor pueden usarla como clave de encriptación cuando se requiere confidencialidad.

Para prevenir el robo, las transacciones están firmadas por el cliente, usando su secreto del cliente, y el vendedor comprueba que el ID-cliente del vale concuerda con el ID-cliente asociado con el secreto del cliente. Si no, entonces el vale sería gastado por alguien distinto al cliente a quien se vendió; en otras palabras, sería robado.

Para proporcionar confidencialidad, un cliente envía su ID-cliente al vendedor y establece un canal seguro usando un algoritmo de encriptación de clave secreta con la clave del cliente como clave de encriptación. El canal seguro es un medio útil para realizar una transacción en secreto.

Hemos descrito Millicent con cierto detalle porque proporciona un ejemplo del mundo real que ilustra la aplicación de muchas de las técnicas de seguridad que se describen en este capítulo. El sistema Millicent es uno de los diversos esquemas de dinero electrónico que se han desarrollado para su uso en comercio electrónico. Fue desarrollado originalmente en Digital Systems Research Center, en Palo Alto, California y lo vende Compaq Corporation [www.millicent.com].

7.7. RESUMEN

Los ataques a la seguridad son parte de la realidad de los sistemas distribuidos. Es esencial proteger los canales e interfaces de comunicación de cualquier sistema que trate con información que sea susceptible de ser atacada. El correo personal, el comercio electrónico y otras transacciones financieras son ejemplos de tal tipo de información. Los protocolos de seguridad se diseñan cuidadosamente para evitar trampas. El diseño de los sistemas seguros parte de un listado de premisas de ataque y un conjunto de «peores casos posibles».

Los mecanismos de seguridad se basan en la criptografía de clave pública y de clave secreta. Los algoritmos criptográficos disfrazan los mensajes de forma que no puede revertirse el proceso sin el conocimiento de la clave de desencriptación. La criptografía de clave secreta es simétrica: la misma clave sirve tanto para la encriptación como para la desencriptación. Si dos partes comparten una clave secreta, podrán intercambiar información encriptada sin riesgo de que sea desvelada o modificada y con garantías de autenticidad.

La criptografía de clave pública es asimétrica: se utilizan claves separadas para la encriptación y la desencriptación, y el conocimiento de una de ellas no implica el de la otra. Una de ellas se

hace pública, de modo que cualquiera pueda enviar mensajes seguros al propietario de la correspondiente clave privada y permitiendo al poseedor de la clave privada firmar mensajes y certificados. Los certificados pueden servir como credenciales para el uso de recursos protegidos.

Los recursos se protegen mediante mecanismos de control de acceso. Los esquemas de control de acceso asignan derechos a principales (esto es, los propietarios de las credenciales) para realizar operaciones sobre objetos y colecciones de objetos distribuidos. Se pueden mantener los derechos en listas de control de acceso (ACL) asociadas a conjuntos de objetos o pueden mantenerse en manos de los principales bajo la forma de habilitaciones: claves infalsificables de acceso a conjuntos de recursos. Las habilitaciones son una forma conveniente de delegación de derechos de acceso pero son difíciles de revocar. Los cambios en una ACL tienen efecto inmediatamente, revocando los derechos de acceso previos, pero son más difíciles y costosas de manejar que las habilitaciones.

Hasta hace muy poco tiempo, el algoritmo de encriptación DES era el esquema de encriptación simétrica más utilizado, pero sus claves de 56 bits ya no son seguras contra ataques por fuerza bruta. La versión triple de DES proporciona una fuerza de 112 bits, que es segura, pero otros algoritmos como IDEA (con claves de 128 bits) son mucho más rápidos y proporcionan igual o mayor robustez.

RSA es el esquema de encriptación asimétrica más utilizado. Por seguridad contra ataques por factorización, debiera usarse con claves de 768 bits o más. Los algoritmos de clave pública (asimétrica) son sobrepasados en cuanto a prestaciones por los algoritmos de clave secreta (simétrica) en varios órdenes de magnitud, de modo que tan sólo se usan en protocolos híbridos como SSL, para el establecimiento de canales seguros que posteriormente usen claves compartidas para los siguientes intercambios.

El protocolo de autenticación Needham-Schroeder fue el primer protocolo práctico de seguridad de propósito general, y aún proporciona el fundamento de muchos sistemas prácticos. Kerberos es un esquema bien diseñado para la autenticación de usuarios y la protección de servicios dentro de una sola organización. Kerberos se basa en Needham-Schroeder y la criptografía simétrica. SSL es el protocolo de seguridad diseñado, y ampliamente usado, para el comercio electrónico. Es un protocolo flexible para el establecimiento y el uso de canales seguros y se basa en la criptografía simétrica y en la asimétrica. Millicent es una implementación de bajo coste para una forma especial de «dinero electrónico» de bajo coste diseñado para transacciones de pequeño importe.

EJERCICIOS

7.1. Describa algunas de las políticas de seguridad de su organización. Expréselas en términos que pudieran ser implementados en un sistema computerizado de bloqueo de puertas.

7.2. Describa algunas de las formas en las que es vulnerable el correo electrónico a la indiscreción, suplantación, modificación, repetición, y denegación de servicio. Sugiera métodos por los que se pudiera proteger el correo electrónico contra cada una de estas formas de ataque.

7.3. Los intercambios iniciales de claves públicas son vulnerables al ataque del *hombre entre medias*. Describa cuantas defensas pueda contra él.

7.4. PGP se usa ampliamente para asegurar la comunicación de correos electrónicos. Describa los pasos que deberían seguir dos usuarios que usan PGP antes de intercambiar mensajes de correo confidencialmente y con garantías de autenticidad. ¿Qué panorama existe para conseguir hacer la negociación de claves preliminares invisible a los usuarios? (La negociación PGP es un ejemplo de un esquema híbrido.)

7.5. ¿Cómo podría enviarse un correo electrónico a una lista de receptores utilizando PGP o un esquema similar? Sugiera un esquema más simple y rápido cuando la lista se usa frecuentemente.

7.6. La implementación del algoritmo de encriptación simétrico TEA que se muestra en las Figuras 7.8 a 7.10 no es portable entre máquinas de diferentes arquitecturas. Explique por qué. ¿Cómo podría transmitirse un mensaje encriptado usando una implementación de TEA de modo que fuera desencriptado en cualquier otra arquitectura?

7.7. Modifique el programa de aplicación de TEA de la Figura 7.10 para utilizar cifrado por encadenamiento de bloques (CBC).

7.8. Construya una aplicación de cifrado de caudal basada en el programa de la Figura 7.10.

7.9. Estime el tiempo requerido para romper una clave DES de 56 bits por fuerza bruta empleando una estación de trabajo de 500 MIPS (millones de instrucciones por segundo), suponiendo que el bucle interno de un programa de ataque por fuerza bruta contiene una 10 instrucciones por valor de la clave, más el tiempo de encriptación de un texto en claro de 8 bytes (véase la Figura 7.14). Realice el mismo cálculo para una clave IDEA de 128 bits. Extrapole sus cálculos para obtener el tiempo de ataque para un procesador paralelo de 50.000 MIPS (o un consorcio Internet con un poder de procesamiento similar).

7.10. En el protocolo de autenticación de Needham y Schroeder con claves secretas, explique por qué la siguiente versión del mensaje 5 no es segura:

$$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$

8

SISTEMAS DE ARCHIVOS DISTRIBUIDOS

8.1. Introducción
8.2. Arquitectura del servicio de archivos
8.3. Sistema de archivos en red de Sun (NFS)
8.4. Sistema de archivos Andrew
8.5. Avances recientes
8.6. Resumen

Los sistemas de archivos distribuidos soportan la compartición de información en forma de archivos a través de Internet. Un servicio de archivos bien diseñado proporciona acceso a los archivos almacenados en un servidor con prestaciones y fiabilidad semejantes (y en algunos casos mejor) que la de archivos almacenados en discos locales. Un sistema de archivos distribuidos permite a los programas almacenar y acceder a archivos remotos del mismo modo que si fueran locales, permitiendo a los usuarios que accedan a archivos desde cualquier computador en una intranet.

Describimos los diseños de dos sistemas de archivos distribuidos que han sido de uso extendido durante una década o más:

- Sistema de archivos en red de Sun, NFS.
- Sistema de archivos Andrew, AFS.

Estos casos de estudio ilustran un rango de soluciones de diseño para la emulación de una interfaz de un sistema de archivos UNIX con distintos grados de escalabilidad y tolerancia a fallos. Cada una exige alguna desviación de la emulación de la semántica de actualización de archivo de *una copia* de UNIX.

Recientes avances en el diseño de sistemas distribuidos han explotado la conectividad de mayor ancho de banda de las redes de área local conmutadas y los nuevos modos de organización de datos en disco para obtener prestaciones muy altas, tolerancia a fallos y sistemas de archivos altamente escalables.

8.1. INTRODUCCIÓN

En los Capítulos 1 y 2, identificamos el hecho de compartir recursos como un objetivo clave de los sistemas distribuidos. El compartir información almacenada es quizás el aspecto más importante de la compartición de recursos distribuidos. Se obtiene a gran escala en Internet principalmente por el uso de los servidores web, pero los requisitos para compartir en redes de área local e intranet conduce a una necesidad de un tipo de servicio diferente, uno que soporte el almacenamiento persistente de datos y programas de todos los tipos, en nombre de los clientes, y la consiguiente distribución de datos actualizados. El propósito de este capítulo es describir la arquitectura e implementación de sistemas de archivos distribuidos *básicos*. Utilizamos aquí la palabra «básico» para señalar sistemas de archivos distribuidos cuyo propósito principal es emular la funcionalidad de sistemas de archivos no distribuidos para programas cliente ejecutándose en múltiples computadores remotos. Éstos no mantienen múltiples réplicas persistentes de archivos, ni soportan el ancho de banda y las garantías de temporización requeridas para flujos de datos multimedia, cuyos requisitos se revisarán en capítulos posteriores. Los sistemas de archivos distribuidos básicos proporcionan un sustento esencial para la organización de la computación basada en intranets.

Comenzamos con un breve repaso del espectro de los sistemas de almacenamiento distribuidos y no distribuidos. Los sistemas de archivos fueron desarrollados originalmente para sistemas centralizados de computadores y computadores portátiles como una disponibilidad del sistema operativo que proporciona una interfaz de programación adecuada para el almacenamiento en disco. Así, adquirieron posteriormente características tales como control de acceso y mecanismos de bloqueo a archivos que les hicieron útiles para la compartición de datos y programas. Los sistemas de archivos distribuidos soportan la compartición de la información en forma de archivos y recursos hardware que dan soporte al almacenamiento persistente a través de una intranet. Un servicio de archivos bien diseñado proporciona acceso a los sistemas almacenados en un servidor con prestaciones y fiabilidad semejantes a, y en algunos casos mejor que, los archivos almacenados en discos locales. Su diseño se adapta para las características de prestaciones y fiabilidad de redes locales y por ello son más efectivos proporcionando almacenamiento persistente compartido para uso en intranets. Los primeros servidores de archivos fueron desarrollados por investigadores en la década de los setenta [Birrell y Needham 1980, Mitchell y Dion 1982, Leach y otros 1983] y el Sistema de Archivos en Red de Sun estuvo disponible en los primeros años ochenta [Sandberg y otros 1985, Callaghan 1999].

Un servicio de archivos permite a los programas almacenar y acceder a los archivos remotos del mismo modo que se hace con los locales, permitiendo a los usuarios acceder a sus archivos desde cualquier computador de una intranet. La concentración de almacenamiento persistente en unos pocos servidores reduce la necesidad de almacenamiento en disco local y (más importante) permite economizar la gestión y el archivo de datos persistentes pertenecientes a una organización. Otros servicios, como el servicio de nombres, el servicio de autenticación de usuarios y el servicio de impresión, pueden ser implementados más fácilmente si hacen llamadas al servicio de archivos, que satisface sus necesidades de almacenamiento permanente. Los servidores web dependen de los sistemas de archivos para el almacenamiento de las páginas web que sirven. En organizaciones que operan con servidores web para acceso externo e interno vía una intranet, los servidores web suelen almacenan y acceder al material desde un sistema de archivos local distribuido.

Con la llegada de la programación orientada a objetos distribuida, aparece la necesidad del almacenamiento persistente y la distribución para los objetos compartidos. Una forma de lograr esto es serializar los objetos (de la forma descrita en la Sección 4.3.2) y almacenar y recuperar estos objetos serializados sobre archivos. Pero este método para obtener persistencia y distribución llega a ser impracticable para objetos que cambian rápidamente, y se han desarrollado otras aproximaciones más directas. La invocación remota de objetos de Java y de los ORB de CORBA proporcionan acceso a objetos remotos, compartidos, pero ninguno de ellos asegura la persistencia de los objetos ni la replicación de los objetos distribuidos.

	Compartición	*Persistencia*	*Caché/ réplicas distribuidas*	*Mantenimiento de consistencia*	*Ejemplo*
Memoria principal	×	×	×	1	RAM
Sistema de archivos	×	✓	×	1	Sistema de archivos UNIX
Sistema de archivos distribuido	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Servidor Web
Memoria compartida distribuida	✓	×	✓	✓	Ivy (Cap. 16)
Objetos remotos (RMI/ORB)	✓	×	×	1	CORBA
Almacén de objetos persistentes	✓	✓	×	1	Servicio de objetos persistentes de CORBA
Almacén de objetos persistentes distribuido	✓	✓	✓	✓	PerDis, Khazana

Figura 8.1. Sistemas de almacenamiento y sus propiedades.

Los desarrollos recientes, sobre la distribución de la información almacenada, se orientan hacia los sistemas de memoria compartida distribuida (DSM) y los almacenes de objetos persistentes. DSM se describe con detalle en el Capítulo 16. Proporciona una emulación de una memoria compartida mediante la replicación de páginas o segmentos de memoria, en cada máquina. Los almacenes de objetos persistentes se presentaron en el Capítulo 5. Su finalidad es proporcionar persistencia para objetos compartidos distribuidos. Ejemplos de ello son el Servicio de Objetos Persistente de CORBA (véase el Capítulo 17) y extensiones de persistencia para Java [Jordan 1996, java.sun.com IV]. Algunas investigaciones recientes han resultado en plataformas que soportan la replicación automática y el almacenamiento persistente de objetos (por ejemplo, PerDiS [Ferreira y otros 2000] y Khazana [Carter y otros 1998]).

La Figura 8.1 proporciona un resumen de las propiedades de los distintos tipos de sistemas de almacenamiento que hemos mencionado. La columna *consistencia* indica qué mecanismos existen para el mantenimiento de la consistencia entre múltiples copias de datos cuando ocurren actualizaciones. Virtualmente todos los sistemas de almacenamiento confían en el uso de caché para optimizar las prestaciones de los programas. Las técnicas de caché se aplicaron en primer lugar a la memoria principal y a los sistemas de archivos no distribuidos, para los que la consistencia es estricta (señalado por un «1», para consistencia de una copia en la Figura 8.1), los programas no contemplan discrepancias entre las copias en la caché y los datos almacenados después de una actualización. Cuando se utilizan réplicas distribuidas, la consistencia estricta es más difícil de obtener. Los sistemas de archivos distribuidos como Sun NFS y el sistema de archivos Andrew replican copias de porciones de archivos en los computadores de los clientes y adoptan mecanismos de consistencia específicos para mantener una aproximación a la consistencia estricta. Esto se indica mediante una marca (✓) en la columna de consistencia de la Figura 8.1. Discutiremos estos mecanismos y el grado en el que se desvían de la consistencia estricta en las Secciones 8.3 y 8.4.

La Web utiliza cachés extensivamente tanto en los computadores cliente como en los servidores proxy mantenidos por la organización de cada usuario. La consistencia entre las copias almacenadas en la caché de un proxy web y los clientes y en el servidor original sólo se consigue mediante acciones específicas del usuario. Cuando se actualiza una página almacenada en el servidor

original no se advierte a los clientes, y son ellos quienes deben realizar comprobaciones periódicas para mantener sus copias actualizadas. Esto sirve adecuadamente para el propósito de navegación web, pero no soporta el desarrollo de aplicaciones cooperativas como el de un tablero compartido distribuido. Los mecanismos de consistencia utilizados en los sistemas DSM se discutirán con detalle en el Capítulo 16. Los sistemas de objetos persistentes varían considerablemente en su aproximación al uso de caché y consistencia. Los esquemas CORBA y Persistent Java mantienen una única copia de cada objeto persistente y para acceder a ellos es preciso realizar una invocación remota, por lo que el único tema de consistencia está entre la copia persistente de un objeto en disco y la copia activa en memoria, que no es visible para los clientes remotos. Los proyectos PerDiS y Khazana que hemos mencionado anteriormente mantienen réplicas en la caché de los objetos y emplean mecanismos de consistencia bastante elaborados para producir formas de consistencia semejantes a las que se encuentran en los sistemas DSM.

Habiendo presentado algunos temas más profundos relacionados con el almacenamiento y distribución de datos persistentes y no persistentes, volvemos al tema principal de este capítulo, el diseño de sistemas básicos de archivos distribuidos. Describimos algunas características relevantes de los sistemas de archivos (no distribuidos) en la Sección 8.1.1, y los requisitos para sistemas de archivos distribuidos en la Sección 8.1.2. La Sección 8.1.3 presenta los casos de estudio que se utilizarán a lo largo del capítulo. En la Sección 8.2 definimos un modelo abstracto para un servicio básico de archivos distribuidos, incluyendo un conjunto de interfaces de programación. El sistema Sun NFS se describe en la Sección 8.3, comparte muchas de las características del modelo abstracto. En la Sección 8.4 describimos el Sistema de Archivos Andrew, un sistema ampliamente usado que emplea mecanismos de caché y consistencia sustancialmente diferentes. La Sección 8.5 revisa algunos de los desarrollos recientes en el diseño de servicios de archivos.

Los sistemas descritos en este capítulo no cubren el espectro total de los sistemas de archivos distribuidos y gestión de datos. Varios sistemas con características más avanzadas serán descritos posteriormente en el libro. El Capítulo 14 incluirá una descripción de Coda, un sistema de archivos distribuidos que mantiene réplicas persistentes de los archivos para la fiabilidad, disponibilidad y el trabajo desconectado. Bayou, un sistema de gestión de datos distribuidos que proporciona una forma de replicación débilmente consistente para alta disponibilidad se tratará también en el Capítulo 14. El Capítulo 15 discutirá el servidor de archivos de vídeo Tiger, que está diseñado para proporciona la entrega oportuna de caudales de datos a gran número de clientes.

8.1.1. CARACTERÍSTICAS DE LOS SISTEMAS DE ARCHIVOS

Los sistemas de archivos son responsables de la organización, almacenamiento, recuperación, nominación, compartición y protección de los archivos. Proporcionan una interfaz de programación característica de la abstracción de archivo, liberando a los programadores de la preocupación por los detalles de la asignación y la disposición del almacenamiento. Los archivos se almacenan en discos y otros medios de almacenamiento no volátiles.

Los archivos contienen *datos* y *atributos*. Los datos consisten en una secuencia de elementos de datos (normalmente bytes de 8 bits), donde cualquier porción de ésta es accesible mediante operaciones de lectura y escritura. Los atributos se alojan como un único registro que contiene información como la longitud del archivo, marcas de tiempo, tipo del archivo, identidad del propietario y listas de control de acceso.

En la Figura 8.3 se muestra una estructura típica del registro de atributos. Los atributos resaltados son administrados por el sistema de archivos y no son modificables habitualmente desde los programas del usuario.

Los sistemas de archivos están diseñados para almacenar y gestionar gran número de archivos, con posibilidades de crear, nombrar y borrar archivos. La nomenclatura de los archivos está respal-

Módulo de directorio:	relaciona nombres de archivos con ID de archivos
Módulo de archivos:	relaciona ID de archivos con archivos concretos
Módulo de control de acceso:	comprueba los permisos para una operación solicitada
Módulo de acceso a archivos:	lee o escribe datos o atributos de un archivo
Módulo de bloques:	accede y asigna bloques de disco
Módulo de dispositivo:	E/S de disco y búferes

Figura 8.2. Módulos del sistema de archivos.

dada por la utilización de directorios. Un *directorio* es un archivo, a menudo de un tipo especial, que relaciona los nombres en texto con los identificadores internos de los archivos. Los directorios pueden incluir los nombres de otros directorios derivando en el esquema familiar de nomenclatura jerárquica de archivos y los *nombres de ruta*, tanto para para los archivos en UNIX como para otros sistemas operativos. Los sistemas de archivo tienen también la responsabilidad del control de acceso a los archivos, restringiendo el acceso a los mismos de acuerdo con las autorizaciones de los usuarios y el tipo de acceso solicitado (lectura, actualización, ejecución y demás).

El término *metadato* se utiliza a menudo para referirse a toda la información extra almacenada por un sistema de archivos que es necesaria para la gestión de los mismos. Incluye los atributos de los archivos, los directorios y todas las demás informaciones persistentes empleadas por el sistema de archivos.

La Figura 8.2 muestra una estructura típica de niveles para la implementación de un sistema de archivos no distribuido en un sistema operativo convencional. Cada nivel depende sólo de los niveles que se encuentran debajo de él. La implementación de un servicio de archivos distribuidos requiere todos los componentes indicados, junto a componentes adicionales para ocuparse de la comunicación cliente-servidor y de la nomenclatura y ubicación de los archivos distribuidos.

Tamaño del archivo
Marca temporal de creación
Marca temporal de lectura
Marca temporal de escritura
Marca temporal de atributos
Contador de referencias
Propietario
Tipo de archivo
Lista de control de acceso

Figura 8.3. Estructuras del registro de atributos de un archivo.

desarchivo = open(nombre, modo) *desarchivo = creat(nombre, modo)*	Abre un archivo existente con un *nombre* determinado. Crea un archivo nuevo con un *nombre* determinado. Ambas operaciones devuelven un descriptor de archivo que referencia el archivo abierto. El *modo* es *read, write* u otro.
estado = close(desarchivo)	Cierra el archivo abierto *desarchivo.*
recuento = read(desarchivo, búfer, n) *recuento = write(desarchivo, búfer, n)*	Transfiere *n* bytes del archivo referenciado por *desarchivo* sobre *búfer.* Transfiere *n* bytes al archivo refererenciado por *desarchivo* desde *búfer.* Ambas operaciones retornan el número de bytes transferidos realmente y adelanta el apuntador de lectura-escritura.
pos = lseek(desarchivo, despl, desde)	Desplaza el puntero de lectura-escritura hasta *despl* (relativo o absoluto, dependiendo del valor de *desde*).
estado = unlink(nombre)	Elimina el *nombre* de archivo de la estructura de directorios. Si el archivo no tuviera otros nombres, sería también eliminado.
estado = link(nombre1, nombre2)	Añade un nombre nuevo (*nombre2*) a un archivo (*nombre1*).
estado = stat(nombre, búfer)	Obtiene los artributos de archivo del archivo *nombre* sobre *búfer.*

Figura 8.4. Operaciones del sistema de archivos de UNIX.

◇ **Operaciones en el sistema de archivos.** La Figura 8.4 resume las principales operaciones sobre archivos que están disponibles para aplicaciones en sistemas UNIX. Ésas son las llamadas del sistema que implementa el núcleo, los programadores de aplicaciones normalmente acceden a ellas a través de bibliotecas de procedimientos, como la librería estándar de entrada-salida de C o las clases archivo de Java. Proporcionamos aquí las primitivas como una indicación de las operaciones que se espera que los servicios de archivo soporten y para poder comparar con las interfaces de servicios de archivo que veremos más adelante.

Las operaciones de UNIX se basan en un modelo de programación en el que se almacena cierta información del estado de un archivo, por el sistema de archivos, para cada programa que lo use. El sistema registra una lista de los archivos abiertos actualmente con un apuntador de lectura-escritura para cada uno, que proporciona la posición en el archivo en la que se aplicará la siguiente operación de lectura o escritura.

El sistema de archivos es responsable de aplicar el control de acceso para los archivos. En sistemas de archivos locales como en UNIX, esto se hace cada vez que se abre un archivo, comprobando los derechos permitidos para la identidad del usuario mediante la lista de control de acceso contra el modo de acceso solicitado en la llamada del sistema *open*. Si los derechos concuerdan con el modo, el archivo es abierto y el *modo* se almacena en la información del estado del archivo abierto.

8.1.2. REQUISITOS DEL SISTEMA DE ARCHIVOS DISTRIBUIDOS

Muchos de los requisitos y potenciales obstáculos en el diseño de servicios distribuidos fueron ya observados en los primeros desarrollos de sistemas de archivos distribuidos. Inicialmente ofrecían transparencia de acceso y transparencia de ubicación, los requisitos de prestaciones, escalabilidad, control de concurrencia, tolerancia a fallos y seguridad surgieron y se fueron satisfaciendo en fases posteriores del desarrollo. Discutimos estos requisitos, y otros relacionados, en las subsecciones siguientes.

◇ **Transparencia.** El servicio de archivos es el servicio más fuertemente cargado en una intranet, por lo que su funcionalidad y prestaciones son críticas. El diseño de un servicio de archivos debe soportar muchos de los requisitos de transparencia para sistemas distribuidos identificados en

la Sección 1.4.7. El diseño debe balancear la flexibilidad y escalabilidad que se derivan de la transparencia frente a la complejidad del software y las prestaciones. Las siguientes formas de transparencia son parcial o totalmente tratadas por los actuales servicios de archivos:

Transparencia de acceso: los programas del cliente no deben preocuparse de la distribución de los archivos. Se proporciona un conjunto sencillo de operaciones para el acceso a archivos locales y remotos. Los programas escritos para trabajar sobre archivos locales serán capaces de acceder a los archivos remotos sin modificación.

Transparencia de ubicación: los programas del cliente deben ver un espacio de nombres de archivos uniforme. Los archivos o grupos de archivos pueden ser reubicados sin cambiar sus nombres de ruta, y los programas de usuario verán el mismo espacio de nombres en cualquier parte que sean ejecutados.

Transparencia de movilidad: ni los programas del cliente ni las tablas de administración de sistema en los nodos cliente necesitan ser cambiados cuando se mueven los archivos. Esta movilidad de archivos permite que archivos o, más comúnmente, conjuntos o volúmenes de archivos puedan ser movidos, ya sea por los administradores del sistema o automáticamente.

Transparencia de prestaciones: los programas cliente deben continuar funcionando satisfactoriamente mientras la carga en el servicio varíe dentro de un rango especificado.

Transparencia de escala: el servicio puede ser aumentado por un crecimiento incremental para tratar con un amplio rango de cargas y tamaños de redes.

◇ **Actualizaciones concurrentes de archivos.** Los cambios en un archivo por un cliente no deben interferir con la operación de otros clientes que acceden o cambian simultáneamente el mismo archivo. Esto es el tema bien conocido del control de concurrencia, discutido con detalle en el Capítulo 12. La necesidad de control de concurrencia para el acceso a datos compartidos en muchas aplicaciones está ampliamente aceptada y las técnicas para su implementación son conocidas, aunque muy costosas. La mayoría de los servicios de archivos actuales siguen los estándares de UNIX moderno proporcionando bloqueo consultivo u obligatorio a nivel de archivo o registro.

◇ **Replicación de archivos.** En un servicio de archivos que soporta replicación, un archivo puede estar representado por varias copias de su contenido en diferentes ubicaciones. Esto tiene dos beneficios, permite que múltiples servidores compartan la carga de proporcionar un servicio a los clientes que acceden al mismo conjunto de archivos, mejorando la escalabilidad del servicio y mejorando la tolerancia a fallos, permitiendo a los clientes localizar otro servidor que mantiene una copia del archivo cuando uno ha fallado. Muy pocos servicios de archivos soportan totalmente la replicación, pero la mayoría soportan la caché local de archivos o porciones de archivos, una forma limitada de replicación. La replicación de datos se discutirá en el Capítulo 14, que incluye una descripción del servicio de archivos replicado Coda.

◇ **Heterogeneidad del hardware y del sistema operativo.** Las interfaces del servicio deben estar definidas de modo que el software del cliente y el servidor pueden estar implementados por diferentes sistemas operativos y computadores. Este requisito es un aspecto importante de la extensibilidad.

◇ **Tolerancia a fallos.** El papel central de un servicio de archivos en los sistemas distribuidos hace que sea esencial que el servicio continúe funcionando aun en el caso de fallos del cliente y del servidor. Afortunadamente un diseño moderadamente tolerante a fallos es inmediato para servidores sencillos. Para manejar los fallos de comunicación transitorios, el diseño puede estar basado en la semántica de invocación de como *máximo una vez* (véase la Sección 5.2.4). O puede utilizarse la semántica más sencilla *al menos una vez* con un protocolo de servidor diseñado en términos de operaciones *idempotentes*, asegurando que solicitudes duplicadas no producen actualizaciones inválidas en los archivos. Los servidores pueden ser *sin estado*, por lo que pueden ser rearrancados

y el servicio restablecido después de un fallo sin necesidad de recuperar el estado previo. La tolerancia a la desconexión o fallos del servidor precisa replicación de los archivos, que es más difícil de alcanzar y será discutida en el Capítulo 14.

◇ **Consistencia.** Los sistemas de archivos convencionales, como los que se proporcionan en UNIX, ofrecen una *semántica de actualización de una copia*. Esto se refiere a un modelo para acceso concurrente a archivos en el que el contenido del archivo visto por todos los procesos que acceden o actualizan a un archivo dado es aquel que ellos verían si existiera un única copia del contenido del archivo. Cuando los archivos están replicados, o en la caché, en diferentes lugares, hay un retardo inevitable en la propagación de las modificaciones hechas en un lugar hacia los otros lugares que mantienen copias, y esto puede producir alguna desviación de la semántica de una copia.

◇ **Seguridad.** Virtualmente todos los sistemas de archivos proporcionan mecanismos de control de acceso basados en el uso de listas de control de acceso. En sistemas de archivos distribuidos, hay una necesidad de autenticar las solicitudes del cliente por lo que el control de acceso en el servidor está basado en identificar al usuario correcto y proteger el contenido de los mensajes de solicitud y respuesta con firmas digitales y (opcionalmente) encriptación de datos secretos. Discutiremos el impacto de estos requisitos en nuestras descripciones de casos de estudio.

◇ **Eficiencia.** Un servicio de archivos distribuidos debe ofrecer posibilidades con la misma potencia y generalidad que las que se encuentran en los sistemas de archivos convencionales y deben proporcionar un nivel de prestaciones comparable. Birrell y Needham [1980] expresaron sus objetivos de diseño para el Servidor de Archivos Cambridge (CFS) en estos términos:

> Nosotros desearíamos tener un servidor de archivos sencillo, de bajo nivel, para compartir un recurso caro, por ejemplo un disco, mientras dejamos libertad para diseñar el sistema de archivos más apropiado para un cliente particular, pero desearíamos también tener disponible un sistema de alto nivel compartido entre clientes.

El cambio del coste de almacenamiento de disco ha reducido la importancia de su primer objetivo, pero su percepción de la necesidad de un rango de servicios que respondan a los requisitos de los clientes con diferentes objetivos, permanece y puede ser conseguido mejor por una arquitectura modular del tipo esbozado anteriormente.

Las técnicas utilizadas para la implementación de los servicios de archivo son una parte importante del diseño de sistemas distribuidos. Un sistema de archivos distribuidos debe proporcionar un servicio que sea comparable con, o mejor que, los sistemas de archivos locales en prestaciones y fiabilidad. Debe ser adecuado para administrar, proporcionando operaciones y herramientas que permitan a los administradores del sistema instalar y operar el sistema convenientemente.

8.1.3. CASOS DE ESTUDIO

Hemos construido un modelo abstracto para un servicio de archivos para actuar como un ejemplo introductorio separando las cuestiones de implementación y proporcionando un modelo simplificado. Describimos el Sistema de Archivos en Red (*Network File System*) de Sun (NFS) con algún detalle, basándonos en nuestro modelo abstracto más sencillo para aclarar su arquitectura. Posteriormente se describe el Sistema de Archivos Andrew (AFS), proporcionando una visión de un sistema de archivos distribuido que considera una aproximación diferente para la escalabilidad y el mantenimiento de la consistencia.

◇ **Arquitectura del servicio de archivos.** Éste es un modelo arquitectónico abstracto sobre el que se sustentan tanto NFS como AFS. Está basado en una división de las responsabilidades entre tres módulos, un módulo cliente que emula la interfaz de un sistema de archivos convencional para los programas de aplicación y módulos servidores, que realizan operaciones para los clientes en directorios y archivos. La arquitectura está diseñada para permitir una implementación *sin estado* del módulo del servidor.

◇ **NFS de SUN.** El sistema de archivos en red de SUN (NFS) ha sido adoptado ampliamente en la industria y en entornos académicos desde su introducción en 1985. El diseño y desarrollo de NFS fueron emprendidos por el personal de Sun Microsystems en 1984 [Sandberg y otros 1985; Sandberg 1987, Callaghan 1999]. Aunque ya habían sido desarrollados varios servicios de archivos distribuidos, y utilizados con éxito, en universidades y laboratorios de investigación, NFS fue el primer servicio de archivos que fue diseñado como un producto. El diseño e implementación de NFS ha obtenido un éxito considerable tanto técnica como comercialmente.

Para animar a su adopción como un estándar, las definiciones de las interfaces fundamentales fueron situadas en el dominio público [Sun 1989], permitiendo a otros vendedores producir implementaciones, y el código fuente fue puesto disponible para una implementación de referencia a otros vendedores de computadores bajo licencia. Actualmente está soportado por muchos vendedores y el protocolo NFS (versión 3) es un estándar de Internet, definido en RFC 1813 [Callaghan y otros 1995]. El libro del mismo Callaghan sobre NFS [Callaghan 1999] es una fuente excelente sobre el diseño y desarrollo de NFS y temas relacionados.

NFS proporciona acceso transparente a archivos remotos desde programas cliente ejecutándose sobre UNIX y otros sistemas. Normalmente, cada computador tiene un cliente NFS y módulos servidor instalados en el núcleo del sistema, al menos en el caso de los sistemas UNIX. La relación cliente-servidor es simétrica: Cada computador en una red NFS puede actuar tanto como cliente como servidor, y los archivos en cada máquina pueden hacerse disponibles para acceso remoto desde otras máquinas. Cualquier computador puede ser un servidor, exportando algunos de sus archivos, y un cliente, accediendo a archivos de otras máquinas. Pero es una práctica habitual configurar instalaciones grandes con algunas máquinas como servidores dedicados y otras como estaciones de trabajo.

Un objetivo importante de NFS es conseguir un elevado nivel de soporte para la heterogeneidad de hardware y el sistema operativo. El diseño es independiente del sistema operativo: Existen implementaciones de servidor y cliente para casi todos los sistemas operativos y plataformas actuales, incluyendo Windows 95, Windows NT, MacOS y VMS así como Linux y casi cualquier otra versión de UNIX. Se han desarrollado implementaciones de NFS en máquinas multiprocesador de altas prestaciones por varios vendedores y éstas se utilizan ampliamente para satisfacer los requisitos de almacenamiento en intranets con muchos usuarios concurrentes.

◇ **Andrew File System.** Andrew es un entorno de computación distribuida desarrollado en la Universidad Carnegie Mellon (CMU) para su utilización como un sistema de información y computación de campus [Morris y otros 1986]. El diseño de Andrew File System (a partir de ahora abreviado como AFS) refleja una intención de soportar la compartición de información en gran escala minimizando la comunicación cliente servidor. Esto fue conseguido transfiriendo archivos completos (o para archivos grandes, trozos de 64-kbytes) entre los computadores del servidor y del cliente y haciendo caché de ellos en los clientes hasta que el servidor reciba una versión más actualizada. La red de computación de campus que servía Andrew se esperaba que creciera hasta incluir entre 5.000 y 10.000 estaciones de trabajo durante el tiempo de vida del sistema. Describiremos AFS-2, la primera implementación de producción, siguiendo las descripciones de Satyanarayanan [1989a, 1989b]. Se pueden encontrar descripciones más recientes en Campbell [1997] y [Linux AFS].

AFS fue implementado inicialmente sobre una red de estaciones de trabajo y servidores trabajando en BSD UNIX y el sistema operativo Mach en CMU y posteriormente estuvo disponible en versiones comerciales y de dominio público. Más recientemente, se ha hecho disponible una implementación de dominio público de AFS para el sistema operativo Linux [Linux AFS]. En 1991, AFS

estaba soportando aproximadamente 800 estaciones de trabajo servidas por aproximadamente 40 servidores en la CMU, con clientes y servidores adicionales en lugares remotos conectados a través de Internet. AFS fue adoptado como la base para el sistema de archivos DCE/DFS en el entorno de computación distribuido (DCE) de la Open Software Foundation [www.open.group.org]. El diseño de DCE/DFS fue mas allá de AFS en varios aspectos importantes, que señalamos en la Sección 8.5.

8.2. ARQUITECTURA DEL SERVICIO DE ARCHIVOS

El alcance de la extensibilidad y configurabilidad se mejora si el servicio de archivos se estructura en tres componentes, un *servicio de archivos plano*, un *servicio de directorio* y un *módulo cliente*. Los módulos relevantes y sus relaciones se ven en la Figura 8.5. El servicio de archivos plano y el servicio de directorio exportan cada uno una interfaz, para su uso por los programas del cliente y sus interfaces RPC, consideradas conjuntamente, proporcionan un conjunto global de operaciones para acceso a archivos. El módulo cliente proporciona una interfaz de programación sencilla con operaciones sobre archivos semejantes a las encontradas en los sistemas de archivos convencionales. El diseño es *abierto* en el sentido que pueden utilizarse diferentes módulos cliente para implementar diferentes interfaces de programación, simulando las operaciones sobre archivos de una variedad de diferentes sistemas operativos y optimizando las prestaciones para diferentes configuraciones de hardware del cliente y el servidor.

La división de las responsabilidades entre los módulos puede definirse como sigue:

◇ **Servicio de archivos planos.** El servicio de archivos planos está relacionado con la implementación de operaciones en el contenido de los archivos. Se utilizan *identificadores únicos de archivos* (UFID) para referirse a los archivos en todas las solicitudes de operaciones del servicio de archivos plano. La división de responsabilidades entre el servicio de archivos y el servicio de directorio, está basada en la utilización de UFID. Cada UFID es una secuencia largas de bits elegidas de forma que cada archivo tiene un UFID que es único entre todos los archivos en un sistema distribuido. Cuando el servicio de archivos planos recibe una solicitud para crear un archivo, genera un nuevo UFID para él y lo devuelve al solicitante.

◇ **Servicio de directorio.** El servicio de directorio proporciona una transformación entre *nombres de texto* para los archivos y sus UFID. Los clientes pueden obtener el UFID de un archivo indicando su nombre de texto al servicio de directorio. El servicio de directorio proporciona las

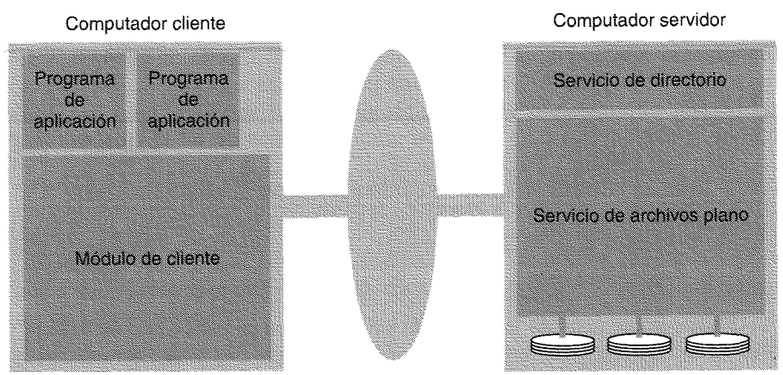


Figura 8.5. Arquitectura del servicio de archivos.

funciones necesarias para generar directorios, para añadir nuevos nombres de archivo a los directorios y para obtener UFID desde los directorios. Es un cliente del servicio de archivos plano, sus archivos de directorio están almacenados en archivos del servicio de archivos plano. Cuando se adopta un esquema jerárquico de nominación de archivos, como en UNIX, los directorios mantienen referencias a otros directorios.

◇ **Módulo cliente.** En cada computador cliente se ejecuta un módulo de cliente, que integra y extiende las operaciones del servicio de archivos plano y el servicio de directorio bajo una interfaz de programación de aplicaciones sencilla, que estará disponible para los programas a nivel de usuario en los computadores cliente. Por ejemplo, en máquinas UNIX, se debe proporcionar un módulo cliente que emula el conjunto total de operaciones UNIX sobre archivos, interpretando los nombres *compuestos* de archivos UNIX mediante reiteradas solicitudes al servicio de directorio. El módulo cliente mantiene también información sobre las ubicaciones en la red del proceso servidor de archivos planos y del proceso servidor de directorio. Finalmente el módulo cliente puede jugar un papel importante consiguiendo unas prestaciones satisfactorias mediante la implementación de una caché de los bloques de archivos utilizados recientemente en el cliente.

◇ **Interfaz del servicio de archivos plano.** La Figura 8.6 contiene una definición de la interfaz para un servicio de archivos planos. Es la interfaz RPC utilizada por los módulos cliente. No es utilizada directamente por los programas a nivel de usuario, normalmente. Un *IdArchivo* es inválido si el archivo a que se refiere no está presente en el servidor que procesa la solicitud o si sus permisos de acceso son inapropiados para la operación solicitada. Todos los procedimientos de la interfaz excepto *Crea* lanzan excepciones si el argumento *IdArchivo* contiene un UFID inválido o el usuario no tiene los derechos de acceso suficientes. Estas excepciones se han omitido por la claridad de la definición.

Las operaciones más importantes son las de lectura y escritura. Tanto la operación de lectura como la de escritura necesitan un parámetro *i* especificando una posición en el archivo. La operación de lectura copia la secuencia de *n* elementos de datos comenzando en el elemento *i* del archivo especificado en *Datos* que se devuelve entonces al cliente. La operación *Escribe* copia la secuencia de elementos de datos de *Datos* en el archivo especificado comenzando en el elemento *i*, reemplazando el contenido anterior del archivo en la posición correspondiente y extendiendo el archivo si es necesario.

Crea crea un archivo nuevo, vacío y devuelve el UFID que se ha generado. *Elimina* borra el archivo especificado.

DameAtributos y *PonAtributos* permiten a los usuarios acceder al registro de los atributos. *DameAtributos* está disponible generalmente para cualquier usuario que acceda al archivo. El acceso a la operación *PonAtributos* suele estar restringida normalmente al servicio de directorio que accede

Lee(IdArchivo, i, n) → Datos — lanza *MalaPosición*	Si $1 \leqslant i \leqslant$ *Tamaño(Archivo)*: Lee una secuencia de hasta *n* elementos del archivo, partiendo del elemento *i* y la devuelve en *Datos*.
Escribe(IdArchivo, i, Datos) — lanza *MalaPosición*	Si $1 \leqslant i$ *Tamaño(Archivo)* + *1*: Escribe una secuencia de *Datos* sobre el archivo, partiendo del elemento *i*, y extendiendo el archivo si es preciso.
Crea() → IdArchivo	Crea un nuevo archivo de tamaño 0 y obtiene un UFID para él.
Elimina(IdArchivo)	Elimina el archivo del almacén de archivos.
DameAtributos(IdArchivo) → Atrib	Obtiene los atributos del archivo.
PonAtributos(IdArchivo, Atrib)	Pone los atributos del archivo (sólo aquellos que le están permitidos en la Figura 8.3)

Figura 8.6. Operaciones del servicio de archivos plano.

al archivo. Los valores de la longitud y de las partes de marcas de tiempo del registro de atributos no son afectadas por *PonAtributos*, son mantenidas por el servicio de archivos planos.

Comparación con UNIX: Nuestra interfaz y las primitivas del sistema de archivos UNIX son funcionalmente equivalentes. No es complicado construir un módulo cliente que emule las llamadas al sistema UNIX en términos de las operaciones de nuestro servicio de archivos planos y el servicio de directorio descritas en la sección siguiente.

En comparación con la interfaz UNIX, nuestro servicio de archivos plano no tiene operaciones *open* y *close*, los archivos pueden ser accedidos inmediatamente proporcionando el UFID adecuado. Las solicitudes de *Lee* y *Escribe* de nuestra interfaz incluyen un parámetro que especifica el punto de partida para cada transferencia desde el archivo, mientras que las operaciones UNIX equivalentes no. En UNIX cada operación *read* y *write* comienza en la posición actual del apuntador de *lectura escritura*, y dicho apuntador se desplaza en una cantidad correspondiente al número de bytes transferidos después de cada lectura y escritura. Se proporciona una operación *seek* para permitir que dicho apuntador de *lectura escritura* sea posicionado explícitamente.

La interfaz de nuestro servicio de archivos planos difiere de la del sistema de archivos de UNIX fundamentalmente por razones de tolerancia a fallos:

Operaciones repetibles: con la excepción de *Crea*, las operaciones son *idempotentes*, permitiendo la utilización de la semántica de *al menos una vez* de RPC, los clientes pueden repetir las llamadas de las que no han recibido contestación. La ejecución repetida de *Crea* produce un archivo diferente en cada llamada.

Servidores sin estado: la interfaz es adecuada para su implementación por servidores sin estado (*stateless*). Los servidores sin estado pueden ser rearrancados después de un fallo y reanudar la operación sin necesitar que ni los clientes ni el servidor restablezcan su estado.

Las operaciones sobre archivos de UNIX no son ni idempotentes ni consistentes con los requisitos para una implementación sin estado. Cuando se abre un archivo se genera un apuntador de *lectura escritura* desde el sistema de archivos de UNIX, y se mantiene, junto con las comprobaciones de los resultados de control de acceso, hasta que se cierra el archivo. Las operaciones de lectura y escritura de UNIX no son idempotentes, si una operación se repite accidentalmente, el avance automático del apuntador de *lectura escritura* produce el acceso a una porción diferente del archivo en la operación repetida. El apuntador del *lectura escritura* es una variable de estado oculta, relacionada con el cliente. Para imitar su comportamiento en un servidor de archivos, se necesitan las operaciones de *open* y *close*, y hay que mantener el valor del apuntador de *lectura escritura* en tanto en cuanto el archivo considerado esté abierto. Eliminando el apuntador de *lectura escritura*, hemos eliminado la necesidad de que un servidor de archivos retenga información sobre el estado en nombre de clientes específicos.

◇ **Control de acceso.** En el sistema de archivos UNIX, se comprueban los derechos de acceso del usuario frente al *modo* de acceso (lectura o escritura) solicitado en la operación de llamada (la Figura 8.4 muestra la interfaz de aplicaciones de UNIX) y el archivo es abierto sólo si el usuario tiene los derechos de acceso necesarios. La identidad del usuario (UID) utilizada en la comprobación de los derechos de acceso es el resultado del *login* autenticado anterior del usuario y no puede ser alterado en las implementaciones no distribuidas. Los derechos de acceso resultantes se retienen hasta que el archivo se cierra y no se necesitan otras comprobaciones en las operaciones posteriores solicitadas sobre el mismo archivo.

En las implementaciones distribuidas, la comprobación de los derechos de acceso se ha de realizar en el servidor porque la interfaz del servidor RPC es un punto desprotegido de acceso para los archivos. La identidad del usuario ha de pasarse con cada solicitud, y el servidor se hace vulnerable a identidades antiguas. Además si se retuvieran en el servidor los resultados de una comprobación de derechos de acceso, y se usaran para otros acceso, el servidor ya no sería sin estado. Se pueden adoptar dos aproximaciones alternativas a este último problema:

- Se realiza una comprobación de acceso cuando se convierte un nombre de archivo en un UFID, y los resultados se codifican en forma de una habilitación (ver Sección 7.2.4), que se devuelve al cliente para su envío con las solicitudes posteriores.
- Se envía la identidad del usuario con cada solicitud del cliente, y las comprobaciones de acceso se realizan en el servidor para cada operación sobre el archivo.

Ambos métodos permiten una implementación de servidor sin estado, y ambas se han utilizado en los sistemas de archivos distribuidos. El segundo es más común, y se utiliza tanto en NFS como en AFS. Ninguna de estas aproximaciones resuelve el problema de seguridad relacionado con las identidades olvidadas del usuario. Hemos visto en el Capítulo 7 que este problema puede abordarse mediante el uso de firmas digitales. Kerberos es un esquema de autenticación efectivo que ha sido aplicado tanto en NFS como en AFS.

En nuestro modelo abstracto, no hacemos suposiciones sobre el método con el que debe estar implementado el control de acceso. La identidad del usuario se pasa como un parámetro implícito y puede utilizarse cuando sea preciso.

◇ **Interfaz del servicio de directorio.** La Figura 8.7 contiene una definición de la interfaz RPC para un servicio de directorio. El propósito principal del servicio de directorio es proporcionar un servicio para trasladar nombres a UFID. Con este fin, se mantienen archivos de directorios que contienen las equivalencias entre los nombres de los archivos y cada UFID. Cada directorio se almacena en un archivo convencional con una UFID, de forma que el servicio de directorio es un cliente del servicio de archivos.

Definimos únicamente las operaciones sobre directorios individuales. Para cada operación, se precisa una UFID para el archivo que contiene el directorio (en el parámetro *Dir*). la operación *Busca* en el servicio básico de directorios realiza una traducción sencilla *Nombre* → *UFID*. Es un bloque básico para su uso en otros servicios o para realizar traducciones más complejas en el módulo del cliente, como la interpretación jerárquica de nombres encontrada en UNIX. Como antes, las excepciones producidas por derechos de acceso no adecuados se omiten de las definiciones.

Hay dos operaciones para modificar directorios: *AñadeNombre* y *DesNombra*. *AñadeNombre* añade una entrada al directorio e incrementa el campo de contador de referencias en el registro de atributos del archivo.

DesNombra elimina una entrada de un directorio y decrementa el contador de referencias. Si esto produce que el contador de referencias llegue a cero, se elimina el archivo. *DameNombres* se proporciona para permitir a los clientes examinar el contenido de los directorios y para implementar operaciones de concordancia de patrones de nombres de archivos como las que se encuentran en el intérprete de órdenes de UNIX. Devuelven todos o un subconjunto de los nombres almacena-

Busca(Dir, Nombre) → *IdArchivo* — lanza *NoEncontrado*	Busca el texto *Nombre* en el directorio y devuelve el UFID relevante. Si *Nombre* no está en el directorio, lanza una excepción.
AñadeNombre(Dir, Nombre, Archivo) — lanza *NombreDuplicado*	Si *Nombre* no está en el directorio, añade *(Nombre, Archivo)* al directorio y actualiza el registro de atributos de archivo.
DesNombra(Dir, Nombre) — lanza *NoEncontrado*	Si *Nombre* está en el directorio: la entrada que contiene *Nombre* se elimina del directorio. Si *Nombre* no está en el directorio: lanza una excepción.
DameNombres(Dir, Patrón) → *SecNombres*	Devuelve todos los nombres del directorio que concuerdan con la expresión regular *Patrón*.

Figura 8.7. Operaciones del servicio de directorio.

dos en un directorio dado. Los nombres son seleccionados mediante comparación de patrones frente a una expresión regular proporcionada por el cliente.

La disponibilidad de reconocimiento de patrones en las operación *DameNombres* permite a los usuarios determinar los nombres de uno o más archivos al proporcionar una especificación incompleta de los caracteres de los nombres. Una expresión regular es una especificación de una clase de cadenas de caracteres en forma de una expresión que contiene una combinación de subcadenas literales y símbolos que representan caracteres variables u ocurrencias repetidas de caracteres o subcadenas.

◇ **Sistema de archivos jerárquico.** Un servicio de archivos jerárquico, como uno de los que proporciona UNIX, consiste en un número de directorios organizados en una estructura de árbol. Cada directorio contiene los nombres de los archivos y otros directorios que son accesibles desde él. Cualquier archivo o directorio puede ser referenciado con un nombre de ruta (*path*), un nombre compuesto de varias partes que representa una ruta a través del árbol. La raíz tiene un nombre que le distingue y cada archivo o directorio tiene un nombre en un directorio. El esquema de nominación de archivos de UNIX no es una jerarquía estricta, los archivos pueden tener varios nombres y pueden estar en varios directorios. Esto se implementa mediante una operación de enlace (*link*), que añade un nuevo nombre para un archivo en un directorio especificado.

Un sistema de nominación de archivos como el de UNIX puede ser implementado desde el módulo cliente utilizando los servicios de directorio y archivos planos que hemos definido. Se construye una red de directorios estructurados en forma de árbol con los archivos en las hojas y los directorios en los otros nodos del árbol. La raíz del árbol es un directorio con una UFID *bien-conocida*. Se da soporte a varios nombres un archivo utilizando la operación *AñadeNombre* y el campo de contador de referencias del registro de atributos.

Se puede proporcionar una función en el módulo del cliente que consiga la UFID de un archivo dado un nombre de ruta. La función interpreta el nombre de ruta comenzando desde la raíz, utilizando *Busca* para obtener la UFID de cada directorio en la ruta.

En un servicio jerárquico de archivos, los atributos asociados con los archivos deben incluir un tipo que distingue a los archivos ordinarios de los directorios. Esto se utiliza cuando se sigue una ruta para garantizar que cada parte del nombre, excepto la última, se refiere a un directorio.

◇ **Agrupación de archivos.** Un *grupo de archivos* es una colección de archivos ubicada en un servidor dado. Un servidor puede mantener varios grupos de archivos y los grupos pueden ser recolocados entre servidores, pero un archivo no puede cambiar el grupo al que pertenece. Una construcción similar (llamada *filesystem-volumen*) se utiliza en UNIX y en la mayoría de los sistemas operativos. Los grupos de archivos se introdujeron inicialmente para trasladar colecciones de archivos almacenados en cartuchos de disco extraíbles entre computadores. En un servicio de archivos distribuidos, los grupos de archivos permiten la asignación de archivos a servidores de archivos almacenados en varios servidores. En un sistema de archivos distribuidos que soporta grupos de archivos, la representación de UFID incluye un componente identificador de grupo de archivos, permitiendo al módulo cliente en cada computador cliente tomar la responsabilidad de enviar las solicitudes al servidor que mantiene el grupo de archivos relevante.

Los identificadores de grupo de archivos deben ser únicos a lo largo de un sistema distribuido. Puesto que se pueden trasladar grupos de archivos, y los sistemas distribuidos que están separados inicialmente se pueden mezclar para formar un sistema único, la única forma de asegurar que los identificadores de grupo serán siempre distintos en un sistema dado es generarlos con un algoritmo que garantice la unicidad global. Por ejemplo, cuando se crea un nuevo grupo de archivos, se puede generar un identificador único de archivos concatenando la dirección IP de 32 bits de la máquina que crea el nuevo grupo con un entero de 16 bits derivado de la fecha, produciendo un entero único de 48 bits:

	32 bits	*16 bits*
identificador de grupo de archivos:	dirección IP	fecha

Observe que la dirección IP no debe utilizarse para el propósito de localizar el grupo de archivos, puesto que puede ser trasladado a otro servidor. En lugar de eso, hay que mantener en el servicio de archivos una traducción entre identificadores de grupo y servidores.

8.3. SISTEMA DE ARCHIVOS EN RED DE SUN (NFS)

La Figura 8.8 representa la arquitectura de NFS de Sun. Sigue el modelo abstracto definido en la sección precedente. Todas las implementaciones de NFS soportan el protocolo de NFS: un conjunto de llamadas a procedimientos remotos que proporcionan el medio para que los clientes realicen operaciones en un almacén de archivos remotos. El protocolo NFS es independiente del sistema operativo pero fue desarrollado originalmente para su utilización en redes de sistemas UNIX, y nosotros describiremos la implementación del protocolo NFS en UNIX (versión 3).

El módulo *servidor NFS* reside en el núcleo de cada computador que actúa como un servidor NFS. Las solicitudes que se refieren a archivos en un sistema de archivos remoto se traducen en el módulo cliente a operaciones del protocolo NFS y después se trasladan al módulo servidor NFS en el computador que mantiene el sistema de archivos relevante.

Los módulos cliente y servidor NFS se comunican utilizando llamadas a procedimientos remotos. El sistema RPC de SUN, descrito en la Sección 5.3.1, se desarrolló para su uso en NFS. Puede configurarse para utilizar UDP o TCP, y el protocolo NFS es compatible con ambos. Se incluye un servicio de enlace que permite a los clientes encontrar los servicios de una máquina a partir del nombre. La interfaz RPC para el servidor NFS es abierta: cualquier proceso puede enviar solicitudes a un servidor NFS. Si las solicitudes son válidas e incluyen credenciales válidas del usuario, serán ejecutadas. El envío de credenciales firmadas del usuario puede requerirse como una característica adicional de seguridad, como lo puede ser la encriptación de los datos para privacidad e integridad.

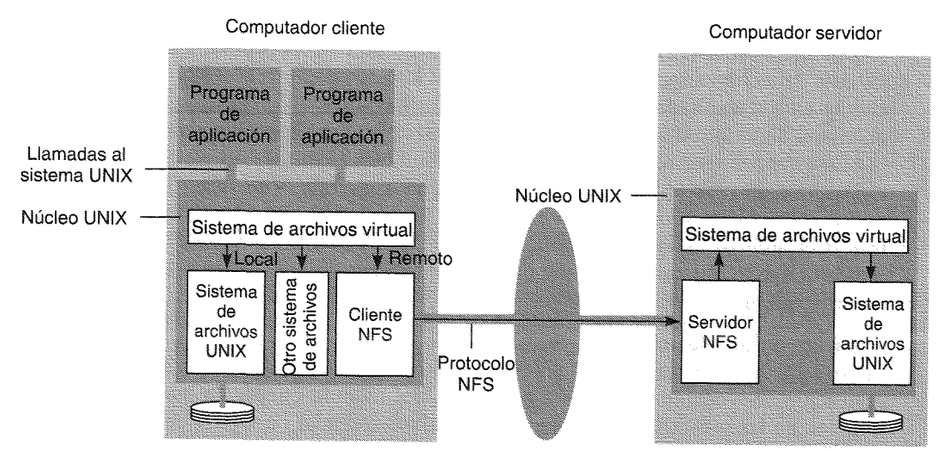


Figura 8.8. Arquitectura NFS.

◇ **Sistema de archivos virtuales.** La Figura 8.8 expone claramente que NFS proporciona acceso transparente: los programas del usuario pueden realizar operaciones sobre los archivos locales o remotos sin distinción. Otros sistemas de archivos distribuidos pueden considerar que permiten las llamadas al sistema de UNIX, y si lo hacen, podrían integrarse de la misma forma.

La integración se obtiene mediante un módulo de sistema de archivos virtual (VFS), que ha sido añadido al núcleo de UNIX para distinguir entre archivos remotos y locales y para traducir los identificadores de archivo, independientes de UNIX, utilizados por NFS en los identificadores de archivo internos utilizados en UNIX y otros sistemas de archivos. En resumen, VFS mantiene la pista de los sistemas de archivos que están actualmente disponibles tanto local como remotamente, pasa cada solicitud al módulo del sistema local apropiado (el sistema de archivos UNIX, el módulo cliente NFS o el módulo de servicio de otro sistema de archivos).

Los identificadores de archivo utilizados en NFS se llaman *apuntadores de archivo* (*file handles*). Un apuntador de archivo es opaco para los clientes y contiene toda la información que necesita el servidor para distinguir un archivo individual. En las implementaciones UNIX de NFS, el apuntador de archivo se deriva del *número de i-nodo* del archivo, añadiendo dos campos extra como sigue (el número de i-nodo de un archivo UNIX es un número que sirve para identificar y localizar el archivo en el sistema de archivos en el que está almacenado):

Apuntador de archivo:	Identificador del *filesystem*	Número de i-nodo del archivo	Número de generación de i-nodos

NFS adopta el *filesystem* (volumen) montable de UNIX como la unidad de agrupación de archivos definida en la sección precedente. (Nota de terminología: el término *filesystem* —*volumen*— se refiere al conjunto de archivos mantenidos en un dispositivo de almacenamiento o partición, mientras que las palabras sistema de archivos se refieren al componente software que proporciona el acceso a los mismos). El campo *identificador del volumen* es un número único que se reserva para cada volumen cuando se crea (y en la implementación UNIX se almacena en el superbloque del sistema de archivos). La *generación del número de i-nodo* es necesaria porque en el sistema de archivos normal de UNIX los números de i-nodo se reutilizan después de la eliminación del archivo. En las extensiones VFS al sistema de archivos UNIX, se almacena un número de generación con cada archivo y se incrementa cada vez que se reutiliza el número de i-nodo (por ejemplo, en una llamada *creat* del sistema UNIX). El cliente obtiene el primer apuntador de archivo para un sistema de archivos remotos cuando lo monta. Los apuntadores de archivos se pasan del servidor al cliente en los resultados de las operaciones *lookup*, *create* y *mkdir* (consultar la Figura 8.9) y del cliente al servidor en la lista de argumentos de todas las operaciones del servidor.

La capa del sistema de archivos virtual tiene una estructura VFS por cada sistema de archivos montado y un *v-nodo* por archivo abierto. Una estructura VFS relaciona un sistema de archivos remoto con el directorio local en el que está montado. El v-nodo contiene un indicador para mostrar si el archivo es local o remoto. Si el archivo es local, el v-nodo contiene una referencia al índice del archivo local (un i-nodo en una implementación UNIX). Si el archivo es remoto, contiene un apuntador al archivo remoto.

◇ **Integración del cliente.** El cliente NFS juega el papel descrito para el módulo cliente en nuestro modelo arquitectónico, proporcionado una interfaz idónea para su uso por programas de aplicación convencionales. Pero a diferencia del módulo cliente de nuestro modelo, emula la semántica de las primitivas del sistema de archivos estándar de UNIX de forma precisa y está integrado con el núcleo UNIX. Está integrado con el núcleo y no proporcionado como una librería para cargar en los procesos clientes de modo que:

- Los programas de usuario pueden acceder a los archivos mediante llamadas del sistema de UNIX sin recompilación o recarga.
- Un único módulo cliente sirve a todos los procesos del nivel del usuario, con una caché compartida de los bloques utilizados recientemente (se describirán posteriormente).
- La clave de encriptación utilizada para autenticar las ID del usuario pasadas al servidor (ver más adelante) puede retenerse en el núcleo, previniendo la impersonación por clientes a nivel de usuario.

El módulo cliente de NFS coopera con el sistema de archivos virtual en cada máquina cliente. Funciona de una manera semejante al sistema de archivos convencional de UNIX, transfiriendo bloques de archivos hacia y desde el servidor y haciendo *caching* de los bloques en la memoria local cuando es posible. Comparte el mismo búfer de caché que el utilizado por el sistema de entrada-salida local. Pero puesto que varios clientes en diferentes máquinas pueden acceder simultáneamente al mismo archivo remoto, se plantea un nuevo y significativo problema de consistencia de caché.

◇ **Control de acceso y autenticación.** A diferencia del sistema de archivos convencional UNIX, el servidor NFS es sin estado y no mantiene archivos abiertos en nombre de sus clientes. Por lo tanto el servidor debe comprobar la identidad del usuario frente a los atributos de acceso del archivo en cada solicitud, para ver si el usuario tiene permiso de acceso al archivo de la forma solicitada. El protocolo Sun RPC requiere que los clientes envíen la información de autenticación del usuario (por ejemplo, los convencionales 16 bits de la ID del usuario y del grupo de UNIX) con cada solicitud y ésta se comprueba frente a los permisos de acceso en los atributos del archivo. Estos parámetros adicionales no se muestran en nuestra revisión del protocolo NFS de la Figura 8.9, vienen dados automáticamente por el sistema RPC.

En la forma más simple, hay una laguna de seguridad en este mecanismo de control de acceso. Un servidor NFS proporciona una interfaz RPC convencional en un puerto bien conocido en cada máquina y cualquier proceso puede comportarse como un cliente, enviando solicitudes al servidor para acceder o actualizar un archivo. El cliente puede modificar las llamadas RPC para incluir la ID de cualquier usuario, haciéndose pasar por el usuario sin su conocimiento o permiso. Esta laguna de seguridad ha sido cerrada con el uso de una opción en el protocolo RPC para la encriptación de la información de autenticación del usuario. Más recientemente, se ha integrado Kerberos en Sun NFS para proporcionar una solución más fuerte y completa a los problemas de autenticación y seguridad del usuario, que nosotros describimos más adelante.

◇ **Interfaz del servidor NFS.** En la Figura 8.9 se ve una representación simplificada de la interfaz RPC proporcionada por el servidor NFS (definida en RFC 1813 [Callaghan y otros 1995]). Las operaciones NFS de acceso al archivo *read*, *write*, *getattr* y *setattr* son casi idénticas a las operaciones *Lee*, *Escribe*, *DameAtributos* y *PonAtributos* definidas en nuestro modelo de servicio de archivos planos (Figura 8.6). La operación *lookup* y la mayoría de las demás operaciones de directorio definidas en la Figura 8.9 son semejantes a las de nuestro modelo de servicios de directorio (Figura 8.7).

Las operaciones de archivo y directorio están integradas en un único servicio, la creación e inserción de nombres de archivo en directorios se realiza en una única operación *create* que toma el nombre del nuevo archivo y el apuntador de archivo del directorio destino (*aadir*) como argumentos. Las otras operaciones NFS sobre directorios son *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* y *statfs*. Se parecen a sus equivalentes UNIX con la excepción de *readdir*, que proporciona un método independiente de la representación para leer el contenido de directorios, y *statfs*, que proporciona la información del estado en los sistemas de archivos remotos.

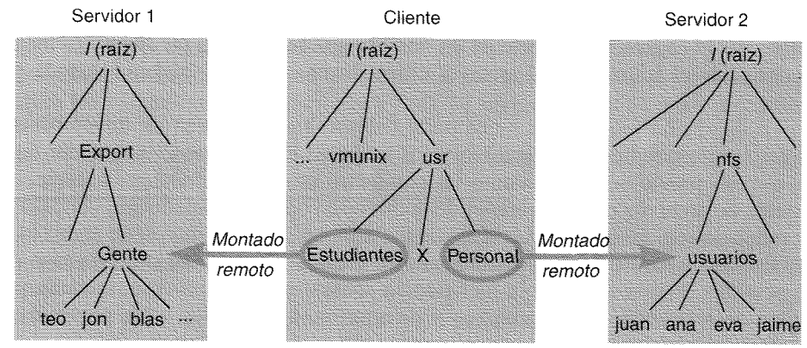
◇ **Servicio de montado.** El montado de los sub-árboles de los sistemas de archivos remotos por los clientes está soportado por un proceso de *servicio de montado* separado que se ejecuta a

lookup(aadir, nombre) → *aa, atrib*	Devuelve el apuntador del archivo y los atributos para el archivo *nombre* en el directorio *aadir*.
create(aadir, nombre, atrib) → *aanuevo, atrib*	Crea un nuevo archivo *nombre* en el directorio *aadir* con los atributos *atrib* y devuelve el nuevo apuntador de archivo y sus atributos.
remove(aadir, nombre) → *estado*	Elimina el archivo *nombre* del directorio *aadir*.
getattr(aa) → *atrib*	Devuelve los atributos del archivo *aa*. (Igual que la llamada UNIX *stat*.)
setattr(aa) → *atrib*	Establece los atributos (modo, ID usuario, ID grupo, tamaño, tiempo de acceso y tiempo de modificación de un archivo). Si se pone el tamaño a 0 se trunca el archivo.
read(aa, despl, conteo) → *atrib, datos*	Devuelve hasta *conteo* bytes de datos desde el archivo, comenzando en *despl*. También devuelve los atributos más recientes del archivo.
write(aa, despl, conteo, datos) → *atrib*	Escribe *conteo* bytes de datos sobre el archivo, comenzando en *despl*. También devuelve los atributos del archivo tras la operación de escritura.
rename(aadir, nombre, destaadir, destnombre) → *estado*	Cambia el nombre del archivo *nombre* del directorio *aadir* en *destnombre* del directorio *destaadir*.
link(nuevoaadir, nuevonombre, aadir, nombre) → *estado*	Crea una entrada *nuevonombre* en el directorio *nuevoaadir* que se refiere al archivo *nombre* en el directorio *aadir*.
symlink(nuevoaadir, nuevonombre, texto) → *estado*	Crea una entrada *nuevonombre* en el directorio *nuevoaadir*, con un *enlace simbólico* con el valor *texto*. El servidor no interpreta *texto* pero crea un archivo de enlace simbólico para alojarlo.
readlink(aa) → *texto*	Devuelve el *texto* asociado con el archivo de enlace simbólico identificado por *aa*.
mkdir(aadir, nombre, atrib) → *nuevoaa, atrib*	Crea un nuevo directorio *nombre* con los atributos *atrib* y devuelve el nuevo apuntador de archivo (*nuevoaa*) y sus atributos.
rmdir(aadir, nombre) → *estado*	Elimina el directorio vacío *nombre* del directorio padre *aadir*. Falla si el directorio no está vacío.
readdir(aadir, cookie, conteo) → *entradas*	Devuelve *conteo* bytes de entradas de directorio desde el directorio *aadir*. Cada entrada contiene un nombre de archivo, un apuntador a archivo, y un puntero opaco a la siguiente entrada del directorio, denominado *cookie* (galletita). *cookie* se emplea en las siguientes llamadas a *readdir* para comenzar la lectura desde la siguiente entrada. Si se pone el valor de *cookie* a 0, lee desde la primera entrada.
statfs(aa) → *estadosf*	Devuelve información sobre el sistema de archivos (tal como tamaño de bloque, número de bloques libres y demás) para el sistema de archivos que contiene el archivo *aa*.

Figura 8.9. Operaciones del servidor NFS (simplificado).

nivel de usuario en cada computador servidor NFS. En cada servidor, hay un archivo con un nombre bien conocido (*/etc/exports*) conteniendo los nombres de los sistemas de archivos locales que están disponibles para montado remoto. Se asocia una lista de acceso con cada nombre del sistema de archivos indicando qué máquinas están autorizadas para montar el sistema de archivos.

Los clientes utilizan una versión modificada del mandato *mount* de UNIX para solicitar el montado de un sistema de archivos remoto, especificando el nombre de la máquina remota, el nombre de la ruta de un directorio en el sistema de archivos remoto y el nombre local con el que va a ser montado. El directorio remoto puede ser cualquier sub-árbol del sistema de archivos remoto solicitado, permitiendo a los clientes montar cualquier parte del sistema de archivos remoto. El



Nota: El sistema montado en */usr/estudiantes* en el cliente, en realidad es un sub-árbol ubicado en */export/gente* en el Servidor 1: el sistema de archivos montado en /usr/personal en el cliente es realmente el sub-árbol */nfs/users* ubicado en el Servidor 2.

Figura 8.10. Sistemas de archivos locales y remotos accesibles desde un cliente NFS.

mandato *mount* modificado comunica con el proceso del servicio de montado en la máquina remota, utilizando un *protocolo de montado*. Éste es un protocolo RPC e incluye una operación que toma un nombre de ruta de directorio y devuelve el asidero de archivo del directorio especificado si el cliente tiene permiso de acceso para el sistema de archivos relevante. La ubicación (dirección IP y número de puerto) del servidor y el asidero de archivo para el directorio remoto se pasan a la capa VFS y al cliente NFS.

La Figura 8.10 muestra un *Cliente* con dos almacenes de archivos montados remotamente. Los nodos *gente* y *usuarios* en los sistemas de archivos en el *Servidor* 1 y *Servidor* 2 están montados sobre los nodos *estudiantes* y *personal* en el almacén de archivos local del *Cliente*. El significado de esto es que programas que se ejecuten en *Cliente* pueden acceder a archivos en *Servidor* 1 y *Servidor* 2 utilizando nombres de ruta como /usr/estudiantes/jon y /usr/personal/ana.

Los volúmenes remotos pueden tener un *montado rígido* (*hard-mounted*) o *flexible* (*soft-mounted*) en el computador del cliente. Cuando un proceso a nivel de usuario accede a un archivo en un volumen montado rígidamente, el proceso se suspende hasta que se completa la solicitud y si la máquina remota no está disponible por alguna razón el módulo cliente NFS continúa reintentando la solicitud hasta que se satisface. Por tanto en el caso de un fallo del servidor, los procesos a nivel de usuario se suspenden hasta que el servidor rearranca y entonces continúan justo como si no hubiera habido fallo. Pero si el volumen relevante tiene un montado flexible, el módulo cliente NFS devuelve una indicación de fallo a los procesos a nivel de usuario después de un pequeño número de reintentos. Los programas construidos adecuadamente detectarán el fallo y tomarán las acciones de recuperación e información adecuadas. Pero muchas utilidades y aplicaciones UNIX no comprueban el fallo de las operaciones de accesos a archivos y éstas se comportan de forma impredecible en el caso de fallo de un volumen con un montado flexible. Por esta razón, muchas instalaciones utilizan exclusivamente montado rígido, con la consecuencia que los programas son incapaces de recuperarse adecuadamente cuando un servidor NFS está indispuesto durante un período de tiempo significativo.

◇ **Traducción de un nombre de ruta.** Los sistemas de archivos UNIX traducen nombres de ruta de archivo compuestos a referencias de i-nodo, en un proceso paso a paso cuando se utilicen las llamadas del sistema *open*, *creat* o *stat*. En NFS, los nombres de ruta no pueden traducirse en el servidor, porque el nombre puede cruzar un «punto de montado» en el cliente, los directorios que mantienen diferentes partes de un nombre compuesto pueden residir en volúmenes en diferentes

servidores. Por lo tanto los nombres de ruta se analizan y su traducción se realiza de manera interactiva por el cliente. Cada parte de un nombre que se refiere a un directorio montado remotamente se traducido al apuntador del archivo utilizando una solicitud *lookup* separada para el servidor remoto.

La operación *lookup* busca una parte del nombre de ruta en un directorio dado y devuelve el asidero correspondiente del archivo y los atributos del mismo. El apuntador devuelto del archivo en el paso previo se utiliza como un parámetro en el paso *lookup* siguiente, el identificado del sistema de archivos en el apuntador de archivo se compara inicialmente con las entradas en la tabla de montado remoto mantenida en el cliente para ver si otro almacén de archivos montados remotamente debiera ser accedido. Manteniendo en caché los resultados de cada paso en las traducciones de nombres de ruta se alivia la ineficiencia aparente de este proceso, sacando partido de la proximidad de referencia de los archivos y directorios, los usuarios y los programas acceden normalmente a los archivos de solo uno o un pequeño número de directorios.

◇ **Automontador.** El automontador se añadió a la implementación UNIX de NFS para poder montar un directorio remoto dinámicamente cuando un punto de montado «vacío» es referenciado por un cliente. La implementación original del automontador se ejecutaba como un proceso UNIX a nivel de usuario en cada computador cliente. Las versiones posteriores (llamadas *autofs*, se implementaron en el núcleo de Solaris y Linux. Aquí describimos la versión original.

El automontador mantiene una tabla de puntos de montado (nombres de ruta) con una referencia a uno o más servidores NFS por cada punto. Se comporta igual que un servidor local NFS en la máquina del cliente. Cuando el módulo cliente NFS intenta resolver un nombre de ruta que incluye uno de estos puntos de montado, pasa una solicitud *lookup( )* al automontador local que localiza el volumen solicitado en su tabla y envía una solicitud de «prueba» a cada servidor de la lista. Se monta el volumen del primer servidor que responda, utilizando el servicio normal de montado. El volumen montado se enlaza con un punto de montado utilizando un enlace simbólico, por lo que el acceso a él no precisará de otras solicitudes al automontador. El acceso al archivo se realiza de la forma normal sin más referencias al automontador a menos que no haya referencias al enlace simbólico durante varios minutos. En este caso, el automontador desmonta el volumen remoto.

Las últimas implementaciones del núcleo reemplazan los enlaces simbólicos con montados reales, impidiendo algunos problemas que se presentan con las aplicaciones que hacen *caching* de los nombres de ruta temporales utilizados en los automontadores a nivel de usuario [Callaghan 1999].

Puede conseguirse una forma sencilla de replicación de sólo lectura listando los distintos servidores que contienen copias idénticas de un volumen o un sub-árbol de archivos frente a un nombre en la tabla de automontador. Esto se utiliza en sistemas de archivos utilizados frecuentemente que cambian con poca frecuencia, como los programas binarios de UNIX. Por ejemplo, se pueden mantener copias del directorio */usr/lib* y su subárbol en más de un servidor. En la primera ocasión que se abre un archivo de */usr/lib* en un cliente, se enviarán mensajes de prueba a todos los servidores, y el primero que responda será montado en el cliente. Esto proporciona un grado limitado de tolerancia a fallos y balance de carga, puesto que el primer servidor que responda será uno que no ha fallado y es probable que sea uno que no está fuertemente ocupado sirviendo otras solicitudes.

◇ **Caché en el servidor.** El mantenimiento de caché tanto en el computador del cliente como en el servidor son características indispensables de las implementaciones NFS con el fin de obtener las prestaciones adecuadas.

En los sistemas UNIX convencionales, las páginas de archivo, los directorios y los atributos de archivo que han sido leídas del disco son retenidas en un *cache búfer* en la memoria principal hasta que se precisa el espacio del búfer para otras páginas. Si un proceso efectúa entonces una solicitud de lectura o escritura de una página que ya está en la caché, podrá satisfacerse sin hacer otro acceso a disco. El modo de *lectura anticipada* (*read-ahead*) se adelanta a los accesos de lectura y busca las páginas que siguen a las que han sido leídas más recientemente, y el modo de *escritura retardada* (*delayed-write*) optimiza las escrituras: cuando una página ha sido alterada (por una solicitud de escritura), su nuevo contenido se escribe en disco sólo cuando se necesita el búfer para otra página. Para salvaguardar los datos frente a pérdidas por una caída del sistema, la operación *sync* de UNIX pasa las páginas actualizadas a disco cada 30 segundos. Las técnicas de caché funcionan en un entorno UNIX tradicional porque todas las solicitudes de lectura y escritura realizadas por los procesos a nivel de usuario pasan a través de una única caché que está implementada en el espacio del núcleo de UNIX. La caché se mantiene actualizada y los accesos a los archivos no pueden evitar la caché.

Los servidores NFS utilizan la caché en la máquina de servidor como se utiliza para otros accesos a archivos. La utilización de la caché del servidor para mantener los bloques de disco leídos recientemente no plantea ningún problema de consistencia, pero cuando un servidor realiza operaciones de escritura, se necesitan medidas extras para garantizar que los clientes pueden estar seguros que los resultados de las operaciones de escritura son persistentes, incluso cuando ocurren caídas del servidor. En la versión 3 del protocolo NFS, la operación *write* ofrece para esto dos opciones (no representadas en la Figura 8.9):

1. Los datos recibidos de los clientes en las operaciones de *escritura* se almacenados en la memoria caché en el servidor *y* se escriben en disco antes de que se envíe una respuesta al cliente. Esto se llama *escritura a través (write-through)* de la caché. El cliente puede estar seguro de que los datos son almacenados persistentemente en el momento en el que se ha recibido la respuesta.
2. Los datos en las operaciones de *escritura* sólo se almacenan en la memoria caché. Éstos serán escritos en disco cuando se reciba una operación de *consumación* (*commit*) para el archivo relevante. El cliente puede estar seguro que los datos están almacenados persistentemente sólo cuando se ha recibido una respuesta a una operación de *consumación* para el archivo relevante. Los clientes de NFS estándar utilizan este modo de operación, emitiendo una *consumación* cuando se cierra un archivo que fue abierto para escritura.

La *consumación* es una operación adicional proporcionada en la versión 3 del protocolo NFS, se añadió para superar un cuello de botella en las prestaciones producido por el modo de operación de *escritura a través* en los servidores que reciben un gran número de operaciones *write*.

El requisito de *escritura a través* en los sistemas de archivos distribuidos es una instancia de los modos independientes de fallos discutidos en el Capítulo 1, los clientes continúan trabajando cuando un servidor falla, y los programas de aplicación pueden realizar acciones en la suposición de que los resultados de las escrituras previas han sido consumados en el almacenamiento del disco. Esto es improbable que ocurra en el caso de actualizaciones de archivos locales, dado que el fallo de un sistema de archivos locales es casi seguro produzca el fallo de todos los procesos de aplicación ejecutándose en el mismo computador.

◇ **Caché en el cliente.** El módulo cliente NFS emplea caché para los resultados de las operaciones *read*, *write*, *getattr*, *lookup* y *readdir*, con el fin de reducir el número de solicitudes transmitidas a los servidores. La caché del cliente introduce el potencial para que existan diferentes versiones de archivos o porciones de archivos en diferentes nodos cliente, porque las escrituras por un cliente no producen la actualización inmediata de las copias en la caché del mismo archivo en otros clientes. En su lugar, los clientes son responsables de sondear al servidor para comprobar la actualidad de los datos en la caché que ellos mantienen.

Para validar los bloques en la caché antes de que sean utilizados se emplea un método basado en marcas de tiempo. Cada elemento de datos o metadatos de la caché se etiqueta con dos marcas de tiempo:

Tc es el tiempo en el que la entrada en la caché fue validada últimamente.
Tm es el tiempo en el que el bloque fue modificado por última vez en el servidor.

Una entrada en la caché es válida en el tiempo T si $T - Tc$ es menor que un intervalo de refresco t, o si el valor registrado para Tm en el cliente concuerda con el valor Tm en el servidor (esto es, los datos no han sido modificados en el servidor desde que se realizó la entrada en la caché). Formalmente la condición de validez es:

$$(T - Tc < t) \vee (Tm_{cliente} = Tm_{servidor})$$

La selección de un valor para t viene del compromiso entre consistencia y eficiencia. Un intervalo de refresco muy corto conlleva una aproximación muy próxima a la consistencia de una copia, al coste de una carga relativamente pesada de llamadas al servidor para comprobar el valor de $Tm_{servidor}$. En los clientes Solaris de Sun, t se coloca de forma adaptativa para los archivos individuales a un valor en el rango de 3 a 30 segundos dependiendo de la frecuencia de actualizaciones en el archivo. Para directorios el rango es de 30 a 60 segundos reflejando el riesgo más bajo de actualizaciones concurrentes.

Puesto que los clientes NFS no pueden determinar si un archivo está siendo compartido o no, el procedimiento de validación debe ser utilizado para todos los accesos al archivo. Cuando se utiliza una entrada en la caché se realiza una comprobación de validez. La primera mitad de la condición de validez puede evaluarse sin acceder al servidor. Si es verdadera, entonces la segunda mitad no necesita ser evaluada; si es falsa, se obtiene el valor actual de $Tm_{servidor}$ (por medio de una llamada *getattr* al servidor) y se compara con el valor local de $Tm_{cliente}$. Si son iguales, entonces la entrada a la caché se considera que es válida y el valor de Tc para la entrada en la caché es actualizada al tiempo actual. Si difieren, entonces los datos en la caché han sido actualizado en el servidor y la entrada en la caché es invalidada, resultando en una solicitud para el servidor por los datos relevantes.

Se utilizan varias medidas para reducir el tráfico de las llamadas *getattr* al servidor:

- Cuando se recibe un nuevo valor de $Tm_{servidor}$ en un cliente, se aplica para todas las entradas en la caché derivadas del archivo relevante.
- Los valores actuales del atributo son enviados se acarrean con los resultados de cada operación en un archivo, y si el valor de $Tm_{servidor}$ ha cambiado el cliente lo utiliza para actualizar las entradas en la caché relativas al archivo.
- El algoritmo adaptativo para fijar el intervalo de refresco t indicado anteriormente reduce considerablemente el tráfico para la mayoría de los archivos.

El procedimiento de validación no garantiza el mismo nivel de consistencia de los archivos que es proporcionado en los sistemas convencionales UNIX, puesto que las actualizaciones recientes no siempre son visibles para los clientes que comparten un archivo; hay dos fuentes de retardo temporal, el retardo después de la escritura antes de que los datos dejen la caché en el núcleo de actualización del cliente y la «ventana» de tres segundos para la validación de la caché. Afortunadamente, la mayoría de las aplicaciones UNIX no dependen de forma crítica de la sincronización de las actualizaciones de archivo, y se han informado pocas dificultades con este origen.

Las escrituras se mantienen de forma diferente. Cuando se modifica una página en la caché se marca como sucia y se planifica para ser volcada al servidor asíncronamente. Las páginas modificadas se vuelcan cuando se cierra el archivo u ocurre un *sync* en el cliente, y esto ocurre más frecuentemente si hay bio-demonios en uso (ver a continuación). Esto no proporciona la misma garantía de persistencia que en la caché del servidor, pero emula el comportamiento para escrituras locales.

Para implementar lectura de la entrada y *escritura retardada*, el cliente NFS necesita realizar asíncronamente algunas lecturas y escrituras. Esto se consigue en las implementaciones UNIX de NFS mediante la inclusión de uno o más procesos *bio-demonio* en cada cliente (*Bio* indica *block imput-output —entrada-salida de bloques—*, el término demonio se utiliza a menudo para referirse a procesos a nivel de usuario que realizan tareas del sistema). El papel de los bio-demonios es realizar operaciones de lectura adelantada y escritura retrasada. Después de cada solicitud se avisa al bio-demonio, y éste solicita la transferencia de el siguiente bloque del archivo desde el servidor a la caché del cliente. En el caso de escritura el bio-demonio enviará un bloque al servidor cuando se haya llenado un bloque una operación del cliente. Los bloques de directorio se envían cuando se ha producido una modificación.

Los procesos bio-demonio mejoran las prestaciones, asegurando que el módulo cliente no se bloquea esperando la vuelta de las lecturas o la consumación de escrituras en el servidor. No hay un requisito lógico, puesto que en ausencia de lectura adelantada, una operación *read* en un proceso de usuario disparará una solicitud síncrona para el servidor relevante, y los resultados de *write* en los procesos de usuario serán transferidos al servidor cuando el archivo relevante se cierra o cuando el sistema de archivos virtual en el cliente lanza una operación *sync*.

◇ **Otras optimizaciones.** El sistema de archivos de Sun está basado en el Sistema Rápido de Archivos (*Fast File System*, FFS) de UNIX BSD que utiliza bloques de disco de ocho kbytes, lo que resulta en menos llamadas al sistema de archivos para acceso a archivos secuenciales que en los sistemas UNIX anteriores. Los paquetes UDP utilizados para la implementación de Sun RPC se extienden a nueve kilobytes, permitiendo que una llamada RPC conteniendo un bloque entero como argumento sea transferida en un único paquete y minimice el efecto de la latencia de red cuando se leen archivos secuencialmente. En NFS versión 3 no hay límite en el tamaño máximo de los bloques de archivo que pueden ser mantenidos en las operaciones *read* y *write*, los clientes y los servidores pueden negociar tamaños más grandes que ocho kbytes si son capaces de mantenerlos.

Como se ha mencionado anteriormente la información del estado del archivo en la caché en los clientes debe actualizarse al menos cada tres segundos para los archivos activos. Para reducir la consecuente carga del servidor resultante de solicitudes *getattr*, todas las operaciones que se refieran a archivos o directorios son tomadas como solicitudes *getattr* implícitas, y los valores de los atributos actuales son acarreados con el resto de los resultados de la operación.

◇ **Haciendo seguro NFS con Kerberos.** En la Sección 7.6.2 se ha descrito el sistema de autenticación desarrollado en el MIT Kerberos, que ha llegado a ser un estándar industrial para servidores de intranet seguros contra accesos no autorizados y ataques de impostores. La seguridad de las implementaciones de NFS ha sido incrementada con el uso del esquema Kerberos para autenticar clientes. En esta subsección, describimos la «Kerberización» de NFS tal y como fue realizada por los diseñadores de Kerberos.

En la implementación estándar original de NFS, la identidad del usuario se incluye en cada solicitud en forma de un identificador numérico no encriptado (el identificador fue encriptado en versiones posteriores de NFS). NFS no realiza ningún otro paso para comprobar la autenticidad del identificador proporcionado. Esto implica un alto grado de confianza en la integridad del computador cliente y en su software por NFS, mientras que el propósito de Kerberos y otros sistemas de seguridad basados en autenticación es reducir a un mínimo el rango de componentes sobre los que se supone dicha confianza. Esencialmente, al utilizar NFS en un entorno «Kerberizado» sólo se aceptarán solicitudes de clientes cuya identidad pueda ser comprobada que ha sido autenticada por Kerberos.

Una solución obvia, considerada por los desarrolladores de Kerberos, fue cambiar la naturaleza de las credenciales solicitadas por NFS para ser un sistema Kerberos basado en tickets y un autenticador de pleno derecho. Pero como NFS está implementado como un servidor sin estado, cada solicitud de acceso a un archivo individual se despacha por su validez intrínseca y entonces los datos de autenticación deberían incluirse en cada solicitud. Esto fue considerado inaceptablemente costoso dado el tiempo requerido para realizar la encriptaciones necesarias y que debieran llevarse a cabo añadiendo la biblioteca cliente Kerberos al núcleo de todas las estaciones de trabajo.

En lugar de eso, se adoptó una aproximación híbrida en la que se proporciona el servidor de montado NFS con todos los datos de autenticación de Kerberos para el usuario cuando se montan sus volúmenes raíz e inicial (*home*). Los resultados de la autenticación, incluyendo el identificador numérico convencional del usuario y la dirección del computador del cliente, se retienen en el servidor con la información de montado de cada volumen (aunque el servidor NFS no retiene el estado relativo de los procesos individuales cliente, retiene los montados actuales en cada computador cliente).

En cada solicitud de acceso al archivo, el servidor NFS comprueba el identificador del usuario y las direcciones del remitente y proporciona el acceso sólo si concuerdan con los almacenados en el servidor para el cliente relevante en el tiempo de montado. Esta aproximación híbrida implica un coste mínimo adicional y es segura contra la mayoría de las formas de ataque, proporciona que sólo un usuario en el tiempo puede entrar en cada computador del cliente. En el MIT, el sistema está configurado de esta forma. Implementaciones recientes de NFS incluyen la autenticación Kerberos como una de las varias opciones de autenticación, y se aconseja que los sitios que también ejecutan servidores Kerberos empleen esta opción.

◇ **Prestaciones.** Las primeras cifras sobre prestaciones comunicadas por Sandberg [1987] mostraron que el uso de NFS no imponía usualmente una penalización de las prestaciones, en comparación con el acceso a los archivos almacenados en los archivos locales. Sandberg identificó dos áreas de problemas restantes:

- El uso frecuente de la llamada *getattr* con el fin de buscar marcas de tiempo de los servidores para la validación de la caché.
- Las prestaciones relativamente pobres de la operación *write* porque se utilizaba la *escritura a través* en el servidor.

También señaló que las escrituras son relativamente infrecuentes en las cargas de trabajo UNIX típicas (alrededor del 5 % de todas las llamadas al servidor), y el coste de la *escritura a través* es, por tanto, tolerable excepto cuando se escriben grandes archivos en el servidor. La versión de NFS que él comprobó no incluía el mecanismo *commit* esbozado anteriormente, que produce una mejora sustancial en las prestaciones de escritura en las versiones actuales. Sus resultados también muestran que la operación *lookup* supone el 50 % de las llamadas al servidor. Esto es consecuencia del método de traducción de nombres de ruta paso a paso requerido por la semántica de nombrado de archivos UNIX.

Hoy en día se realizan medidas regularmente, por Sun y otros implementadores de NFS, utilizando una versión actualizada de un conjunto exhaustivo de programas de prueba conocidos como LADDIS [Keith y Wittle 1993]. Los resultados actuales y pasados están disponibles en [www.spec.org]. Las prestaciones se resumen aquí para implementaciones del servidor NFS de diferentes vendedores y variadas configuraciones hardware. Resultados recientes incluyen un rendimiento de 5.011 operaciones del servidor por segundo, con una latencia promedio de 3,71 milisegundos y una latencia máxima de 8 ms (1 × 450 MHz Pentium III CPU, 34 discos en un controlador Ethernet de un Gbps y sistema operativo de tiempo real dedicado) y un rendimiento de 29.083 operaciones del servidor por segundo con (latencia promedio)/máxima de 4,25/7,8 ms (24 × 450 MHz IBM RS64-III CPU, 289 discos en cuatro controladores, 5 redes de 1 Gbps, AIX UNIX). Estas figuras indican que es probable que NFS ofrezca una solución muy efectiva para las necesidades de almacenamiento distribuido en las intranets de la mayoría de los tamaños y tipos de uso, oscilando por ejemplo de una carga tradicional UNIX de desarrollo por varios cientos de ingenieros de software a una batería de servidores web accediendo y sirviendo material de un servidor NFS.

◇ **Resumen NFS.** NFS de Sun sigue fuertemente nuestro modelo abstracto. El diseño resultante proporciona buena transparencia de ubicación y acceso si el servicio de montado NFS se utiliza adecuadamente, para producir espacios de nombre semejantes en todos los clientes. NFS soporta hardware y sistemas operativos heterogéneos. La implementación del servidor NFS es sin estado, permitiendo a los clientes y servidores recuperar la ejecución después de un fallo sin necesidad de ningún procedimiento de recuperación. La migración de archivos o sistemas de archivos no está soportada, excepto en el nivel de intervención manual para reconfigurar las directivas de montado después de un movimiento de un sistema de archivos a una nueva ubicación.

Las prestaciones de NFS mejoran mucho gracias a la caché de bloques de archivo en cada computador cliente. Esto es importante para la obtención de prestaciones satisfactorias pero produce alguna desviación de la semántica de actualización de una copia de archivo estricta UNIX.

Los otros objetivos de diseño de NFS y las extensiones a ellos que han sido conseguidas, se discuten a continuación.

Transparencia de acceso: el módulo cliente NFS proporciona una interfaz de programación de aplicación para los procesos locales que es idéntica a la interfaz del sistema operativo local. Por tanto en un cliente UNIX, los accesos a archivos remotos se realizan utilizando llamadas al sistema normales de UNIX. No se precisa modificar los programas existentes para poder trabajar correctamente con archivos remotos.

Transparencia de ubicación: cada cliente establece un espacio de nombres de archivo añadiendo directorios montados en volúmenes remotos sobre su espacio local de nombres. Los sistemas de archivos han de ser *exportados* por el nodo que los mantiene y *montados remotamente* por un cliente antes que ellos puedan ser accedidos por procesos ejecutándose en el cliente (véase la Figura 8.10). El punto de la jerarquía de nombres del cliente donde aparece montado remotamente un sistema de archivos lo determina el cliente, por tanto NFS no fuerza un espacio de nombres de archivo único a través de la red; cada cliente ve un conjunto de sistemas de archivos remotos que es determinado localmente, y los archivos remotos pueden tener diferentes nombres de ruta en diferentes clientes, pero se puede establecer un espacio de nombres uniforme en cada cliente mediante las tablas de configuración apropiadas, logrando el objetivo de transparencia de ubicación.

Transparencia de movilidad: los volúmenes (en el sentido UNIX, esto es, subárboles de archivos) pueden ser reubicados entre servidores, pero las tablas de montado remoto en cada cliente deben ser actualizadas separadamente para permitir a los clientes el acceso al sistema de archivos en su nueva ubicación, por lo que la transparencia de migración no está totalmente conseguida en NFS.

Escalabilidad: las cifras de prestaciones publicadas muestran que se pueden construir servidores NFS que mantengan cargas reales muy grandes, de una manera eficiente y beneficiosa. Se pueden incrementar las prestaciones de un único servidor mediante la adición de procesadores, discos y controladores. Cuando los límites de esos procesos son alcanzados, hay que instalar servidores adicionales y los sistemas de archivos deben ser reubicados entre ellos. La efectividad de esa estrategia está limitada por la existencia de archivos de uso muy frecuente, son archivos sencillos que son accedidos tan frecuentemente que el servidor llega al límite de prestaciones. Cuando las cargas exceden el máximo de prestaciones disponible con esa estrategia, una solución mejor es emplear un sistema de archivos distribuidos que soporte la replicación de los archivos actualizables (como Coda, descrito en el Capítulo 14), o uno como AFS que reduce el tráfico del protocolo haciendo caché de los archivos completos. En la Sección 8.5 discutiremos otras aproximaciones a la escalabilidad.

Replicación de archivos: los almacenes de archivos de *sólo lectura* pueden ser replicados en varios servidores NFS, pero NFS no soporta la replicación de archivos actualizables. El Servicio de Información de Red de Sun (*Network Information Service*, NIS) es un servicio separado disponible para su uso con NFS que soporta la replicación de bases de datos sencillas organiza-

das como pares clave-valor (por ejemplo los archivos del sistema UNIX */etc/passwd* y */etc/hosts*). Éste gestiona la distribución de actualizaciones y accesos a los archivos replicados basada en un modelo de replicación sencilla maestro-esclavo (también conocido como modelo de *copia primaria*, que se discutirá en el Capítulo 14) que provee la replicación de parte o toda la base de datos en cada sitio. NIS proporcionan un almacén compartido para información del sistema que cambia infrecuentemente y no requiere que las actualizaciones ocurran simultáneamente en todos los sitios.

Heterogeneidad del hardware y del sistema operativo: NFS ha sido implementado para casi todos los sistemas operativos y plataformas hardware conocidos y está soportado por una variedad de sistemas de archivos.

Tolerancia a fallos: la naturaleza sin estado e idempotente del protocolo de acceso a archivos NFS asegura que los modos de fallo observados por los clientes cuando acceden a archivos remotos son similares a aquéllos de acceso a archivos locales. Cuando un servidor falla, el servicio que proporciona se suspende hasta que se rearranca el servidor, pero una vez que ha sido rearrancado los procesos cliente a nivel de usuario proceden desde el punto en el que fue interrumpido el servicio sin darse cuenta del fallo (excepto en el caso de acceso a sistemas de archivos remotos *con montado flexible*). En la práctica el montado rígido se ha utilizado en la mayoría de los casos, y esto propende a impedir que los programas de aplicación manejen los fallos del servidor de modo elegante.

El fallo de un computador cliente o de un proceso a nivel de usuario en un cliente no tiene efecto sobre ningún servidor que el pueda estar utilizando, puesto que los servidores no mantienen el estado en nombre de sus clientes.

Consistencia: hemos descrito el comportamiento de las actualizaciones con cierto detalle. NFS proporciona una cercana aproximación a la semántica de *una copia* y coincide con las necesidades de la gran mayoría de las aplicaciones, pero no podemos recomendar el uso de archivos compartidos vía NFS para la comunicación o coordinación fuerte entre procesos en diferentes computadores.

Seguridad: las necesidades de seguridad en NFS sólo emergen al conectar la mayoría de las intranets con Internet. La integración de Kerberos con NFS fue un salto fundamental hacia delante. Otros desarrollos recientes incluyen la opción de utilizar una implementación RPC segura (RPCSEC- GSS, documentada en RFC 2203 [Eisler y otros 1997]) para la autenticación y la privacidad y seguridad de los datos transmitidos con las operaciones de lectura y escritura. Abundan las instalaciones que todavía no han desplegado estos mecanismos, y son inseguras.

Eficiencia: las prestaciones medidas de varias implementaciones de NFS y su amplia adopción para uso en situaciones que generan cargas muy pesadas son indicaciones claras de la eficiencia con la que puede ser implementado el protocolo NFS.

8.4. SISTEMA DE ARCHIVOS ANDREW

Como NFS, AFS proporciona acceso transparente a archivos compartidos remotos para los programas UNIX que se ejecutan en estaciones de trabajo. El acceso a los archivos AFS se hace mediante las primitivas usuales de los archivos UNIX, permitiendo a los programas UNIX existentes acceder a archivos AFS sin modificación o recompilación. AFS es compatible con NFS. Los servidores AFS mantienen los archivos UNIX «locales», pero el sistema de archivos en los servidores está basado en NFS, por lo que los archivos se referencian mediante apuntadores de archivo de estilo NFS más que mediante números de i-nodo, y los archivos pueden ser accedidos remotamente a través de NFS.

AFS difiere notoriamente de NFS en su diseño e implementación. Las diferencias son atribuibles principalmente a la identificación de la escalabilidad como el objetivo de diseño más importante. AFS está diseñado para trabajar bien con gran número de usuarios activos más que otros sistemas de archivos distribuidos. La estrategia clave para alcanzar la escalabilidad es la caché de los archivos completos en los nodos cliente. AFS tiene dos carcterísticas de diseño inusuales:

Servicio de archivo completo: el contenido entero de los directorios y los archivos es transmitido a los computadoress cliente por los servidores AFS (en AFS-3, los archivos más grandes de 64 kbytes son transferidos en trozos de 64 kbytes).

Caché de archivo completo: una vez que una copia de un archivo, o un trozo, ha sido transferido a un computador cliente es almacenado en una caché en el disco local. La caché contiene varios cientos de archivos, los utilizados más recientemente en ese computador. La caché es permanente sobreviviendo a los rearranques del computador cliente. Las copias locales de los archivos son utilizadas para satisfacer las solicitudes *open* de los clientes con preferencias sobre las copias remotas cuando es posible.

◇ **Escenario.** Aquí presentamos un escenario simple que ilustra el funcionamiento de AFS:

- Cuando un proceso de usuario en un computador cliente emite una llamada del sistema *open* para un archivo en un espacio de archivos compartido y no hay una copia actual del archivo en la caché local, el servidor que mantiene el archivo se localiza y se envía una solicitud para una copia del archivo.
- La copia es almacenada en el sistema de archivos UNIX local en el computador del cliente; se *abre* la copia y el descriptor UNIX resultante del archivo es devuelto al cliente.
- Las operaciones posteriores: *read*, *write* y demás, sobre el archivo por los procesos en el computador cliente, se aplican a la copia local.
- Cuando el proceso en el cliente emite una llamada del sistema *close*, si la copia local ha sido actualizada su contenido es enviado de vuelta al servidor. El servidor actualiza el contenido del archivo y las marcas de tiempo del mismo. La copia en el disco local del cliente es retenida en el caso de que se necesite de nuevo por un proceso a nivel de usuario en la misma estación de trabajo.

Discutiremos las prestaciones de AFS observadas más adelante, pero podemos hacer aquí algunas observaciones y predicciones generales basadas en las características de diseño descritas anteriormente:

- Para archivos compartidos que son actualizados infrecuentemente (aquellos que contienen el código de los comandos y librerías de UNIX) y para archivos que son accedidos normalmente por un único usuario (como la mayoría de los archivos en un directorio *home* del usuario y su sub-árbol), las copias en la caché localmente probablemente permanezcan válidas durante períodos largos, en el primer caso porque no se actualizan y en el segundo porque si se actualizan, la copia actualizada estará en la caché de la estación de trabajo del propietario. Esta clase de archivo es la abrumadora mayoría de los accesos.
- La caché local puede reservar una proporción sustancial del espacio de disco en cada estación de trabajo, pongamos 100 megabytes. Esto es normalmente suficiente para el establecimiento de un conjunto de trabajo de los archivos utilizados por un usuario. La provisión de almacenamiento caché suficiente para el establecimiento de un conjunto de trabajo asegura que los archivos de uso regular en una estación de trabajo dada son normalmente retenidos en la caché hasta que son necesarios de nuevo.
- La estrategia de diseño está basada en algunas suposiciones sobre los tamaños promedio y máximo de los archivos y la proximidad de referencia de los mismos en sistemas UNIX. Estas suposiciones se derivan de las observaciones de las cargas de trabajo UNIX típicas en entornos académicos y otros [Satyanarayanan 1981; Ousterhout y otros 1985; Floyd 1986].

Las observaciones más importantes son:

— Los archivos son pequeños; la mayoría son menores de 10 kilobytes de tamaño.
— Las operaciones de lectura en los archivos son mucho más comunes que la escritura (unas seis veces más habituales).
— El acceso secuencial es lo habitual, y el acceso aleatorio es inusual.
— La mayoría de los archivos son leídos y escritos por sólo un usuario. Cuando se comparte un archivo normalmente sólo lo modifica un usuario.
— Los archivos son referenciados a ráfagas. Si un archivo ha sido referenciado recientemente hay una alta probabilidad que sea referenciado en el futuro próximo.

Estas observaciones fueron utilizadas para guiar el diseño y optimización de AFS, *no* para restringir la funcionalidad vista por los usuarios.

- AFS trabaja mejor con las clases de archivo identificadas en el primer punto anterior. Hay un tipo importante de archivo que no concuerda con ninguna de estas clases, las bases de datos son compartidas normalmente por muchos usuarios y generalmente son actualizadas muy a menudo. Los diseñadores de AFS han excluido explícitamente las provisión de facilidades de almacenamiento para bases de datos de sus objetivos de diseño, afirmando que las restricciones impuestas por las diferentes estructuras de nombrado (esto es, acceso basado en contenido) y la necesidad para acceso a datos con gran detalle, control de concurrencia y atomicidad de las actualizaciones hace difícil el diseño de un sistema de base de datos distribuida que es también un sistema de archivos distribuido. Ellos arguyen que la provisión de funcionalidades de bases de datos distribuidas debiera ser considerada separadamente [Satyanarayanan 1989a].

8.4.1. IMPLEMENTACIÓN

El escenario anterior ilustra la operación de AFS pero deja muchas preguntas no contestadas sobre su implementación. Entre las más importantes están:

- ¿Cómo consigue el control AFS cuando una llamada del sistema *open* o *close* refiriéndose a un archivo en el espacio de archivos compartidos es emitida por un cliente?
- ¿Cómo está manteniendo el servidor el archivo localizado requerido?
- ¿Qué espacio está reservado para los archivos en la caché en las estaciones de trabajo?
- ¿Cómo asegura AFS que las copias en la caché de los archivos están actualizadas cuando los archivos pueden ser actualizados por varios clientes?

A continuación respondemos a estas preguntas.

AFS está implementado como dos componentes software que existen como procesos UNIX llamados *Vice* y *Venus*. La Figura 8.11 muestra la distribución de los procesos Vice y Venus. Vice es el nombre dado al servidor software que se ejecuta como un proceso UNIX a nivel de usuario en cada computador servidor, y Venus es un proceso a nivel de usuario que se ejecuta en cada computador cliente y corresponde al módulo cliente en nuestro modelo abstracto.

Los archivos disponibles para los procesos de usuario que se ejecutan en estaciones de trabajo son o *locales* o *compartidos*. Los archivos locales se manejan como archivos normales UNIX. Están almacenados en el disco de la estación de trabajo y están disponibles sólo para los procesos de usuario locales. Los archivos compartidos están almacenados en los servidores y las copias de ellos son introducidas en la caché en los discos locales de las estaciones de trabajo. El espacio de nombres visto por los procesos del usuario es el ilustrado en la Figura 8.12. Es una jerarquía convencional de directorios UNIX, con un subárbol específico (llamado *cmu*) conteniendo todos los archivos compartidos. Este desdoblamiento del espacio de nombres de archivos en archivos locales y compartidos conduce a alguna pérdida de la transparencia de ubicación, pero esto apenas evidente para usuarios distintos de los administradores del sistema. Los archivos locales son utilizados sólo para archivos temporales (*/tmp*) y procesos que son esenciales para el arranque de la estación de trabajo. Otros archivos estándar UNIX (como aquellos encontrados normalmente en */bin*, */lib*, etc.) están implementados como enlaces simbólicos desde los directorios locales a los archivos mantenidos en el espacio compartido. Los directorios de los usuarios están en el espacio compartido permitiendo a los usuarios acceder a sus archivos desde cualquier estación de trabajo.

El núcleo UNIX de cada estación de trabajo y servidor es una versión modificada de UNIX BSD. Las modificaciones están diseñadas para interceptar *open*, *close* y algunas otras llamadas del sistema para archivos cuando ellas se refieren a archivos en el espacio de nombres compartido y pasan entonces al proceso Venus en el computador cliente (como se ve en la Figura 8.13). Se incluye otra modificación del núcleo por razones de prestaciones y se describe más adelante.

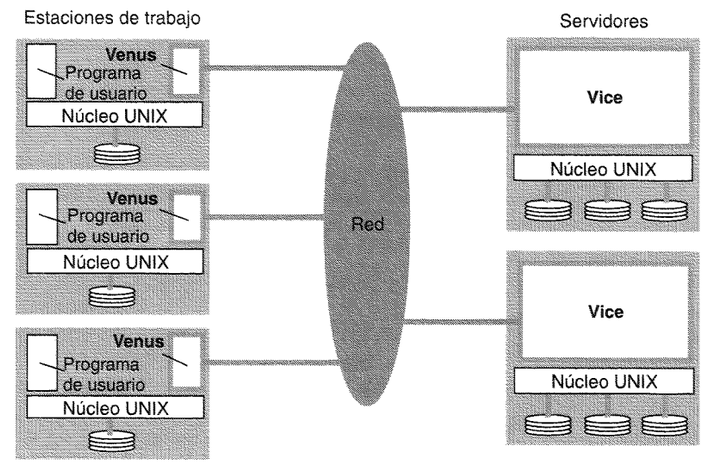


Figura 8.11. Distribución de procesos en el Sistema de Archivos Andrew.

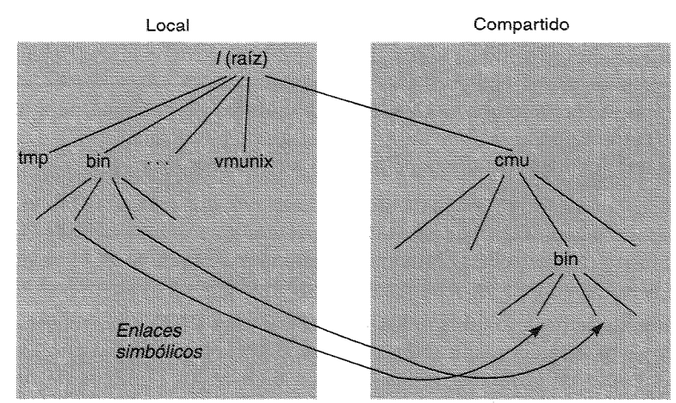


Figura 8.12. Espacio de nombres visto por los clientes de AFS.

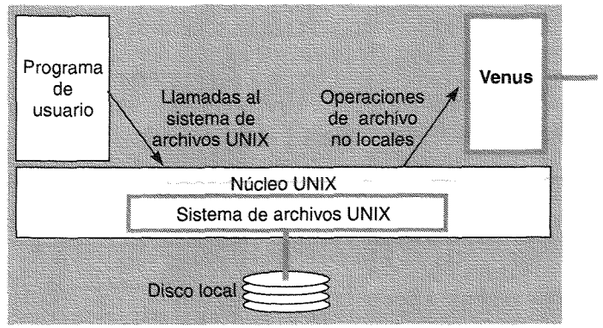


Figura 8.13. Interpretación de llamadas al sistema en AFS.

Una de las particiones de archivos en el disco local de cada estación de trabajo se usa como una caché, que mantiene las copias cacheadas de los archivos del espacio compartido. Venus administra la caché, eliminando los archivos menos recientemente utilizados cuando se adquiere un nuevo archivo desde un servidor para conseguir el espacio requerido si la partición está llena. La caché de la estación de trabajo es generalmente lo suficientemente grande para acomodar varios cientos de archivos de tamaño promedio en gran parte que la estación de trabajo sea independiente de los servidores Vice una vez que se ha introducido en la caché un espacio de trabajo de los archivos actuales del usuario y de los archivos del sistema utilizados frecuentemente.

AFS se parece al modelo abstracto de archivos descrito en la Sección 8.2 en estos aspectos:

- Los servidores Vice implementan un servicio de archivos planos, y la estructura jerárquica de directorios requerida por los programas de usuario UNIX es implementada en el conjunto de procesos Venus en las estaciones de trabajo.
- Cada archivo y directorio en el espacio de archivos compartido se reconoce por un identificador de archivo único de 96 bits (*ida*) semejante a una UFID. Los procesos Venus trasladan los nombres de rutas emitidos por los clientes a *ida*.

Los archivos se agrupan en *volúmenes* para facilitar la localización y el movimiento. Estos volúmenes son generalmente más pequeños que los volúmenes UNIX, que son la unidad de agrupamiento de archivos en NFS. Por ejemplo, los archivos personales de cada usuario están ubicados generalmente en un volumen separado. Se reservan otros volúmenes para los binarios del sistema, la documentación y el código de las bibliotecas.

La representación de cada *ida* incluye el número de volumen para el volumen que contiene el archivo (como el *identificador de grupo del archivo* en cada UFID) y un *identificador único* para asegurar que los identificadores de archivo no son reutilizados:

32 bits	*32 bits*	*32 bits*
Número de volumen	Apuntador de archivo	Unificador

Lo programas de usuario utilizan los nombres de ruta UNIX para referirse a los archivos, pero AFS utiliza los *idas* en la comunicación entre los procesos Venus y Vice. Los servidores Vice aceptan solicitudes sólo en términos de *idas*. Venus traduce los nombres de ruta proporcionados por los clientes en *idas* utilizando una búsqueda paso a paso para obtener la información de los archivos directorios mantenidos en los servidores Vice.

Proceso de usuario	*Núcleo UNIX*	*Venus*	*Red*	*Vice*
open(NombreArchivo, modo)	Si *NombreArchivo* se refiere a un archivo del espacio de archivos compartido, pasa la petición a Venus.	Comprueba la lista de archivos en la caché local. Si no está presente o no hay una *promesa de devolución de llamada*, envía una petición de archivo al servidor Vice que es el custodio del volumen que contiene el archivo.	→	Transfiere una copia del archivo y una promesa de *devolución de llamada* a la estación de trabajo. Registra la promesa de devolución de llamada.
	Abre el archivo local y devuelve el descriptor de archivo a la aplicación.	Sitúa la copia del archivo en el sistema de archivos local, inserta su nombre local en la caché de lista local y devuelve el nombre local a UNIX.	←	
read(DescriptorArchivo, Búfer, tamaño)	Realiza una operación de lectura UNIX normal sobre la copia local.			
write(DescriptorArchivo, Búfer, tamaño)	Realiza una operación de escritura UNIX normal sobre la copia local.			
close(DescriptorArchivo)	Cierra la copia local y notifica a Venus que el archivo ha sido cerrado.	Si la copia local ha sido cambiada, envía una copia al servidor Vice que es el custodio del archivo.	→	Reemplaza los contenidos del archivo y envía una *devolución de llamada* a todos los otros clientes que posean *promesas de devolución de llamada* sobre el archivo.

Figura 8.14. Implementación de las llamadas al sistema en AFS.

La Figura 8.14 describe las acciones tomadas por Vice, Venus y el núcleo UNIX cuando un proceso de usuario emite cada llamada del sistema mencionada en nuestro bosquejo de escenario anterior. La promesa de *devolución de llamada (callback)* mencionada aquí es un mecanismo para asegurar que las copias en la caché de los archivos son actualizadas cuando otro cliente cierra el mismo archivo después de actualizarlo. Este mecanismo se discute en la sección siguiente.

8.4.2. CONSISTENCIA DE LA CACHÉ

Cuando Vice proporciona una copia de un archivo al proceso Venus también le proporciona una *promesa de devolución de llamada*, un testigo emitido por el servidor Vice que es el custodio del

archivo, que garantiza que notificará al proceso Venus cuando otro cliente modifica el archivo. Las promesas de devolución de llamada se almacenan con los archivos en la caché sobre los discos de las estaciones de trabajo y tienen dos estados: *válida* o *cancelada*. Cuando un servidor realiza una solicitud para actualizar un archivo notifica a todos los procesos Venus a los que ha emitido promesas de devolución de llamada enviando *una devolución de llamada* a cada uno, una promesa de devolución de llamada es una llamada a un procedimiento remoto desde el servidor al proceso Venus. Cuando el proceso Venus recibe una devolución de llamada, coloca el testigo de la *promesa de devolución de llamada* para el archivo relevante a *cancelada*.

Cuando Venus usa *open* en nombre de un cliente, comprueba la caché. Si el archivo requerido se encuentra en la caché, se comprueba el testigo. Si su valor es *cancelada*, entonces debe buscarse una copia reciente del archivo en el servidor Vice, pero si el testigo es *válida*, entonces puede ser abierta y utilizada la copia de la caché sin referenciar a Vice.

Cuando una estación de trabajo se rearranca después de un fallo o una parada, Venus intenta retener tanto archivos de la caché en el disco local como sea posible, pero no puede presuponer que los testigos de promesa de devolución de llamada sean correctos, puesto que algunas devoluciones de llamada pueden haberse perdido. Por cada archivo con un testigo válido, Venus debe enviar una solicitud de validación de la caché conteniendo la marca de tiempo de modificación del archivo al servidor que es el custodio del archivo. Si la marca de tiempo es actual, el servidor responde con *válida* y el testigo es rehabilitado. Si la marca de tiempo muestra que el archivo está caducado, el servidor responde con *cancelada* y el testigo es colocado a *cancelada*. Las promesas de devolución de llamada deben ser renovadas antes de un *open* si ha transcurrido un tiempo *T* (normalmente del orden de unos pocos minutos) desde que el archivo fue introducido en la caché sin comunicación con el servidor. Esto se hace para tratar los posibles fallos de comunicación, que producen la pérdida de mensajes de devolución de llamada.

Este mecanismo basado en devoluciones de llamada para mantener la consistencia de la caché se adoptó porque ofrecía la aproximación más escalable, siguiendo la evaluación en el prototipo (AFS-1) de un mecanismo basado en marca de tiempo semejante al utilizado en NFS. En AFS-1, un proceso Venus que mantiene una copia en la caché de un archivo interroga al proceso Vice en cada *open* para determinar si la marca de tiempo en la copia local concuerda con la del servidor. La aproximación basada en devoluciones de llamada es más escalable porque produce comunicación entre el cliente y el servidor y actividad en el servidor sólo cuando el archivo ha sido actualizado, mientras que la aproximación por marca de tiempo produce una interacción cliente-servidor en cada *open*, incluso cuando hay una copia válida en la caché. Puesto que la mayoría de los archivos no son accedidos concurrentemente, y las operaciones *read* predominan sobre las *write*, el mecanismo de *devolución de llamada* redunda en una reducción dramática del número de interacciones cliente-servidor.

El mecanismo de devolución de llamada utilizado en AFS-2 y versiones posteriores de AFS requiere que los servidores Vice mantengan algo del estado en nombre de sus clientes Venus a diferencia de AFS-1, NFS y nuestro modelo de servicio de archivos. El estado precisado dependiente del cliente precisado consiste en una lista de los procesos Venus para los que se han planteado promesas de devolución de llamada para cada archivo. Estas listas de devoluciones de llamada deben retenerse aun en el caso de fallos en el servidor, mantenerse en los discos del servidor y ser actualizadas utilizando operaciones atómicas.

La Figura 8.15 muestra las llamadas RPC proporcionadas por los servidores AFS para operaciones sobre archivos (esto es, la interfaz proporcionada por los servidores AFS para los procesos Venus).

◇ **Semántica de actualización.** El objetivo de este mecanismo de coherencia de caché es conseguir la mejor aproximación a la semántica de *una copia* de archivo que sea practicable sin una degradación considerable de las prestaciones. Una implementación estricta de la semántica de

Fetch(ida) → *atrib, datos*	Devuelve los atributos (estado) y, opcionalmente, los contenidos del archivo identificado por el *ida* y registra una promesa de devolución de llamada sobre él.
Store(ida, atrib, datos)	Actualiza los atributos y (opcionalmente) los contenidos de un archivo especificado.
Create() → *ida*	Crea un nuevo archivo y registra una promesa de devolución de llamada sobre él.
Remove(ida)	Elimina el archivo especificado.
SetLock(ida, modo)	Establece un bloqueo sobre el archivo o directorio especificado. El modo del bloqueo puede ser compartido o exclusivo. Los bloqueos no eliminados expiran a los 30 minutos.
ReleaseLock(ida)	Desbloquea el archivo o directorio especificado.
RemoveCallback(ida)	Informa al servidor de que un proceso Venus ha volcado un archivo desde su caché.
BreakCallback(ida)	Esta llamada se realiza desde un servidor Vice a un proceso Venus. Cancela la promesa de devolución de llamada del archivo en cuestión.

Nota: No se muestran las operaciones de directorio ni las administrativas (*Rename*, *Link*, *Removedir*, *GetTime*, *CheckToken*, y demás).

Figura 8.15. Componentes principales de la interfaz de servicio Vice.

una copia para las primitivas de acceso a archivos UNIX requeriría que los resultados de cada *write* a un archivo fueran distribuidos a todos los sitios en los que se mantenga el archivo en la caché antes de que puedan ocurrir otros accesos. Esto no es practicable en los sistemas de gran escala, en su lugar, el mecanismo de promesa de devolución de llamada mantiene una aproximación bien definida a la semántica de una copia.

Para AFS-1 la semántica de actualización puede ser establecida formalmente en muy pocos términos. Para un cliente *C* operando sobre un archivo *F* cuyo custodio es un servidor *S*, se mantienen las siguientes garantías de actualidad de las copias de *F*:

después de un *open* con éxito:	*latest*(*F*, *S*) -el último
después de un *open* fallado:	*failure*(*S*) -fallo
después de un *close* con éxito:	*updated*(*F*, *S*) -actualizado
después de un *close* fallado:	*failure*(*S*) -fallo

Donde *latest*(*F*, *S*) indica una garantía de que el valor actual de *F* en *C* es el mismo que el valor en S, *failure*(*S*) indica que la operación *open* o *close* no ha sido realizada en *S* (y el fallo puede ser detectado por *C*), y *updated*(*F*, *S*) indica que el valor de *F* en *C* ha sido propagado con éxito hacia *S*.

Para AFS-2, la garantía actual es ligeramente más débil, y la declaración formal correspondiente de la garantía es más compleja. Esto es porque un cliente puede abrir una copia antigua de un archivo después de que haya sido actualizado por otro cliente. Esto ocurre si se pierde un mensaje de *devolución de llamada*, por ejemplo como resultado de un fallo de la red. Pero hay un máximo de tiempo *T* para el que un cliente puede permanecer sin darse cuenta de una versión más nueva de un archivo. Aquí tenemos la garantía siguiente:

después de un *open* con éxito: *latest*(*F*, *S*, *0*)
o (lostCallback(S, T) e inCache(F) y
latest(F, S, T))

Donde *latest*(*F*, *S*, *T*) indica que la copia de *F* vista por el cliente está caducada no más allá de *T* segundos, *lostCallback*(*S*, *T*) indica que se ha perdido un mensaje de devolución de llamada desde *S* hasta *C* en algún instante durante los últimos *T* segundos, y *inCache*(*F*) que el archivo *F* estaba en la caché en *C* antes que fuera intentada la operación *open* intentada. La declaración formal anterior expresa el hecho que la copia en la caché de *F* en *C* después de una operación *open* es la

versión más reciente en el sistema o que un mensaje de devolución de llamada se ha perdido (debido a un fallo en la comunicación) y la versión que ya estaba en la caché ha sido utilizada, la versión en la caché no estará caducada más que en *T* segundos. (*T* es una constante del sistema que representa el intervalo al que deben ser renovadas las promesas de devolución de llamada. En la mayoría de las instalaciones el valor de *T* se pone en diez minutos aproximadamente). En línea con su objetivo, proporcionar a gran escala, un servicio de archivos distribuidos compatible UNIX, AFS no proporciona ningún otro mecanismo para el control de las actualizaciones concurrentes. El algoritmo de consistencia de la caché descrito anteriormente entra en acción sólo en las operaciones *open* y *close*. Una vez un archivo ha sido abierto, el cliente puede acceder y actualizar la copia local de cualquier forma que él elija sin el conocimiento de ninguno de los procesos en las otras estaciones de trabajo. Cuando se cierra un archivo, se devuelve una copia al servidor reemplazando la versión actual.

Si los clientes en diferentes estaciones de trabajo hacen *open*, *write* y *close* sobre el mismo archivo concurrentemente, todas las actualizaciones menos la resultante del último *close* se perderán silenciosamente (no se da ningún informe de error). Los clientes deben implementar el control de concurrencia independientemente si lo precisan. Por otro lado, cuando dos procesos cliente en la misma estación de trabajo abren un archivo, comparten la misma copia en la caché y las actualizaciones son realizadas en la forma normal UNIX, bloque a bloque.

Aunque las semánticas de actualización difieren, dependiendo de las ubicaciones de los procesos concurrentes que acceden a un archivo, y no son precisamente las mismas que las proporcionadas por el sistema de archivos estándar UNIX, son lo suficientemente próximas para que la gran mayoría de programas UNIX existentes trabajen correctamente.

8.4.3. OTROS ASPECTOS

◇ **Modificaciones al núcleo UNIX.** Hemos indicado que el servidor Vice es un proceso a nivel de usuario ejecutándose en el computador del servidor y la máquina del servidor está dedicada a la provisión de un servicio AFS. El núcleo UNIX en las máquinas AFS está alterado para que Vice pueda realizar operaciones en archivos en términos de apuntadores de archivo en lugar de los convencionales descriptores de archivo UNIX. Ésta es la única modificación al núcleo requerida por AFS y es necesaria si Vice no mantiene ningún estado del cliente (como descriptores de archivo).

◇ **Base de datos de ubicaciones.** Cada servidor contiene una copia de una base de datos de ubicaciones totalmente replicada proporcionando una correspondencia de los nombres de volúmenes para los servidores. Pueden ocurrir inexactitudes temporales en esta base de datos cuando se recoloca un volumen, pero son inocuas porque la información emitida se deja en el servidor desde el que se mueve el volumen.

◇ **Hilos.** Las implementaciones de Vice y Venus hacen uso de un paquete de hilos sin desalojo para permitir que las solicitudes sean procesadas concurrentemente, tanto en el cliente (donde varios procesos de usuario pueden tener solicitudes de acceso al archivo en progreso concurrentemente) como en el servidor. En el cliente, las tablas que describen los contenidos de la caché y la base de datos de volúmenes son mantenidas en memoria, que es compartida entre los hilos de Venus.

◇ **Réplicas de sólo lectura.** Los volúmenes conteniendo archivos que son leídos frecuentemente pero modificados raramente, como los directorios */bin* y */usr/bin* de comandos del sistema UNIX y el directorio */man* de páginas del manual, pueden estar replicados como volúmenes de sólo lectura en varios servidores. Cuando se hace esto, sólo hay una réplica de *lectura escritura* y todas las actualizaciones se dirigen a ella. La propagación de los cambios en la réplica de sólo lectura, se realiza después de la actualización por un procedimiento operativo explícito. Las entradas en la base de datos de ubicaciones para volúmenes que están replicados de esta forma son *una a muchas*, y el servidor para cada solicitud del cliente se selecciona en base a las cargas y accesibilidad de los servidores.

◇ **Transferencias al por mayor.** AFS transfiere archivos entre los clientes y los servidores en trozos de 64-kilobytes. El uso de paquetes de tan gran tamaño es una ayuda importante para las prestaciones, minimizando el efecto de latencia de la red. Por tanto el diseño de AFS permite optimizar el uso de la red.

◇ **Cacheado parcial de archivos.** La necesidad de transferir el contenido completo de los archivos a los clientes incluso cuando el requisito de la aplicación es para leer tan sólo una pequeña porción del archivo, es una fuente de ineficiencia obvia. La versión 3 de AFS elimina este requisito permitiendo que los datos del archivo sean transferido e introducidos en la caché en bloques de 64- kbytes mientras se mantiene todavía la semántica de consistencia y otras características del protocolo AFS.

◇ **Prestaciones.** El objetivo principal de AFS es la escalabilidad, por lo que sus prestaciones con un gran número de usuarios son de particular interés. Howard y otros [1988] dan detalles de una extensa comparativa de medidas de prestaciones, que fueron acometidas utilizando *AFS benchmark* desarrollado con este fin, y que ha sido utilizado posteriormente ampliamente para la evaluación de sistemas de archivos distribuidos. Como era de esperar, el cacheado de los archivos completos y el protocolo de devolución de llamadas derivaron una drástica reducción de las cargas en los servidores. Satyanarayanan [1989a] sostiene que se midió una carga del servidor del 40 % con 18 nodos cliente ejecutando una prueba estándar, frente a una carga del 100 % para NFS evaluando la misma prueba. Satyanarayanan atribuye muchas de las ventajas de prestaciones de AFS a la reducción en la carga del servidor derivada del uso de devoluciones de llamada para notificar a los clientes de las actualizaciones en los archivos, comparado con el mecanismo de timeout utilizado en NFS para la comprobación de la validez de las páginas en las cachés de los clientes.

◇ **Soporte de área amplia.** La versión 3 de AFS soporta múltiples celdas administrativas, cada una con sus propios servidores, clientes, administradores de sistema y usuarios. Cada celda es un entorno completamente autónomo, pero una federación de celdas puede cooperar presentando a los usuarios un espacio de nombres de archivo uniforme de una pieza. El sistema resultante fue ampliamente utilizado por Transarc Corporation y fue publicado un resumen detallado de los patrones de uso de las prestaciones resultantes [Spasojevic y Styanarayanan 1996]. El sistema se instaló en unos 1.000 servidores en 150 sitios. El resumen mostró proporciones de éxito de la caché en el rango de 96-98 % para accesos a una muestra de 32.000 volúmenes de archivos manteniendo 200 Gbytes de datos.

8.5. AVANCES RECIENTES

Desde la aparición de NFS y AFS se han realizado varios avances en el diseño de sistemas de archivos distribuidos. En esta sección, describimos los avances que mejoran las prestaciones, disponibilidad y escalabilidad de los sistemas de archivos distribuidos convencionales. Avances más radicales se describen en otras partes del libro, incluyendo el mantenimiento de la consistencia en sistemas de archivos de *lectura escritura* replicados, para soportar la operación de modo desconectado y alta disponibilidad en los sistemas Bayou y Coda (véase las Secciones 14.4.2 y 14.4.3), y una arquitectura altamente escalable para la entrega de secuencias de datos en tiempo real con calidad garantizada en el servidor de archivos de vídeo Tiger (véase la Sección 15.6).

◇ **Mejoras NFS.** Varios proyectos de investigación han tratado la cuestión de la semántica de actualización de una copia extendiendo el protocolo NFS para incluir las operaciones *open* y *close* y añadir un mecanismo de devolución de llamada para permitir al servidor notificar a los clientes la necesidad de invalidar entradas en la caché. Aquí describimos dos de tales esfuerzos, sus resultados parecen indicar que estas mejoras pueden ser acomodadas sin complejidad excesiva o costes de comunicación extras.

Agunos esfuerzos recientes de Sun y otros desarrolladores NFS se han dirigido a hacer más accesibles y útiles los servidores NFS en redes de área ancha. Mientras que el protocolo HTTP soportado por los servidores web ofrece un método efectivo y altamente escalable para hacer disponibles los archivos completos a los clientes a trevés de Internet, es menos útil para los programas de aplicación que requieren el acceso a porciones de grandes archivos o aquellos que actualizan porciones de archivos. El desarrollo WebNFS (descrito posteriormente) hace posible que los programas de aplicación lleguen a ser clientes de los servidores NFS en cualquier sitio de Internet (utilizando el protocolo NFS directamente en lugar de hacerlo indirectamente a través de un modulo del núcleo). Esto, junto con las bibliotecas apropiadas para Java y otros lenguajes de programación de red, deben ofrecer la posibilidad de implementar aplicaciones de Internet para compartir datos directamente, como juegos multiusuario o clientes de grandes bases de datos dinámicas.

Obtención de la semántica de actualización de *una copia*: La arquitectura de servidor sin estado de NFS proporciona grandes ventajas en robustez y facilidad de implementación para NFS, pero evita la consecución de una semántica de actualización de *una copia* requerida (los efectos de las escrituras concurrentes por diferentes clientes en el mismo archivo no está garantizado que sean las mismas que serían en un sistema único UNIX cuando múltiples procesos escriben en un archivo local). También impide la utilización de devoluciones de llamada para notificar a los clientes los cambios en los archivos, y esto deviene en solicitudes frecuentes de *getattr* de los clientes para comprobar la modificación de archivos.

Se han desarrollado dos sistemas de investigación que tratan con estas desventajas. Spritely NFS [Srinivasan y Mogul 1989, Mogul 1994] es un desarrollo de un sistema de archivos desarrollado para el sistema operativo distribuido Sprite en Berkeley [Nelson y otros 1988]. Spritely NFS es una implementación del protocolo NFS con la adición de llamadas *open* y *close*. Los módulos de los clientes deben enviar una operación *open* cuando un proceso local a nivel de usuario abre un archivo que está en el servidor. Los parámetros de una operación *open* en Sprite especifican un modo (lectura, escritura o ambas) e incluyen contadores del número de procesos locales que tienen el archivo abierto para lectura y para escritura. De modo semejante, cuando un proceso local cierra un archivo remoto se envía una operación *close* al servidor con los contadores actualizados de los lectores y los escritores. El servidor registra esos números en una *tabla de archivos abiertos* con la dirección IP y el número de puerto del cliente.

Cuando un servidor recibe un *open*, comprueba la *tabla de archivos abiertos* para otros clientes que tengan el mismo archivo abierto y envía mensajes de devolución de llamada a dichos clientes instruyéndoles para modificar su estrategia de caché. Si *open* especifica el modo de escritura, entonces fallará si cualquier otro cliente tiene el archivo abierto para escritura. Los otros clientes que tengan el archivo abierto para lectura serán instruidos para invalidar cualquier porción del archivo en la caché local.

Para operaciones *open* que especifican modo de lectura, el servidor envía un mensaje de devolución de llamada a cualquier cliente que esté escribiendo, indicándole que detenga la introducción en la caché (*caching*) (es decir, que utilice estrictamente el modo de escritura a través), e instruye a todo los clientes que están leyendo dejen de introducir en la caché el archivo (por lo que todas las llamadas de lectura locales producen una solicitud al servidor).

Estas medidas conducen a un servicio de archivos que mantienen la semántica de actualización de una copia UNIX con el coste de llevar algo del estado relacionado con el cliente al servidor.

También permiten ganar alguna eficiencia en el mantenimiento de las escrituras en la caché. Si el estado relacionado con el cliente se mantiene en memoria volátil en el servidor se vuelve vulnerable a los fallos del servidor. Spritely NFS implementa un protocolo de recuperación que interroga a una lista de clientes con archivos abiertos recientemente en el servidor para recuperar la *tabla de archivos abiertos* completa. La lista de clientes se almacena en el disco, se actualiza relativamente con poca frecuencia y es «pesimista», Puede incluir de forma segura más clientes que los que tenían archivos abiertos en el momento de una caída. Los clientes fallidos pueden producir también entradas en exceso en la *tabla de archivos abiertos*, pero se eliminarán cuando el cliente rearranque.

Cuando se evaluó Spritely NFS frente a NFS versión 2, mostró una modesta mejora de prestaciones. Esto se debió a la introducción de una mejorada caché de escritura. Los cambios en NFS versión 3 resultarían probablemente en una mejora al menos tan grande, pero los resultados del proyecto Spritely NFS indican claramente que es posible conseguir la semántica de actualización de *una copia* sin pérdida sustancial de prestaciones, aunque a costa de alguna complejidad de implementación extra en los módulos cliente y servidor y la necesidad de un mecanismo de recuperación, para restablecer el estado después de una caída del servidor.

NQNFS: El proyecto NQNFS (Not Quite NFS) [Macklem 1994] tenía objetivos similares a Spritely NFS, añadir consistencia de caché más precisa al protocolo NFS y mejorar las prestaciones a través de un mejor uso de la caché. Un servidor NQNFS mantiene un estado relativo al cliente relacionado con los archivos abiertos similar, pero utiliza concesiones (véase la Sección 5.2.6) para ayudar a la recuperación después de una caída del servidor. El servidor coloca un límite superior en el tiempo para que el cliente puede mantener una concesión en un archivo abierto. Si el cliente desea continuar más allá de ese tiempo, debe renovar la concesión. Las devoluciones de llamada se utilizaban de una manera similar a Spritely NFS para solicitar a los clientes que vuelquen sus cachés cuando ocurre una solicitud de escritura, pero si los clientes no contestan, el servidor simplemente espera hasta que sus concesiones expiren antes de responder a la nueva solicitud de escritura.

WebNFS: La llegada del Web y los applets Java condujeron al reconocimiento, por el equipo de desarrollo de NFS y otros, de que algunas aplicaciones Internet podrían beneficiarse del acceso directo a los servidores NFS sin muchas de las sobrecargas asociadas mediante la emulación de las operaciones de archivos UNIX incluida en los clientes estándar de NFS.

El objetivo de WebNFS (descrito en las RFC 2055 y 2056 [Callaghan 1996a, 1996b]) es permitir a los navegadores web, programas Java y otras aplicaciones interaccionar con un servidor NFS directamente para acceder a archivos que se encuentran «publicados» utilizando un apuntador de archivos público para acceder a los archivos relativos a un directorio raíz público. Este modo de utilización evita el servicio de montado y el servicio de enlace de puertos (*portmapper*, descrito en el Capítulo 5). Los clientes WebNFS interaccionan con un servidor NFS en un número de puerto bien conocido (2049). Para acceder a los archivos por nombre de ruta ellos lanzan solicitudes *lookup* utilizando un apuntador público de archivo. El apuntador público de archivo tiene un valor bien conocido que es interpretado especialmente por el sistema de archivos virtual en el servidor. Debido a la alta latencia de las redes de área amplia, se utiliza una variante *multicomponente* de la operación *lookup* para buscar un nombre de ruta compuesta en una única solicitud.

Por tanto WebNFS permite escribir clientes que accedan a porciones de archivos almacenados en servidores NFS en sitios remotos con mínimas sobrecargas de inicialización. Se proporciona control de acceso y autenticación, pero en muchos casos el cliente sólo requerirá acceso de lectura a archivos públicos y en ese caso la opción de autenticación puede ser deshabilitada. Para leer una porción de un único archivo localizado en un servidor NFS que soporta WebNFS se precisa el establecimiento de una conexón TCP y dos llamadas RPC, un *lookup* compuesto y una operación *read*. El tamaño del bloque leído no está limitado por el protocolo NFS.

Por ejemplo, un servicio meteorológico debería publicar un archivo en su servidor NFS que contiene una gran base de datos de datos meteorológicos actualizados frecuentemente con un URL como:

nfs://data.weather.gov/weatherdata/global.data

Un cliente interactivo WeatherMap, que muestra mapas del tiempo, podría estar construido en Java u otro lenguaje que soporte una biblioteca de procedimientos de WebNFS. El cliente lee sólo aquellas porciones del archivo /weatherdata/global.data que fueran necesarias para construir los mapas particulares solicitados por un usuario, mientras que una aplicación semejante que utilizara HTTP para acceder a los datos meteorológicos debiera haber transferido las base de datos completa al cliente o debiera solicitar el soporte de un programa del servidor de propósito especial para proporcionarle los datos que precisa.

NFS versión 4: Se encuentra en desarrollo una nueva versión del protocolo NFS en el momento de publicación de este libro. Los objetivos de NFS version 4 están descritos en la RFC 2624 [Shepler 1999] y en el libro de Ben Callaghan [Callaghan 1999]. Como WebNFS, pretende convertirlo y hacerlo práctico para utilizar NFS en redes de área extendida y aplicaciones Internet. Incluye las características de WebNFS, pero la introducción de un nuevo protocolo también ofrece una oportunidad para hacer mejoras más radicales. (WebNFS estaba restingido a cambios en el servidor que no impliquen la adición de nuevas operaciones al protocolo.)

El grupo de trabajo que desarrolla NFS versión 4 intenta explotar los resultados que han emergido de la investigación en el diseño de servidores de archivos en la década pasada, como el uso de devoluciones de llamada o contratos para mantener la consistencia. NFS soportará la recuperación al vuelo de los fallos en el servidor permitiendo a los sistemas de archivos sean trasladados a nuevos servidores transparentemente. La escalabilidad será mejorada utilizando servidores proxy de una forma análoga a su uso en la Web.

◇ **Mejoras en AFS.** Hemos mencionado que DCE/DFS, el sistema de archivos distribuidos incluido en el Entorno de Computación Distribuido (DCE) de la Open Software Foundation [www.opengroup.org], estaba basado en el Andrew File System. El diseño de DCE/DFS va más allá que AFS, particularmente en su aproximación a la consistencia de caché. En AFS, las devoluciones de llamada se generan sólo cuando el servidor recibe una operación *close* para un archivo que ha sido actualizado. DFS adoptó una estrategia semejante a la de Spritely NFS o NQNFS para generar devoluciones de llamada tan pronto como ha sido actualizado el archivo. Para actualizar un archivo, el cliente debe obtener un testigo *write* del servidor, especificando el rango de bytes del archivo en el que se permite al cliente actualizar el archivo. Cuando se solicita un testigo *write*, los clientes que mantienen copias del mismo archivo para lectura reciben devoluciones de llamada de revocación. Los testigos de otros tipos son utilizados para conseguir consistencia para los atributos y otros metadatos del archivo en la caché. Todos los testigos tienen un tiempo de vida asociado, y los clientes los deben renovar después de que dicho tiempo ha transcurrido.

◇ **Mejoras en la organización del almacenamiento.** Ha habido un progreso considerable en la organización de los archivos de datos almacenados en discos. El ímpetu por mucho de este trabajo surgió de las cargas incrementadas y de la mayor fiabilidad que los sistemas de archivos distribuidos necesitan soportar, y ha producido sistemas de archivos con prestaciones mejoradas sustancialmente. Los principales resultados de este trabajo son:

Redundant Arrays of Inexpensive Disks (*Cadenas Redundantes de Discos Baratos*, RAID): éste es un modo de almacenamiento [Patterson y otros 1988, Chen y otros 1994] en el que los bloques de datos están segmentados en trozos de tamaño fijo y almacenados en *ristras* entre varios discos, junto con códigos correctores de errores que permiten que los bloques de datos sean reconstruidos completamente y la operación continúe normalmente en el caso de fallos de disco. RAID produce también mejores prestaciones que un único disco, porque las listas que representan un bloque son leídas y escritas concurrentemente.

Log-structured file storage (*Almacen de archivo estructurado en histórico*, LFS): como Spritely NFS, esta técnica originada en el proyecto de sistema operativo Sprite en Berkeley [Rosenblum y Ousterhout 1992]. Los autores observaron que a medida que cantidades más grandes de memoria principal llegaban a estar disponibles para caché en los servidores de archivos, un incremento en el nivel de éxitos en la caché producía excelentes prestaciones en la lectura, pero las prestaciones en la escritura seguían siendo mediocres. Esto se debía a las altas latencias asociadas con las escrituras de bloques de datos individuales a disco y las actualizaciones asociadas con los datos de los metabloques (esto es, los bloques conocidos como *i-nodos* que mantienen los atributos de los archivos y un vector de apuntadores a los bloques en un archivo).

La solución LFS es acumular un conjunto de escrituras en memoria y entonces realizarlas a disco en segmentos de tamaño fijo, grandes, contiguos. Éstos se llaman *segmentos log* porque los bloques de datos y metadatos son almacenados estrictamente en el orden en el que fueron actualizados. Las copias frescas de los bloques de datos y los metadatos actualizados se escriben siempre, precisando el mantenimiento de un mapa dinámico (en memoria con una copia de respaldo persistente) apuntando a los bloques de i-nodos. Se precisa también la recolección de basura de los bloques rancios, con la compactación de los bloques *vivos* para dejar áreas contiguas de almacenamiento libres para el almacenamiento de los *segmentos log*. El último es un proceso bastante complejo, es realizado como una actividad en segundo plano de un componente llamado el *limpiador*. Se han desarrollado algunos algoritmos sofisticados para el limpiador basados en los resultados de simulaciones.

A pesar de estos costes extra, la ganancia en prestaciones globales es excelente. Rosenblum y Ousterhout midieron un rendimiento de escritura tan alto como el 70 % de ancho de banda disponible del disco, comparado con menos del 10 % para un sistema de archivos UNIX convencional. La estructura log simplifica también la recuperación después de caídas del servidor. El sistema de archivos Zebra [Hartman y Ousterhout 1995], desarrollado como una continuación del trabajo original LFS, combina escrituras estructuradas en históricos con una aproximación RAID distribuida, los *segmentos log* se subdividen en secciones con corrección de error en los datos y escrituras en los discos en nodos de red separados. Se indican prestaciones de cuatro a cinco veces las de NFS, para las escrituras de archivos grandes, con ganancias más pequeñas para archivos pequeños.

◇ **Nuevas aproximaciones de diseño.** La disponibilidad de redes conmutadas de altas prestaciones (como ATM o Ethernet de alta velocidad conmutado) han provocado varios esfuerzos para proporcionar sistemas de almacenamiento persistente que distribuyan los archivos de datos de una manera altamente escalable y tolerante a fallos entre muchos nodos en una intranet, separando las responsabilidades de lectura y escritura de los datos de las de gestión de los metadatos y solicitudes de servicio de los clientes. A continuación, esbozamos dos de tales desarrollos.

Estas aproximaciones escalan mejor que las de los servidores más centralizados que han sido descritas en secciones precedentes. Demandan generalmente un alto nivel de confianza entre los computadores que cooperan para proporcionar el servicio, porque incluyen un protocolo de bastante bajo nivel para la comunicación con los nodos que mantienen los datos (algo análogo a una API de *disco virtual*). Su alcance aquí está probablemente limitado a una única red local.

xFS: Un grupo de la Universidad de Berkeley en California, propuso una arquitectura de sistema de archivos en red sin servidor y desarrolló un prototipo de implementación, xFS [Anderson y otros 1996]. Su aproximación estuvo motivada por tres factores:

1. La oportunidad proporcionada por las redes de área local (LAN) conmutadas rápidamente para múltiples servidores de archivos en una red de área local para transferir volúmenes de datos a los clientes concurrentemente.
2. La expansión de las demandas para acceso a datos compartidos.
3. Las limitaciones fundamentales de los sistemas basados en servidores de archivos centrales.

En lo concerniente a (3), se refieren al hecho que la construcción de servidores NFS con altas prestaciones precisa un hardware relativamente costoso con múltiples CPU, discos y controladores de red, y que hay límites para el proceso de dividir el espacio de archivos, colocando los archivos compartidos en sistemas de archivos separados montados en diferentes servidores. También apuntan el hecho de que un servidor central representa un punto de fallo único.

xFS es *sin servidor* en el sentido que distribuye las responsabilidades de proceso del servidor de archivos entre un conjunto de computadores disponibles en una red local con la granularidad de archivos individuales. Las responsabilidades de almacenamiento están distribuidas independientemente de la gestión y otras responsabilidades de servicio: xFS implementa un software de sistema de almacenamiento RAID, reparto de los archivos de datos entre los discos de múltiples computadores (desde este punto de vista es un precursor del sistema de archivos de vídeo Tiger que se describirá en el Capítulo 15) junto con una técnica de *estructuración en históricos* de una manera similar a las del sistema de archivos Zebra.

La responsabilidad de la gestión de cada archivo puede reservarse a cualquiera de los computadores que soportan el servicio xFS. Esto se consigue mediante una estructura de metadatos llamada *gestor de correspondencias* (*map manager*), que se replica en todos los clientes y servidores. Los identificadores de archivo incluyen un campo que actua como un índice en el gestor de correspondencias, y cada entrada en la relación identifica el computador que es responsable actualmente de la gestión del archivo correspondiente. Otras varias estructuras de metadatos, semejantes a las encontradas en otros sistemas de almacenamiento *estructuración en históricos* y RAID, son utilizadas para la gestión del almacenamiento de archivos *estructurada en históricos* y el almacenamiento en disco en *ristras*.

Se construyó un prototipo preliminar de xFS y se evaluaron sus prestaciones. El prototipo estaba incompleto en el momento de realización de la evaluación, la implementación de recuperación de fallos estaba indefinida y el esquema de almacenamiento *estructurado en históricos* carecía de un componente limpiador para recuperar el espacio ocupado por los históricos de estado y compactar los archivos.

Las evaluaciones de prestaciones llevadas a cabo con este prototipo preliminar utilizaron 32 máquinas de un solo procesador y SPARCStations de Sun con dos procesadores conectados a una red de alta velocidad. Las evaluaciones compararon el servicio de archivos xFS corriendo en hasta 32 estaciones de trabajo con NFS y AFS, ejecutándose en una única SPARCStation de Sun con dos procesadores. Los anchos de Banda de lectura y escritura obtenidos con xFS con 32 servidores excedieron a los de NFS y AFS con servidores de dos procesadores en aproximadamente un factor de 10. La diferencia en prestaciones fue mucho menos marcada cuando xFS fue comparado con NFS y AFS utilizando las pruebas estándar de AFS. Pero en conjunto los resultados indican que el proceso altamente distribuido y la arquitectura de almacenamiento de xFS ofrecen una dirección prometedora para conseguir una mejor escalabilidad en los sistemas de archivos distribuidos.

Frangipani: Frangipani es un sistema de archivos distribuidos desarrollado y utilizado en el Centro de Investigación de Digital Systems (ahora Centro de Investigación de Compaq Systems) [Thekkath y otros 1997]. Sus objetivos son muy semejantes a los de xFS, y como xFS, los aproxima con un diseño que separa las responsabilidades de almacenamiento persistente de otras acciones del servicio de archivos. Pero el servicio de Frangipani está estructurado en dos niveles totalmente independientes. El nivel más bajo está proporcionado por el sistema de discos distribuido Petal [Lee y Thekkath 1996].

Petal proporciona una abstracción de discos virtuales distribuidos entre muchos discos ubicados en múltiples servidores en una red local conmutada. La abstracción de disco virtual tolera la mayoría de los fallos de hardware y software con la ayuda de réplicas de los datos almacenados y balancea automáticamente la carga en los servidores para reubicar datos. Los discos virtuales de Petal son accedidos mediante un manejador de disco UNIX utilizando operaciones estándar de entrada-salida de bloques, por lo que pueden ser utilizados para soportar la mayoría de los sistemas de archivos. Petal añade entre el 10 y 100 % a la latencia de los accesos a disco, pero la estrategia de caché produce rendimientos en las lecturas y escrituras al menos tan buenos como los manejadores de disco subyacentes.

Los módulos de servidor Frangipani se ejecutan en el núcleo del sistema operativo. Como en xFS, la responsabilidad de gestionar archivos y las tareas asociadas (incluyendo la provisión de un servicio de bloqueo de archivos para los clientes) se ha asignado dinámicamente a las máquinas, y todas las máquinas ven un nombre unificado de archivo con accesos coherentes (con la semántica similar a *una copia*) para los archivos compartidos actualizables. Los datos se almacenan en un formato *estructurado en históricos* y en *ristras* en el depósito de discos virtuales Petal. El uso de Petal libera a Frangipani de la necesidad de administrar el espacio físico de disco, implementando un sistema de archivos distribuidos mucho más sencillo, Frangipani puede emular las interfaces de servicio de varios servicios de archivos existentes, incluyendo NFS y DCE/DFS. Las prestaciones de Frangipani son al menos tan buenas como las de la implementación de Digital del sistema de archivos UNIX.

8.6. RESUMEN

Las características fundamentales de diseño para sistemas de archivos distribuidos son:

- La utilización efectiva de la memoria caché en el cliente para conseguir iguales prestaciones o mejores que las de los sistemas de archivos locales.
- El mantenimiento de la consistencia entre múltiples copias de archivos en las cachés de los clientes cuando son actualizadas.
- La recuperación después de un fallo en el servidor o en el cliente.
- El alto rendimiento en la lectura y escritura de archivos de todos los tamaños.
- La escalabilidad.

Los sistemas de archivos distribuidos son empleados intensamente en la computación de las organizaciones, y sus prestaciones han estado sujetas a muchos ajustes. NFS tiene un protocolo sin estado sencillo, que ha mantenido su posición inicial como la tecnología dominante de los sistemas de archivos distribuidos con la ayuda de mejoras relativamente menores al protocolo, implementaciones muy ajustadas y soporte de hardware de altas prestaciones.

AFS demostró la viabilidad de una arquitectura relativamente sencilla utilizando el estado del servidor para reducir el coste del mantenimiento de la coherencia de las cachés en los clientes. AFS sobrepasa a NFS en muchas situaciones. Los avances recientes han empleado reparto de los datos entre múltiples discos y escritura *estructurada en históricos* para mejorar más las prestaciones y la escalabilidad.

En el estado actual del arte, los sistemas de archivos distribuidos son altamente escalables, proporcionan buenas prestaciones tanto entre las redes de área local como las de áreas grandes, mantienen la semántica de actualización de *una copia* y toleran y se recuperan de los fallos. Los requisitos futuros incluyen soporte para usuarios móviles con operación desconectada y garantías de reintegración automática y calidad del servicio para satisfacer la necesidad de un almacenamiento persistente y la entrega de secuencias de multimedia y otros datos dependientes del tiempo. Los soluciones a estos requisitos se discutirán en los Capítulos 14 y 15.

EJERCICIOS

8.1. ¿Por qué no hay operación *Abre* y *Cierra* en nuestra interfaz de servicio de archivos planos o en el servicio de directorio? ¿Cuáles son las diferencias entre la operación *Busca* de nuestro servicio de directorio y la *open* de UNIX?

8.2. Esboce métodos mediante los cuales un módulo cliente podría emular la interfaz del servicio de archivos UNIX utilizando nuestro modelo de servicio de archivos.

8.3. Escriba un procedimiento *BuscaRuta*(*Ruta*, *Dir*) → *UFID* que implemente *Busca* para nombres de ruta del estilo UNIX basados en nuestro modelo de servicio de directorio.

8.4. ¿Por qué deben ser únicos los UFID entre todos los posibles sistemas de archivos? ¿Cómo se asegura la unicidad para los UFID?

8.5. ¿Qué extensión hace que NFS se desvíe de la semántica de actualización de una sola copia? Construir un escenario en el que dos procesos a nivel de usuario que compartan un archivo debieran funcionar correctamente en una única máquina UNIX pero se observen inconsistencias cuando funcionaran en máquinas diferentes.

8.6. NFS de Sun pretende soportar sistemas distribuidos heterogéneos proporcionando un servicio de archivos independiente del sistema operativo. ¿Cuáles son las decisiones fundamentales que debiera tener en cuenta el implementador de un servidor NFS en un sistema operativo distinto de UNIX? ¿Qué restricciones debiera obedecer el sistema de archivos subyacente para ser adecuado para la implementación de servidores NFS?

8.7. ¿Qué datos debe mantener el módulo cliente NFS en nombre de cada proceso a nivel de usuario?

8.8. Esboce implementaciones de un módulo cliente para las llamadas del sistema UNIX *open*( ) y *read*( ), utilizando las llamadas RPC de NFS de la Figura 8.9, (i) sin y (ii) con cliente caché.

8.9. Explique por qué la interfaz RPC de las implementaciones iniciales de NFS es potencialmente insegura. La laguna de seguridad ha sido solucionada en NFS 3 mediante el uso de encriptación. ¿Cómo se mantiene la clave de encriptación secreta? ¿Es adecuada la seguridad de la clave?

8.10. Después de un timeout de una llamada RPC para acceder a un archivo o a un sistema de archivos montado hard de NFS el módulo cliente no devuelve el control al proceso a nivel de usuario que ha originado la llamada. ¿Por qué?

8.11. ¿Cómo ayuda el automontador de NFS a mejorar las prestaciones y la escalabilidad de NFS?

8.12. ¿Cuántas llamadas de búsqueda son necesarias para resolver un nombre de ruta de 5 partes (por ejemplo /usr/users/jaime/codigo/xyz.c)para un archivo que está almacenada en el servidor de NFS? ¿Cuál es la razón de realizar la traducción paso a paso?

8.13. ¿Qué condición debe satisfacerse por la configuración de las tablas de montado en los computadores cliente para que se consiga la transparencia de acceso en un sistema de archivos basado en NFS?

8.14. ¿Cómo consigue el control AFS cuando una llamada *open* o *close* del sistema, referida a un archivo en el espacio de archivos compartidos, es emitida por un cliente?

8.15. Compare la semántica de actualización de UNIX cuando se accede a archivos locales con las de NFS y AFS. ¿Bajo qué circunstancias pueden los clientes llegar a despreocuparse de la diferencias?

8.16. ¿Cómo se ocupa AFS del riesgo de que los mensajes de devolución de llamada (*callback*) sean perdidos?

8.17. ¿Qué características del diseño de AFS lo hacen más escalable que NFS? ¿Cuáles son los límites de su escalabilidad, suponiendo que se pueden añadir servidores cuando sea preciso? ¿Qué desarrollos recientes ofrecen una gran escalabilidad?

9

SERVICIOS DE NOMBRES

9.1. Introducción
9.2. Servicios de nombres y el Sistema de Nombres de Dominio
9.3. Servicios de directorio y descubrimiento
9.4. Estudio del caso del Servicio de Nombres Global
9.5. Estudio del caso del Servicio de Directorio X.500
9.6. Resumen

Este capítulo presenta el servicio de nombres como un servicio independiente que es utilizado por los procesos cliente para obtener atributos como las direcciones de los recursos u objetos, dados los nombres de éstos. Las entidades nombradas pueden ser de muchos tipos y pueden gestionarse desde diferentes servicios. Por ejemplo, los servicios de nombres se utilizan a menudo para manejar direcciones y otros detalles de los usuarios, computadores, dominios de red, servicios y objetos remotos. Análogamente a los servicios de nombres, se describen servicios de directorio y descubrimiento, los cuales buscan servicios cuando se conoce alguno de sus atributos.

Se muestran a grandes rasgos, las cuestiones básicas de diseño para los sistemas de nombres, como la estructura y gestión del espacio de nombres reconocido por el servicio y las operaciones que el servicio de nombres soporta, y se ilustran dentro del contexto del Servicio de Nombres de Dominio.

También se examina cómo se implementan los servicios de nombres, tratando aspectos como la navegación a través de una colección de servidores cuando se pretende resolver un nombre, las cachés de datos con nombre y la replicación de los datos con nombre para incrementar las prestaciones y la disponibilidad.

Se incluyen dos estudios del caso: el Servicio de Nombres Global y el Servicio de Directorio X.500 y LDAP.

9.1. INTRODUCCIÓN

En un sistema distribuido los nombres se utilizan para hacer referencia a una amplia variedad de recursos como computadores, servicios, objetos remotos y archivos, así como a usuarios. El dar nombres es una cuestión que puede ser fácilmente olvidada pero que es fundamental en el diseño de sistemas distribuidos. Los nombres facilitan la comunicación y la compartición de recursos. Se necesita un nombre para solicitar que un sistema computacional actúe sobre un recurso específico elegido entre muchos posibles; por ejemplo, se necesita un nombre en forma de URL para acceder a una página web particular. Los procesos no pueden compartir recursos particulares gestionados por un sistema computacional a no ser que puedan nombrarlos de forma consistente. Los usuarios no pueden comunicarse entre ellos a través de un sistema distribuido a no ser que puedan nombrarse, por ejemplo mediante la dirección de correo electrónico.

Los nombres no constituyen el único medio útil de identificación: otro procedimiento son los atributos descriptivos. A veces los clientes no conocen el nombre de una entidad particular que están buscando, pero tienen información que la describe. O puede que el cliente necesite un servicio (en lugar de la entidad particular que lo implementa) y conozca algunas de las características que debe tener el servicio que solicita.

Se presentan los servicios de nombres, los cuales proporcionan a los clientes datos sobre los objetos nombrados en sistemas distribuidos; también se presentan los conceptos relacionados sobre servicios de directorio y de descubrimiento, los cuales proporcionan datos sobre objetos que cumplen una cierta descripción. Describimos diferentes posibles aproximaciones en el diseño e implementación de esos servicios, utilizando el Servicio de Nombres de Dominio (DNS), GNS y X500 como casos de estudio. Comenzaremos examinando los conceptos fundamentales de nombres y atributos.

9.1.1. NOMBRES, DIRECCIONES Y OTROS ATRIBUTOS

Cualquier proceso que necesite acceder a un recurso específico debe poseer su nombre o un identificador. Ejemplos de nombres fácilmente legibles son nombres de archivos como */etc/passwd*, URL's como *http://www.cdk3.net/* y nombres de dominio de Internet, como *dcs.qmw.ac.uk*. El término *identificador* se utiliza a veces para referirse a nombres que sólo se interpretan en los programas. Las referencias a objetos remotos y administradores de archivos NFS son ejemplos de identificadores. Los identificadores se eligen por la eficiencia con la que pueden ser buscados y almacenados por el software.

Needham [1993] distingue entre nombre *puro* y otros. Los nombres puros son simplemente patrones de bits sin interpretar. Los nombres *no puros* contienen información acerca del objeto al que nombran; en particular pueden contener información sobre la ubicación del objeto. Los nombres puros siempre deben buscarse antes de poder ser utilizados. En el otro extremo de un nombre puro se sitúa la *dirección* de un objeto: un valor que identifica la ubicación del objeto en lugar del objeto en sí mismo. Las direcciones son eficaces para acceder a los objetos, pero los objetos a veces cambian de localización, por lo que las direcciones no siempre resultan adecuadas como medio de identificación. Por ejemplo, las direcciones de correo electrónico de los usuarios normalmente cambian cuando los usuarios se mueven entre organizaciones o entre proveedores de servicio a Internet; no son suficientes por ellas mismas para referirse a un individuo específico a lo largo del tiempo.

Decimos que un nombre está *resuelto* cuando está traducido a datos relacionados con el recurso u objeto nombrado, a menudo con el objetivo de realizar una acción de invocación sobre él. La asociación entre un nombre y un objeto se llama *enlace*. En general los nombres se enlazan a los *atributos* de los objetos nombrados en lugar de enlazarlos a la implementación de los propios objetos.

Un atributo es el valor de una propiedad asociada con un objeto. Un atributo clave de una entidad, que es normalmente relevante en un sistema distribuido, es su dirección. Por ejemplo:

- DNS relaciona los nombres de dominio con los atributos de un cierto computador: su dirección IP, el tipo de entrada (por ejemplo una referencia a un servidor de correo o a otro tipo de nodo) y, por ejemplo, el período de tiempo durante el que la entrada del nodo será válida.
- Se puede utiliza el servicio de directorio X.500 para relacionar un nombre de persona sobre atributos que incluyen la dirección de correo electrónico y el número de teléfono.
- El Servicio de Nombres y el Servicio de Comercio de CORBA se presentará en el Capítulo 17. El servicio de nombres relaciona el nombre de un objeto remoto con su referencia de objeto remoto, mientras que el servicio comercial relaciona el nombre de un objeto remoto con su referencia de objeto remoto, junto con un número arbitrario de atributos que describen el objeto en términos comprensibles para el usuario humano.

Obsérvese que una *dirección* puede ser considerada a menudo simplemente como otro nombre que debe ser buscado o bien que puede contener dicho nombre. Cada dirección IP debe ser buscada para obtener una dirección de red, como una dirección Ethernet. De forma similar los navegadores web y clientes de correo electrónico utilizan DNS para interpretar los nombres de dominio de los URL y las direcciones de correo electrónico. La Figura 9.1 muestra la porción de nombres de dominio de un URL resuelto vía DNS para conseguir una dirección IP en primer lugar, y a continuación la dirección Ethernet del servidor web conseguida a través de ARP. La última parte del URL se resuelve en el sistema de archivos del servidor web para encontrar el archivo relevante.

◇ **Nombres y servicios.** Muchos de los nombres utilizados en un sistema distribuido son específicos de algún servicio particular. Un cliente utiliza dicho nombre al solicitar un servicio con objeto de realizar una operación sobre el objeto nombrado o sobre un recurso que éste maneja. Por ejemplo, al servicio de archivos se le proporciona un nombre de archivo cuando se solicita la eliminación de dicho archivo; al servicio de gestión de procesos se le proporciona un identificador de proceso cuando se solicita el envío de una señal a dicho proceso. Esos nombres sólo se utilizan en el contexto del servicio que gestiona los objetos nombrados, exceptuando cuando los clientes se comunican mediante objetos compartidos.

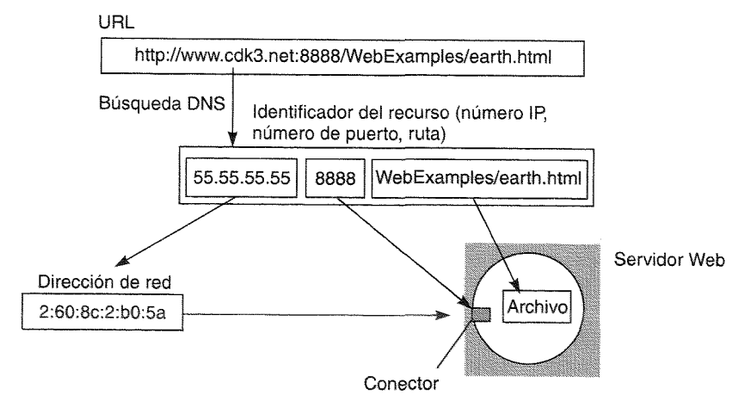


Figura 9.1. Dominios de nominación compuestos utilizados para acceder a un recurso desde un URL.

Los nombres también son necesarios para referirse a entidades del sistema distribuido que están fuera del ámbito de un único servicio. Los principales ejemplos de estas entidades son los usuarios (con nombres propios, nombres de login, identificadores de usuario y direcciones de correo electrónico), los computadores (con *nombres de host* como *bruno* o *bronwyn*) y los propios servicios (como el *servicio de archivos* o el *servicio de impresión*). En el middleware orientado a objetos, los nombres hacen referencia a objetos remotos que ofrecen servicios o proporcionan aplicaciones. Nótese que todos estos nombres deben ser legibles y tener significado para los humanos, ya que los usuarios y los administradores de sistema necesitan referirse a los principales componentes y elementos de configuración de los sistemas distribuidos; los programadores necesitan referirse a los servicios en los programas; y los usuarios necesitan comunicarse entre ellos en el sistema distribuido y determinar qué servicios están disponibles en sus diferentes partes. Partiendo de la conectividad proporcionada por Internet, estos requisitos de nominación tienen potencialmente un ámbito a escala mundial.

◇ **Identificadores de Recurso Unificados.** En la Sección 1.3 se presentó el URL como el principal medio de identificación de recursos web. De hecho los *Localizadores* Uniformes de Recursos (*Uniform Resource Locator*) son un tipo particular de *Identificadores* Uniformes de Recursos (*Uniform Resource Identificator*, URI) [URI].

Los URL presentan la importante propiedad de la escalabilidad, de modo que pueden hacer referencia a un conjunto de recursos web sin límite, a la vez que apuntan de forma eficiente a los recursos. El acceder a un recurso es fácil, partiendo de la información en su URL (un nombre DNS de computador y un camino en esa máquina). Aunque debido a que los URL son esencialmente *direcciones* de recursos web, sufren el inconveniente de que si el recurso se borra o se reubica, por ejemplo de un sitio web a otro, entonces habrá generalmente enlaces desconectados del recurso asociado al antiguo URL. Si un usuario trata de acceder a un enlace desconectado, el servidor web responderá que el recurso no ha sido encontrado o, posiblemente peor, proporcionará un recurso diferente que actualmente ocupa la misma situación.

El otro tipo principal de URI es el *Nombre Uniforme de Recurso* (*Uniform Resource Name*, URN). Los URN tratan de resolver el problema de los enlaces desconectados y proporcionan modos más completos de encontrar recursos en el Web. La idea consiste en tener un URN permanente para un recurso en el Web, incluso si el recurso se recoloca. El propietario de un recurso registrará su nombre, junto con el URL actual, frente a un servicio de búsqueda URN que proporcionará el URL a partir del URN. El propietario debe anotar el nuevo URL si se traslada el recurso. Un URN tiene la forma *urn*:*espacioNombres*: *nombreEspecífico-espacioNombres*. Por ejemplo, urn:ISBN:0-201-62433-8 podría referirse al libro que lleva el nombre 0-201-62433-8 en el esquema de nombres estándar ISBN. El nombre (inventado) *urn*:*doi*:10.555/*music-pop*1234 se refiere a la publicación llamada *music-pop*-1234 conocida en el esquema de nombres del editor como 10.555 en el esquema Identificador de Objetos Digital (*Digital Object Identifier*) [www.doi.org]. De forma similar, un URN como *urn*:*dcs.gormenghast.ac.uk*:*TR*2000-56 podría, en principio, utilizarse para obtener el último URL para el informe técnico conocido como TR2000-56, registrado con servicio de búsqueda de URN en *dcs.gormenghast.ac.uk*, en el Departamento de Ciencias de la Computación de la Universidad de Gormenghast.

Las *Características de Recurso Uniformes* (*Uniform Resource Characteristics*, URC, también conocidas como Citas de Recursos Uniformes) son un subconjunto de los URN's. Un URC es una descripción de un recurso web que consta de atributos del recurso, del tipo de *author* = *Leslie Lamport*, *keywords* = *tiempo*,... Los URC's sirven para describir recursos web y para realizar búsquedas de recursos web que cumplan con su especificación de atributos. Discutiremos en las dos secciones siguientes los servicios que se necesitan para buscar recursos partiendo de sus nombres y descripciones.

9.2. SERVICIOS DE NOMBRES Y EL SISTEMA DE NOMBRES DE DOMINIO

Un *servicio de nombres* almacena una colección de uno o más *contextos* de nominación, es decir, conjuntos de enlaces entre nombres textuales y atributos de objetos como usuarios, computadores, servicios y objetos remotos. La principal tarea que facilita un servicio de nombres es la resolución de un nombre, es decir, la búsqueda de atributos dado un cierto nombre. En la Sección 9.2.2 describimos la implementación de la resolución de nombres. También son necesarias otras operaciones, por ejemplo, crear nuevos enlaces, eliminar enlaces, listar los nombres enlazados y añadir y eliminar contextos.

La gestión de nombres estará muy separada de los otros servicios en virtud del carácter abierto de los sistemas distribuidos, el motivo de esto se encuentra en:

Unificación: a menudo es conveniente que los recursos que se gestionan desde diferentes servicios utilicen el mismo esquema de nomenclatura. Los URL son un buen ejemplo.

Integración: no es siempre posible predecir el ámbito de la compartición en un sistema distribuido. Puede ser necesario compartir y por lo tanto nombrar recursos que fueron creados en diferentes dominios administrativos. Sin un servicio de nombres común podría ocurrir que los dominios administrativos utilizasen nomenclaturas completamente diferentes.

◇ **Requisitos de un servicio de nombres general.** En sus inicios, los servicios de nombres eran muy simples ya que se diseñaron para cubrir únicamente las necesidades de vincular nombres con direcciones en un único dominio de gestión, correspondiente a una única LAN o WAN. La interconexión de redes y el incremento de escala de los sistemas distribuidos han hecho que el problema de la correspondencia de nombres sea mucho más complejo.

Grapevine [Birrell y otros 1982] fue uno de los primeros servicios de nombres extensibles y multidominio. se diseñó explícitamente para ser escalable por encima de, al menos, dos órdenes de magnitud el número de nombres y en la carga de solicitudes que podía manejarse.

El Servicio de Nombres Global (*Global Name Service*), desarrollado en el Centro de Investigación de Sistemas de Digital Equipment Corporation [Lampson 1986], desciende de Grapevine aunque con objetivos más ambiciosos:

Gestionar un número arbitrario de nombres y servir un número arbitrario de organizaciones administrativas: por ejemplo, el sistema debería ser capaz, entre otras cosas, de gestionar direcciones de correo electrónico para todos los usuarios de computadores del mundo.

Tiempo de vida elevado: durante el tiempo de vida del servicio ocurrirán muchos cambios en la organización del conjunto de nombres y en los componentes que lo implementan.

Alta disponibilidad: la mayor parte del resto del sistema dependerá del servicio de nombres; y no podrán funcionar si este servicio no está disponible.

Aislamiento de fallos: de forma que los fallos locales no provoquen el fallo del sistema completo.

Tolerancia a la ausencia de autenticación: en un sistema abierto a gran escala no hay ningún componente que esté autenticado por *todos* los clientes del sistema.

Dos ejemplos de servicios de nombres que se han concentrado en el objetivo de la escalabilidad para un gran número de objetos son el servicio de nombres Globe [van Steen y otros 1998] y el sistema Handle [www.handle-net]. El Sistema de Nombres de Dominio de Internet (DNS), presentado en el Capítulo 3, es menos ambicioso en el número de objetos que es capaz de manejar, pero continúa usándose de forma muy amplia. Este sistema nombra objetos (en la práctica computadores) en Internet. Para proporcionar un servicio satisfactorio, se basa en gran medida en la replicación y el almacenamiento en caché de los datos de nominación. El diseño de DNS y de otros

servicios de nombres parte de la suposición de que la consistencia caché no debe gestionarse estrictamente como en el caso de copias caché de archivos, ya que las actualizaciones son menos frecuentes y el uso de una copia sin actualizar, en una traducción de nombres, puede detectarse generalmente en el software del cliente.

En esta sección discutiremos las principales cuestiones de diseño de los servicios de nombres, proporcionando ejemplos de DNS. Posteriormente estudiaremos un caso detallado de DNS.

9.2.1. ESPACIOS DE NOMBRES

Un *espacio de nombres* es la colección de todos los nombres válidos reconocidos por un servicio particular. Que un nombre sea válido significa que el servicio intentará su búsqueda, incluso si ese nombre resulta no estar asociado a ningún objeto, es decir, está *desvinculado*. Los espacios de nombres requieren una definición sintáctica. Por ejemplo, el nombre *Dos* posiblemente no se corresponda con el nombre de un proceso UNIX, sin embargo el entero 2 seguramente sí lo sea. De forma similar, el nombre «...» no es aceptable como nombre DNS de un computador.

Los nombres pueden tener una estructura interna que representa su posición en un espacio de nombres jerárquico, como ocurre en el sistema de archivos de UNIX, o en una organización jerárquica, como es el caso de los nombres de dominio de Internet; o bien pueden elegirse de entre un conjunto plano de identificadores numéricos o simbólicos. La ventaja más importante de los espacios de nombres jerárquicos es que cada parte de un nombre se resuelve con relación a un contexto separado, y puede usarse el mismo nombre en diferentes contextos con diferentes significados. En el caso de los sistemas de archivos, cada directorio representa un contexto. Por ello */etc/passwd* es un nombre jerárquico con dos componentes. El primero, *etc*, se resuelve con relación al contexto «/» o raíz, y la segunda parte, *passwd*, lo está con relación al contexto */etc*. El nombre */antiguoetc/passwd* puede tener otro significado ya que su segundo componente se resuelve sobre un contexto diferente. De forma similar, el mismo nombre */etc/passwd* puede resolverse sobre diferentes archivos en los contextos de dos computadores diferentes.

Los espacios de nombres jerárquicos son potencialmente infinitos, de forma que permiten a un sistema crecer de forma indefinida. Los espacios de nombres planos son normalmente finitos; su tamaño se determina estableciendo una longitud máxima permitida para los nombres. Si en un espacio de nombres plano no se pone un límite en el tamaño de los nombres, entonces se convierte en potencialmente infinito. Otra ventaja potencial de un espacio de nombres jerárquico es que diferentes contextos pueden ser gestionados por diferentes personas.

La estructura de URL se presentó en el Capítulo 1. El espacio de nombres URL incluye *nombres relativos* del tipo de *../imagenes/figura1.jpg*. En este esquema URL, el nombre del host servidor y el directorio del servidor al que se refiere este nombre de camino son consideradas por el navegador como los mismos que los del documento en el que está insertado.

Los nombres DNS se denominan *nombres de dominio* y son cadenas similares a los nombres absolutos de archivos en UNIX. Algunos ejemplos son *bruno.dcs.qmw.ac.uk* (un computador), *dcs.qmw.ac.uk*, *com* y *purdue.edu* (los tres últimos son dominios).

El espacio de nombres DNS tiene una estructura jerárquica: un nombre de dominio está formado por una o más cadenas, separadas por el delimitador «.», llamadas *componentes de nombre* o *etiquetas*. No existe delimitador en el comienzo o en el final de un nombre de dominio, a pesar de que a veces nos referimos mediante «.» a la raíz del espacio de nombres DNS, con fines administrativos. Los componentes de nombre son cadenas imprimibles que no contienen «.». En general, un *prefijo* de un nombre es una sección inicial del nombre que contiene únicamente cero o más componentes completos.

Por ejemplo, en DNS, *dcs* y *dcs.qmw* son prefijos de *dcs.qmw.ac.uk*. Los nombres DNS no son sensibles a mayúsculas y minúsculas, de forma que *ac.uk* y *AC.UK* tienen el mismo significado.

Los servidores DNS no reconocen los nombres relativos: todos los nombres se refieren a la raíz global. Sin embargo, en la implementación práctica, el software de cliente mantiene una lista de nombres de dominio que se añaden automáticamente a cualquier nombre de componente único antes de la resolución. Por ejemplo, el nombre *bruno* sobre el dominio *dcs.qmw.ac.uk* se refiere probablemente a *bruno.dcs.qmw.ac.uk*; el software de cliente añadirá el dominio por defecto *dcs.qmw.ac.uk* e intentará resolver el nombre resultante. Si esto falla, entonces se añadirán otros nombres de dominio por defecto; finalmente, el nombre (absoluto) *bruno* se presentará a la raíz para su resolución. Sin embargo, los nombres con más de un componente se presentan normalmente de forma intacta al DNS como nombres absolutos.

◇ **Alias.** Desgraciadamente, los nombres con más de uno o dos componentes son desagradables de teclear y recordar. En general, un *alias* es similar a los enlaces simbólicos de tipo UNIX, permitiendo que un nombre práctico sea sustituto de otro más complicado. DNS permite alias, en los que se define un nombre de dominio para representar a otro. La razón para disponer de alias es la de proporcionar transparencia. Por ejemplo, los alias se usan habitualmente para especificar los nombres de máquinas que ejecutan un servicio web o un servicio FTP. El nombre *www.dcs.qmw.ac.uk* es un alias de *copper.dcs.qmw.ac.uk*. Esto tiene la ventaja de que los clientes pueden hacer referencia al servidor web mediante un nombre genérico que no se refiere a una máquina particular, y si el servidor web se mueve a otro computador, todo lo que hay que hacer es actualizar el alias en la base de datos DNS.

◇ **Dominios de nombres.** Un *dominio de nombres* es un espacio de nombres para el que existe una única autoridad administrativa global para asignar nombres. Esta autoridad ejerce, en conjunto, un control de los nombres que se pueden enlazar al dominio, aunque pueda delegar libremente esta tarea.

Los dominios en DNS son colecciones de nombres de dominio; sintácticamente el nombre de un dominio es el sufijo común de los nombres de dominio que hay dentro de él, aunque por otra parte no se pueda distinguir, por ejemplo, del nombre de un computador. Por ejemplo, *qmw.ac.uk* es un dominio que contiene *dcs.qmw.ac.uk*. Observe que el término *nombre de dominio* es potencialmente confuso ya que sólo algunos nombres de dominio identifican dominios. Un computador puede llegar a tener el mismo nombre que un dominio: por ejemplo, *yahoo.com* es el nombre de un servidor web en el dominio llamado *yahoo.com*.

El administrador de dominios puede estar involucrado en sub-dominios. El dominio *dcs.qmw.ac.uk* (Department of Computer Science, en Queen Mary and Westfield College, UK) puede contener cualquier nombre que desee el departamento. Sin embargo, el propio nombre de dominio *dcs.qmw.ac.uk* debe estar de acuerdo con las autoridades del colegio, quienes gestionan el dominio *qmw.ac.uk*. De forma análoga, *qmw.ac.uk* debe acordar con la autoridad registrada para *ac.uk* y así consecutivamente.

La responsabilidad de un dominio de nombres va normalmente en paralelo con la responsabilidad de gestionar y mantener actualizada la porción correspondiente de la base de datos almacenada en un servidor de nombres autorizado y utilizado por el servicio de nombres. Los datos de nominación pertenecientes a diferentes dominios de nombres se almacenan generalmente en distintos servidores de nombres, gestionados por sus autoridades correspondientes.

◇ **Combinación y personalización de los espacios de nombres.** DNS proporciona un espacio de nombres global y homogéneo en el que un cierto nombre se refiere a la misma entidad, independientemente de qué proceso, en qué computador busque dicho nombre. Por el contrario, algunos servicios de nombres permiten distintos espacios de nombres (algunas veces espacios de nombres heterogéneos) incluidos en ellos; también algunos servicios de nombres permiten personalizar el espacio de nombres para acomodarse a las necesidades de grupos individuales, usuarios e incluso procesos.

Fusionado: El montado de sistemas de archivos UNIX y NFS (véase la Sección 8.3) es un ejemplo en el que una parte de un espacio de nombres se inserta cómodamente en otro. Pero, considérese cómo mezclar el sistema de archivos UNIX *completo* de dos (o más) computadores llamados *rojo* y *azul*. Cada computador tiene su propia raíz, con nombres de archivos que se solapan. Por ejemplo */etc/passwd* se refiere a un archivo en *rojo* y a otro diferente en *azul*. La forma más obvia de mezclar los sistemas de archivos es reemplazar la raíz de cada computador por una *súper raíz* y montar cada sistema de archivos de cada computador en esta súper raíz, sea por ejemplo */rojo* y */azul*. Los usuarios y los programas se referirán entonces a */rojo/etc/passwd* y a */azul/etc/passwd*. Sin embargo, la nueva asignación de nombres podría provocar que fallaran aquellos programas que todavía manejaran el nombre antiguo */etc/passwd*. Una solución es dejar los contenidos de la raíz antigua en cada computador e insertar los sistemas de archivos */rojo* y */azul* en ambos computadores (asumiendo que esto no genera conflictos de nombres con los contenidos de la antigua raíz).

La moraleja es que siempre se puede mezclar espacios de nombres mediante la creación de un contexto raíz de mayor nivel, pero esto puede generar un problema de compatibilidad hacia atrás. Al arreglar el problema de la compatibilidad resultan espacios de nombres híbridos y el inconveniente de tener que traducir los nombres viejos entre los usuarios de los dos computadores.

Heterogeneidad: El espacio de nombres del Entorno de Computación Distribuida (*Distributed Computing Environment*, DCE) [OSF 1997] permite incluir en su interior espacios de nombres heterogéneos. Los nombres DCE pueden contener *uniones*, similares a los puntos de montado en NFS y en UNIX (véase la Sección 8.3), excepto por el hecho de que permiten montar espacios de nombres heterogéneos. Por ejemplo, sea el nombre completo DCE */.../dcs.qmw.ac.uk/principals/Jean.Dollimore*. La primera parte de este nombre */.../dcs.qmw.ac.uk* denota un contexto, denominado *celda*. El siguiente componente es una unión. Por ejemplo, la unión *principals* es un contexto que contiene los directores de seguridad donde puede buscarse el componente final *Jean.Dollimore*. De forma similar, en */.../dcs.qmw.ac.uk/files/pub/reports/TR2000-99*, la unión *files* es un contexto correspondiente a un directorio del sistema de archivos, en el que buscar el componente final *pub/reports/TR*2000-99. Los dos empalmes *principals* y *files* son las raíces de espacios de nombres heterogéneos, implementados mediante servicios de nombres heterogéneos.

Personalización: Vimos en el ejemplo anterior, sobre sistemas de archivos empotrados montados en NFS, que a veces los usuarios prefieren construir sus espacios de nombres de forma independiente en lugar de compartir un espacio de nombres único. El montado de sistemas de archivos permite a los usuarios importar archivos que almacenados en los servidores y compartirlos, mientras que el resto de nombres continúan refiriéndose a archivos locales no compartidos y pueden ser administrados de forma autónoma. Sin embargo, incluso los mismos archivos, accedidos desde dos computadores diferentes, pueden ser montados en diferentes puntos y por lo tanto tener diferentes nombres. Debido a que no comparten el espacio de nombres completo, los usuarios deberán traducir los nombres entre computadores.

Otro ejemplo que justifica la personalización es si el mismo nombre se refiere a diferentes archivos en diferentes computadores. Por ejemplo, el mismo nombre */bin/netscape* puede, en principio, estar asociado a un programa en formato binario x86 en un computador basado en Pentium, y a un binario SPARC en un computador Sun. Esta asociación de los mismos nombres sobre diferentes archivos hace posible escribir scripts asociados a esos nombres que se ejecutan correctamente en todas las configuraciones de máquina.

El servicio de nombres Spring [Radia y otros 1993] ofrece la posibilidad de construir espacios de nombres dinámicamente y compartir contextos de nominación individuales de forma selectiva. A diferencia de los anteriores ejemplos, incluso dos procesos diferentes en el mismo computador pueden tener contextos de nominación diferentes. Los contextos de nominación Spring son objetos de primera clase que pueden ser compartidos en un sistema distribuido. Por ejemplo, sea un usuario en el computador *rojo* que quiere ejecutar un programa en *azul* que publica caminos de archivos del tipo */etc/passwd*, pero esos nombres deben resolverse en el sistema de archivos de *rojo*, no en el de *azul*. Esto se logra en Spring pasando a *azul* una referencia al contexto de nominación local de *rojo* y utilizándola como el contexto de nominación del programa. Plan 9 [Pike y otros 1993] también permite a los procesos tener su propio espacio de nombres del sistema de archivos. Una característica novedosa de Plan 9 (que también puede implementarse en Spring) es que pueden ordenarse y fusionarse los directorios físicos dentro de un único directorio lógico. El efecto es que la búsqueda de un nombre en el directorio lógico único se realizará en la sucesión de los directorios físicos hasta que se encuentre, momento en el que se devolverán los atributos. Esto elimina la necesidad de proveer listas de rutas cuando se buscan programas o archivos de biblioteca.

9.2.2. RESOLUCIÓN DE NOMBRES

En general, la resolución es un proceso iterativo en el que se presenta de forma repetitiva un nombre sobre los contextos de nominación. Un contexto de nominación puede asociar un cierto nombre sobre un conjunto de atributos básicos (como los del usuario) bien de forma directa o bien asociando el nombre sobre otro contexto de nominación y sobre otro nombre derivado para presentarse en ese contexto. Para resolver un nombre, en primer lugar se presenta sobre un contexto de nominación inicial; el proceso de resolución se reitera mientras se generen otros contextos y otros nombres derivados. Esto se ilustró al principio de la Sección 9.2.1, con el ejemplo de */etc/passwd*, en el que *etc* se presenta al contexto /, y a continuación *passwd* se presenta sobre el contexto */etc*.

Otro ejemplo de la naturaleza iterativa de la resolución es la utilización de alias. Por ejemplo, siempre que se pregunte a un servidor DNS para resolver un alias como *www.dcs.qmw.ac.uk*, el servidor primeramente resuelve el alias en otro nombre de dominio (en este caso *copper.dcs.qmw.ac.uk*), el cual deberá ser resuelto a continuación para producir la dirección IP.

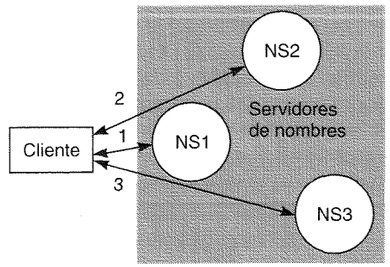
En general, la utilización de alias permite la presencia de ciclos en el espacio de nombres, en cuyo caso puede que la resolución no termine nunca. Existen dos soluciones, la primera consiste en abandonar el proceso de resolución si el número de resoluciones supera un cierto umbral; y, la segunda, permitir a los administradores vetar cualquier alias que pueda inducir ciclos.

◇ **Servidores de nombres y navegación.** Cualquier servicio de nombres, como DNS, que almacena una gran base de datos y es utilizado por una población grande no almacenará toda su información de nominación en un único computador servidor. Ese servidor sería un cuello de botella y un punto de fallo crítico. Cualquier servicio de nominación fuertemente utilizado deberá usar replicación para conseguir la suficiente disponibilidad. Veremos que DNS especifica que cada subconjunto de sus bases de datos sea replicada en, al menos, dos servidores independientes.

Hemos mencionado previamente que los datos que pertenecen a un dominio de nombres se almacenan normalmente en un servidor de nombres local gestionado por la autoridad responsable de ese dominio. A pesar de que, en algunos casos, un servidor de nombres puede almacenar datos relativos a más de un dominio, normalmente es correcto decir que los datos se reparten sobre los servidores de acuerdo a sus dominios. Veremos que en DNS, la mayor parte de las entradas se utilizan en los computadores locales. Aun así, también hay servidores de nombres para dominios superiores, como *yahoo.com* y *ac.uk*, y para el dominio raíz.

La partición de los datos implica que el servidor de nombres local no podrá responder a todas las solicitudes sin la ayuda de otros servidores de nombres. Por ejemplo, el servidor de nombres en el dominio *dcs.qmw.ac.uk* no es capaz de proporcionar la dirección IP de un computador en el dominio *cs.purdue.edu* a no ser que esa información esté en la caché (por supuesto eso no ocurre la primera vez que se realiza la consulta).

El proceso de localización de los datos con nombre entre más de un servidor de nombres, para resolver un nombre, se llama *navegación*. El software cliente de resolución de nombres realiza



Un cliente contacta de forma iterativa con los servidores de nombres NS1-NS3 para resolver un nombre

Figura 9.2. Navegación iterativa.

la navegación en nombre del cliente. Se comunica con los servidores de nombres que sea necesario para resolver un nombre. Puede tomar la forma de una biblioteca de código que se enlaza a los clientes, por ejemplo en las implementaciones BIND de DNS (véase la Sección 9.2.3) o en Grapevine [Birrell y otros 1982]. La alternativa, utilizada con X500, es la de proporcionar la resolución de nombres como un proceso separado que se comparte por todos los procesos clientes de ese computador.

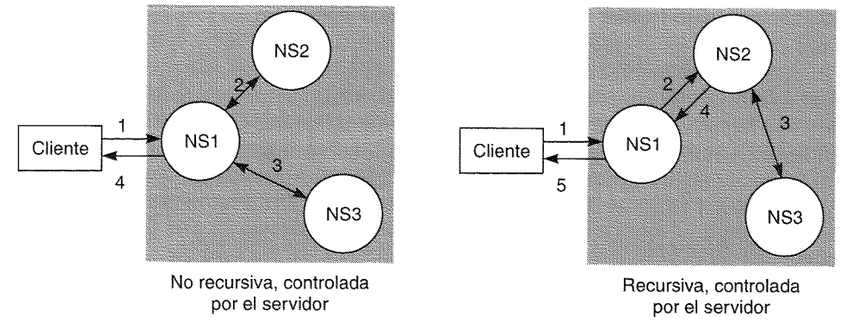
DNS soporta el modelo conocido como *navegación iterativa* (véase la Figura 9.2). Para resolver un nombre, un cliente lo presenta al servidor de nombres local, el cual intenta resolverlo. Si el servidor de nombres local tiene dicho nombre, devuelve el resultado inmediatamente. Si no lo tiene, se lo enviará a otro servidor capaz de ayudarle. La resolución avanza en el nuevo servidor, el cual puede seguir reenviándola hasta que el nombre sea localizado o bien se descubra que no existe.

Debido a que DNS está diseñado para gestionar las entradas de millones de dominios y es accedido por un enorme número de clientes, no sería factible hacer que todas las solicitudes comenzaran en un servidor raíz, incluso aunque éste estuviera ampliamente replicado. La base de datos de DNS se divide entre servidores de forma que se permite que muchas de las solicitudes puedan satisfacerse localmente y otras se completen sin necesidad de resolver cada parte del nombre de forma separada. El esquema para la resolución de nombres en DNS se describirá con mayor detalle en la Sección 9.2.3.

NFS también utiliza navegación iterativa en la resolución de un nombre de archivo, implementándola componente a componente (véase el Capítulo 8). Esto es así porque el servicio de archivos puede encontrase un enlace simbólico cuando al resolver un nombre. Un enlace simbólico debe interpretarse en el espacio de nombres del sistema de archivos del cliente porque puede apuntar a un archivo en un directorio almacenado en otro servidor. El computador cliente deberá determinar de qué servidor se trata, porque únicamente el cliente conoce sus puntos de montaje.

En la *navegación por multidifusión*, cada cliente envía de forma simultánea el nombre a resolver junto con el tipo de objeto requerido al grupo de servidores de nombres. Únicamente el servidor que almacena los atributos buscados responde a la solicitud. Sin embargo, si el nombre no se encuentra, entonces la solicitud no genera ninguna respuesta. Cheriton y Mann [1989] describen un esquema de navegación por multidifusión en el que cuando el nombre solicitado no haya podido resolverse, se incluye un servidor diferente en el grupo. La navegación por multidifusión también se utiliza en los servicios de descubrimiento (véase la Sección 9.3).

Otra alternativa al modelo de navegación iterativa es que un servidor de nombres coordine la resolución del nombre y devuelva los resultados al agente de usuario. Ma [1992] distingue entre *navegación recursiva* y *no recursiva controlada por el servidor* (véase la Figura 9.3). En la navegación no recursiva controlada por el servidor, el cliente puede elegir cualquier servidor de nombres. El servidor, como si fuera un cliente, se comunica mediante multidifusión o de forma



Un cliente de nombres NS1 se comunica con otros servidores de nombres representando a un cliente

Figura 9.3. Navegación no recursiva y recursiva controlada por el servidor.

iterativa con sus parejas en la forma descrita previamente. En la navegación recursiva controlada por el servidor, una vez más el cliente contacta con un solo servidor. Si el nombre no está en este servidor, el servidor contacta con un igual que almacene un prefijo (más largo) del nombre, el cual a su vez intenta su resolución. Este procedimiento continúa de forma recursiva hasta que se resuelve el nombre.

Si un servicio de nombres abarca diferentes dominios administrativos, entonces se puede prohibir que los clientes que se estén ejecutando en un cierto dominio administrativo accedan a los servidores de nombres pertenecientes a otro dominio. Además, incluso puede prohibirse que los servidores de nombres descubran de la disposición de datos de nominación sobre servidores de nombres en otros dominios administrativos. En ese caso, tanto la navegación controlada por cliente como la no recursiva controlada por servidor son inapropiadas, y deberá utilizarse la navegación recursiva controlada por servidor. Los servidores de nombres autorizados solicitan datos de servicio de nombres a los servidores designados por los diferentes administradores, los cuales devuelven los atributos sin revelar las zonas de la base de datos de nominación en las que estaban almacenados.

◇ **Caché.** En DNS y otros servicios de nombres, el software de resolución de nombres del cliente y los servidores mantienen una caché de resultados de resoluciones previas. Cuando un cliente solicita la búsqueda de un nombre, el software de resolución de nombres consulta su caché. Si encuentra un resultado reciente de una búsqueda previa de dicho nombre, la devuelve al cliente; en otro caso se la envía al servidor. A su vez, dicho servidor puede devolver datos provenientes de otros servidores pero almacenados en su caché.

El almacenamiento de datos en caché es clave en las prestaciones de un servicio de nombres y ayuda en el mantenimiento de la disponibilidad tanto del servicio como de otros servicios que seguirán funcionando incluso después de que el servidor de nombres falle. La mejora en los tiempos de respuesta que generan gracias al ahorro en las comunicaciones con los servidores de nombres es obvia. Las cachés pueden utilizarse para eliminar a los servidores de nombres (por ejemplo, el servidor raíz) del camino de navegación, permitiendo que se realice la resolución a pesar del posible fallo de algún servidor.

El uso de caché por parte de los sistemas de resolución de nombres en los clientes está ampliamente difundido en los servicios de nombres, y con mucho éxito ya que los datos de nominación raramente cambian. Por ejemplo, la información del tipo de dirección de computador o de servicio tiende a permanecer inalterada durante meses o años. Sin embargo, existe la posibilidad de que un servicio de nombres devuelva atributos caducados, por ejemplo una dirección obsoleta, durante el proceso de resolución.

9.2.3. EL SISTEMA DE NOMBRES DE DOMINIO

El Sistema de Nombres de Dominio es un diseño de servicio de nombres, y su base de datos principal se utiliza a lo largo de Internet. Fue ideado principalmente por Mockapetris [1987] con el objetivo de reemplazar el esquema de nombres original de Internet, en el que todos los nombres de host y las direcciones se mantenían en un único archivo maestro central y se descargaban vía FTP a todos los computadores que los necesitaban [Harrenstien y otros 1985]. Pronto se vio que este esquema inicial sufría de fuertes limitaciones:

- No era escalable hasta un número grande de computadores.
- Las organizaciones locales deseaban administrar sus propios sistemas de nombres.
- Se necesitaba un servicio de nombres general, no uno que sólo sirviera para buscar direcciones de computadores.

Los objetos nombrados por DNS son, en primer lugar, computadores (para las cuales en su mayor parte lo que se almacena como atributos son sus direcciones IP) y lo que en este capítulo hemos denominado como dominios de nombres son llamados simplemente *dominios* en DNS. Sin embargo, en principio se puede nombrar cualquier tipo de objeto, y su arquitectura tienen la capacidad suficiente para múltiples implementaciones. Las organizaciones y departamentos pueden manejar sus propios datos de nominación. En Internet hay millones de nombres enlazados mediante DNS, y se realizan búsquedas contra dicho sistema a lo largo de todo el mundo. Cualquier nombre puede ser resuelto por cualquier cliente. Esto se consigue mediante una partición jerárquica de la base de datos de nombres, mediante la replicación de los datos de nombres y mediante el uso de cachés.

◇ **Nombres de dominio.** DNS está diseñado para utilizarse de variadas formas, cada una de las cuales puede tener su propio espacio de nombres. Sin embargo, en la práctica sólo uno se utiliza ampliamente, el usado para los nombres de Internet. El espacio de nombres DNS de Internet se divide de acuerdo a criterios de organización y geográficos. Los nombres se escriben con el dominio de mayor importancia en la derecha. Los dominios de organización de primer nivel (también llamados *dominios genéricos*) utilizados actualmente en Internet son:

com	–	Organizaciones comerciales
edu	–	Universidades y otras instituciones de educación
gov	–	Agencias de gobierno de los EE.UU.
mil	–	Organizaciones militares de los EE.UU.
net	–	Principales centros de soporte de la red
org	–	Otras organizaciones no mencionadas anteriormente
int	–	Organizaciones internacionales

Además, cada país tiene sus propios dominios:

us	–	Estados Unidos
uk	–	Reino Unido
fr	–	Francia
...	–	...

Los diferentes países, excepto los EE.UU., utilizan su propio dominio para distinguir sus organizaciones. El Reino Unido, por ejemplo, tiene los dominios *co.uk* y *ac.uk*, que corresponden a *com* y *edu* respectivamente (*ac* significa *comunidad académica*). Nótese que, a pesar de que el sufijo *uk* es claramente geográfico, un dominio como *doit.co.uk* podría ubicarse en la oficina española de Doit Ltd, una compañía británica. En otras palabras, incluso los nombres de dominio con connotaciones claramente geográficas son completamente independientes de sus ubicaciones físicas.

◇ **Solicitudes DNS.** DNS en Internet se utiliza, inicialmente, como sistema de resolución de nombres de host y para la búsqueda de hosts de correo electrónico, como se explica a continuación:

Resolución de nombres de host: en general, las aplicaciones utilizan DNS para resolver nombres de host en direcciones IP. Por ejemplo, cuando se proporciona a un navegador [BROWSER] web un URL que contiene el nombre de dominio *www.dcs.qmw.ac.uk*, éste realiza una solicitud a DNS y obtiene la correspondiente dirección IP. Como se señaló en el Capítulo 4, los navegadores utilizan HTTP para comunicarse con los servidores web en una dirección IP con un número de puerto reservado. Los servicios FTP y SMTP trabajan de una forma similar; por ejemplo se puede proporcionar a un programa FTP el nombre de dominio *ftp.dcs.qmw.ac.uk* el cual realiza una solicitud DNS para obtener su dirección IP y utiliza a continuación TCP/IP para comunicarse con él mediante un número de puerto reservado. Los nombres *www*, *ftp* y *smtp* pueden ser alias para los nombres de dominio de los computadores en los que se ejecutan realmente dichos servicios. Para un ejemplo sin alias, suponga la utilización de un programa *telnet* para contactar con un host cuyo nombre de dominio es *jeans-pc.dcs.qmw.ac.uk*; *telnet* realiza una solicitud DNS para obtener la dirección IP correspondiente y a continuación utiliza el número de puerto por defecto.

Localización de hosts de correo: el software de correo electrónico utiliza DNS para resolver nombres de dominio en la dirección IP de los hosts de correo (computadores que aceptan correo para sus dominios). Por ejemplo, cuando se necesita resolver la dirección *tom@dcs.rnx.ac.uk*, la solicitud DNS se realiza con la dirección *dcs.rnx.ac.uk* y el tipo se designa como *mail*. Esta solicitud devuelve una lista de nombres de dominio de hosts que pueden aceptar correo para *dcs.rnx.ac.uk*, si existe (y, opcionalmente, las correspondientes direcciones IP). DNS puede devolver más de un nombre de dominio de forma que el software de correo puede intentar diferentes alternativas si el host de correo principal es, por alguna razón, inalcanzable. DNS devuelve un valor entero de preferencia para cada host de correo, indicando así el orden en el que deben utilizarse los hosts de correo.

Existen otros tipos de solicitudes que están implementadas en algunas instalaciones, pero se utilizan mucho menos que las explicadas anteriormente:

Resolución inversa: algunos programas necesitan obtener un nombre de dominio dada la dirección IP. Esto es justo lo contrario a una solicitud normal de nombre de host, aunque el servidor de nombres que recibe la solicitud responde únicamente si la dirección IP está en su propio dominio.

Información de host: DNS puede almacenar el tipo de arquitectura y de sistema operativo de los nombres de dominio de los hosts. Se ha sugerido que esta opción no debería implementarse, ya que proporciona información útil para aquellos que tratan de obtener acceso no autorizado a los computadores.

Servicios bien conocidos: a partir del nombre de dominio del computador puede obtenerse una lista de servicios en ejecución en ésta (por ejemplo, telnet, FTP) y el protocolo utilizado para acceder a ellos (es decir, UDP o TCP en Internet), siempre que el servidor de nombres dé soporte a esta información.

En principio, DNS puede usarse para almacenar atributos arbitrarios. Una solicitud se especifica mediante un nombre de dominio, una clase y un tipo. Para los nombres de dominio en Internet, la clase es IP. El tipo de las solicitudes especifica si se solicita una dirección IP, un host de correo, un servidor de nombres o algún otro tipo de información. El dominio especial in-addr.arpa, existe para mantener las direcciones IP en las búsquedas inversas. La clase atributo se utiliza para distinguir, por ejemplo, la base de datos de nombres de Internet de otras bases de datos experimentales de nominación en DNS. Se define un conjunto de tipos para una base de datos dada; los pertenecientes a la base de datos de Internet se proporcionan en la Figura 9.5.

◇ **Servidores de nombres DNS.** El problema de la escalabilidad se trata mediante una combinación de particionado de la base de datos de nombres y replicación y almacenamiento en caché de

aquellas partes de la base de datos cercanas a los puntos donde se necesitan. La base de datos DNS se distribuye a lo largo de una red lógica de servidores. Cada servidor mantiene parte de la base de datos de nombres (principalmente datos para el dominio local). La mayor parte de las solicitudes se refieren a computadores en el dominio local y son satisfechas mediante servidores dentro de dicho dominio. Sin embargo, cada servidor almacena los nombres de dominio y direcciones de otros servidores de nombres, de forma que puedan satisfacerse las solicitudes referentes a objetos de fuera del dominio.

Los datos de nominación de DNS se dividen en *zonas*. Una zona contiene los siguientes datos:

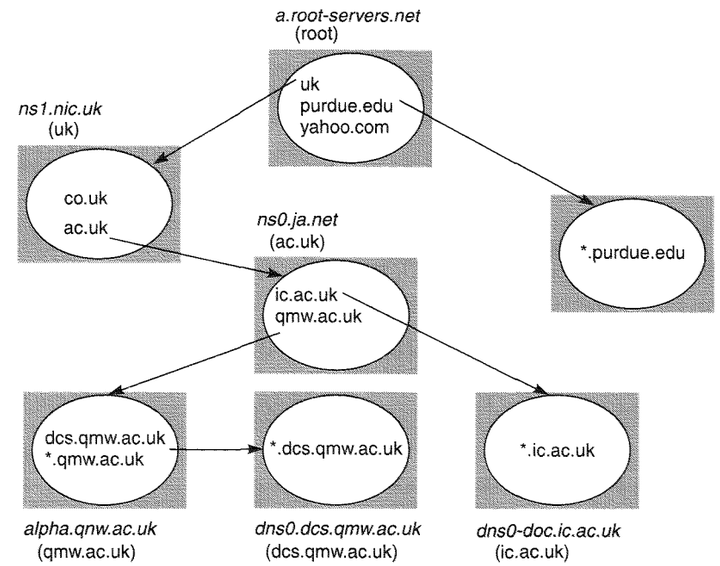
- Datos de atributos para nombres en el dominio, excepto los sub-dominios administrados por autoridades de menor nivel. Por ejemplo, una zona podría contener datos para el Colegio Queen Mary y Westfield (*qmw.ac.uk*) excepto los datos gestionados por los departamentos, por ejemplo el Department of Computer Science (*dcs.qmw.ac.uk*).
- Los nombres y direcciones de al menos dos servidores de nombres que proporcionan datos *autorizados* para la zona. Se trata de versiones de datos de zona para los que se puede confiar que estén razonablemente actualizados.
- Los nombres de servidores de nombres que mantienen datos autorizados para sub-dominios delegados; y datos de *enlace* que proporcionen las direcciones IP de esos servidores.
- Parámetros de gestión de zona, para, por ejemplo, gestionar la caché y la replicación de datos de zona.

Un servidor puede mantener datos autorizados para cero o más zonas. Para que los datos de nombres estén disponibles incluso cuando un único servidor falla, la arquitectura DNS especifica que cada zona debe replicarse de forma autorizada en al menos dos servidores.

Los administradores de sistema insertan los datos de una zona dentro de un archivo maestro, el cual es la fuente de datos autorizados para esa zona. Hay dos tipos de servidores a los que se permite proporcionar datos autorizados. Un *servidor maestro* o *primario* lee los datos de zona directamente desde un archivo maestro local. Los *servidores secundarios* descargan los datos de zona desde un servidor primario. Se comunican periódicamente con el servidor primario para comprobar si la versión que tienen almacenada coincide con la mantenida en el servidor primario. Si una copia secundaria ha caducado, el primario envía la última versión. La frecuencia de comprobación de los secundarios se configura por los administradores como un parámetro de zona y su valor es generalmente de una o dos veces al día.

Cualquier servidor puede almacenar en su caché los datos provenientes de otros servidores para evitar tener que contactar con ellos cuando la resolución de nombres solicite los mismos datos de nuevo; esto lo hace bajo la condición de que los clientes estén avisados de que dichos datos son no autorizados. Cada entrada en una zona tiene un valor de tiempo de vida. Cuando un servidor no autorizado almacena datos desde uno autorizado, anota el tiempo de vida. No se proporcionarán a los clientes datos por encima de dicho tiempo; cuando se le solicitan datos cuyo período de tiempo ha expirado, se vuelve a contactar con el servidor autorizado para comprobar los datos. Ésta es una característica útil que minimiza el tráfico de red mientras se mantiene la flexibilidad para los administradores de sistema. Cuando se espera que los atributos cambien de forma esporádica se les puede asignar un tiempo de vida grande. Si un administrador sabe que los atributos cambiarán en breve puede reducir el tiempo de vida en la medida correspondiente.

En la Figura 9.4 se muestra la organización de una base de datos DNS. Nótese que, en la práctica, los servidores raíz como *a.root-servers.net* mantienen entradas para diferentes niveles de dominio, al igual que entradas para los nombres de dominio de primer nivel. Esto permite reducir el número de pasos de navegación al resolver nombres de dominio. Los servidores de nombres raíz mantienen entradas autorizadas de los servidores de nombres en los dominios de mayor nivel. También son servidores autorizados para los dominios de mayor nivel genéricos como *com* y *edu*. Sin embargo, los servidores de nombres raíz no son servidores de nombres para los dominios aso-



Nota: los nombres de los servidores de nombres están en cursiva, y los dominios correspondientes están entre paréntesis. Las flechas indican entradas en el servidor de nombres

Figura 9.4. Servidores de nombres DNS.

ciados a países. Por ejemplo, el dominio *uk* actualmente tiene siete servidores de nombres, uno de los cuales se llama *ns1.nic.net*. Estos servidores de nombres conocen los servidores de nombres para los dominios de segundo nivel del Reino Unido como *ac.uk* y *co.uk*. Los servidores de nombres (actualmente cinco) del dominio *ac.uk* conocen los servidores de nombres de todos los dominios universitarios en el país, como *qmw.ac.uk* o *ic.ac.uk*. En algunos casos, un dominio universitario delega algunas de sus responsabilidades en un sub-dominio como *dcs.qmw.ac.uk*.

La información del dominio raíz se replica desde un servidor primario sobre una docena de servidores secundarios, como se ha descrito previamente. A pesar de esto, algunos servidores raíz sirven alrededor de 1.000 solicitudes por segundo, según Liu y Albitz [1998]. Todos los servidores DNS almacenan las direcciones de uno o más servidores de nombres raíz, los cuales no cambian a menudo. También almacenan a menudo la dirección de un servidor autorizado para el dominio padre. Una solicitud formada por un nombre de dominio de tres componentes del tipo de *www.berkeley.edu* puede ser satisfecha utilizando dos pasos de navegación en el peor caso: uno hacia un servidor raíz que almacena la correspondiente entrada de servidor de nombres y un segundo paso al servidor cuyo nombre es devuelto.

Remitiéndonos a la Figura 9.4, el nombre de dominio *jeans-pc.dcs.qmw.ac.uk* puede ser buscado desde *dcs.qmw.ac.uk* utilizando el servidor local *dns0.dcs.qmw.ac.uk*. Este servidor no almacena una entrada para el servidor web *www.ic.ac.uk* pero mantiene una entrada en caché para *ic.ac.uk* (que ha sido obtenida desde el servidor autorizado *ns0.ja.net*). El servidor *dns0-doc.ic.ac.uk* puede ser accedido para resolver el nombre completo.

◇ **Navegación y procesamiento de solicitudes.** Los clientes DNS son conocidos como *resolvedores* [RESOLVER]. Normalmente se implementan en forma de biblioteca software. Aceptan solicitudes, las formatean y las insertan en mensajes válidos del protocolo DNS y después se

Tipo de registro	*Significado*	*Contenidos principales*
A	Una dirección de computador	Número IP
NS	Un servidor de nombres autorizado	Nombre de dominio para un servidor
CNAME	El nombre canónico de un alias	Nombre de dominio para un alias
SOA	Marca el comienzo de datos de una zona	Parámetros que gobiernan una zona
WKS	Una descripción de servicio bien conocido	Lista de nombres de servicio y protocolos
PTR	Puntero de nombre de dominio (búsquedas inversas)	Nombre de dominio
HINFO	Información de host	Arquitectura de la máquina y sistema operativo
MX	Intercambio de correo	Lista de pares *<preferencia,host>*
TXT	Cadena de texto	Texto arbitrario

Figura 9.5. Registros de recursos DNS.

comunican con uno o más servidores de nombres para resolver las solicitudes. Se utiliza un protocolo simple de tipo petición-respuesta, normalmente a través de paquetes UDP sobre Internet (los servidores DNS utilizan un número de puerto conocido). El resolvedor detecta un retardo excesivo y reenvía la solicitud, si es necesario. Además puede configurarse para contactar con una lista de servidores de nombres en orden de preferencia para el caso de que uno o más estén fuera de servicio.

La arquitectura DNS permite la navegación recursiva al igual que la iterativa. El resolvedor especifica qué tipo de navegación es la requerida cuando contacta con un servidor de nombres. Sin embargo, los servidores de nombres no están obligados a implementar la navegación recursiva. Como se mencionó previamente, la navegación recursiva puede embotellar los hilos del servidor, retrasando otras peticiones.

Para ahorrar comunicaciones sobre la red, el protocolo DNS permite empaquetar varias solicitudes en el mismo mensaje de petición, y permite que los servidores de nombres empaqueten varias respuestas en sus mensajes de contestación.

◇ **Registros de recursos.** Los datos de zona se almacenan en los servidores de nombres en archivos que contienen datos de entre varios tipos fijos de registros de recurso. Para la base de datos de Internet estos tipos aparecen en la Figura 9.5. Cada registro se refiere a un nombre de dominio, que no es mostrado. Las entradas en la tabla se refieren a ítems ya mencionados, excepto las entradas TXT, incluidas para permitir el almacenamiento de otro tipo arbitrario de información asociada a los nombres de dominio.

Los datos de una zona comienzan con registros de tipo SOA, que contienen los parámetros de zona, especificando, por ejemplo, el número de versión y la frecuencia con la que los secundarios deben refrescar sus propias copias. A continuación aparece una lista de registros de tipo *NS* que especifican los servidores de nombres para el dominio y una lista de registros de tipo *MX* que indican las preferencias y los nombres de dominio de los hosts de correo. Por ejemplo, la base de datos para el dominio *dcs.qmw.ac.uk* está formada por los siguientes datos, donde el tiempo de vida *1D* significa 1 día:

Nombre de dominio	*Tiempo de vida*	*Clase*	*Tipo*	*valor*
	1D	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	NS	cancer.ucs.ed.ac.uk
	1D	IN	MX	1 mail1.qmv.ac.uk
	1D	IN	MX	2 mail2.qmv.ac.uk

Los registros adicionales de tipo *A* en la base de datos proporcionan las direcciones IP de los dos servidores de nombres *dns0* y *dns1*. Las direcciones IP de los hosts de correo y del tercer servidor de nombres se proporcionan en las bases de datos correspondientes a sus dominios.

La mayor parte del resto de registros de una zona inferior como *dcs.qmw.ac.uk* serán de tipo *A* y enlazarán el nombre de dominio de un computador sobre su dirección IP. Pueden contener algunos alias para servicios conocidos, por ejemplo:

Nombre de dominio	*Tiempo de vida*	*Clase*	*Tipo*	*valor*
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

Si el dominio tiene sub-dominios, habrá registros adicionales de tipo *NS* especificando sus servidores de nombres, los cuales tendrán también entradas *A* individuales. Por ejemplo, la base de datos de *qmw.ac.uk* contiene los siguientes registros para los servidores de nombres en su sub-dominio *dcs.qmw.ac.uk*:

Nombre de dominio	*Tiempo de vida*	*Clase*	*Tipo*	*valor*
dcs	1D	IN	NS	dns0.dcs
dns0.dcs	1D	IN	A	138.37.88.249
dcs	1D	IN	NS	dns1.dcs
dns1.dcs	1D	IN	A	138.37.94.248
dcs	1D	IN	NS	cancer.ucs.ed.ac.uk

Compartición de carga en los servidores de nombres: En algunos puntos, servicios fuertemente utilizados como Web y FTP pueden estar soportados por un grupo de computadoress sobre la misma red. En este caso se utiliza el mismo nombre de dominio para cada miembro del grupo. Cuando un nombre de dominio se comparte por varios computadores, hay un registro para cada computador en el grupo, el cual proporciona sus direcciones IP. El servidor de nombres responde a las solicitudes que se refieren a múltiples registros con el mismo nombre devolviendo la dirección IP mediante una planificación *round robin*. Los clientes sucesivos tienen acceso a servidores diferentes de forma que los servidores puedan compartir la carga de trabajo. La caché puede, potencialmente, malograr este esquema, ya que un servidor de nombres no autorizado o un cliente pueden tener la dirección del servidor en su caché y continuar usándola. Para contrarrestar este efecto, se proporciona a los registros un tiempo de vida muy corto.

◇ **La implementación BIND de DNS.** El Dominio de Nombres Internet de Berkeley (*Berkeley Internet Name Domain*, BIND) es una implementación de DNS para computadores que ejecutan UNIX. Los programas cliente enlazan el resolvedor en las bibliotecas software. Los servidores de nombres DNS ejecutan el demonio *named*.

BIND soporta tres categorías de servicio de nombres: servidores primarios, servidores secundarios y servidores *sólo caché*; el programa *named* implementa sólo uno de esos tipos en función de los contenidos de un archivo de configuración. Las dos primeras categorías son como las descritas previamente. Los servidores *sólo caché* leen desde un archivo de configuración suficientes nombres y direcciones de servidores autorizados para resolver cualquier nombre. Por lo tanto únicamente almacenan esos datos y los que aprenden al resolver nombres de clientes.

Una organización típica está formada por un servidor primario, con uno o más servidores secundarios que proporcionan servicio de nombres sobre diferentes redes de área local en el mismo sitio. Adicionalmente, cada computador individual suele ejecutar sus propios servicios *sólo caché*, para reducir el tráfico de red y acelerar aún más los tiempos de respuesta.

◇ **Discusión sobre DNS.** La implementación Internet de DNS consigue tiempos medios de respuesta en búsquedas relativamente cortos, teniendo en cuenta la cantidad de datos de nombres y la escala de las redes implicadas. Hemos visto que consigue estos resultados mediante una combinación de particionado, replicación y almacenamiento en caché de los datos de nombres. Los objetos nombrados son, principalmente, computadores, servidores de nombres y hosts de correo. La relación entre un nombre de computador (host) y una dirección IP raramente cambia, al igual que los identificadores de los servidores de nombres y de los hosts de correo, de forma que el almacenamiento en caché y la replicación se realizan en un entorno relativamente favorable.

En un sistema DNS puede ocurrir que los datos de nominación se vuelvan inconsistentes. Es decir, si se cambian los datos de nominación, puede que otros servidores proporcionen a los clientes datos obsoletos durante períodos que pueden ser del orden de días. No se aplica ninguna de las técnicas de replicación exploradas en el Capítulo 14 es aplicada. Sin embargo, la inconsistencia no tiene consecuencias hasta que el cliente no intenta acceder a datos obsoletos. DNS no resuelve la detección de direcciones obsoletas.

Además de computadores, DNS también implementa la nominación de un tipo particular de servicio: el correo, siempre en base a dominios. DNS asume que existe sólo un servicio de correo por cada dominio direccionado, de forma que los usuarios no deben incluir el nombre de dicho servicio explícitamente en los nombres. Las aplicaciones de correo electrónico seleccionan de forma transparente este servicio utilizando el tipo de solicitud apropiado al contactar con los servidores DNS.

En resumen, DNS almacena una variedad limitada de datos de nominación, pero hasta el momento ha sido suficiente ya que aplicaciones como el correo electrónico imponen sus propios esquemas de nominación sobre los nombres de dominio. Puede argumentarse que la base de datos de DNS representa el mínimo común denominador de lo que debería ser considerado útil por la mayor parte de la comunidad de usuarios de Internet. DNS no fue diseñado para ser el único servicio de nombres en Internet; coexiste con nombres locales y servicios de directorio que almacenan datos más oportunos para las necesidades locales (como el Servicio de Información de Red de Sun —*Network Information System*—, que almacena contraseñas codificadas, por ejemplo, o bien el Servicio de Directorio Activo de Microsoft (*Active Directory Service*) [www.microsoft.com I], el cual almacena información detallada sobre todos los recursos en un cierto dominio).

Lo que continúa siendo un problema potencial del diseño de DNS es su rigidez en relación con los cambios en la estructura del espacio de nombres, y la falta de habilidad en la personalización del espacio de nombres para cumplir las necesidades locales. Estos aspectos del diseño de nominación se retomarán para el estudio del caso del servicio de nombres global en la Sección 9.4. Antes de eso, estudiaremos los servicios de directorio y descubrimiento.

9.3. SERVICIOS DE DIRECTORIO Y DESCUBRIMIENTO

Hemos descrito cómo los servicios de nombres almacenan colecciones de parejas <*nombre*, *atributo*>, y cómo los atributos pueden ser localizados a partir del nombre. Es natural considerar la dualidad de esta estructura, en la que los atributos son usados como valores a ser buscados. En estos servicios, los nombres de texto pueden ser considerados simplemente como otro atributo. Algunas veces los usuarios quieren encontrar una persona o recurso en particular, pero no conocen su nombre, únicamente alguno de sus atributos. Por ejemplo, un usuario puede preguntar: *¿cuál es el nombre del usuario con número de teléfono* 020-555 9980*?* Algunas veces los usuarios necesitan un servicio, pero no conocen qué entidad del sistema lo proporciona, a pesar de que el servicio es accesible de forma cómoda. Por ejemplo, un usuario puede preguntar *¿qué computadores en este edificio son Macintosh ejecutando el sistema operativo MacOS* 8.6*?* o bien *¿dónde puedo imprimir una imagen en color de alta resolución?*

Un servicio que almacene colecciones de enlaces entre nombres y atributos y que realice búsquedas de entradas que emparejan especificaciones basadas en atributos se llama *servicio de directorio*. Por ejemplo el Servicio de Directorio Activo de Microsoft, X.500 y su primo LDAP (descrito en la Sección 9.5), Univers [Bowman y otros 1990] y Profile [Peterson 1988]. Los servicios de directorio a veces son llamados *servicios de páginas amarillas*, y los servicios de nombres convencionales a veces son llamados *servicios de páginas blancas*, utilizando una analogía obvia con los diferentes tipos de directorios telefónicos. A veces los servicios de directorio son conocidos como *servicios de nombres basados en atributos*.

Un servicio de directorio devuelve los atributos de cualquier objeto encontrado que coincida con los atributos especificados. De forma que, por ejemplo, la solicitud *NumeroTelefono* = = 020-555 9980 debe devolver *Nombre* = *Juan Herrero*, *NumeroTelefono* = 020-555 9980, *DireccionCorreo* = *juan@dcs.gormenghast.ac.uk*, ... El cliente puede especificar que únicamente un subconjunto de los atributos es de su interés; por ejemplo, simplemente las direcciones de correo electrónico de los objetos encontrados. X.500 y algunos otros servicios de directorio también permiten buscar objetos mediante nombres de texto de una jerarquía convencional.

Los atributos son claramente más potentes que los nombres como designadores de objetos: es posible escribir programas pueden que seleccionen objetos de acuerdo con especificaciones precisas de los atributos mientras que los nombres puede que no sean conocidos. Otra ventaja de los atributos es que no desvelan la estructura de las organizaciones al resto del mundo como lo harían los nombres divididos usando criterios asociados a una cierta organización. Sin embargo, la relativa simplicidad asociada a la utilización de nombres de texto hace que, para muchas aplicaciones, no puedan ser reemplazados por la nominación basada en atributos.

◇ **Servicios de descubrimiento.** Un servicio de descubrimiento es un servicio de directorio que registra los servicios proporcionados en un entorno de red espontáneo. Como se explicó en la Sección 2.2.3, en las redes espontáneas los dispositivos tienen tendencia a conectarse sin previo aviso y sin preparación administrativa. El requisito consiste en integrar, sin intervención de usuarios, un conjunto de clientes y servicios que cambian dinámicamente. Para cumplir esas necesidades, un servicio de descubrimiento proporciona una interfaz que automáticamente registra y desregistra servicios, así como una interfaz para que los clientes busquen los servicios que necesitan de entre los que están actualmente disponibles.

Por ejemplo, considérese un visitante ocasional a una organización o un hotel (véanse las Secciones 1.2.3 y 2.2.3), que necesita imprimir un documento. No se puede esperar que el usuario disponga en la configuración de su computador portátil de los nombres de las impresoras, o que adivine dichos nombres (*\\myrtle\titus* y *\\lionel\frederick*). En lugar de forzar a que el usuario configure su máquina, es preferible que el portátil pueda hacer uso de una interfaz de *búsqueda* de un servicio de descubrimiento para encontrar el conjunto de impresoras de red disponible que cumplen las necesidades del usuario. Puede seleccionarse una impresora concreta mediante la interacción con el usuario o consultando un registro de preferencias del usuario. Esta posibilidad resultará familiar a los usuarios del sistema operativo Macintosh.

Los atributos requeridos para el servicio de impresión pueden, por ejemplo, especificar si debe ser *láser* o de *chorro de tinta*; si se debe proporcionar o no impresión en color; y la situación física respecto al usuario (por ejemplo, su número de habitación).

De forma similar, los servicios notifican su existencia al servicio de descubrimiento a través de una interfaz de *registro*. Por ejemplo, una impresora (o el servicio que la gestiona) puede registrar sus atributos con el servicio de descubrimiento de la siguiente forma:

clasederecurso = impresora, tipo = laser, color = si, resolucion = 600dpi, localizacion = habitacion101, url = http://www.hotelDuLac.com/services/printer57.

El URL especifica la localización en la red de la impresora.

Nótese que la búsqueda de un servicio mediante un servicio de descubrimiento no supone necesariamente la intervención de un usuario. Por ejemplo, un frigorífico podría descubrir un servicio de gestión de errores si se produce un fallo en el test de auto-diagnóstico, con el fin de notificar a su propietario el fallo utilizando el PC del hogar.

En los servicios de descubrimiento, el contexto para el descubrimiento se denomina frecuentemente *ámbito*. Algunos servicios, como el protocolo simplificado de descubrimiento de servicios, están pensados para utilizarse en ciertos ámbitos en función de la accesibilidad de una red local, como el ámbito de todos los recursos conectados a una red sin cables en el hogar. Esto es conveniente en diversidad ocasiones. Por ejemplo, cuando un usuario solicita una impresora normalmente quiere una que esté físicamente cercana, lo cual suele implicar que esté situada en una red local. Por el contrario, los servicios de directorio como X.500 se estructuran jerárquicamente para reflejar ámbitos geográficos y de organización. Es así que, X.500 puede utilizarse tanto para buscar un persona en la organización local como en un país.

Recientes desarrollos en servicios de descubrimiento incluyen el servicio de descubrimiento Jini (véase a continuación), el protocolo de localización de servicios [Guttman 1999], el sistema de nominación intencional [Adjie-Winoto y otros 1999], el protocolo simplificado de descubrimiento de servicios, el cual está en el corazón de la iniciativa *plug and play* universal [www.upnp.com] y el servicio de descubrimiento de servicios seguros [Czerwinski y otros 1999].

◇ **Jini.** Jini [Waldo 1999, Arnold y otros 1999] es un sistema diseñado para ser utilizado en entornos de red espontáneos. Está completamente basado en Java, asumiendo que se ejecuta una JVM (*Java Virtual Machine*) en todos los computadores, permitiendo la comunicación mediante RMI (véase el Capítulo 5) y la descarga de código según se necesite. Jini proporciona recursos para descubrimiento de servicios, para transacciones, para espacios de datos compartidos llamados *javaSpaces* (similares a los espacios de tuplas [Carriero y Gelernter 1989]) y para eventos. Describiremos únicamente el sistema de descubrimiento.

Los componentes relacionados con el descubrimiento en un sistema Jini son servicios de *búsqueda*, servicios Jini y clientes Jini (véase la Figura 9.6). El servicio de *búsqueda* implementa lo que hemos denominado un servicio de descubrimiento, a pesar de que Jini utiliza el término *descubrimiento* sólo para el descubrimiento del propio servicio de búsqueda. El servicio de búsqueda permite a los servicios Jini registrar los servicios que ofrece, y a los clientes Jini solicitar servicios que cumplan sus requisitos. Un servicio Jini, como un servicio de impresión, puede registrarse frente a uno o más servicios de búsqueda. Un servicio Jini proporciona, y el servicio de búsqueda almacena, un objeto que realiza el servicio, así como los atributos del servicio. Los clientes Jini solicitan búsquedas de servicios para encontrar los servicios Jini que cumplan con sus requisitos; si la búsqueda tiene éxito, se descarga, desde el servicio de búsqueda, un objeto que proporciona el servicio. El emparejamiento de servicios ofrece a los clientes la posibilidad de basar las solicitudes en atributos o en tipos Java, por ejemplo permitiendo a un cliente solicitar una impresora color para la que disponga de la correspondiente interfaz Java.

Al igual que cualquier otro sistema que soporte entornos de red espontáneos, Jini se enfrenta con el reto de la conectividad en frío. Si un cliente o servicio Jini entra en una red, debe utilizar el servicio de búsqueda. Pero, ¿cómo puede localizar el servicio de búsqueda? En algunos casos, los clientes y servicios conocerán *a priori* la dirección de un servicio de búsqueda particular (podría haber sido proporcionado por un usuario). Pero el caso más interesante es cuando el cliente y los servicios son unos *recién llegados al sistema*: no han sido configurados de ninguna forma y todavía deben localizar el servicio de búsqueda. Jini resuelve este problema de la misma forma que

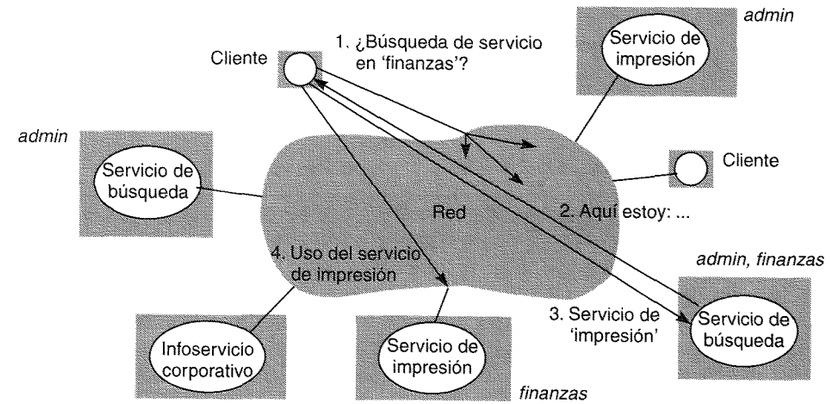


Figura 9.6. Servicio de descubrimiento en Jini.

cualquier otro servicio de descubrimiento: mediante la multidifusión a una dirección de multidifusión IP bien conocida. Esta dirección es conocida por todas las instancias del software Jini.

Cuando un cliente o servicio Jini comienza, envía una petición a esta dirección de multidifusión. Para ello se utiliza un valor de *tiempo de vida* que restringe el alcance del datagrama que lleva la solicitud a su vecindario en la red. Los servicios de búsqueda escuchan en un conector (*socket*) enlazado en la misma dirección para recibir dichas solicitudes. Cualquier servicio de búsqueda que reciba la solicitud y pueda responderla (véase a continuación) responderá con la dirección de contacto en la que recibe solicitudes de servicio. El solicitante puede realizar una invocación remota para buscar o registrar un servicio (el registro se denomina *joining* en Jini).

Los servicios de búsqueda también anuncian su existencia mediante datagramas enviados a la misma dirección de multidifusión. Los clientes y servicios Jini también pueden escuchar en la dirección de multidifusión para apercibirse de nuevos servicios de búsqueda.

Puede haber varias instancias del servicio de búsqueda accesibles mediante multidifusión desde un cierto cliente o servicio Jini. Cada una de esas instancias se configura con uno o más nombres de *grupo* como *admin*, *finanzas* y *ventas*, los cuales actúan como etiquetas de ámbito. Cuando clientes y servicios solicitan un servicio de búsqueda, pueden especificar cualquier grupo en el que estén especialmente interesados, y únicamente responderán los servicios de búsqueda enlazados a los mismos nombres de grupo. Por ejemplo, en una organización dividida en departamentos, un dispositivo que necesite una impresora *admin* puede encontrar una que pertenezca al grupo *admin*, y un dispositivo que necesite una impresora *finanzas* encontrará una enlazada al grupo *finanzas*.

En la Figura 9.6 se muestra un cliente Jini descubriendo y utilizando un servicio de impresión. El cliente necesita un servicio de búsqueda en el grupo *finanzas* de forma que multidifunde una solicitud enfocada al nombre del grupo. Sólo un servicio de búsqueda está enlazado al grupo *finanzas* (el mismo servicio que también está enlazado al grupo *admin*), y dicho servicio responde. La respuesta del servicio de búsqueda incluye su dirección por lo que el cliente se comunica directamente con él mediante RMI para localizar todos los servicios de tipo *impresión*. Sólo un servicio de impresión se ha registrado con dicho servicio de búsqueda bajo el grupo *finanzas*, devolviéndose un objeto para acceder a dicho servicio. A continuación el cliente utiliza el servicio de impresión directamente, utilizando el objeto devuelto. La figura también muestra otro servicio de impresión, el situado en el grupo *admin*. También hay un servicio de información corporativa que no está enlazado a ningún grupo particular (y que puede ser registrado con todos los servicios de búsqueda).

Jini utiliza *concesiones*, discutidas en el Capítulo 5. Cuando los servicios de Jini se registran con el servicio de búsqueda, se les proporciona una concesión que garantiza su entrada de registro por un período de tiempo limitado. Si un servicio no se comunica con el servicio de búsqueda para renovar su concesión antes de que éste expire, entonces se asume que ha fallado y el servicio de búsqueda puede eliminar la entrada.

9.4. ESTUDIO DEL CASO DEL SERVICIO DE NOMBRES GLOBAL

Un Servicio de Nombres Global (*Global Name Service*, GNS) fue diseñado e implementado por Lampson y algunos colegas en el Centro de Investigación de Sistemas de DEC [Lampson 1986] para proporcionar servicios de localización de recursos, direccionamiento de correo y autentificación. Los objetivos de diseño de GNS ya se mostraron al final de la Sección 9.1; reflejan el hecho de que un servicio de nombres, para su uso entre diferentes tipos de redes, debe soportar una base de datos de nombres que pueda extenderse para incluir los nombres de millones de computadores y (finalmente) direcciones de correo electrónico para billones de usuarios. Los diseñadores de GNS también admitieron que la base de datos de nombres tiende a tener un tiempo de vida largo, que debe continuar operando efectivamente mientras crece y mientras la red en la que está basada evoluciona. Durante ese tiempo, la estructura del espacio de nombres puede cambiar para reflejar los cambios en las estructuras de organización. El servicio debe proporcionar cabida a los cambios en los nombres de los individuos, organizaciones y grupos que gestiona; y cambios en la estructura de nominación, del tipo de los que ocurren cuando una empresa es absorbida por otra. En esta descripción, realizaremos especial hincapié en aquellas características de diseño que permiten acomodarse a dichos cambios.

La potencialmente enorme base de datos de nombres y la escala del entorno distribuido en el que GNS ha sido ideado para operar hacen que el uso de las cachés sea esencial, y provocan que sea extremadamente difícil mantener la consistencia completa entre todas las copias de una entrada de la base de datos. La estrategia adoptada para la consistencia caché se basa en la suposición de que las actualizaciones a las bases de datos serán poco frecuentes y que una diseminación lenta de las actualizaciones es aceptable, ya que los clientes pueden detectar y recuperarse de la utilización de datos de nominación caducados.

GNS gestiona una base de datos de nombres compuesta de un árbol de directorios que contienen nombres y valores. Los directorios se identifican mediante nombres de caminos compuestos referidos a una raíz, o relativos a un directorio de trabajo, muy parecidos a los nombres de archivos del sistema de archivos de UNIX. Cada directorio tiene también asignado un entero el cual sirve como *identificador de directorio* (DI) único. En esta sección utilizamos nombres en cursiva cuando nos referimos a un DI o a un directorio, de forma que *EC* es el identificador del directorio EC. Un directorio contiene una lista de nombres y referencias. Los valores almacenados en las hojas del árbol de directorio se organizan en *árboles de valores*, de forma que los atributos asociados con los nombres pueden ser valores estructurados.

Los nombres en GNS tienen dos partes: <*nombre de directorio, nombre de valor*>. La primera parte identifica un directorio; la segunda se refiere a un árbol de valor, o a alguna porción de un árbol de valor. Por ejemplo, véase la Figura 9.7 en la que cada DI se muestra como un entero corto a pesar de que en realidad se eligen de entre un rango de enteros que aseguran su singularidad. Los atributos de un usuario Pedro.Herrero en el directorio QMW deberían estar almacenados en el árbol de valor llamado <*EC*/UK/AC/QMW, Pedro.Herrero>. El árbol de valor incluye una contraseña, que puede referenciarse como <*EC*/UK/AC/QMW, Pedro.Herrero/clave>, además de varias direcciones de correo, cada una de las cuales se debe mostrar en el árbol de valor bajo un único nodo llamado <*EC*/UK/AC/QMW, Pedro.Herrero/buzones>.

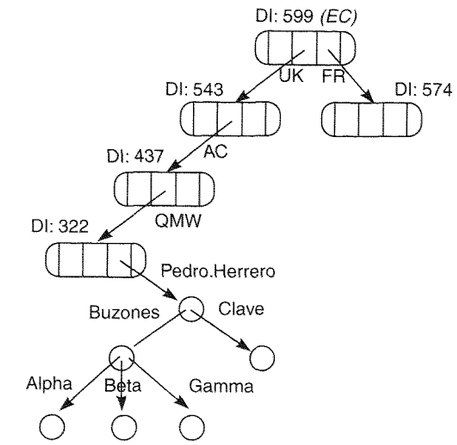


Figura 9.7. Árbol de directorio de GNS y valor del árbol para el usuario Pedro.Herrero.

El árbol de directorio está dividido y almacenado en muchos servidores, con cada partición replicada en varios servidores. La consistencia del árbol se mantiene frente a dos o más actualizaciones concurrentes; por ejemplo, dos usuarios pueden intentar crear entradas con el mismo nombre de forma simultánea, y sólo uno de ellos lo conseguirá. Los directorios replicados presentan un segundo problema de consistencia; se trata mediante un algoritmo de actualización distribuida asíncrona que asegura la consistencia final, pero sin garantías de que todas las copias estén siempre actualizadas. Este nivel de consistencia se considera satisfactorio.

◇ **Adecuación de los cambios.** Trataremos ahora los aspectos de diseño involucrados en la implementación del crecimiento y cambio en la estructura de la base de datos de nominación. En el nivel de clientes y administradores, el crecimiento se sirve de la forma habitual, es decir, mediante la extensión del árbol de directorio. Sin embargo, podemos querer integrar los árboles de nombres de dos servicios GNS que previamente estaban separados. Por ejemplo, ¿cómo podemos integrar la base de datos mostrada en la Figura 9.7 cuya raíz es el directorio *EC* con otra base de datos para NORTEAMÉRICA? En la Figura 9.8 se muestra la nueva raíz MUNDO insertada por encima de las raíces existentes de los dos árboles que se quieren mezclar. Esta técnica es sencilla, pero, ¿cómo afecta a los clientes que continúan utilizando nombres referidos a lo que era *la raíz* antes de que se realizara la integración? Por ejemplo <*/UK/AC/QMW, Pedro.Herrero*> es un nombre uti-

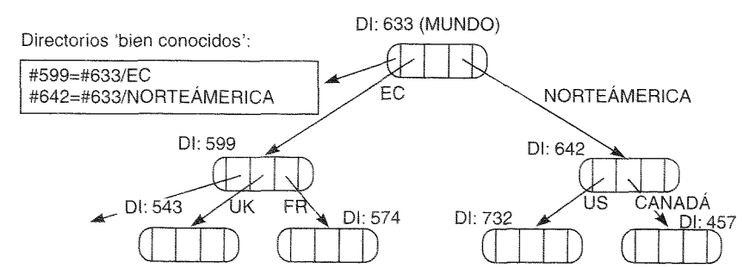


Figura 9.8. Firmas digitales con claves públicas.

lizado por los clientes antes de la integración. Se trata de un nombre absoluto (al comenzar con el símbolo raíz «/»), pero se refiere a la raíz *EC*, no a *MUNDO*. *EC* y *NORTEAMÉRICA* son *raíces de trabajo*, es decir, contextos iniciales sobre los que se buscan los nombres que comienzan con la raíz «/».

La existencia de identificadores de directorio únicos puede utilizarse para resolver este problema. La raíz de trabajo para cada programa debe ser identificada como parte de su entorno de ejecución (al igual que se hace para el directorio de trabajo de un programa). Cuando un cliente en la Comunidad Europea utiliza un nombre en la forma <*/UK/AC/QMW*, Pedro.Herrero>, su agente de usuario local, el cual conoce la raíz de trabajo, prefija el identificador de directorio EC (número 599), produciendo así el nombre <#599/UK/AC/QMW, Pedro.Herrero>. El agente de usuario entrega este nombre derivado al servidor GNS en la solicitud de búsqueda. El agente puede seguir el mismo criterio para el tratamiento de los nombres relativos referidos a directorios de trabajo. Los clientes que están al corriente de la nueva configuración también pueden proporcionar nombres absolutos al servidor GNS, los cuales se refieren al directorio conceptual súper raíz que contiene todos los identificadores de directorio, por ejemplo <MUNDO/EC/UK/AC/QMW, Pedro.Herrero>, pero el diseño no puede asumir que todos los clientes serán actualizados para tener en cuenta dicho cambio.

La técnica descrita resuelve el problema lógico, permitiendo a los usuarios y programas cliente continuar utilizando nombres definidos de forma relativa a una raíz antigua, incluso cuando se ha insertado una nueva raíz real, pero deja un problema de implementación: en una base de datos de nominación distribuida que pueda contener millones de directorios, ¿cómo puede el servicio GNS localizar un directorio partiendo únicamente de su identificador, como el número #599? La solución adoptada por GNS es listar los directorios utilizados como raíces de trabajo, como EC, en una tabla de *directorios bien conocidos* mantenida en el actual directorio raíz de la base de datos de nominación. Siempre que la raíz real de la base de datos de nombres cambie, como ocurre en la Figura 9.8, todos los servidores GNS son informados de la nueva localización de la raíz real. Estos servidores pueden entonces interpretar los nombres en la forma usual MUNDO/EC/UK/AC/QMW (referida a la raíz real) y también pueden interpretar los nombres en la forma #599/UK/AC/QMW ya que mediante la utilización de la tabla de *directorios bien conocidos* pueden traducirlos a nombres de camino completos comenzando en la raíz real.

GNS también soporta la reestructuración de la base de datos para acomodar cambios en la organización. Suponga que los Estados Unidos entran a formar parte de la Comunidad Europea (!). En la Figura 9.9 se muestra el nuevo árbol de directorio. Sin embargo, si el sub-árbol US se mueve directamente al directorio EC, los nombres que comiencen por MUNDO/NORTEAMÉRICA/US dejarán de funcionar. La solución adoptada por GNS consiste en insertar un *enlace simbólico* en lugar de la entrada original US (mostrada en negrilla en la Figura 9.9). El procedimiento de búsqueda de directorio de GNS interpreta el enlace como una redirección al directorio US en su nueva localización.

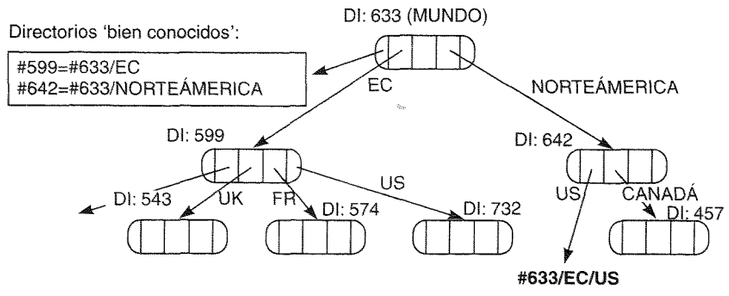


Figura 9.9. Reestructuración del directorio.

◇ **Discusión de GNS.** GNS desciende de Grapevine [Birrell y otros 1982] y Clearinghouse [Oppen y Dalal 1983], dos sistemas de nominación de mucho éxito desarrollados fundamentalmente para la gestión del correo en la Corporación Xerox. GNS resuelve perfectamente las necesidades de escalabilidad y reconfigurabilidad, pero la solución adoptada para mezclar y mover los árboles de directorio implica el requisito de que la base de datos (la tabla de directorios bien conocidos) deba estar replicada sobre cada nodo. En una red a gran escala, las reconfiguraciones pueden suceder en cualquier nivel, y esta tabla podría crecer hasta un tamaño muy elevado, entrando en conflicto con el objetivo de la escalabilidad.

9.5. ESTUDIO DEL CASO DEL SERVICIO DE DIRECTORIO X.500

X.500 es un servicio de directorio en el sentido definido en la Sección 9.3. Puede ser utilizado de la misma forma que un servicio de nombres convencional, pero se usa principalmente para satisfacer solicitudes descriptivas, diseñadas para descubrir los nombres y atributos de otros usuarios o recursos de sistema. Los usuarios pueden tener múltiples requisitos para la búsqueda en un directorio de usuarios en red, organizaciones y recursos de sistema para obtener información acerca de las entidades contenidas en el directorio. Los posibles usos de ese tipo de servicio pueden ser variados. Éstos van desde la realización de consultas análogas a la utilización de directorios telefónicos, como accesos corrientes a las *páginas blancas* para obtener la dirección de correo electrónico de un usuario o una solicitud a las *páginas amarillas* con el objetivo, por ejemplo, de obtener los nombres y números telefónicos de garajes especializados en la reparación de un cierto tipo particular de coche, al uso del directorio para acceder a detalles personales, como puesto de trabajo, hábitos dietéticos o incluso imágenes fotográficas de los individuos.

Este tipo de solicitudes pueden provenir de los usuarios, como el ejemplo de la consulta sobre garajes realizada en las *páginas amarillas* explicada previamente, o desde procesos, cuando pueden ser utilizados para la identificación de servicios que cumplen un cierto requisito funcional.

Los individuos y las organizaciones pueden utilizar un servicio de directorio para proporcionar una amplia cantidad de información sobre ellos mismos y sobre los recursos que ofrecen para ser utilizados desde la red. Los usuarios pueden realizar búsquedas sobre el directorio para encontrar información específica basándose únicamente en un conocimiento parcial de su nombre, estructura o contenido.

Las organizaciones de estandarización ITU e ISO han definido el *Servicio de Directorio X.500* [ITU/ISO 1997] como un servicio de red orientado al cumplimiento de dichos requisitos. El estándar lo describe como un servicio para el acceso a información de *entidades del mundo real*, pero puede utilizarse igualmente para el acceso a información sobre servicios y dispositivos hardware y software. X.500 se especifica como un servicio de nivel de aplicación en el conjunto de estándares de Interconexión de Sistemas Abiertos (OSI), pero su diseño no depende de forma significativa de otros estándares OSI por lo que puede verse como el diseño de un servicio de directorio de propósito general. Esbozaremos a continuación el diseño del servicio de directorio X.500 y su implementación. Los lectores interesados en una descripción más detallada de X.500 y de los métodos para su implementación pueden referirse al libro de Rose sobre este tema [Rose 1992]. X.500 es también la base para LDAP (se explicará posteriormente); y se utiliza en el servicio de directorio DCE [OSF 1997].

Los datos almacenados en los servidores X.500 se organizan en una estructura de árbol en la que los nodos tienen nombres, como en otros servidores de nombres descritos en este capítulo, pero en X.500 es posible almacenar una gran cantidad de atributos en cada nodo del árbol y el

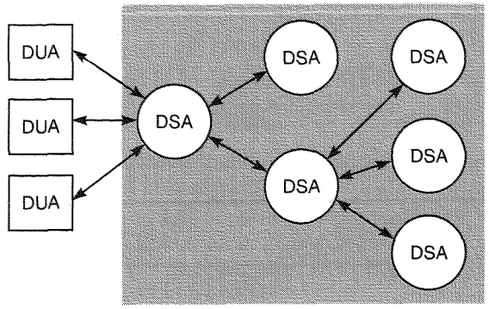


Figura 9.10. Arquitectura del servicio X.500.

acceso no se realiza simplemente por nombre sino que también se pueden realizar búsquedas basadas en cualquier combinación de atributos.

El árbol de nombres de X.500 se llama *Árbol de Información de Directorio* (*Directory Information Tree*, DIT), y la estructura de directorio completa incluyendo los datos asociados a los nodos se llama *Base de Información de Directorio* (*Directory Information Base*, DIB). Está pensado para tener un único DIB integrado que contiene la información proporcionada por las organizaciones de todo el mundo, con porciones de DIB situadas en servidores X.500 individuales. Normalmente, una organización de tamaño medio o grande debería proporcionar al menos un servidor.

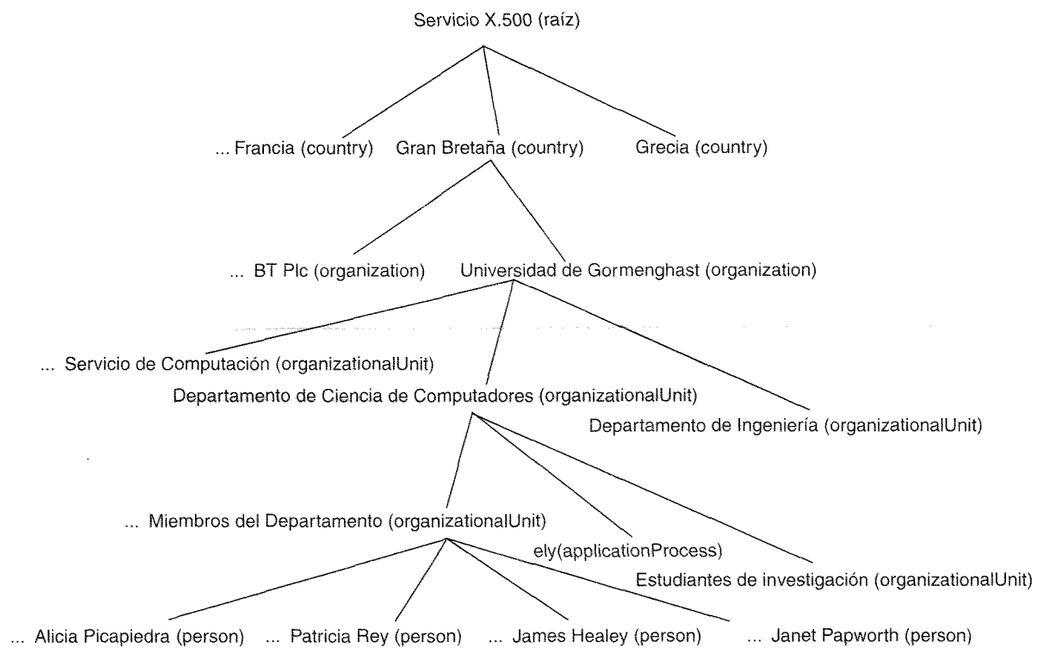


Figura 9.11. Parte del Árbol de Información de Directorio X.500.

info
Alicia Picapiedra, Miembro del Departamento, Departamento de Ciencia de los Computadores, Universidad de Gormenghast, GB

commonName
Alicia.L.Picapiedra
Alice.Picapiedra
Alice Picapiedra
A. Picapiedra

surname
Picapiedra

telephoneNumber
+44 986 33 4604

uid
alf

mail
alp@dcs.gormenghast.ac.uk
Alicia. Picapiedra @dcs.gormenghast.ac.uk

roomNumber
Z42

userClass
Investigador

Figura 9.12. Una Entrada DIB en X.500.

Los clientes acceden al directorio estableciendo una conexión al servidor y enviándole solicitudes de acceso. Los clientes pueden enviar una solicitud de información a cualquier servidor. Si los datos solicitados no están en el segmento de la DIB gestionado por el servidor sobre el que se ha contactado, éste realizará una invocación a otros servidores para resolver la solicitud o bien redirigirá al cliente sobre otro servidor.

Utilizando la terminología del estándar X.500, los servidores se llaman *Agentes de Servicio de Directorio* (*Directory Service Agent*, DSA) y los clientes son *Agentes de Usuario de Directorio* (*Directory User Agents*, DUA). En la Figura 9.10 se muestra la arquitectura del software y uno de los posibles modelos de navegación, en el que cada proceso cliente DUA interactúa con un único proceso DSA, el cual accede a otros DSA según se necesite para resolver la solicitud.

Cada entrada en la DIB está formada por un nombre y un conjunto de atributos. Al igual que en otros servidores de nombres, el nombre completo de una entrada se corresponde con un camino a través del DIT, desde la raíz del árbol hasta la entrada. Además de nombres completos o *absolutos*, un DUA puede establecer un contexto formado por un nodo base, y usar nombres relativos más cortos que proporcionan el camino desde el nodo base a la entrada nombrada.

En la Figura 9.11 se muestra la porción del Árbol de Información de Directorio que incluye la Universidad de Gormenghast, en Gran Bretaña, y en la Figura 9.12 aparece una de las entradas DIB asociadas. La estructura de datos para las entradas en la DIB y en la DIT es muy flexible. Una entrada en la DIB está formada por un conjunto de atributos, en el que cada atributo tiene un *tipo* y uno o más *valores*. El tipo de cada atributo distingue por un nombre de tipo (por ejemplo, *countryName*, *organizationName*, *commonName*, *telephoneNumber*, *mailbox*, *objectClass*). Si es necesario se pueden definir nuevos tipos para los atributos. Para cada tipo distinto existe una definición de tipo, la cual incluye una descripción de tipo y una definición sintáctica en Notación ASN.1 (un estándar de notación para definiciones sintácticas) definiendo representaciones para todos los valores permitidos del tipo.

Las entradas DIB se clasifican de forma similar a las estructuras de clases de objetos que aparecen en lenguajes de programación orientados a objetos. Cada entrada incluye un atributo *objectClass*, el cual determina la clase (o clases) del objeto al que se refiere dicha entrada. Algunos ejemplos de valores de *objectClass* son *Organization*, *organizationalPerson* y *document*. Si es necesario se pueden definir nuevas clases. La definición de una clase debe indicar qué atributos son obligatorios y cuáles son opcionales para las entradas en dicha clase. Las definiciones de clases se organizan en una jerarquía de herencias en la que todas las clases excepto una (llamada *topClass*) deben contener un atributo *objectClass* y el valor de dicho atributo debe ser el nombre de una

o más clases. Si existen varios valores *objectClass*, el objeto hereda los atributos obligatorios y opcionales de cada una de las clases.

El nombre de una entrada DIB (el nombre que determina su posición en el DIT) se determina seleccionando como *atributos distinguidos* a uno o más de sus atributos. Los atributos seleccionados para este propósito son llamados *Nombres Distinguidos* (DN) de una entrada.

Ahora podemos considerar los métodos utilizados para acceder a un directorio. Existen dos tipos principales de solicitudes de acceso:

read: debe proporcionarse un nombre absoluto o relativo (un *nombre de dominio* en la terminología X.500) de una entrada, junto con una lista de atributos para ser leídos (o una indicación de que se solicitan todos los atributos). El DSA localiza la entrada nombrada mediante la navegación en el DIT y enviando solicitudes a otros servidores DSA para aquellas partes del árbol que no mantenga. Obtiene los atributos requeridos y se los envía al cliente.

search: se trata de una solicitud de acceso basada en atributos. Se proporcionan en forma de argumentos un nombre base y una expresión de filtro. El nombre base especifica el nodo en el DIT desde el cual debe comenzar la búsqueda; la expresión de filtro es una expresión booleana que debe ser evaluada para cada nodo que se encuentre por debajo del nodo base. El filtro especifica un criterio de búsqueda: una combinación lógica de tests sobre los valores de cualquier atributo en una entrada. El mandato *search* devuelve una lista de nombres (Nombres de Dominio) para todas aquellas entradas por debajo del nodo base para las que la evaluación del filtro haya resultado *TRUE*.

Por ejemplo, se puede construir y aplicar un filtro para encontrar los *commonNames* de las personas que ocupan la habitación Z42 en el Departamento de Ciencias de la Computación de la Universidad de Gormenghast (véase la Figura 9.12). A continuación, se utiliza una solicitud de lectura para obtener uno o todos los atributos de las entradas DIB encontradas.

La búsqueda puede ser costosa cuando se aplica a grandes porciones del árbol de directorio (los cuales pueden residir en varios servidores). Se pueden proporcionar argumentos adicionales a *search* para restringir el ámbito de esta búsqueda, el tiempo durante el que se permite que una búsqueda continúe y el tamaño de la lista de entradas que se devuelve.

◇ **Administración y actualización de la DIB.** La interfaz DSA incluye operaciones para añadir, eliminar y modificar entradas. Se proporciona control de acceso tanto para solicitudes como para operaciones de actualización, de forma que el acceso a partes del DIT pueda restringirse a ciertos usuarios o clases de usuarios.

La DIB se divide bajo la suposición de que cada organización proporcionará como mínimo un servidor que gestione los detalles de las entidades en dicha organización. Se pueden replicar porciones de la DIB sobre diferentes servidores.

Al ser X.500 un estándar (*recomendación* si se utiliza la terminología CCITT) no cuestiona la implementación. Sin embargo, resulta evidente que cualquier implementación sobre múltiples servidores en una red de área amplia se debe basar en una utilización extensiva de las técnicas de replicación y caché para reducir las redirecciones generadas en las solicitudes.

Existe una implementación, descrita en Rose [1992], en un sistema desarrollado en el University College, en Londres, conocido como QUIPU [Kille 1991]. En esta implementación, tanto la caché como la replicación se realizan al nivel de entradas DIB individuales, y al nivel de colecciones de entradas que descienden del mismo nodo. Se asume que los valores pueden volverse inconsistentes después de una actualización y que el intervalo de tiempo que tarda en resolverse la inconsistencia puede ser de varios minutos. Esta forma de diseminación de las actualizaciones se considera normalmente aceptable para aplicaciones de servicio de directorios.

◇ **Protocolo de peso ligero para el acceso a directorios.** La interfaz estándar de X.500 utiliza un protocolo que implica a los niveles superiores de la pila de protocolos ISO. En la Universidad de Michigan un grupo propuso una aproximación de peso ligero llamada *Protocolo de Peso Ligero para el Acceso a Directorios* (LDAP), en la que un DUA accede a un servicio de directorio X.500 directamente utilizando TCP/IP. Véase RFC 2251 [Wahl y otros 1997]. LDAP simplifica la interfaz de X.500 de otras formas. Proporciona un API relativamente simple; reemplaza la codificación ASN.1 mediante la codificación textual.

A pesar de que la especificación LDAP se basa en X.500, LDAP no lo necesita. Cualquier otro servidor de directorio que cumpla la especificación LDAP más simple (al contrario que la especificación X.500) puede ser utilizado por una implementación LDAP. Por ejemplo, el Servicio de Directorio Activo de Microsoft proporciona una interfaz LDAP. LDAP ha sido ampliamente adoptado, sobre todo en servicios de directorio de intranet. Proporciona acceso seguro a datos de directorio mediante autenticación.

◇ **Discusión de X.500.** X.500 especifica un modelo detallado para servicios de directorio que van desde el ámbito de organizaciones individuales hasta directorios globales, siendo por ello muy importante. Su mayor impacto se ha producido a nivel de intranet, donde su influencia se ha conseguido a través de software LDAP de amplia difusión. El futuro de X.500 como estándar de directorio global (Internet) no está claro. En primer lugar, el requisito de un sistema de directorio global de existir en absoluto (basado o no en X.500) no está claro, sobre todo a la vista de los problemas de privacidad que pueden llegar a existir sobre dicho directorio. Segundo, dicho sistema de directorio global necesitará integrarse con los estándares de nombres existentes en Internet, incluyendo nombres DNS y direcciones de correo. Finalmente, las decisiones sobre el ámbito de la información que deberá ser proporcionada en los directorios deberá ser tomada a nivel nacional e internacional para asegurar la uniformidad de las clases de objetos almacenados en la DIB.

9.6. RESUMEN

En este capítulo se ha descrito el diseño e implementación de los servicios de nombres en sistemas distribuidos. Los servicios de nombres almacenan los atributos de objetos en un sistema distribuido (en especial sus direcciones) y devuelven esos atributos cuando se realiza una búsqueda sobre un cierto nombre de texto.

Los principales requisitos que debe poseer un servicio de nombres son la habilidad para manejar un número arbitrario de nombres; tiempo de vida grande; alta disponibilidad; aislamiento de los fallos; y tolerancia frente a la falta autenticación.

Las principales cuestiones de diseño son, en primer lugar, la estructura del espacio de nombres, es decir, las reglas sintácticas que gobiernan la formación de nombres. Una cuestión relacionada es el modelo de resolución: las reglas mediante las que un nombre multi-componente se resuelve en un conjunto de atributos. El conjunto de nombres enlazados debe ser gestionado. La mayor parte de los diseños dividen el espacio de nombres en dominios (secciones discretas del espacio de nombres), asociando a cada uno de ellos una única autoridad que controla el enlazado de nombres dentro del dominio.

La implementación del servicio de nombres puede abarcar diferentes organizaciones y comunidades de usuarios. En otras palabras, la colección de enlaces entre nombres y atributos, se almacena en varios servidores de nombres, cada uno de los cuales almacena al menos parte del conjunto de nombres del dominio de nombres. Se plantea así la cuestión de la navegación, es decir, el procedimiento utilizado para resolver un nombre cuando la información necesaria es almacenada en más de un punto. Los tipos de navegación soportados son la iterativa, la multidifusión, la recursiva controlada por el servidor y la no recursiva controlada por el servidor.

Otro aspecto importante sobre la implementación de un servicio de nombres es la utilización de la replicación y de la caché. Ambos esquemas sirven para ayudar a incrementar la disponibilidad del servicio y, también, ambos reducen el tiempo necesario para resolver un nombre.

En este capítulo se han considerado dos ejemplos básicos de diseños e implementaciones de servicios de nombres. El Sistema de Nombres de Dominio se utiliza ampliamente para la nominación de computadores y el direccionamiento de correo electrónico en Internet; consigue buenos tiempos de respuesta gracias a la replicación y al uso de la caché. El Servicio de Nombres Global es un diseño que ha abordado el problema de la reconfiguración del espacio de nombres provocado por los cambios en las organizaciones.

En este capítulo también se han estudiado los servicios de directorio y de descubrimiento. Son sistemas que buscan datos sobre objetos y servicios partiendo de información descriptiva basada en atributos proporcionada por los clientes. X.500 es un servicio de directorio definido en forma de estándar por la CCITT y la ISO. Puede ser utilizado por directorios cuyo ámbito varía desde una intranet hasta Internet. Se ha descrito además el servicio de descubrimiento Jini, diseñado para entornos de red espontáneos. Un servicio de búsqueda en Jini proporciona las interfaces para que los servicios se registren ellos mismos y para que los clientes descubran servicios que cumplan con sus requisitos.

EJERCICIOS

9.1. Describa los nombres (incluyendo identificadores) y los atributos utilizados en un servicio de archivos distribuido del tipo de NFS (véase el Capítulo 8).

9.2. Discuta los problemas surgidos por el uso de alias en un servicio de nombres, e indique cómo, si es posible, pueden ser resueltos.

9.3. Explique por qué la navegación iterativa es necesaria en un servicio de nombres en el que diferentes espacios de nombres están integrados parcialmente, como ocurre en el esquema de nominación de archivos proporcionado en NFS.

9.4. Describa el problema de los nombres desvinculados en la navegación por multidifusión. ¿Qué aspecto de la instalación de un servidor está implicado en la respuesta a búsquedas sobre nombres desconectados?

9.5. ¿Cómo ayuda el almacenamiento en caché a la disponibilidad de un servicio de nombres?

9.6. Discuta la ausencia de una distinción sintáctica (como el uso de un punto final «.») entre los nombres absolutos y relativos en DNS.

9.7. Investigue su configuración local de dominios y servidores DNS. Puede encontrar un programa llamado *nslookup* instalado en los sistemas UNIX, el cual le permite generar solicitudes individuales al servidor de nombres.

9.8. ¿Por qué los servidores raíz DNS gestionan entradas para nombres de dos niveles como *yahoo.com* y *purdue.edu*, en lugar de nombres de un solo nivel como *edu* y *com*?

9.9. ¿Qué otras direcciones de servidores de nombres mantienen por defecto servidores de nombres DNS y por qué?

9.10. ¿Por qué razones puede un cliente DNS elegir la navegación recursiva en lugar de la navegación iterativa? ¿Cuál es la relevancia de la navegación recursiva sobre la concurrencia en un servidor de nombres?

9.11. ¿Cuándo podrá un servidor DNS proporcionar múltiples respuestas a la búsqueda de un solo nombre, y por qué?

9.12. El servicio de búsqueda de Jini busca coincidencias entre las ofertas de servicio con las solicitudes de clientes basándose en atributos o en tipos Java. Explique con ejemplos la diferencia entre esas dos formas de buscar coincidencias. ¿Cuál es la ventaja de proporcionar ambos tipos?

9.13. Explique cómo el servicio de búsqueda de Jini utiliza concesiones para asegurar que la lista de servicios registrados en un servidor de búsquedas permanece actualizada a pesar de que los servicios puedan fallar o volverse inaccesibles.

9.14. Describa la utilización de multidifusión IP y grupos de nombres en el servicio de *descubrimiento* de Jini, el cual permite que clientes y servidores localicen servidores de búsquedas.

9.15. GNS no garantiza que todas las copias de las entradas en la base de datos de nombres estén actualizadas. ¿Cómo pueden los clientes GNS detectar que se les ha respondido con una entrada caducada? ¿Bajo qué circunstancias puede esto ser perjudicial?

9.16. Discuta las ventajas e inconvenientes potenciales de utilizar un servicio de directorio X.500 en lugar de DNS y los programas de manejo de correo de Internet. Esboce el diseño de un sistema de manejo de correo en un entorno de red en el que todos los usuarios y hosts de correo estén registrados en una base de datos X.500.

9.17. ¿Qué cuestiones de seguridad tienden a ser relevantes en un servicio de directorio como X.500 operando dentro de una organización de tipo universitario?

10

TIEMPO Y ESTADOS GLOBALES

10.1. Introducción
10.2. Relojes, eventos y estados de proceso
10.3. Sincronización de relojes físicos
10.4. Tiempo lógico y relojes lógicos
10.5. Estados globales
10.6. Depuración distribuida
10.7. Resumen

En este capítulo, presentamos algunos temas relacionados con la cuestión del tiempo en los sistemas distribuidos. El tiempo es una tema práctico importante. Por ejemplo, necesitamos computadores en todo el mundo para marcar en el tiempo las transacciones de comercio electrónico consistentemente. El tiempo es también una construcción teórica importante para comprender cómo se desarrollan las transacciones distribuidas. Pero el tiempo es problemático en los sistemas distribuidos. Cada computador puede tener su propio reloj físico, pero los relojes se desvían normalmente, y no podemos sincronizarlos perfectamente. Examinaremos algoritmos para sincronizar relojes físicos aproximadamente, y después pasaremos a explicar los relojes lógicos, incluyendo relojes vectoriales que son una herramienta para ordenar los eventos sin conocer de forma precisa cuándo ocurren.

La ausencia de tiempo físico global hace difícil descubrir el estado de nuestros programas distribuidos cuando se ejecutan. A menudo necesitamos conocer cuál es el estado de proceso A cuando el proceso B está en un cierto estado, pero no podemos confiar en los relojes físicos para conocer lo que es verdad al mismo tiempo. La segunda mitad del capítulo examina algoritmos para determinar estados globales de computaciones distribuidas a pesar de la falta de tiempo global.

10.1. INTRODUCCIÓN

Este capítulo presenta conceptos y algoritmos fundamentales relacionados con la monitorización de cómo se desarrolla la ejecución en los sistemas distribuidos, y para temporizar los eventos que ocurren en sus ejecuciones.

El tiempo es una característica importante e interesante en los sistemas distribuidos, por varias razones. Primero, porque el tiempo es una cantidad que a menudo queremos medir de forma precisa. Para saber en qué hora del día ocurrió un evento particular en un computador concreto es necesario sincronizar su reloj, con una fuente de tiempo externa, fidedigna. Por ejemplo, una transacción de *comercio electrónico* implica eventos en el computador del comerciante y en un computador del banco. Es importante, para propósitos de auditoría, que esos eventos sean marcados en el tiempo de forma precisa.

Segundo, se han desarrollado algoritmos que dependen de la sincronización de reloj para varios problemas en distribución [Liskov 1993]. Éstos incluyen el mantenimiento de la consistencia de los datos distribuidos (el uso de marcas de tiempo para serializar transacciones se discutirá en la Sección 12.6), la comprobación de la autenticidad de una solicitud enviada a un servidor (una versión del protocolo de autenticación de Kerberos, discutido en el Capítulo 7, depende de relojes sincronizados no fuertemente); y eliminando el procesamiento de actualizaciones duplicadas (ver, por ejemplo, Ladin y otros [1992]).

Einstein demostró, en su Teoría Especial de la Relatividad, las intrigantes consecuencias que se siguen de la observación que la velocidad de la luz es constante para todos los observadores, a pesar de su velocidad relativa. Él probó, a partir de esta suposición entre otras cosas, que dos sucesos que se considera que son simultáneos en un marco de referencia no son necesariamente simultáneos de acuerdo con los observadores en otros marcos de referencia que se mueven con relación a ellos. Por ejemplo, un observador en la Tierra y un observador alejándose de la Tierra en una nave espacial no coincidirán en el intervalo de tiempo entre dos sucesos, que será mayor a medida que aumente su velocidad relativa.

Sin embargo, el orden relativo de dos sucesos puede ser incluso invertido por dos observadores diferentes. Pero esto no puede suceder si un suceso pudiera haber sido la causa de que ocurra el otro. En este caso, el efecto físico sigue a la causa física para todos los observadores, aunque el tiempo transcurrido entre causa y efecto puede variar. Se probó, entonces la relatividad con respecto al observador de la temporización de los sucesos físicos, así como se desacreditó la noción de tiempo físico absoluto de Newton. No existe un reloj físico especial en el universo al que podamos invocar cuando queramos medir intervalos de tiempo.

La noción de tiempo físico también es problemática en un sistema distribuido. Esto no es debido a los efectos de la relatividad especial, que son despreciables o no existentes para computadores normales (¡a menos que uno considere los computadores viajando en las naves espaciales!). El problema se basa en una limitación similar de nuestra capacidad para marcar sucesos en diferentes nodos de una manera suficientemente precisa para conocer el orden en el que ocurrieron cualquier par de sucesos, o si ellos ocurrieron simultáneamente. No hay un tiempo absoluto, global al que podamos invocar. Y aún más, a veces necesitamos observar sistemas distribuidos y establecer si ciertos estados de asuntos ocurrieron al mismo tiempo. Por ejemplo, en los sistemas orientados a objetos necesitamos ser capaces de establecer las referencias a un objeto particular ya apenas existe, ya si el objeto ha llegado a ser desechable (en cuyo caso podemos liberar su memoria). Establecer esto requiere observaciones de los estados de los procesos (para encontrar si ellos contienen referencias) y de los canales de comunicación entre los procesos (en el caso de los mensajes que contienen referencias están en tránsito).

En la primera mitad de este capítulo, examinamos métodos en los que los relojes de los computadores pueden ser sincronizados aproximadamente, utilizando paso de mensajes. A continuación presentaremos los relojes lógicos, incluyendo los relojes vectoriales, que son utilizados para definir un orden de sucesos sin medir el tiempo físico en el que ellos han ocurrido.

En la segunda mitad, describimos algoritmos cuyo propósito es capturar los estados globales de los sistemas distribuidos cuando se ejecutan.

10.2. RELOJES, EVENTOS Y ESTADOS DE PROCESO

En el Capítulo 2 se presentó un modelo introductorio de interacción entre procesos en un sistema distribuido. Nosotros refinaremos ese modelo con el fin de ayudarnos a comprender cómo caracterizar la evolución del sistema a medida que se ejecuta, y cómo marcar los eventos de la ejecución del sistema que interesan a los usuarios. Comenzamos considerando cómo ordenar y colocar marcas de tiempo a los eventos que ocurren en un único proceso.

Consideramos que un sistema distribuido consta de una colección $\wp$ de N procesos p_i, $i = 1, 2, \ldots N$. Cada proceso se ejecuta en un único procesador, y los procesadores no comparten memoria (el Capítulo 17 considera el caso de procesos que comparten memoria). Cada proceso p_i en $\wp$ tiene un estado s_i que, en general, se transforma a medida que se ejecuta. El estado del proceso incluye los valores de todas sus variables. El estado puede incluir también los valores de cualquiera de los objetos en el entorno de su sistema operativo local que lo afecte, como los archivos. Suponemos que los procesos no se pueden comunicar entre ellos de ninguna forma, excepto mediante el envío de mensajes a través de la red. Por lo tanto, por ejemplo, si el proceso opera sobre los brazos de un robot conectados a sus respectivos nodos en el sistema, no les está permitido comunicarse ¡agitando los brazos de otro robot!

A medida que cada proceso p_i se ejecuta toma una serie de acciones, cada una de las cuales es un mensaje Envía o Recibe, o una operación que transforma el estado p_i, que cambia uno o más valores de s_i. En la práctica, podemos elegir si utilizar una descripción de alto nivel de las acciones, de acuerdo con la aplicación. Por ejemplo, si los procesos en $\wp$ están incluidos en una aplicación de comercio electrónico, entonces las acciones pueden ser del estilo de *el cliente envió un mensaje de pedido* o *el servidor del comerciante anotó la transacción en el registro*.

Decimos que un evento es la ocurrencia de una única acción que un proceso realiza a medida que se ejecuta, una acción de comunicación o una acción de transformación del estado. La secuencia de sucesos en un único proceso p_i puede ser colocada en orden único, total, que indicaremos con la relación $\rightarrow_i$ entre eventos. Es decir, $e \rightarrow_i e'$ si y sólo si el evento e ocurre antes del e' en p_i. Esta ordenación está bien definida, sea o no el proceso multihilo, puesto que hemos supuesto que el proceso se ejecuta en un procesador único.

Ahora podemos definir la *historia* del proceso p_i como la serie de eventos que tienen lugar en él, ordenados de la forma que hemos descrito con la relación $\rightarrow_i$:

$$historia(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots>$$

◇ **Relojes.** Hemos visto cómo ordenar los eventos en un proceso pero no cómo proporcionarles marcas de tiempo, para asignarles una fecha y una hora del día. Los computadores pueden disponer de su propio reloj físico. Estos relojes son dispositivos electrónicos que cuentan las oscilaciones que ocurren en un cristal a una frecuencia definida, y que normalmente dividen esta cuenta y almacenan el resultado en un registro contador. Los dispositivos de reloj pueden estar programados para generar impulsos a intervalos regulares con el fin de que, por ejemplo, pueda ser implementada la división de tiempo, sin embargo nosotros no trataremos con este aspecto del funcionamiento de los relojes.

El sistema operativo lee el valor del reloj hardware $H_i(t)$ del nodo, lo escala y añade una compensación para producir un reloj software $C_i(t) = \alpha H_i(t) + \beta$ que mide aproximadamente el tiempo

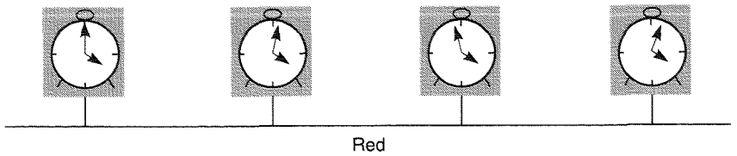


Figura 10.1. Sesgo entre los relojes de los computadores en un sistema distribuido.

real, físico t para el proceso p_i. En otras palabras, cuando el tiempo real en un marco de referencia absoluto es t, $C_i(t)$ es la lectura del reloj software. Por ejemplo, $C_i(t)$ podría ser el valor de 64-bits del número de nanosegundos que han transcurrido en el instante t desde una referencia conveniente de tiempo. En general, el reloj no es suficientemente preciso, y por tanto $C_i(t)$ difiere de t. Sin embargo, si C_i se comporta suficientemente bien (examinaremos la noción de corrección de reloj dentro de poco), podemos utilizar su valor para marcar el tiempo de cualquier evento en p_i. Hay que indicar qué eventos sucesivos corresponderán con marcas de tiempo diferentes solo si la *resolución del reloj*, el período entre actualizaciones del valor del reloj, es más pequeño que el intervalo de tiempo entre eventos sucesivos. La tasa a la que ocurren los sucesos dependen de factores tales como la longitud del ciclo de instrucción del procesador.

◇ **Sesgo y deriva de reloj.** Los relojes de los computadores, como los otros, tienden a no estar en perfecto acuerdo (véase la Figura 10.1). La diferencia instantánea entre las lecturas de dos relojes cualquiera se llama su *sesgo*. También los relojes basados en cristal utilizados en los computadores están, como los otros relojes, sujetos a *derivas de reloj*, que significa que ellos cuentan el tiempo a diferentes ritmos, y por lo tanto divergen. Los osciladores subyacentes están sujetos a variaciones físicas, con la consecuencia que sus frecuencias de oscilación difieren. Aun más, la misma frecuencia de los relojes varía con la temperatura. Existen diseños que intentan compensar esta variación, pero no la pueden eliminar. La diferencia en el período de oscilación entre dos relojes puede ser extremadamente pequeña, pero la diferencia acumulada sobre muchas oscilaciones conduce a una diferencia observable en los contadores registrados por los dos relojes, independientemente de con qué precisión fueron inicializados con el mismo valor. Un *ritmo de deriva* de reloj es el cambio en la compensación (diferencia en la lectura) entre el reloj y un reloj de referencia nominal perfecto por unidad de tiempo medido por el reloj de referencia. Para relojes ordinarios basados en un cristal de cuarzo, suele ser de aproximadamente 10^{-6} segundos/segundo, dando una diferencia de 1 segundo cada 1.000.000 segundos, o 11,6 días. El ritmo de deriva de los relojes de cuarzo de *alta precisión* es de aproximadamente 10^{-7} o 10^{-8}.

◇ **Tiempo Universal Coordinado.** Los relojes de computador pueden sincronizarse con fuentes externas de tiempo de gran precisión. Los relojes físicos más precisos utilizan osciladores atómicos, cuyo ritmo de deriva es aproximadamente de una parte en 10^{13}. La salida de estos relojes atómicos se utiliza como estándar para el tiempo real transcurrido, conocido como *Tiempo Atómico Internacional*. Establecido en 1967, el segundo estándar se ha definido como 9.192.631.770 períodos de transición entre dos niveles hiperfinos del estado fundamental del Cesio-133 (Cs^{133}).

Los segundos, los años y otras unidades de tiempo que nosotros utilizamos están arraigadas en el tiempo astronómico. Fueron definidos originalmente en términos de la rotación de la tierra sobre su eje y su rotación alrededor del sol. Sin embargo, el período de rotación de la tierra alrededor de su eje va siendo gradualmente más largo, principalmente por la fricción de las mareas; los efectos atmosféricos y las corrientes de convección dentro del núcleo de la tierra también pueden producir incrementos y decrementos de duración corta en el período. Por lo tanto el tiempo astronómico y el tiempo atómico tienen una tendencia a separarse.

El *Tiempo Universal Coordinado* —abreviado como UTC (del equivalente francés)— es un estándar internacional de cronometraje. Está basado en el tiempo atómico, aunque ocasionalmente se inserta un salto de un segundo —o, más raramente, borrado— para mantenerse en sintonía con el tiempo astronómico. Las señales UTC se sincronizan y difunden regularmente desde las estaciones de radio terrestres y los satélites, que cubren muchas partes del mundo. Por ejemplo, en USA la estación de radio WWV difunde señales de tiempo en frecuencias de onda corta. Las fuentes de los satélites incluyen el *Sistema de Posicionamiento Global* (*Global Positioning System*, GPS).

Hay disponibles receptores comerciales. Comparados con UTC *perfecto*, las señales recibidas desde las estaciones terrestres tienen una precisión del orden de 0,1-10 milisegundos, dependiendo de la estación utilizada. Las señales recibidas desde GPS tienen una precisión de alrededor de 1 microsegundo. Los computadores con receptores adjuntos pueden sincronizar sus relojes con estas señales de tiempo. Los computadores pueden también recibir el tiempo con una precisión de unos pocos milisegundos sobre una línea telefónica con organizaciones tales como el Instituto Nacional para los Estándares y Tecnología (*National Institute for Standars and Technology*, NIST) en USA.

10.3. SINCRONIZACIÓN DE RELOJES FÍSICOS

Para conocer en qué hora del día ocurren los sucesos en los procesos de nuestro sistema distribuido $\wp$ —por ejemplo, para propósitos de contabilidad— es necesario sincronizar los relojes de los procesos C_i con una fuente de tiempo externa autorizada. Esto es la *sincronización externa*. Y si los relojes C_i están sincronizados con otro con un grado de precisión conocido, entonces podemos medir el intervalo entre dos eventos que ocurren en diferentes computadores llamando a sus relojes locales, incluso aunque ellos no estén necesariamente sincronizados con una fuente externa de tiempo. Esto es *sincronización interna*. Definimos estos dos modos de sincronización más detalladamente como sigue, sobre un intervalo de tiempo real I:

Sincronización externa: para una sincronización dada $D > 0$, y para una fuente S de tiempo UTC, $|S_i(t) - C_i(t)| < D$, para $i = 1, 2, \ldots N$ y para todos los tiempos reales t en I. Otra forma de decir esto es que los relojes C_i son *precisos* con el límite D.

Sincronización interna: para una sincronización dada $D > 0$, $|C_i(t) - C_j(t)| < D$, para $i = 1, 2, \ldots N$ y para todos los tiempos reales t en I. Otra forma de decir esto es que los relojes C_i *concuerdan* con el límite D.

Los relojes que están sincronizados internamente no están necesariamente sincronizados externamente, puesto que pueden desplazarse colectivamente desde una fuente de tiempo externa incluso aunque estén de acuerdo entre sí. Sin embargo, se deduce de las definiciones que si el sistema $\wp$ está sincronizado externamente con un límite D entonces el mismo sistema está sincronizado internamente con un límite de $2D$.

Se han sugerido varias nociones de *corrección* para relojes. Es común definir que un reloj hardware H es correcto si su ritmo de deriva cae dentro de un límite conocido $\rho > 0$ (un valor derivado de uno proporcionado por el fabricante, como 10^{-6} segundos/segundo). Esto significa que el error midiendo el intervalo entre tiempos reales t y t' ($t' > t$) está limitado por:

$$(1 - \rho)(t' - t) \leqslant H(t') - H(t) \leqslant (1 + \rho) \leqslant (t' - t)$$

Esta condición prohíbe saltos en el valor de los relojes hardware (durante su operación normal). A veces nosotros requerimos que nuestros relojes software obedezcan la condición. Pero una condición mas débil de *monotonicidad* puede bastar. Monotonicidad es la condición que un reloj C sólo avanza siempre que:

$$t' > t \quad \Rightarrow \quad C(t') > C(t)$$

Por ejemplo, la utilidad *make* de UNIX es una herramienta que se utiliza para compilar sólo aquellos archivos fuente que han sido modificados desde la última vez que fueron compilados. Las fechas de modificación de cada par correspondiente de archivos fuente y objeto son comparadas para determinar esta condición. Si un computador cuyo reloj estaba corriendo rápido retrasa su reloj después de compilar un archivo fuente pero antes de que el archivo sea cambiado, puede parecer que el archivo fuente ha sido modificado antes de la compilación. Erróneamente, *make* no recompilará el archivo fuente.

Podemos conseguir monotonicidad a pesar del hecho de que un reloj se encuentre corriendo rápido. Sólo necesitamos cambiar el ritmo al que se hacen las actualizaciones al tiempo dado a las aplicaciones. Esto puede ser conseguido por software sin cambiar el ritmo al que los ticks del reloj hardware subyacente; recuerde que $C_i(t) = \alpha H_i(t) + \beta$, donde somos libres de elegir los valores de α y β.

Una condición híbrida de corrección que es aplicada a veces es requerir que un reloj obedezca la condición de monotonicidad, y que su ritmo de deriva esté limitado entre los puntos de sincronización, pero permitir que el valor del reloj salte por delante de los puntos de sincronización.

Un reloj que no se atiene a ninguna de las condiciones de corrección aplicadas se dice que es *defectuoso*. Un *fallo de ruptura* de reloj se dice que ocurre cuando el reloj deja de pulsar por completo; cualquier otro fallo de reloj es un *fallo arbitrario*. Un ejemplo de un fallo arbitrario es el de un reloj con el *error del año* 2000, que rompe la condición de monotonicidad registrando la fecha de después del 31 de diciembre de 1999 como 1 de enero de 1900 en lugar de 2000; otro ejemplo es un reloj cuyas baterías están muy bajas y cuyo ritmo de deriva llega a ser de repente muy grande.

Tenga en cuenta que los relojes no tienen por qué ser precisos para ser correctos, de acuerdo con las definiciones. Puesto que el objetivo puede ser la sincronización interna más que la externa, los criterios para corrección están relacionados únicamente con el funcionamiento adecuado del *mecanismo* del reloj, no con su ajuste absoluto.

Describimos ahora algoritmos para sincronización externa e interna.

10.3.1. SINCRONIZACIÓN EN UN SISTEMA SÍNCRONO

Comenzamos considerando el caso más simple posible: El de sincronización interna entre dos procesos en un sistema distribuido síncrono. En un sistema síncrono se conocen los límites para el ritmo de deriva de los relojes, el máximo retardo de transmisión de mensajes y el tiempo para ejecutar cada paso de un proceso (véase la Sección 2.3.1).

Un proceso envía el tiempo t de su reloj local a otro en un mensaje m. En principio, el proceso receptor podría tener su reloj en el tiempo $t + T_{trans}$, donde T_{trans} es el tiempo preciso para transmitir m entre ellos. Los dos relojes podrían coincidir (puesto que la meta es la sincronización interna, no importa si el reloj del proceso emisor es preciso).

Desafortunadamente, T_{trans} está sujeto a variaciones y es desconocido. En general, otros procesos están compitiendo por recursos con los procesos a sincronizar en sus respectivos nodos y otros mensajes compiten con m por la red. A pesar de todo, hay siempre un tiempo mínimo de transmisión *min* que podría obtenerse si no se ejecutara ningún otro proceso y no existiera más tráfico en la red; *min* puede ser medido o estimado de forma conservadora.

En un sistema síncrono, por definición hay también un límite superior *max* del tiempo tomado para transmitir cualquier mensaje. Sea la incertidumbre en el tiempo de transmisión del mensaje u, donde $u = (max - min)$. Si el receptor coloca su reloj a $t + min$, entonces el sesgo del reloj puede ser como mucho u, puesto que el mensaje puede de hecho haber necesitado un tiempo *max* para llegar. Similarmente, si coloca su reloj a $t + max$, el sesgo podría ser otra vez como mucho u. Sin embargo, si coloca su reloj al punto mitad, $t + (max + min)/2$, entonces el sesgo es como mucho $u/2$. En general, para un sistema síncrono, el límite óptimo que puede ser conseguido en el sesgo de reloj cuando sincronizamos N relojes es $u(1 - 1/N)$ [Lindelius y Linch 1984].

La mayoría de los sistemas distribuidos encontrados en la práctica son asíncronos: Los factores que conducen a retardos de los mensajes no están limitados en su efecto, y no hay límite superior *max* en los retardos de transmisión de mensajes. Esto es particularmente así para Internet. Para un sistema asíncrono, nosotros sólo podremos decir que $T_{trans} = min + x$, donde $x \geqslant 0$. El valor de x no es conocido en un caso particular, aunque para una instalación particular podría medirse una distribución de valores.

10.3.2. MÉTODO DE CRISTIAN PARA SINCRONIZAR RELOJES

Cristian [1989] sugirió la utilización de un servidor de tiempo, conectado a un dispositivo que recibe señales de una fuente de UTC, para sincronizar computadores externamente. Bajo solicitud, el proceso servidor S proporciona el tiempo de acuerdo con su reloj tal como se muestra en la Figura 10.2.

Cristian observó que aunque no hay límite superior en los retardos de transmisión de mensajes en un sistema asíncrono, los tiempos de ida y vuelta de los mensajes intercambiados entre cada par de procesos son a menudo razonablemente cortos, una pequeña fracción de un segundo. El describe el algoritmo como *probabilístico*: el método consigue sincronización sólo si los tiempos de ida y vuelta entre el cliente y el servidor son suficientemente cortos comparados con la precisión requerida.

Un proceso p solicita el tiempo en un mensaje m_r, y recibe el valor del tiempo t en un mensaje m_t (t se inserta en m_t en el último instante posible antes de la transmisión desde un computador de S). El proceso p registra el tiempo total de ida y vuelta T_{round} tomado para enviar la solicitud m_r y recibir la respuesta m_t. Se puede medir este tiempo con precisión razonable si su ritmo de deriva de reloj es pequeño. Por ejemplo, el tiempo de ida y vuelta debiera ser del orden de 1 a 10 milisegundos en una LAN, sobre dicho tiempo un reloj con un ritmo de deriva de 10^{-6} milisegundos/segundo varía como mucho 10^{-5} milisegundos.

Una estimación sencilla del tiempo al que p debe fijar su reloj es $t + T_{round}/2$, que supone que el tiempo transcurrido se desdoble igualmente antes y después de que S coloque t en m_t. Esto es normalmente una suposición razonable de precisión, a menos que los dos mensajes sean transmitidos sobre redes diferentes. Si el valor del tiempo mínimo de transmisión *min* es conocido o puede ser estimado conservativamente, entonces podemos determinar la precisión de este resultado como sigue.

El instante más temprano en el que S podría haber colocado el tiempo en m_t fue *min* después de que p enviara m_r. El punto más tarde al que podría haberse hecho esto sería *min* antes de que m_t llegue a p. El tiempo del reloj de S cuando el mensaje de respuesta llega estará en el rango $[t + min, t + T_{round} - min]$. La anchura de este rango es $T_{round} - 2min$, por lo que la precisión es $\pm(T_{round}/2 - min)$.

Se puede tratar con la variabilidad en alguna medida haciendo varias solicitudes a S (espaciando las solicitudes de modo que se pueda eliminar la congestión transitoria) y tomando el valor

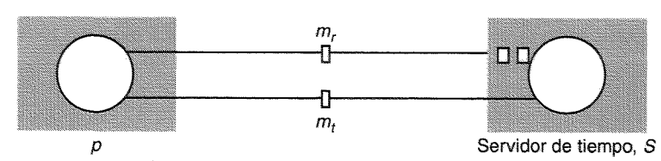


Figura 10.2. Sincronización de relojes utilizando un servidor de tiempo.

mínimo de T_{round} para conseguir una estimación más precisa. Cuanto más grande es la precisión requerida, más pequeña es la probabilidad de conseguirla. Esto es porque los resultados más precisos son aquellos en los que ambos mensajes son transmitidos en una tiempo próximo a *min*, un suceso improbable en una red ocupada.

◇ **Discusión del algoritmo de Cristian.** Como se ha descrito, el método de Cristian sufre del problema asociado con todos los servicios implementados por un servidor único, por los que el único servidor de tiempo puede caer y hacer que la sincronización sea imposible temporalmente. Cristian sugirió, por esta razón, que el tiempo debiera ser proporcionado por un grupo de servidores de tiempo sincronizados, cada uno con un receptor para señales de tiempo UTC. Por ejemplo, un cliente podría repartir su solicitud a todos los servidores y usar sólo la primera respuesta obtenida.

Hay que tener en cuenta que un servidor de tiempo erróneo que responda con valores espurios de tiempo, o un servidor de tiempo impostor que responde con tiempos erróneos deliberadamente, podrían causar muchos daños en un sistema de computadores. Estos problemas fueron más allá del alcance del trabajo descrito por Cristian [1989], que supone que las señales de tiempo externo se autocomprueban. Cristian y Fetzer [1994] describen una familia de protocolos probabilísticos para la sincronización de relojes internos, cada uno de los cuales toleran ciertos fallos. Srikanth y Toueg [1987] describen primero un algoritmo que es óptimo con respecto a la precisión de relojes sincronizados, al mismo tiempo que tolera algunos fallos. Dolev y otros [1986] mostraron que si f es el número de relojes defectuosos de un total de N, entonces debemos tener $N > 3$ si otros relojes, correctos, son capaces todavía de alcanzar el acuerdo. El problema de tratar con relojes defectuosos se aborda parcialmente por el algoritmo de Berkeley, que se describe a continuación. El problema de la interferencia maliciosa con la sincronización del tiempo puede ser tratado con técnicas de autenticación.

10.3.3. EL ALGORITMO DE BERKELEY

Gusella y Zatty [1989] describieron un algoritmo para sincronización interna que ellos desarrollaron para colecciones de computadores ejecutando el UNIX Berkeley. En este algoritmo se elige, un computador coordinador para actuar como *maestro*. A diferencia del protocolo de Cristian, este computador consulta periódicamente a los otros computadores cuyos relojes están para ser sincronizados, llamadas *esclavos*. Los esclavos le devuelven sus valores de reloj. El maestro estima sus tiempos locales de reloj observando los tiempos de ida y vuelta (semejantes a los de la técnica de Cristian), y promedia los valores obtenidos (incluyendo la lectura de su propio reloj). El balance de las probabilidades es que este promedio contrarresta las tendencias de los relojes individuales a funcionar rápido o lento. La precisión del protocolo depende de un tiempo de ida y vuelta máximo nominal entre el maestro y los esclavos. El maestro elimina cualquier lectura adicional asociada con tiempos más grandes que el máximo.

En lugar de reenviar el tiempo actual actualizado a los demás computadoress, que introduciría una nueva imprecisión debido al tiempo de transmisión del mensaje, el maestro envía la cantidad que precisa el esclavo para hacer su ajuste. Esto puede ser un valor positivo o negativo.

El algoritmo elimina las lecturas de relojes defectuosos. Dichos relojes podrían tener un efecto adverso significativo si se tomara un promedio ordinario. El maestro toma un *promedio tolerante a fallos*. Esto es, se elige un subconjunto de los relojes que no difieran entre ellos más de una determinada cantidad, y el promedio se toma de las lecturas de sólo estos relojes.

Gusella y Zatti describieron un experimento implicando 15 computadoress cuyos relojes fueron sincronizados en alrededor de 10 a 25 milisegundos utilizando su protocolo. El ritmo de deriva de los relojes locales que fue medido era menos que 2×10^{-5}, y el tiempo máximo de ida y vuelta era de 10 milisegundos.

Si el maestro fallara, entonces puede ser elegido otro para asumir y funcionar exactamente como su predecesor. La Sección 11.3 discute algunos algoritmos de elección de propósito general. Hay que tener en cuenta que éstos no garantizan la elección de un nuevo maestro en un tiempo limitado y por tanto la diferencia entre dos relojes sería ilimitada si fueran utilizados.

10.3.4. EL PROTOCOLO DEL TIEMPO DE RED

El método de Cristian y el algoritmo de Berkeley están pensados principalmente para utilizar en intranets. El Protocolo de Tiempo de Red (*Network Time Protocol*, NTP) [Mills 1995] define una arquitectura para un servicio de tiempo y un protocolo para distribuir la información del tiempo sobre Internet.

Los objetivos y las metas principales de diseño de NTP son los siguientes.

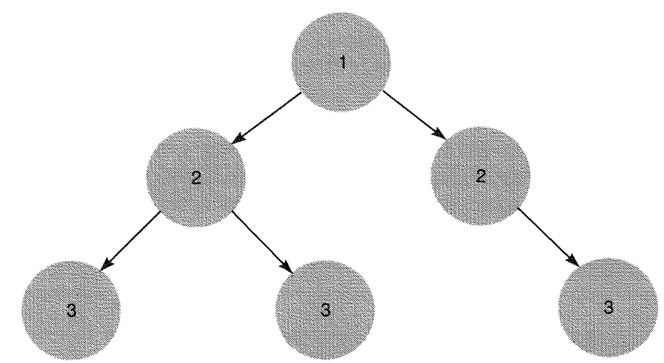
Proporcionar un servicio que permita a los clientes a lo largo de Internet estar sincronizados de forma precisa a UTC: a pesar de los retardos largos y variables de los mensajes encontrados en la comunicación Internet, NTP emplea técnicas estadísticas para el filtrado de los datos de tiempo y discrimina entre la calidad de los datos de tiempo de los diferentes servidores.

Proporcionar un servicio fiable que pueda sobrevivir a pérdidas largas de conectividad: hay servidores redundantes y recorridos redundantes entre los servidores. Los servidores pueden reconfigurarse para continuar proporcionando el servicio si uno de ellos llega a ser inalcanzable.

Permitir a los clientes resincronizar con suficiente frecuencia para compensar las tasas de deriva encontradas en la mayoría de los computadores: el servicio está diseñado para escalar a gran número de clientes y servidores.

Proporcionar protección contra la interferencia con el servicio de tiempo, ya sea maliciosa o accidental: el servicio de tiempo utiliza técnicas de autenticación para comprobar que los datos de tiempo se originan de las fuentes verdaderas reclamadas. También valida las direcciones de devolución de los mensajes enviados hacia él.

El servicio NTP está proporcionado por una red de servidores localizados a través de Internet. *Los servidores primarios* están conectados directamente a una fuente de tiempo como un radioreloj recibiendo UTC; los servidores secundarios están sincronizados, en el fondo, con servidores primarios. Los servidores están conectados en una jerarquía lógica llamada una *subred de sincronización* (véase la Figura 10.3), cuyos niveles se llaman *estratos*. Los servidores primarios ocupan el estra-



Nota: Las flechas indican control de la sincronización, los números indican estratos.

Figura 10.3. Un ejemplo de una subred de sincronización en una implementación NTP.

to 1: ellos están en la raíz. Los servidores del estrato 2 son servidores secundarios que están sincronizados directamente con los servidores primarios; los servidores del estrato 3 están sincronizados con los del estrato 2, y así sucesivamente. Los servidores del nivel más bajo (hojas) se ejecutan en las estaciones de trabajo de los usuarios.

Los relojes que pertenecen a los servidores con números altos de estrato tienen tendencia a ser menos fiables que aquéllos con números bajos de estrato, porque se introducen errores en cada nivel de sincronización. NTP tiene en cuenta también los retardos totales de ida y vuelta de los mensajes hacia la raíz valorando la calidad de los datos de tiempo mantenidos por un servidor particular.

La subred de sincronización se puede reconfigurar cuando los servidores llegan a ser inalcanzables o se producen fallos. Si, por ejemplo, falla la fuente UTC de un servidor primario, entonces él puede llegar a ser un servidor secundario del estrato 2. Si falla la fuente normal de sincronización de un servidor secundario o llega a ser inalcanzable, entonces él puede sincronizarse con otro servidor.

Los servidores NTP se sincronizan entre sí en uno de estos tres modos: multidifusión, llamada a procedimiento y modo simétrico. *El modo multidifusión* está pensado para su uso en una LAN de alta velocidad. Uno o más servidores reparten periódicamente el tiempo a los servidores que se ejecutan en otros computadores conectados en la LAN, que fijan sus relojes suponiendo un pequeño retardo. Este modo puede alcanzar sólo precisiones relativamente bajas, pero que no obstante son consideradas suficientes para muchos propósitos.

El modo de llamada a procedimiento es similar al funcionamiento del algoritmo de Cristian, descrito anteriormente. En este modo, un servidor acepta solicitudes de otros computadores, que el procesa respondiendo con su marca de tiempo (lectura actual del reloj). Este modo es adecuado donde se requieren precisiones más altas que las que se pueden conseguir con multidifusión, o donde la multidifusión no viene soportada por hardware. Por ejemplo, servidores de archivos en la misma o en LAN vecinas, que necesitan mantener una información precisa del tiempo para acceso a los archivos, no pueden contactar con un servidor local en el modo de llamada procedimiento.

Finalmente, el *modo simétrico* está pensado para su utilización por servidores que proporcionan información del tiempo en LANs y por los niveles más altos (estratos más bajos) de la subred de sincronización, donde se deben obtener las precisiones más altas. Un par de servidores operando en modo simétrico intercambian mensajes llevando información del tiempo. Los datos del tiempo son retenidos como parte de una asociación entre los servidores que se mantiene con el fin de mejorar la precisión de su sincronización en el tiempo.

En todos los modos, los mensajes se entregan de modo no fiable, utilizando el protocolo de transporte estándar Internet UDP. En el modo de llamada a procedimiento y en el simétrico, los procesos intercambian pares de mensajes. Cada mensaje lleva marcas de tiempo de los sucesos del mensaje reciente: los tiempos locales cuando el mensaje anterior NTP entre el par fue enviado y recibido y el tiempo local cuando el mensaje actual fue transmitido. El receptor del mensaje NTP anota el tiempo local cuando el recibe el mensaje. Los cuatro tiempos T_{i-3}, T_{i-2}, T_{i-1} y T_i se muestran en la Figura 10.4 para los mensajes m y m' enviados entre los servidores A y B. Observe que en el modo simétrico, a diferencia del algoritmo de Cristian descrito anteriormente, puede haber un retardo no despreciable entre la llegada de un mensaje y el envío del siguiente. También, se pueden perder mensajes, pero las tres marcas de tiempo llevadas por cada mensaje son sin embargo válidas.

Por cada par de mensajes enviados entre dos servidores NTP calcula una *compensación* o^i, que es una estimación de la deriva actual entre los dos relojes, y un *retardo* d_i, que es el tiempo total de transmisión para los dos mensajes. Si el verdadero desplazamiento del reloj en B con relación al de A es o, y si los tiempos de transmisión actuales para m y m' son t y t' relativamente, entonces tenemos:

$$T_{i-2} = T_{i-3} + t + o \quad \text{y} \quad T_i = T_{i-1} + t' + o$$

Esto conduce a:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

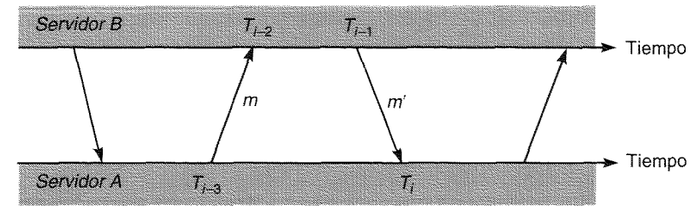


Figura 10.4. Mensajes intercambiados entre un par de iguales NTP.

También se verifica que:

$$o = o_i + (t' - t)/2, \quad \text{donde} \quad o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Usando el hecho de que t, $t' \geqslant 0$ se puede ver que $o_i - d_i/2 \leqslant o \leqslant o_i + d_i/2$. Por tanto o_i es una estimación de la deriva, y d_i es una medida de la precisión de esta estimación.

Los servidores NTP aplican un algoritmo de filtrado de datos a pares sucesivos de $<o_i, d_i>$, que estima la deriva o y calculan la calidad de esta estimación como una cantidad estadística llamada el *filtro de dispersión*. Un filtro de dispersión relativamente alto implica datos relativamente poco fiables. Los ocho pares $<o_i, d_i>$ más recientes son retenidos. Como con el algoritmo de Cristian, se elige como estimación de o el valor de o_j que corresponde con el mínimo valor d_i.

Sin embargo, el valor del desplazamiento derivado de la comunicación con una única fuente no se utiliza necesariamente en sí mismo para controlar el reloj local. En general un servidor NTP conecta en los intercambios de mensajes con varios de su iguales, o colegas. Además del filtrado de datos aplicados a los intercambios con cada igual, NTP aplica un algoritmo de selección de colega. Éste examina los valores obtenidos de los intercambios con cada uno de los distintos colegas, buscando los valores relativamente menos fiables. La salida de este algoritmo puede provocar que un servidor cambie el colega que utilizaba inicialmente para la sincronización.

Los colegas con número de estrato más bajo son menos favorecidos que aquéllos en el estrato más alto porque están más *próximos* a las fuentes de tiempo primarias. También, aquéllos con la *dispersión de sincronización* más baja son favorecidos relativamente. Ésta es la suma de las dispersiones del filtro medidas entre el servidor y la raíz de la subred de sincronización. (Los colegas intercambian la dispersión de sincronización en mensajes, permitiendo que se calcule este total.)

NTP emplea un modelo de bucle de bloqueo de fase (*phase lock*) [Mills 1995], que modifica la frecuencia de actualización del reloj de acuerdo con las observaciones de su tasa de deriva. Para considerar un ejemplo sencillo, si se descubre que un reloj siempre se adelanta a una tasa de, digamos, cuatro segundos por hora, entonces puede reducirse su frecuencia ligeramente (en software o hardware) para compensarlo. La deriva del reloj es reducida por tanto en los intervalos entre sincronización.

Mills acota precisiones de sincronización del orden de decenas de milisegundos sobre recorridos en Internet, y un milisegundo en LAN.

10.4. TIEMPO LÓGICO Y RELOJES LÓGICOS

Desde el punto de vista de un único proceso, los sucesos están ordenados de forma única por los tiempos mostrados en el reloj lógico. Sin embargo, como apuntó Lamport [1978], puesto que no podemos sincronizar perfectamente los relojes a lo largo de un sistema distribuido, no podemos usar, en general, el tiempo físico para obtener el orden de cualquier par arbitrario de sucesos que ocurran en él.

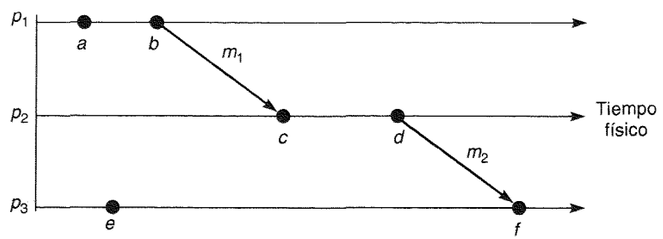


Figura 10.5. Sucesos ocurridos en tres procesos.

En general, podemos utilizar un esquema que es similar a la causalidad física, pero que se aplica en los sistemas distribuidos, para ordenar algunos de los sucesos que ocurren en diferentes procesos. Esta ordenación está basada en dos puntos sencillos e intuitivamente obvios:

Si dos sucesos han ocurrido en el mismo proceso $p_i (i = 1, 2, ..., N)$, entonces ocurrieron en el orden en el que les observa p_i, éste es el orden $\rightarrow_i$ que hemos definido anteriormente.

Cuando se envía un mensaje entre procesos, el suceso de enviar el mensaje ocurrió antes del de recepción del mismo.

Lamport llamó a la ordenación parcial obtenida al generalizar estas dos relaciones la realización *suceder antes*. Esto también se conoce a veces como la relación de *orden causal* o *ordenación causal potencial*.

Definimos la relación *suceder antes*, indicada por $\rightarrow$, como sigue:

SA1: Si $\exists$ un proceso p_i: $e \rightarrow_i e'$, entonces $e \rightarrow e'$.

SA2: Para cualquier mensaje m, *envía*$(m) \rightarrow$ *recibe*(m);
donde *envía*(m) es el suceso de enviar el mensaje, y *recibe*(m) el de recibirlo.

SA3: Si e, e' y e'' son sucesos tal que $e \rightarrow e'$ y $e' \rightarrow e''$, entonces $e \rightarrow e''$.

Por tanto si e y e' son sucesos, y si $e \rightarrow e'$, entonces podemos encontrar una serie de sucesos $e_1, e_2, ..., e_n$ ocurriendo en uno o más sucesos tal que $e = e_1$, y $e' = e_n$, y para $i = 1, 2, ... N - 1$ o bien se aplica SA1 o SA2 entre e_i y e_{i+1}. Esto es, o bien ocurren en sucesión en el mismo proceso, o hay mensaje m tal que e_1 = *envía*(m) y e_{i+1} = *recibe*(m). La secuencia de sucesos $e_1, e_2, ..., e_n$ no necesita ser única.

En la Figura 10.5 se ilustra la relación $\rightarrow$ para el caso de tres procesos p_1, p_2 y p_3. Se puede ver que $a \rightarrow b$, puesto que los sucesos ocurren en este orden en el proceso $p_1 (a \rightarrow_1 b)$ y de modo similar $c \rightarrow d$. Además $b \rightarrow c$, puesto que estos sucesos son el envío y recepción del mensaje m_1, y de modo semejante $d \rightarrow f$. Combinando estas relaciones, podemos decir también, por ejemplo, que $a \rightarrow f$.

De la Figura 10.5 se puede ver también que no todos los sucesos están relacionados con la relación $\rightarrow$. Por ejemplo, $a \not\rightarrow e$ y $e \not\rightarrow a$, puesto que ocurren en diferentes procesos, y no hay una cadena de mensajes que intervengan entre ellos. Decimos que sucesos como a y e que no están ordenados por $\rightarrow$ son concurrentes y se escribe $a \parallel e$.

La relación $\rightarrow$ captura un flujo de información entre dos eventos. Nótese, sin embargo, que en principio la información pueden fluir de formas distintas de la de paso de mensajes. Por ejemplo, si Pérez presenta un mandato a su proceso para que envíe un mensaje, acto seguido telefonea a Gómez, quien ordena a su proceso que envíe otro mensaje, luego el envío del primer mensaje claramente *sucedió antes* que el segundo. Desafortunadamente, como no se han enviado mensajes de red entre los procesos que los emitieron, no podemos modelar este tipo de relaciones en nuestro sistema.

Otro punto a señalar es que aun produciéndose la relación *sucedió antes* entre dos sucesos, el primero podría o no haber causado realmente el segundo. Por ejemplo, si un servidor recibe un mensaje de petición y consecuentemente envía una respuesta, entonces la transmisión de respuesta está causada por la transmisión de petición. Sin embargo, la relación $\rightarrow$ captura sólo la causalidad potencial, y dos sucesos pueden estar relacionados por $\rightarrow$ incluso aunque no haya una conexión real entre ellos. Un proceso podría, por ejemplo, recibir un mensaje y consecuentemente enviar otro mensaje, pero uno que él emite cada cinco minutos en cualquier caso y no tiene ninguna relación específica con el primer mensaje. No se ha supuesto ninguna causalidad real, pero la relación $\rightarrow$ debe ordenar estos sucesos.

◇ **Relojes lógicos.** Lamport inventó un mecanismo simple por el que la relación *sucedió antes* puede capturarse numéricamente, denominado *reloj lógico*. Un reloj lógico de Lamport es un contador software que se incrementa monótonamente, cuyos valores no necesitan tener ninguna relación particular con ningún reloj físico. Cada proceso p_i mantiene su propio reloj lógico, L_i, que él utiliza para aplicar las llamadas *marcas de tiempo de Lamport* a los sucesos. Representamos la marca de tiempo e del suceso en p_i por $L_i(e)$, y representamos por $L(e)$ la marca de tiempo del suceso e cualquiera que sea el proceso en el que ocurrió.

Para capturar la relación *sucedió antes* $\rightarrow$, los procesos actualizan sus relojes lógicos y transmiten los valores de sus relojes lógicos en mensajes como sigue:

RL1: L_i se incrementa antes de emitir cada suceso en el proceso p_i:
$L_i = L_i + 1$

RL2: (a) Cuando un proceso p_i envía un mensaje m, acarrea en m el valor de $t = L_i$.
(b) Al recibir (m, t), cada proceso p_i calcula $L_j := max(L_j, t)$ y entonces aplica RL1 antes de realizar la marca de tiempo del suceso *recibe*(m).

Aunque nosotros incrementamos los relojes en 1, podríamos haber elegido cualquier valor positivo. Se puede ver fácilmente, por inducción en la longitud de cualquier secuencia de sucesos relacionando dos sucesos e y e', que $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Hay que señalar que el inverso no es verdadero. Si $L(e) < L(e')$, no podemos inferir que $e \rightarrow e'$. En la Figura 10.6 ilustramos el uso de los relojes lógicos para el ejemplo dado en la Figura 10.5. Cada uno de los procesos p_1, p_2 y p_3 tienen su reloj lógico inicializado a 0. Los valores del reloj dados son aquéllos inmediatamente después del suceso para el que son adyacentes. Tenga en cuenta, también, $L(b) > L(e)$ pero $b \parallel e$.

◇ **Relojes lógicos totalmente ordenados.** Algunos pares de sucesos distintos, generados por diferentes procesos, tienen marcas de tiempo de Lamport numéricamente idénticas. Sin embargo, podemos crear un orden total sobre los sucesos, esto es, uno para el que todos los pares de sucesos distintos están ordenados, teniendo en cuenta los identificadores de los procesos en los que

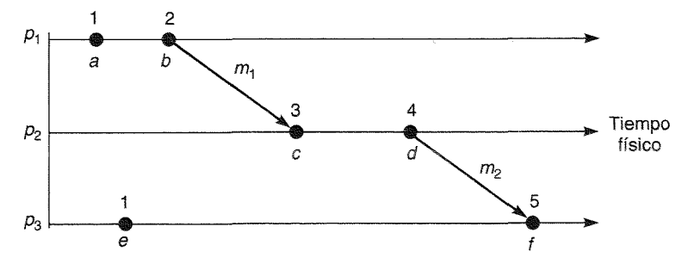


Figura 10.6. Marcas temporales de Lamport para los eventos mostrados en la Figura 10.5.

ocurren los sucesos. Si e es un suceso que ocurre en p_i con marca de tiempo local T_i, y e' es un suceso que ocurre en p_j con marca de tiempo local T_j, definimos las marcas de tiempo globales para esos sucesos como (T_i, i) y (T_j, j) respectivamente. Y definimos $(T_i, i) \leqslant (T_j, j)$ si y sólo si $T_i < T_j$, o $T_i = T_j$, e $i < j$. Esta ordenación no tiene significado físico general (porque los identificadores de los procesos son arbitrarios), pero a veces es útil. Lamport la utilizó, por ejemplo, para ordenar la entrada de procesos en una sección crítica.

◇ **Relojes vectoriales.** Mattern [1989] y Fidge [1991] desarrollaron relojes vectoriales para vencer la deficiencia de los relojes de Lamport: del hecho que $L(e) < L(e')$ no podemos deducir que $e \rightarrow e'$. Un reloj vectorial para un sistema de N procesos es un vector de N enteros. Cada proceso mantiene su propio reloj vectorial V_i, que utiliza para colocar marcas de tiempo en los sucesos locales. Como las marcas de tiempo de Lamport, cada proceso adhiere el vector de marcas de tiempo en los mensajes que envía al resto, y hay unas reglas sencillas para actualizar los relojes, como las siguientes:

RV1: Inicialmente, $V_i[j] = 0$, para $i, j = 1, 2..., N$.

RV2: Justo antes que p_i coloque una marca de tiempo en un suceso, coloca $V_i[i] := V_i[i] + 1$.

RV3: p_i incluye el valor $t = V_i$ en cada mensaje que envía.

RV4: Cuando p_i recibe una marca de tiempo y en un mensaje, establece $V_i[j] := max(V_i[j], t[j])$, para $j = 1, 2 ..., N$. Esta operación de *mezcla* toma el vector máximo entre dos, componente a componente.

Para un vector V_i, $V_i[i]$ es el número de sucesos a los que p_i ha puesto una marca de tiempo, y $V_i[j] (j \neq i)$ es el número de sucesos que han ocurrido en p_j que han sido potencialmente afectados por p_i. (El proceso p_j puede haber puesto marcas de tiempo en más sucesos por este método, pero la información no ha fluido hacia p_i en mensajes todavía.)

Podemos comparar vectores de marcas de tiempo de la forma siguiente:

$V = V'$ si y sólo si $V[j] = V'[j]$ para $j = 1, 2..., N$

$V \leqslant V'$ si y sólo si $V[j] \leqslant V'[j]$ para $j = 1, 2..., N$

$V < V'$ si y sólo si $V \leqslant V' \wedge V \neq V'$

Sea $V(e)$ el vector de marcas de tiempo aplicadas por el proceso en el que ocurre e. Es sencillo mostrar, por inducción sobre la longitud de cualquier secuencia de sucesos que relacione los sucesos e y e', que $e \rightarrow e' \Rightarrow V(e) < V(e')$. El Ejercicio 10.13 solicita que el lector muestre el recíproco: Si $V(e) < V(e')$, entonces $e \rightarrow e'$.

La Figura 10.7 muestra los vectores de marcas de tiempo de los eventos de la Figura 10.5. Se puede ver, por ejemplo, que $V(a) < V(f)$, que refleja el hecho que $a \rightarrow f$. De forma semejante, podemos decir si dos sucesos son concurrentes comparando sus marcas de tiempo. Por ejemplo, que $c \parallel e$ pueda ser deducido del hecho que ni $V(c) \leqslant V(e)$ ni $V(e) \leqslant V(c)$.

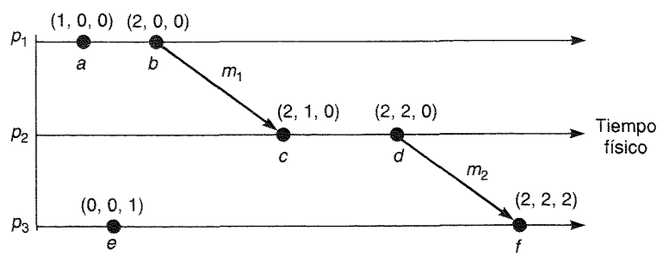


Figura 10.7. Vector de marcas temporales para los eventos mostrados en la Figura 10.5.

Los vectores de marcas de tiempo tienen la desventaja, comparados con las marcas de tiempo de Lamport, de precisar una cantidad de almacenamiento y de carga real de mensajes que es proporcional a N, el número de procesos. Charron-Bost [1991] mostró que, si somos capaces de decir si dos sucesos son o no concurrentes inspeccionando sus marcas de tiempo, entonces la dimensión N es inevitable. Sin embargo, existen técnicas para almacenar y transmitir cantidades más pequeñas de datos, a costa del procesamiento precisado para reconstruir los vectores completos. Raynal y Singhal [1996] dan cuenta de alguna de estas técnicas. También describen la noción de *relojes matriciales*, en la que los procesos mantienen estimaciones de los vectores de tiempo de otros procesos así como las suyas propias.

10.5. ESTADOS GLOBALES

En esta sección y en la siguiente examinaremos el problema de descubrir si una propiedad particular de un sistema distribuido es cierta cuando éste se ejecuta. Comenzamos dando ejemplos de compactación de memoria distribuida, detección de bloqueos indefinidos, detección de terminación y depuración.

Compactación automática de memoria: un objeto se considera desechable si no hay posteriormente ninguna referencia a él desde cualquier parte del sistema distribuido. La memoria ocupada por el objeto puede ser reclamada, una vez que se sabe que éste es desechable. Para comprobar que un objeto es desechable, debemos verificar que no hay referencias a él en cualquier parte del sistema. En la Figura 10.8a, el proceso p_1 tiene dos objetos, ambos con referencias, uno tiene una referencia al propio p_1, y el otro tiene una referencia a p_2. El proceso p_2 tiene un objeto desechable, al que no existe ninguna referencia en el sistema. Hay también otro objeto al que ni p_1 ni p_2 tienen acceso, pero hay una referencia a él en un mensaje que transita entre ambos. Esto nos muestra que cuando consideramos las propiedades de un sistema, debemos incluir el estado de los canales de comunicación así como el de los procesos.

Detección distribuida de bloqueos indefinidos: un bloqueo indefinido distribuido ocurre cuando cada uno de los procesos de una colección espera por otro proceso para enviarle un mensaje, y donde hay un ciclo en el grafo de esta relación *espera por*. La Figura 10.8b muestra que cada uno de los procesos p_1 y p_2 espera un mensaje del otro, por lo que el sistema nunca progresará.

Detección de la terminación distribuida: el problema aquí es detectar que un algoritmo distribuido ha terminado. La detección de la terminación es un problema que parece muy fácil de resolver: al principio parece sólo necesario comprobar si cada proceso se ha detenido. Para ver que esto no es así, consideremos un algoritmo distribuido ejecutado por dos procesos p_1 y p_2, cada uno de los cuales puede necesitar valores del otro. Instantáneamente, podemos encontrar que un proceso es activo o pasivo, un proceso pasivo no está ligado a ninguna actividad por sí mismo pero está preparado para responder con un valor solicitado por el otro. Supongamos que descubrimos que p_1 es pasivo y que p_2 también lo es (véase la Figura 10.8c). Para ver que no podemos determinar que el algoritmo ha terminado, consideremos el siguiente escenario: cuando hemos comprobado p_1 para ver si es pasivo, un mensaje estaba en su camino desde p_2, que se convirtió en pasivo inmediatamente después de enviarlo. Cuando reciba el mensaje, p_1 se convertirá de nuevo en activo, después de encontrar que era pasivo. El algoritmo no había terminado.

Los fenómenos de terminación y bloqueo indefinido son de alguna forma semejantes, pero son problemas diferentes. En primer lugar, un bloqueo indefinido puede afectar sólo a un subconjunto de procesos en el sistema, mientras que todos los procesos tienen que haber terminado. En segundo lugar, la pasividad de un proceso no es lo mismo que la espera en un ciclo de bloqueo indefinido: un proceso bloqueado está intentando realizar otra acción, por lo que espera a otro proceso; un proceso pasivo no está ligado con ninguna actividad.

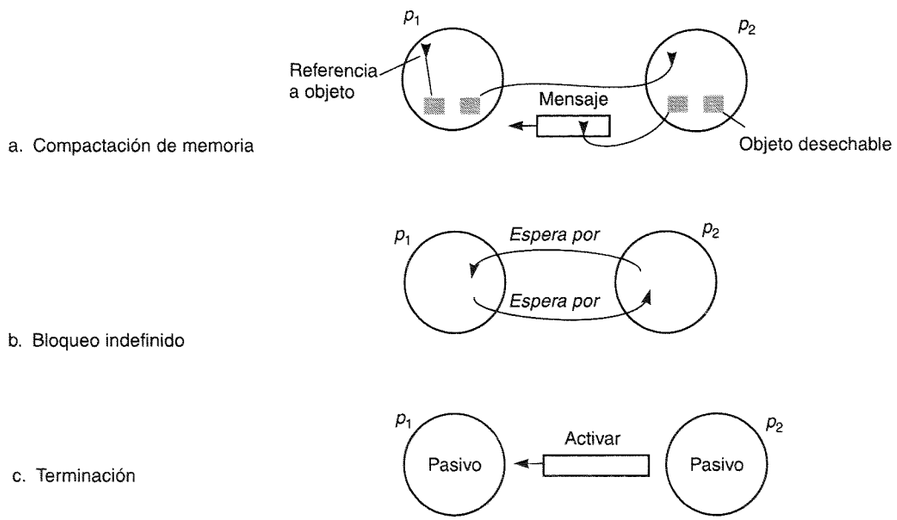


Figura 10.8. Detección de propiedades globales.

Depuración distribuida: Los sistemas distribuidos son complejos de depurar [Bonnaire y otros, 1995], y es preciso tener cuidado al establecer qué es lo que ocurre durante la ejecución. Por ejemplo, Pérez ha escrito una aplicación en la que cada proceso p_i contiene una variable $x_i(i = 1, 2..., N)$. Las variables cambian a medida que se ejecuta el programa, pero se precisa que estén a menos de cierto valor δ de otras. Desafortunadamente hay un error en el programa, y ella sospecha que bajos ciertas circunstancias $|x_i - x_j| > \delta$ para algún i y j, incumpliendo sus restricciones de consistencia. Su problema es que hay que evaluar esta relación para valores de las variables que ocurran al mismo tiempo.

Cada uno de los problemas señalados anteriormente tiene soluciones específicas ligadas a él; pero todos ilustran la necesidad de observar un estado global, y eso motiva una aproximación general.

10.5.1. ESTADOS GLOBALES Y CORTES CONSISTENTES

En principio, es posible observar la sucesión de estados de un proceso individual, pero la cuestión de cómo establecer un estado global del sistema, el estado de la colección de procesos, es más difícil de tratar.

El problema esencial es la ausencia de tiempo global. Si todos los procesos tuvieran los relojes perfectamente sincronizados ellos podrían estar de acuerdo en un tiempo en el que cada proceso debía registrar su estado, el resultado sería un estado global actual del sistema. De una colección de estados de proceso deberíamos decir, por ejemplo, qué procesos estaban bloqueados indefinidamente. Pero no podemos obtener una sincronización perfecta de los relojes, por lo que este método no está disponible.

Debiéramos, por tanto, preguntarnos si podemos elaborar un estado global significativo de los estados locales registrados en diferentes tiempos reales. La respuesta es un *sí* matizado, pero para ver esto debemos presentar antes algunas definiciones.

Retornemos a nuestro sistema general $\wp$ de N procesos $p_i(i = 1, 2..., N)$, cuya ejecución deseamos estudiar. Hemos dicho anteriormente que ocurre una serie de eventos en cada proceso, y que podemos caracterizar la ejecución de un proceso por su historia:

$$historia(p_i) = h_i = <e_i^0, e_i^1, e_i^2, ...>$$

De forma semejante, podemos considerar cualquier prefijo finito de la historia del proceso:

$$h_i^k = <e_i^0, e_i^1, ... e_i^k>$$

Cada evento es una acción interna del proceso (por ejemplo, la actualización de una de sus variables) o el envío o la recepción de un mensaje sobre los canales de comunicación que conectan los procesos.

En principio, podemos registrar qué ocurrió en la ejecución de $\wp$. Cada proceso puede registrar los sucesos que se producen allí, y la sucesión de estados por la que pasa. Denotamos con s_i^k el estado del proceso p_i inmediatamente antes que ocurra el suceso k, por lo que e_i^0 es el estado inicial de p_i. Hemos tenido en cuenta en los ejemplos anteriores que el estado de los canales de comunicación es relevante a veces. Más que introducir un nuevo tipo de estado, hacemos que el proceso registre el envío o la recepción de todos los mensajes como una parte de su estado. Si encontramos que el proceso p_i ha registrado que envió un mensaje m al proceso $p_j(i \neq j)$, entonces examinando si p_j ha recibido el mensaje podemos inferir o no que m es parte del estado del canal entre p_i y p_j.

Podemos formar la historia global de $\wp$ como la unión de las historias individuales de los procesos:

$$H = h_0 \cup h_1 \cup \cdots \cup h_{N-1}$$

Matemáticamente, podemos tomar cualquier conjunto de estados de los procesos individuales para formar un estado global $(s_1, s_2, ... s_N)$. Pero qué estados son significativos; esto es, ¿cuál de los estados de los procesos podrían haber ocurrido al mismo tiempo? Un estado global corresponde a los prefijos iniciales de las historias individuales de los procesos. Un *corte* de la ejecución del sistema es un subconjunto de su historia global que es la unión de los prefijos de las historias de los procesos:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_N^{c_N}$$

El estado s_i en el estado global S correspondiente al corte C es el de p_i inmediatamente después del último suceso procesado por p_i en el corte, $e_i^{c_i}(i = 1, 2, ..., N)$. El conjunto de sucesos $\{e_i^{c_i}: i = 1, 2, ..., N\}$ se llama la *frontera* del corte.

Consideremos los sucesos que ocurren en los procesos p_1 y p_2 mostrados en la Figura 10.9. La figura representa dos cortes, uno con frontera $<e_1^0, e_2^0>$ y otro con frontera $<e_1^2, e_2^2>$. El corte de más a la izquierda es *inconsistente*. Esto es porque p_2 incluye la recepción del mensaje m_1, pero en p_1 no incluye el envío del mensaje. Esto se considera como un *efecto* sin una *causa*. La ejecución actual no estuvo nunca en un estado global correspondiente a los estados del proceso en la frontera, y podemos decir esto, en principio, examinando la relación $\rightarrow$ entre los sucesos. Por el contrario, el corte de la derecha es *consistente*. Incluye tanto el envío como la recepción del mensaje m_1. Incluye el envío pero no la recepción del mensaje m_2. Esto es consistente con la ejecución actual; después de todo, el mensaje se tomó algún tiempo para llegar.

Un corte C es consistente si, para cada suceso que contiene, también contiene todos los sucesos que *sucedieron antes* del suceso:

$$\text{Para todos los sucesos } e \in C,\ f \rightarrow e \quad \Rightarrow \quad f \in C$$

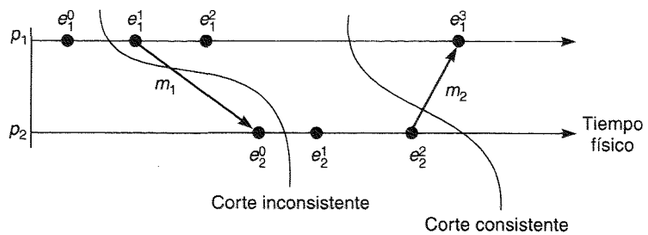


Figura 10.9. Cortes.

Un *estado global consistente* es uno que corresponde con un corte consistente. Podemos caracterizar la ejecución de un sistema distribuido como una serie de transiciones entre los estados globales del sistema:

$$S_0 \rightarrow S_1 \rightarrow S_2 \ldots$$

En cada transición, ocurre exactamente un evento en algún proceso simple del sistema. Este suceso es el envío de un mensaje, la recepción de un mensaje o un suceso interno. Si dos sucesos ocurrieron simultáneamente, podemos considerar no obstante que han ocurrido en un orden definido: es decir ordenados de acuerdo con los identificadores de los procesos. (Los sucesos que ocurren simultáneamente deben ser concurrentes: ninguno *sucedió antes* que el otro.) De esta forma un sistema evoluciona a través de estados consistentes.

Una *ejecución* es una ordenación de orden total de todos los sucesos en una historia global que es consistente con cada ordenación de la historia local, $\rightarrow_i (i = 1, 2, ..., N)$. Una *linealización o ejecución consistente* es una ordenación de los sucesos en una historia global que es consistente con su relación *sucedió antes* $\rightarrow$ en H. Observe que una linealización es también una ejecución.

No todas las ejecuciones pasan a través de estados consistentes globales, pero todas las linealizaciones pasan sólo a través de estados globales consistentes. Decimos que un estado S' es *alcanzable* desde un estado S si existe una linealización que pasa a través de S y después de S'.

A veces podemos alterar la ordenación de los sucesos concurrentes con una linealización, y derivar una ejecución que aún pasa sólo a través de estados globales consistente. Por ejemplo, si dos eventos sucesivos en una linealización son la recepción de mensajes por dos procesos, entonces podemos conmutar el orden de estos dos sucesos.

10.5.2. PREDICADOS DE ESTADO GLOBAL, ESTABILIDAD, SEGURIDAD Y VITALIDAD

Detectar una condición tal como un bloqueo indefinido o una terminación equivale a evaluar un *predicado de estado global*. Un predicado de estado global es una función que pasa del conjunto de estados globales de los procesos en el sistema $\wp$ a *Verdadero*, *Falso*. Una de las características útiles de los predicados asociados con el estado de un objeto que desechable, un sistema que está siendo bloqueado indefinidamente o un sistema que está terminando es que todos son *estables*: una vez que el sistema entra en un estado en el que el predicado es *Verdadero*, permanece *Verdadero* en todos los estados futuros alcanzables desde ese estado. Por contraste, cuando monitorizamos o depuramos una aplicación a menudo estamos interesados en predicados no estables, como lo que ocurre en nuestro ejemplo de las variables cuya diferencia se supone está limitada. Incluso si la aplicación alcanza un estado en el que se obtiene el límite, no se necesita permanecer en ese estado.

También anotamos aquí otras dos nociones relevantes para los predicados de estado global: seguridad y vitalidad. Supongamos que hay una propiedad indeseable α que es un predicado del estado global del sistema: por ejemplo, α podría ser la propiedad de estar bloqueado indefinidamente. Sea S_0 el estado original del sistema. *Seguridad* con respecto a α es la aserción que α evalúa a *Falso* para todos los estados S alcanzables desde S_0. A la inversa, sea β una propiedad deseable de un estado global del sistema: por ejemplo, la propiedad de alcanzar la terminación. *Vitalidad* con respecto a β es la propiedad que, para cualquier linealización L comenzando en el estado S_0, β se evaluará a *Verdadero* para algún estado S_l alcanzable desde S_0.

10.5.3. EL ALGORITMO DE *INSTANTÁNEA* DE CHANDY Y LAMPORT

Chandy y Lamport [1985] describen un algoritmo de *instantánea* para determinar estados globales de sistemas distribuidos, que presentamos ahora. El objetivo del algoritmo es registrar un conjunto de estados de procesos y canales (una *instantánea*) para un conjunto de procesos $p_i (i = 1, 2, ..., N)$ tal que, incluso a través de una combinación de estados registrados que nunca podrían haber ocurrido al mismo tiempo, el estado global registrado sea consistente.

Veremos que el estado que registra el algoritmo de *instantánea* tiene propiedades convenientes para evaluar predicados del estado global.

El algoritmo registra el estado localmente en los procesos; no proporciona un método para recoger el estado global en un sitio. Un método obvio para recoger el estado es que todos los procesos envíen el estado que ellos han registrado a un proceso recolector designado, pero no tratamos aquí más ese tema.

El algoritmo supone que:

- No fallan ni los canales ni los procesos; la comunicación es fiable por lo que cada mensaje enviado es recibido intacto, exactamente una vez.
- Los canales son unidireccionales y proporcionan la entrega de los mensajes con ordenación FIFO.
- El grafo de los procesos y canales está fuertemente conectado (hay un recorrido entre dos procesos cualquiera).
- Cualquier proceso puede iniciar una instantánea global en cualquier instante.
- Los procesos pueden continuar su ejecución y enviar y recibir mensajes normales mientras tiene lugar la instantánea.

Para cada proceso p_i, se considera que los canales entrantes son aquéllos por medio de los cuales otros procesos envían mensajes a p_i; del mismo modo los canales salientes de p_i son aquellos sobre los que él envía mensajes a otros procesos. La idea esencial del algoritmo es como sigue. Cada proceso registra su estado y para cada canal entrante también un conjunto de mensajes enviados a él. El proceso registra, para cada canal, todos los mensajes que entraron después de que él registrara el estado y antes que el emisor registrara su propio estado. Este planteamiento nos permite registrar los estados de los procesos en momentos diferentes menos para dar cuenta de las diferencias entre los estados del proceso en términos de mensajes transmitidos pero todavía no recibidos. Si el proceso p_i ha enviado un mensaje m al proceso p_j; pero p_j no lo ha recibido todavía, entonces consideramos a m como perteneciente al estado del canal entre ellos.

El algoritmo procede mediante el uso de mensajes *marcador* especiales, que son distintos de cualquiera de los otros mensajes que envían los procesos, y que los procesos podrán enviar y recibir mientras continúan con su ejecución normal. El marcador tiene un papel dual: como un aviso para que el receptor guarde su propio estado, si no lo ha hecho aún; y como un medio para determinar qué mensajes incluir en el estado del canal.

Regla de recepción del marcador para el proceso p_i
Cuando p_i recibe un mensaje *marcador* sobre el canal *c*:
si (p_i no ha registrado todavía su estado)
registra su estado de proceso ahora;
registra el estado de *c* como el conjunto vacío;
activa el registro de los mensajes que llegan sobre otros canales entrantes;
sino
p_i registra el estado de *c* como el conjunto de mensajes que ha recibido sobre *c*
desde que guardó su estado
fin si

Regla de envío del marcador para el proceso p_i
Después p_i ha registrado su estado para cada canal de salida *c*:
p_i envía un mensaje marcador sobre *c*
(antes que envíe otro mensaje sobre *c*).

Figura 10.10. Algoritmo de *instantánea* de Chandy y Lamport.

El algoritmo se define mediante de dos reglas, la *regla de recepción del marcador* y *la regla de envío del marcador* (véase la Figura 10.10). La regla de envío del marcador obliga a los procesos a enviar un marcador después de haber registrado su estado, pero antes de que envíen cualquier otro mensaje.

La regla de recepción del marcador obliga a un proceso que no ha registrado su estado a hacerlo. En ese caso, éste es el primer marcador que ha recibido. Él observará qué mensajes llegan posteriormente en los otros canales entrantes. Cuando un proceso que ya ha guardado su estado recibe un marcador (en otro canal), registra el estado de ese canal bajo la forma del conjunto de mensajes que recibió desde que guardó su estado.

Cualquier proceso puede comenzar el algoritmo en cualquier instante. Actúa como si hubiera recibido un marcador (sobre un canal no existente) y sigue la regla de recepción del marcador. Por tanto registra su estado y comienza a registrar los mensajes que llegan sobre todos los canales entrantes. Varios procesos pueden comenzar registrando concurrentemente de esta forma (en tanto en cuanto los marcadores que utilizan puedan ser distinguidos).

Ilustramos el algoritmo para un sistema con dos procesos p_1 y p_2 conectados por dos canales unidireccionales c_1 y c_2. Los dos procesos comercian con *cosas*. El proceso p_1 envía órdenes de compra de cosas sobre c_2 hacia p_2, incluyendo el pago al precio de 10 dólares por cosa. Algún tiempo después, el proceso p_2 envía cosas a través del canal c_1 hacia p_1. Los procesos están en el estado inicial que se ve en la Figura 10.11. El proceso p_2 ha recibido ya una orden por cinco cosas, que serán enviadas en breve por p_1.

La Figura 10.12 muestra una ejecución del sistema mientras se registra el estado. El proceso p_1 registra su estado en el estado global S_0, cuando el estado de p_1 es $<1000\$, 0>$. Siguiendo la regla de envío del marcado, el proceso p_1 emite entonces un mensaje marcador sobre su canal saliente c_2 antes de que él envíe el siguiente mensaje a nivel de aplicación: (pedir 10, 100$) sobre el canal c_2. El sistema entra en el estado global actual S_1.

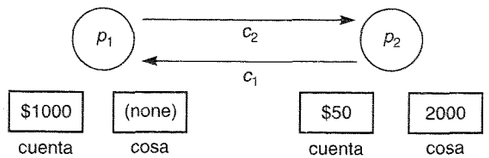


Figura 10.11. Dos procesos y sus estados iniciales.

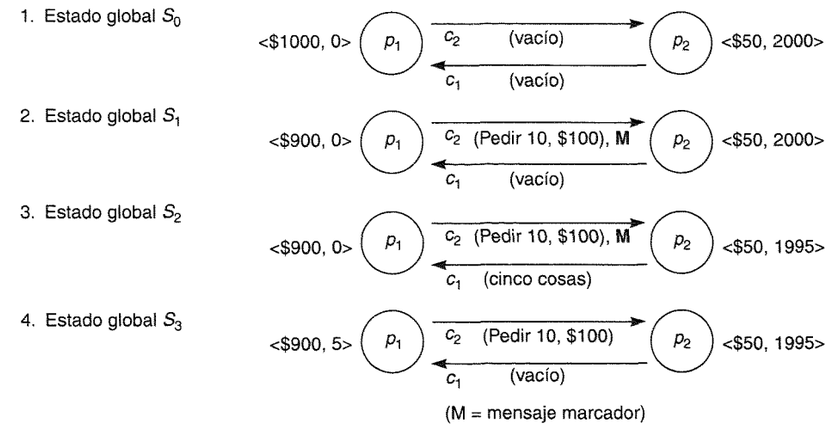


Figura 10.12. La ejecución de los procesos de la Figura 10.11.

Antes de que p_2 reciba el marcador, emite un mensaje de aplicación (cinco cosas) sobre c_1 como respuesta a la orden previa de p_1, produciendo un nuevo estado actual S_2.

Ahora el proceso p_1 recibe el mensaje de p_2 (cinco cosas), y p_2 recibe el marcador. Siguiendo la regla de recepción del marcador, p_2 registra su estado como $<50\$, 1995>$ y el del canal c_2 como la secuencia vacía. Siguiendo la regla de envío del marcador, envía un mensaje marcador sobre c_1.

Cuando el proceso p_1 recibe el mensaje marcador de p_2, registra el estado del canal c_1 como un único mensaje (cinco cosas) que recibe después que ha registrado su estado. El estado global final actual es S_3.

El estado final registrado es p_1: $<1000\$, 0>$; p_2: $<50\$, 1995>$; c_1: <(cinco cosas)>; c_2: $<>$. Hay que considerar que el estado difiere de todos los estados globales a través de los que ha pasado realmente el sistema.

◇ **Terminación del algoritmo de instantánea.** Suponemos que un proceso que ha recibido un mensaje marcador registra su estado en un tiempo finito y envía mensajes marcador sobre cada canal saliente en un tiempo finito (incluso cuando no se necesite enviar más mensajes sobre esos canales). Si hay un recorrido de canales de comunicación y procesos desde un proceso p_i hasta un proceso $p_j (j \neq i)$, entonces está claro de estas suposiciones que p_j registrará su estado un tiempo finito después que p_i haya registrado el suyo. Puesto que estamos suponiendo que el grafo de procesos y canales está conectado fuertemente, se deduce que todos los procesos habrán registrado sus estados y los de los canales entrantes un tiempo finito después que algunos procesos hayan registrado inicialmente su estado.

◇ **Características del estado observado.** El algoritmo de instantánea selecciona un corte de la historia de la ejecución. El corte, y por tanto el estado registrado por este algoritmo, es consistente. Para ver esto, sean e_i y e_j sucesos que ocurren en p_i y p_j, respectivamente, tal que $e_i \rightarrow e_j$. Afirmamos que si e_j está en el corte entonces e_i también lo está. Esto es, si e_j *sucedió antes* que p_j registrara su estado, entonces e_i debe haber *sucedido antes* que p_i registrara el suyo. Esto es obvio si los dos procesos son el mismo, por lo que deberíamos haber supuesto que $j \neq i$. Supongamos, por el momento, lo opuesto de lo que deseamos probar: que p_i registró su estado antes que ocurriera e_i. Consideremos la secuencia H de mensajes $m_1, m_2 \ldots, m_H (H \geqslant 1)$, produciendo la relación

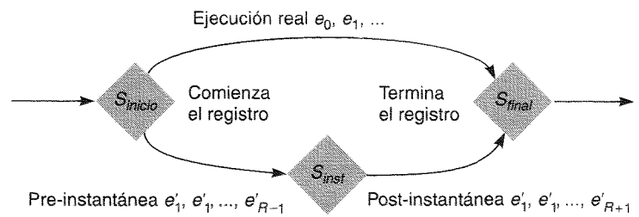


Figura 10.13. Alcanzabilidad entre estado en el algoritmo de «instantánea».

$e_i \rightarrow e_j$. Mediante una ordenación FIFO sobre los canales que atraviesan estos mensajes, y por las reglas de envío y recepción del marcador, un mensaje marcador debiera haber alcanzado p_j por delante de m_1, m_2 ..., m_H. Por la regla de recepción del marcador, p_j debería haber registrado su estado antes que el suceso e_j. Esto contradice nuestra suposición que e_j está en el corte, y hemos acabado.

Podríamos establecer además una relación de alcanzabilidad entre el estado global observado y los estados inicial y final cuando se ejecuta el algoritmo. Sea $Sis = e_0, e_1, \ldots$ una linealización del sistema tal y como se ejecuta (donde a dos sucesos ocurridos exactamente al mismo tiempo, los ordenamos de acuerdo con sus identificadores de proceso). Sea S_{inicio} el estado global inmediatamente antes que el primer proceso registrara su estado; sea S_{final} el estado global cuando termina el algoritmo de *instantánea*, inmediatamente después de la última acción de registro de estado; y sea S_{inst} el estado global registrado.

Encontraremos una permutación de Sis, $Sys' = e'_0, e'_1, e'_2, \ldots$ tal que los tres estados S_{inicio}, S_{inst} y S_{final} ocurren en Sis', S_{inst} es alcanzable desde S_{inicio} en Sis', y es alcanzable desde S_{ins} en Sis'. La Figura 10.13 muestra esta situación, en la que la linealización superior es Sis, y la inferior Sis'.

Deducimos Sis' de Sis categorizando inicialmente todos los sucesos en Sis como sucesos *pre-instantánea* o *post-instantánea*. Un suceso pre-instantánea en el proceso p_i es uno que *sucedió en* p_i *antes* de que registrara su estado; todos los demás sucesos son post-instantánea. Es importante comprender que un suceso post-instantánea puede suceder antes que uno pre-instantánea en Sis, si los sucesos ocurren en diferentes procesos. (Naturalmente ningún suceso post-instantánea puede *suceder antes* que otro pre-instantánea en el mismo proceso.)

Mostraremos cómo podemos ordenar todos los sucesos pre-instantánea antes que los post-instantánea para obtener Sis'. Supongamos que e_j es un suceso post-instantánea de otro proceso, y e_{j+1} es un suceso pre-instantánea de un proceso diferente. No puede ser que $e_j \rightarrow e_{j+1}$. Para ello estos dos sucesos deberían ser el envío y recepción de un mensaje, respectivamente. Un mensaje marcador debería tener que haber precedido al mensaje, haciendo de la recepción del mensaje un suceso post-instantánea, pero por suposición e_{j+1} es un suceso pre-instantánea. Podríamos, por tanto, conmutar ambos sucesos sin violar la relación *sucedió antes* (esto es, la secuencia resultante de los sucesos mantiene una linealización). La conmutación no introduce nuevos estados de procesos, puesto que no hemos alterado el orden en el que ocurren los sucesos en cualquier proceso individual.

Continuamos conmutando tantos pares de sucesos adyacentes de esta forma como sea necesario hasta que hemos ordenado todos los sucesos pre-instantánea $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$ anteriores a todos los sucesos post-instantánea $e'_R, e'_{R+1}, e'_{R+2}, \ldots$ de la ejecución resultante Sis'. Para cada proceso el conjunto de sucesos en $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$ que ocurrieron en él es exactamente el conjunto de sucesos que experimentó antes de que registrara su estado. Por tanto el estado de cada proceso en ese punto, y el estado de los canales de comunicación, es el del estado global S_{inst} registrado por el algoritmo. No hemos desordenado ninguno de los estados S_{inicio} o S_{final} con los que la linealización comienza y termina. Por lo tanto hemos establecido una relación de alcanzabilidad.

◇ **Estabilidad y alcanzabilidad del estado observado.** La propiedad de alcanzabilidad del algoritmo de instantánea se utiliza para detectar predicados estables. En general, cualquier predicado no estable que nosotros establezcamos como *Verdadero* en el estado S_{inst} puede o no haber sido *Verdadero* en la ejecución actual cuyo estado global hemos registrado. Sin embargo, si un predicado estable es *Verdadero* en el estado S_{inst} entonces podemos concluir que el predicado es *Verdadero* en el estado S_{final}, puesto que por definición un predicado estable que es *Verdadero* en un estado S lo es también en cualquier otro estado alcanzable de S. De forma similar, si el predicado se evalúa a *Falso* para S_{inst}, entonces debe ser también *Falso* para S_{inicio}.

10.6. DEPURACIÓN DISTRIBUIDA

Examinamos ahora el problema del registro de un estado global de un sistema de forma que podremos hacer afirmaciones útiles sobre si un estado transitorio, como opuesto a uno estable, ocurrió en la actual ejecución. Esto es lo que precisamos, en general, cuando depuramos un sistema distribuido. Ya dimos un ejemplo en el que cada conjunto de procesos p_i tiene una variable x_i. La condición de seguridad requerida en este ejemplo es $|x_i - x_j| \leqslant \delta$ $(i, j = 1, 2, \ldots, N)$; esta restricción será satisfecha incluso aunque un proceso pueda cambiar el valor de su variable en cualquier instante. Otro ejemplo es un sistema distribuido controlando un sistema de tuberías en una fábrica donde estamos interesados en si todos las válvulas (controladas por diferentes procesos) fueron abiertas al mismo tiempo. En estos ejemplos, no podemos observar, en general, los valores de las variables o los estados de las válvulas simultáneamente. El reto es monitorizar la ejecución del sistema a lo largo del tiempo, para capturar información de la *traza* mas que una simple instantánea, por lo que podemos establecer *a posteriori* si la condición de seguridad requerida fue violada o pudo haberlo sido.

El algoritmo de instantánea de Chandy y Lamport recoge el estado de una forma distribuida y hemos indicado ya cómo los procesos del sistema podrían enviar el estado que ellos conforman a un proceso monitor para su recogida. El algoritmo que describiremos (debido a Marzullo y Neiger [1991]) es centralizado. Los procesos observados envían sus estados a un proceso llamado monitor, que ensambla estados globalmente consistentes de los que recibe. Consideramos que el monitor se encuentra fuera del sistema observando su ejecución.

Nuestro objetivo es determinar casos en los que dado un predicado de estado global ϕ era sin duda alguna *Verdadero* en algún punto de la ejecución que observamos, y los casos en los que posiblemente era *Verdadero*. La noción *posiblemente* surge como un concepto natural porque podemos extraer un estado global consistente S de un sistema en ejecución y encontrar que $\phi(S)$ es *Verdadero*. Una observación única de un estado global consistente no nos permite concluir si un predicado no estable será evaluado siempre a *Verdadero* en la ejecución actual.

La noción *sin duda alguna* se aplica a la ejecución actual y no a una ejecución que hayamos extrapolado de ella. Nos puede parecer paradójico considerar qué sucedió en la ejecución actual. Sin embargo, es posible evaluar si ϕ fue sin duda alguna *Verdadero* considerando todas las linealizaciones de los sucesos observados.

Definimos ahora las nociones de *posiblemente* ϕ y *sin duda alguna* ϕ para un predicado ϕ en función de las linealizaciones de H, la historia de la ejecución del sistema.

posiblemente ϕ — La afirmación *posiblemente* ϕ significa que hay un estado consistente S a través del cual pasa una linealización H tal que $\phi(S)$ es *Verdadero*.

sin duda alguna ϕ — La afirmación *sin duda alguna* ϕ significa que para todas las linealizaciones L de H, hay un estado consistente S a través del cual pasa L tal que $\phi(S)$ es *Verdadero*.

Cuando utilizamos el algoritmo de instantánea de Chandy y Lamport y obtenemos el estado global S_{inst} podemos afirmar *posiblemente* ϕ si $\phi(S_{inst})$ si ocurre que es *Verdadero*. Pero, en general, evaluar *posiblemente* ϕ implica una búsqueda a través de todos los estados globales consistentes derivados de la ejecución observada. Únicamente si $\phi(S)$ se evalúa a *Falso* para todos los estados globales consistentes S no es el caso de *posiblemente* ϕ. Hay que tener en cuenta que mientras que podemos concluir *sin duda alguna* ($\neg\phi$) de $\neg$ *posiblemente* ϕ, no podemos concluir $\neg$*posiblemente* ϕ de *sin duda alguna* ($\neg\phi$). La última es la afirmación que $\neg\phi$ se mantiene en algún estado en cada linealización: puede mantenerse en otros estados.

Ahora describimos:

- Cómo se recogen los estados del proceso.
- Cómo los monitores extraen estados consistentes globales.
- Cómo el monitor evalúa *posiblemente* ϕ y *sin duda alguna* $\neg\phi$ tanto en sistemas síncronos como asíncronos.

◇ **Recolectando el estado.** Los procesos observados p_i ($i = 1, 2, ..., N$) envían su estado inicial al proceso monitor inicialmente, y después de tiempo en tiempo, en *mensajes de estado*. El proceso monitor registra los mensajes de estado de los procesos p_i en una cola separada Q_i, para cada $i = 1, 2, ..., N$.

La actividad de preparar y enviar mensajes de estado puede retrasar la ejecución normal de los procesos observados, pero aparte de eso no interfiere con ellos. No hay necesidad de enviar el estado inicialmente o cuando cambia. Hay dos optimizaciones para reducir el tráfico de mensajes de estado hacia el monitor. Primero, el predicado de estado global puede depender solo de ciertas partes de los estados de los procesos. Por ejemplo, puede depender solo de los estados de variables particulares. Por lo que los procesos observados solo necesitan enviar el estado relevante al proceso monitor. Segundo, solo necesitan enviar el estado en los instantes en que el predicado ϕ puede llegar a ser *Verdadero* o dejar de serlo. No tiene sentido enviar lo cambios de estado que no afectan al valor del predicado.

Así, en el ejemplo del sistema de procesos p_i que se supone deben obedecer la restricción $|x_i - x_j| \leqslant \delta$, ($i = 1, 2, ..., N$), los procesos necesitan notificar al monitor sólo cuando los valores de sus propias variables x_i cambian. Cuando éstos envían su estado, también proporcionan el valor de x_i pero no necesitan enviar ninguna otra variable.

10.6.1. OBSERVACIÓN DE ESTADOS GLOBALES CONSISTENTES

El monitor debe recoger los estados globales consistentes frente a los que evaluar ϕ. Recuerde que un corte C es consistente si y sólo si para todos los sucesos en el corte C, $f \rightarrow e \Rightarrow f \in C$.

Por ejemplo, la Figura 10.14 muestra dos procesos p_1 y p_2 con variables x_1 y x_2 respectivamente. Los sucesos mostrados en las líneas de tiempo (con los vectores de marcas de tiempo) están ajustados a los valores de las dos variables. Inicialmente $x_1 = x_2 = 0$. El requisito es que $|x_i - x_j| \leqslant 50$. El proceso hace ajustes en sus variables, pero *grandes* ajustes producen que se envíe a los otros procesos un mensaje conteniendo el nuevo valor. Cuando cualquiera de los procesos recibe un mensaje de ajuste de otro, coloca su variable homóloga al valor contenido en el mensaje.

Cuando uno de los procesos p_1 o p_2 ajusta el valor de sus variable (tanto si es un ajuste *pequeño* como uno *grande*), envía el valor en un mensaje de estado al proceso monitor. El último mantiene los mensajes de estado en las colas *por proceso* para el análisis. Si los procesos monitor usaron valores del corte inconsistente C_1 de la Figura 10.14, entonces se debería encontrar que $x_1 = 1$, $x_2 = 100$, incumpliendo la restricción $|x_i - x_j| \leqslant 50$. Pero ese estado de cosas nunca llegó a ocurrir. Por otro lado, los valores del corte consistente C_2 muestra $x_1 = 105$, $x_2 = 90$.

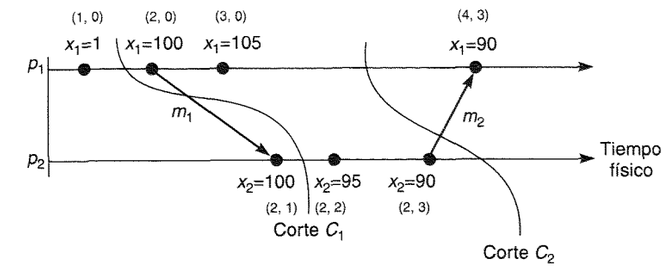


Figura 10.14. Vectores de marcas de tiempo y valores variables para la ejecución de la Figura 10.9.

Con el fin que el monitor pueda distinguir estados globales consistentes de estados globales inconsistentes, los procesos observados engloban sus valores de reloj vectorial con sus mensajes de estado. Cada cola Q_i se mantiene ordenada por el orden de envío, que puede ser establecido inmediatamente examinando el *i-ésimo* componente del vector de marcas de tiempo. Naturalmente el proceso monitor no puede deducir nada sobre la ordenación de los estados enviados por procesos diferentes a partir de su orden de llegada, a causa de las latencias de los mensajes. En su lugar debe examinar los vectores de marcas de tiempo de los mensajes de estado.

Sea $S = (s_1, s_2, ..., s_N)$ un estado global obtenido de los mensajes de estado que ha recibido el proceso monitor. Sea $V(s_i)$ el vector de marcas de tiempo del estado s_i recibido desde p_i. Se puede mostrar que S es un estado global consistente si y sólo si:

$$V(s_i)[i] \geqslant V(s_j)[i] \quad \text{para} \quad i, j = 1, 2, ..., N - \text{(Condición } EGS\text{)}$$

Esto viene a decir que el número de sucesos de p_i conocidos en p_j cuando éste envía s_j no es más que el número de sucesos que han ocurrido en p_i cuando envió s_i. En otras palabras, si el estado de un proceso depende del estado del otro (de acuerdo con la ordenación *sucedió antes*), entonces el estado global también abarca el estado del que depende.

En resumen, ahora poseemos un método por el que el proceso monitor puede establecer si un estado global dado es consistente, usando el vector de marcas de tiempo mantenido por los procesos observados e incluido en los mensajes de estado que ellos le enviaron.

La Figura 10.15 muestra la red de estados globales consistentes correspondiente a la ejecución de los dos procesos de la Figura 10.14. Esta estructura captura la relación de alcanzabilidad entre estados globales consistentes. Los nodos representan estados globales, y los arcos representan posibles transiciones entre los estados. El estado global S_{00} tiene a ambos procesos en su estado inicial; S_{10} tiene a p_2 aún en su estado inicial y a p_1 en el siguiente estado en su historia local. El estado S_{01} no es consistente, porque el mensaje m_1 se envió desde p_1 a p_2, por lo que no aparece en la red.

La red está organizada en niveles con, por ejemplo, S_{00} en el nivel 0, S_{10} en el nivel 1. En general, S_{ij} está en el nivel $(i + j)$. Una linealización atraviesa la red desde cualquier estado global a cualquier otro estado global alcanzable desde él en el siguiente nivel, es decir, en cada paso algún proceso experimenta un suceso. Por ejemplo, S_{22} es alcanzable desde S_{20}, pero no lo es desde S_{30}.

La red nos muestra todas las linealizaciones correspondientes a una historia. Ahora está claro, en principio, como un proceso monitor debiera evaluar *posiblemente* ϕ y *sin duda alguna* ϕ. Para evaluar *posiblemente* ϕ, el proceso monitor comienza en el estado inicial y salta a través de todos los estados consistentes alcanzables desde ese punto, evaluando ϕ en cada etapa. Éste se detendrá

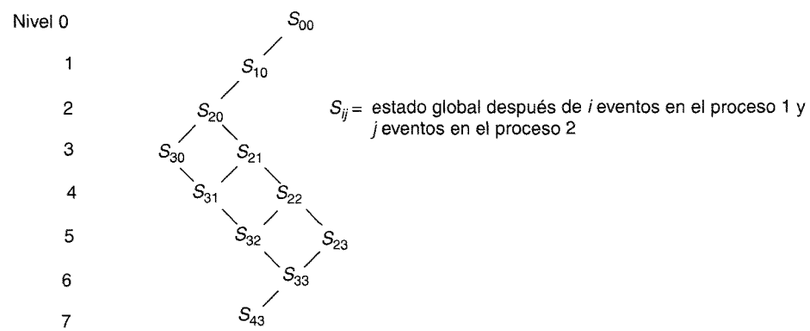


Figura 10.15. La red de estados globales para la ejecución de la Figura 10.14.

cuando ϕ se evalúa a *Verdadero*. Para evaluar *sin duda alguna* ϕ, el proceso monitor deberá intentar encontrar el conjunto de estados a través de los cuales deben pasar todas las linealizaciones, y en cuál de ellos ϕ se evalúa a *Verdadero*. Por ejemplo, si $\phi(S_{30})$ y $\phi(S_{21})$ en la Figura 10.15 son ambos *Verdaderos* entonces, puesto que todas las linealizaciones pasan a través de estos estados, se mantiene *sin duda alguna* ϕ.

10.6.2. EVALUANDO *POSIBLEMENTE* ϕ

Para evaluar *posiblemente* ϕ, el proceso monitor debe atravesar la red de estados alcanzables, comenzando desde el estado inicial $(s_1^0, s_2^0, ..., s_N^0)$. El algoritmo se muestra en la Figura 10.16. Dicho algoritmo supone que la ejecución es infinita. Puede ser adaptado fácilmente para una ejecución finita.

El proceso monitor puede descubrir el conjunto de estados consistentes en el nivel $L+1$ alcanzable desde un estado consistente en el nivel L por el método siguiente. Sea $S = (s_1, s_2, ..., s_N)$ un

---

1. *Evaluando posiblemente ϕ para la historia global H de N procesos*
   $L := 0$
   Estados $:= \{ (s_1^0, s_2^0, ..., s_N^0)\}$;
   mientras (ϕ (S) = *Falso* para todos los $S \in$ *Estados*)
      $L := L + 1$;
      Alcanzable$:= \{S'$: S' alcanzable en H desde algún $S \in$ *Estados* $\wedge$ *nivel* $(S') = L\}$;
      Estados := *Alcanzable*
   fin mientras
   salida "*posiblemente ϕ*";

2. *Evaluando sin duda alguna ϕ para la historia global H de N procesos*
   $L := 0$;
   si ($\phi(s_1^0, s_2^0, ..., s_N^0)$) *entonces Estados* $:= \{\}$ *sino Estados* $:= \{ (s_1^0, s_2^0, ..., s_N^0)\}$;
   mientras (*Estados* $\neq \{\}$)
      $L := L + 1$;
      Alcanzable$:= \{S'$: S' alcanzable en H desde algún $S \in$ *Estados* $\wedge$ *nivel* $(S') = L\}$;
      Estados$:= \{S \in$ *Alcanzable*: $\phi(S)$ = *Falso*$\}$
   fin mientras
   salida "*sin duda alguna ϕ*";

Figura 10.16. Algoritmos para evaluar posiblemente ϕ y *sin duda alguna* ϕ.

estado consistente. Entonces un estado consistente en el siguiente nivel alcanzable desde S es de la forma $S' = (s_1, s_2, ..., s_i', ..., s_N)$, que difiere de S sólo porque contiene el siguiente estado de algún proceso p_i. El monitor puede encontrar todos esos estados atravesando las colas de estados Q_i $(i = 1, 2, ..., N)$. el estado S' es alcanzable desde S si y sólo si:

$$\text{para } j = 1, 2, ..., N, j \neq i\text{: } V(s_j)[j] \geqslant V(s_i')[j]$$

Esta condición se deduce de la anterior condición EGC y del hecho que S ya estaba en un estado consistente. Un estado dado, en general, puede ser alcanzado desde varios estados del nivel anterior, por lo que el proceso monitor debería tener cuidado de evaluar la consistencia de cada estado sólo una vez.

10.6.3. EVALUANDO SIN DUDA ALGUNA ϕ

Para evaluar *sin duda alguna* ϕ, el proceso monitor atraviesa de nuevo la red de estados alcanzables en un tiempo, comenzando desde el estado inicial $(s_1^0, s_2^0, ..., s_N^0)$. El algoritmo (presentado en la Figura 10.16) supone de nuevo que la ejecución es infinita pero puede ser adaptado fácilmente para una ejecución finita. Mantiene el conjunto *Estados*, que contiene aquellos estados en el nivel actual que pueden ser alcanzados en una linealización desde el estado inicial atravesando sólo estados en los que ϕ se evalúa a *Falso*. Por el hecho que exista una linealización, no podemos afirmar *sin duda alguna* ϕ: la ejecución podría haber tomado esta linealización, y ϕ sería *Falso* en cada etapa a lo largo de ella. Si alcanzamos un nivel en el que no existe dicha linealización, podemos concluir que *sin duda alguna* ϕ.

En la Figura 10.17, en el nivel 3 el conjunto *Estados* consta de un solo estado, que es alcanzable por una linealización en la que todos los estados están a *Falso* (marcado en líneas oscuras). El único estado considerado en el nivel 4 es uno marcado con «F». (El estado a su derecha no se considera, puesto que sólo puede ser alcanzado vía un estado para el que ϕ se evalúa a *Verdadero*.) Si ϕ se evalúa a *Verdadero* en el estado del nivel 5, entonces podemos concluir que *sin duda alguna* ϕ. En otro caso el algoritmo debe continuar más allá de este nivel.

◇ **Coste.** Los algoritmos que acabamos de describir explotan combinatoriamente. Supongamos que k es el máximo número de sucesos de un único proceso. Entonces los algoritmos que hemos descrito implican $O(k^N)$ comparaciones (el proceso monitor compara los estados de cada uno de los N procesos observados con todos los demás).

Hay también un coste de espacio para estos algoritmos de $O(k^N)$. Sin embargo, observamos que el monitor puede borrar un mensaje conteniendo el estado s_i de la cola Q_i cuando ningún otro ítem

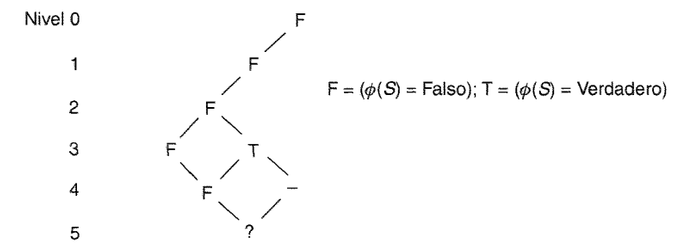


Figura 10.17. Evaluando *sin duda alguna* ϕ.

de estado llegando desde otro proceso pudiera estar posiblemente implicado en un estado global consistente que contenga s_i. Esto es, cuando:

$$V(s_j^{último})[i] > V(s_i)[i] \quad \text{para} \quad j = 1, 2, ..., N, j \neq i$$

donde $s_j^{último}$ es el último estado que el proceso monitor ha recibido desde el proceso p_j.

10.6.4. EVALUANDO *DEFINITIVAMENTE* ϕ Y *SIN DUDA ALGUNA* ϕ EN SISTEMAS SÍNCRONOS

Los algoritmos que hemos dado anteriormente trabajan en un sistema asíncrono: no hemos hecho suposiciones temporales. Pero el precio pagado por esto es que el monitor debe examinar un estado global consistente $S = (s_1, s_2, ..., s_N)$ para el que dos estados locales cualquiera s_i y s_j ocurrieron en un tiempo arbitrariamente distante en la ejecución actual del sistema. Nuestro requisito, por contraste, es considerar sólo aquellos estados por los que la ejecución actual podría haber pasado.

En un sistema síncrono, suponemos que los procesos mantienen sus relojes físicos sincronizados internamente con un límite conocido, y que los procesos observados proporcionan marcas de tiempo físicas así como un vector de marcas de tiempo en sus mensajes de estado. Entonces el proceso monitor necesita considerar sólo aquellos estados globales consistentes cuyos estados locales podrían posiblemente haber existido simultáneamente, dada la sincronización aproximada de los relojes. Con suficientemente buena sincronización de los relojes, éstos serían un número mucho menor que todos los estados consistentes globalmente.

Damos ahora un algoritmo para explotar los relojes sincronizados de esta forma. Suponemos que cada proceso observado $p_i (i = 1, 2, ..., N)$ y el proceso monitor, que podríamos llamar p_i, mantiene un reloj físico $C_i (i = 1, 2, ..., N)$. Éstos están sincronizados con un límite conocido $D > 0$; es decir, en el mismo tiempo real:

$$|C_i(t) - C_j(t)| < D \quad \text{para} \quad i, j = 1, 2, ..., N$$

Los procesos observados envían al proceso monitor tanto su tiempo vectorial como el tiempo físico con sus mensajes de estado. El proceso monitor aplica ahora una condición que no sólo comprueba la consistencia de un estado global $S = (s_1, s_2, ..., s_N)$, sino también comprueba cualquier para de estados que pudieran haber ocurrido en algún instante real, dado los valores del reloj físico. En otras palabras, para $i, j = 1, 2, ..., N$:

$$V(s_i)[i] \geqslant V(s_j)[i] \quad \text{y} \quad s_i \text{ y } s_j \text{ podrían haber ocurrido en el mismo tiempo real.}$$

La primera cláusula es la condición que hemos utilizado anteriormente. Para la segunda cláusula, hay que señalar que p_i está en el estado s_i desde el momento en que notifica su estado al proceso monitor, $C_i(s_i)$, hasta algún tiempo local posterior $L_j(s_j)$, por ejemplo, cuando ocurre la transición al siguiente estado en p_i. Para que s_i y s_j se hayan obtenido en el mismo instante real tenemos, por tanto, que permitir para el límite en la sincronización del reloj:

$$C_i(s_i) - D \leqslant C_j(s_j) \leqslant L_i(s_i) + D, \text{ o viceversa (cambiando } i \text{ por } j).$$

El proceso debe calcular un valor para $L_i(s_i)$, que se mide contra el reloj de p_i. Si el proceso monitor ha recibido un mensaje de estado para el siguiente estado de p_i, entonces $L_i(s_i)$ es $C_i(s_i')$. En caso contrario el proceso monitor estima $L_i(s_i)$ como $C_0 - max + D$, donde C_0 es el valor actual del reloj del monitor, y *max* el tiempo máximo de transmisión para un mensaje de estado.

10.7. RESUMEN

Este capítulo comenzó describiendo la importancia del mantenimiento de la precisión del tiempo en los sistemas distribuidos. Después ha descrito algoritmos para sincronización de relojes a pesar de sus derivas entre ellos y la variabilidad de los retardos en los mensajes entre computadores.

El grado de precisión que es obtenible en la sincronización satisface, en la práctica, muchos requisitos pero no es suficiente, sin embargo, para determinar la ordenación de un par arbitrario de sucesos que ocurran en diferentes computadores. La relación *sucedió antes* es un orden parcial entre sucesos que refleja un flujo de información entre ellos, dentro de un proceso o vía mensajes entre procesos. Algunos algoritmos requieren que los sucesos estén ordenados en un orden *sucedió antes*, por ejemplo las actualizaciones sucesivas hechas sobre copias separadas de datos. Los relojes de Lamport son contadores que se actualizan de acuerdo con la relación *sucedió antes* entre eventos. Los relojes vectoriales son una mejora de los relojes de Lamport, porque es posible determinar, examinando sus vectores de marcas de tiempo, si dos sucesos están ordenados en la relación *sucedió antes* o son concurrentes.

Hemos presentado los conceptos de eventos (sucesos), historias locales y globales, recortes, estados globales y locales, ejecuciones, estados consistentes, linealizaciones (ejecuciones consistentes) y alcanzabilidad. Un estado o ejecución consistente es uno que está de acuerdo con la relación *sucedió antes*.

Hemos continuado considerando el problema de registrar un estado global consistente observando la ejecución de un sistema. Nuestro objetivo ha sido evaluar un predicado sobre este estado. Una clase importante de predicados son los predicados estables. Hemos descrito el algoritmo de instantánea de Chandy y Lamport, que captura un estado consistente global y permite hacer aserciones sobre si un predicado estable se mantiene en la ejecución actual. Continuamos, dando el algoritmo de Marzullo y Neiger para derivar afirmaciones sobre si se mantiene un predicado o podría haberse mantenido en la ejecución actual. El algoritmo emplea un proceso de monitor para recoger estados. El monitor examina el vector de marcas de tiempo para extraer estados consistentes globales, y construye y examina la red de todos los estados existente globales. Este algoritmo presenta gran complejidad computacional pero es valioso de comprender y puede ser de algún beneficio práctico en sistemas reales donde relativamente pocos sucesos cambian el valor del predicado global. El algoritmo tiene una variante más eficiente para sistemas síncronos, donde los relojes pueden estar sincronizados.

EJERCICIOS

10.1. ¿Por qué es necesaria la sincronización del reloj de los computadores? Describa los requisitos de diseño para un sistema para sincronizar los relojes en un sistema distribuido.

10.2. Un reloj proporciona la lectura 10:27:54.0 (hr:min:seg) cuando de descubre que adelanta 4 segundos. Explicar por qué es deseable ajustarlo al tiempo correcto en ese punto y mostrar (numéricamente) cómo debiera ser ajustada de modo que sea correcto después de transcurridos 8 segundos.

10.3. Un esquema para implementar la entrega como mucho una vez de mensajes fiable utiliza relojes sincronizados para rechazar mensajes duplicados. Los procesos colocan su valor local del reloj (una *marca de tiempo*) en los mensajes que envían. Cada receptor mantienen una tabla, dando para cada proceso emisor, el mensaje con la marca de tiempo más grande que ha visto. Suponer que los relojes están sincronizados en 100 ms, y que los mensajes pueden llegar como mucho 50 ms después de la transmisión.

(i) ¿Cuándo ignora un proceso un mensaje llevando una marca de tiempo T, si ha registrado el último mensaje recibido desde el proceso que tiene una marca de tiempo T'?
(ii) ¿Cuándo puede un receptor eliminar una marca de tiempo 175.000 (ms) de su tabla? (Sugerencia: utilizar el valor del reloj local del receptor).
(iii) ¿Los relojes deben estar sincronizados interna o externamente?

10.4. Un cliente intenta sincronizarse con un servidor de tiempo. Registra los tiempos de ida y vuelta y las marcas de tiempo devuelto por el servidor en la tabla posterior.

¿Cuál de estos tiempos debiera utilizarse para sintonizar el reloj? ¿A qué tiempo debiera colocarse? Estimar la precisión de la sintonización con respecto al reloj del servidor. Si se sabe que el tiempo entre enviar y recibir un mensaje en el sistema implicado es de al menos 8 ms, ¿cambiarían tus respuestas?

Ida y vuelta (ms)	*Tiempo (hr:min:seg)*
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

10.5. En el sistema del Ejercicio 10.4 se requiere sincronizar un reloj de un servidor de archivos con ± 1 milisegundo. Discutir esto en relación con el algoritmo de Cristian.

10.6. ¿Qué reconfiguraciones crees que ocurran en la subred de sincronización NTP?

10.7. Un servidor B de NTP recibe un mensaje del servidor A a las 16:34:23.480 llevando una marca de tiempo 16:34:13.430 y lo responde. A recibe el mensaje a las 16:34:15.725, llevando una marca de tiempo 16:34:25.7 de B. Estimar la deriva entre B y A y la precisión de la estimación.

10.8. Discutir los factores que se toman en cuenta cuando se decide con qué servidor NTP debe sincronizar un cliente su reloj.

10.9. Discutir cómo es posible compensar la deriva de reloj entre puntos de sincronización observando el ritmo de deriva sobre el tiempo. Discutir cualquier limitación a su tiempo.

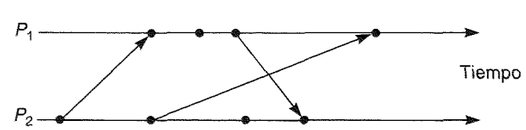
10.10. Considerando una cadena de cero o más mensajes conectando sucesos e y e' y utilizando la inducción, mostrar que $e \rightarrow e' \quad \Rightarrow \quad L(e) \rightarrow L(e')$.

10.11. Mostrar que $V_i[j] \leqslant V_i[i]$.

10.12. De una forma similar al Ejercicio 10.10, mostrar que $e \rightarrow e' \quad \Rightarrow \quad V(e) < V(e')$.

10.13. Utilizando el resultado del Ejercicio 10.11, mostrar que si los sucesos e y e' son concurrentes entonces no se cumple ni $V(e) \leqslant V(e')$ ni $V(e') \leqslant V(e)$. De esta manera, si $V(e) < V(e')$ entonces $e \rightarrow e'$.

10.14. Dos procesos P y Q están conectados en un anillo utilizando dos canales, y permanentemente rotan un mensaje m. En un instante cualquiera, sólo hay una copia de m en el sistema. El estado de cada proceso consiste en el número de veces que ha recibido el mensaje m, y P es el que envía el primer mensaje m. En un cierto punto, P tiene el mensaje y sus



estado es 101. Inmediatamente después de enviar m, P inicia el algoritmo de instantánea. Explicar la operación del algoritmo es este caso, proporcionando el posible estado global devuelto por él.

10.15. La figura anterior muestra sucesos que ocurren en dos procesos, p_1 y p_2. Las flechas entre procesos indican transmisión de mensaje.

Dibuje y etiquete la red de estados consistentes (estado p_1, estado p_2), comenzando con el estado inicial (0, 0).

10.16. Pérez está ejecutando una colección de procesos $p_1, p_2, \ldots p_N$. Cada proceso p_i contiene una variable v_i. Ella desea determinar si todas las variables $v_1, v_2, \ldots v_N$ han sido siempre iguales en el curso de la ejecución.

(i) Los procesos de Pérez se ejecutan en un sistema síncrono. Ella utiliza un proceso monitor para determinar si las variables fueron siempre iguales. ¿Cuándo deberían los procesos de la aplicación comunicarse con el proceso monitor, y qué deberían contener sus mensajes?
(ii) Explicar la declaración posiblemente ($v_1 = v_2 \ldots = v_N$). ¿Cómo puede Pérez determinar si esta declaración es verdadera durante su ejecución?

11

COORDINACIÓN Y ACUERDO

11.1. Introducción
11.2. Exclusión mutua distribuida
11.3. Elecciones
11.4. Comunicación por multidifusión
11.5. Consenso y sus problemas relacionados
11.6. Resumen

En este capítulo, se presentan algunos temas y algoritmos relacionados con la forma en la que los procesos coordinan sus acciones y realizan acuerdos sobre valores compartidos en sistemas distribuidos aun en presencia de fallos. El capítulo empieza presentando algoritmos para conseguir exclusión mutua entre una colección de procesos y para coordinar sus accesos a los recursos compartidos. Continúa examinando cómo puede implementarse una elección en un sistema distribuido. Esto es, describe cómo un grupo de procesos pueden ponerse de acuerdo acerca de un nuevo coordinador de sus actividades tras fallar el coordinador anterior.

La segunda mitad del capítulo examinará los problemas relacionados con la comunicación por multidifusión, el consenso, los acuerdos bizantinos y la consistencia interactiva. En el caso de la comunicación por multidifusión se centra en la consecución de acuerdos en temas tales como el orden en el cual se entregan los mensajes. El consenso y el resto de los problemas se generalizan a partir de la siguiente idea: ¿cómo puede cualquier colección de procesos ponerse de acuerdo en algún valor, con independencia del dominio de los valores en cuestión? Encontramos un resultado fundamental en la teoría de sistemas distribuidos: bajo ciertas condiciones, entre las que se incluyen condiciones de fallo sorprendentemente benignas, es imposible garantizar que los procesos alcanzarán un consenso.

11.1. INTRODUCCIÓN

Este capítulo presenta una colección de algoritmos con objetivos variados, pero que comparten un fin que es fundamental en los sistemas distribuidos: dado un conjunto de procesos, coordinar sus acciones o ponerse de acuerdo en uno o más valores. Por ejemplo, en el caso de un mecanismo complejo como el de una nave espacial, es esencial que los computadores que realizan el control acuerden condiciones tales como si la misión de la nave prosigue o si ésta ha sido abortada. Además, los computadores deben coordinar sus acciones correctamente con respecto a los recursos compartidos (los sensores y actuadores de la nave espacial). Los computadores han de ser capaces de hacerlo incluso cuando no hay una relación establecida maestro-esclavo entre los componentes (que haría la coordinación particularmente simple). La razón para evitar relaciones fijas maestro-esclavo es que, frecuentemente, requerimos de nuestros sistemas que se mantengan trabajando correctamente incluso si ocurre un fallo, por lo que es necesario evitar puntos de fallo individuales tales como maestros prefijados.

Una distinción importante, como se hizo en el Capítulo 10, será si el sistema distribuido en estudio es síncrono o asíncrono. En un sistema asíncrono, no podemos hacer suposiciones con relación a la coordinación temporal. En un sistema síncrono, supondremos que hay límites para el retraso máximo en la transmisión de mensajes, en el tiempo para ejecutar cada proceso y en el promedio de derivas de los relojes. Las suposiciones de sincronía permiten que se usen timeouts para detectar caídas en los procesos.

Otra meta importante del capítulo, cuando se discuta sobre algoritmos, será la de considerar la presencia de fallos y cómo tenerlos en cuenta cuando se diseñan dichos algoritmos. En la Sección 2.3.2 se presentó un modelo de fallo que se usará en este capítulo. El abordar el tema de los fallos es una tarea sutil; por lo tanto, se empezará considerando algoritmos que no toleran fallos, se avanzará introduciendo fallos benignos y se llegará a considerar cómo tolerar fallos arbitrarios. Además, se encuentra un resultado fundamental en la teoría de sistemas distribuidos. Incluso bajo condiciones de fallo sorprendentemente benignas, es imposible garantizar en un sistema asíncrono que una colección de procesos puedan ponerse de acuerdo en un valor compartido; por ejemplo, que todos los procesos involucrados en el control de la nave espacial se pongan de acuerdo en si «la misión sigue» o «la misión aborta».

La Sección 11.2 examina el problema de la exclusión mutua distribuida. Ésta es la extensión a los sistemas distribuidos del conocido problema de evitar condiciones de competición continua en los núcleos y aplicaciones multi-hilo. La exclusión mutua es un problema importante a resolver, dado que lo que ocurre fundamentalmente en un sistema distribuido es que los procesos comparten recursos. A continuación, la Sección 11.3 presenta un asunto relacionado, aunque más general, que es cómo «elegir» dentro de una colección de procesos a uno para que desarrolle un papel especial. Por ejemplo, en el Capítulo 10 se vio cómo los procesos sincronizaban sus relojes con un servidor de tiempos previamente designado. Si el servidor falla y han sobrevivido otros servidores que pueden desempeñar ese papel, entonces es necesario elegir un servidor que lo releve con el fin de mantener la consistencia.

La comunicación por multidifusión es el tema de la Sección 11.4. Tal y como se explicó en la Sección 4.5.1, la técnica de multidifusión es un paradigma de comunicación muy útil, pudiendo aplicarse en temas tan dispares como la localización de recursos o la coordinación de actualizaciones de datos replicados. La Sección 11.4 examina la fiabilidad y la semántica de la ordenación en la técnica de multidifusión y proporciona algoritmos para lograr las distintas variantes. La entrega en la multidifusión es esencialmente un problema de llegar a un acuerdo entre procesos: los receptores acuerdan qué mensajes recibirán y en qué orden. La Sección 11.5 describe el problema de acuerdos de forma más general, principalmente en las formas conocidas como consenso y acuerdos bizantinos.

El tratamiento que se sigue en este libro implica el establecimiento de las hipótesis y los objetivos que deben cumplirse, y dar una explicación informal de por qué los algoritmos que se presentan son correctos. No puede proporcionarse una aproximación más rigurosa dados los problemas de espacio. Debido a esto, remitiremos al lector a los textos que proporcionan informes minuciosos de algoritmos distribuidos, tales como Attiya y Welch [1998] y Lynch [1996].

Antes de introducir los problemas y los algoritmos, discutiremos las hipótesis sobre fallos y la forma práctica de detectar fallos en un sistema distribuido.

11.1.1. SUPOSICIONES SOBRE FALLOS Y DETECTORES DE FALLOS

Para simplificar, este capítulo supone que cada par de procesos están conectados por canales fiables. Esto es, aunque los componentes de la red subyacente puedan sufrir fallos, los procesos utilizan un protocolo de comunicación fiable que enmascara estos fallos, por ejemplo mediante la retransmisión de mensajes perdidos o corruptos. También por simplificar, se supone que ningún fallo en un proceso implica una amenaza para la capacidad de otros procesos de comunicarse. Esto significa que ningún proceso depende de otro para enviar mensajes.

Nótese que un canal fiable *al final* entrega un mensaje en el búfer de entrada del receptor. En un sistema síncrono, suponemos que existe una redundancia en el hardware allí donde sea necesario de tal forma que un canal fiable no sólo entrega finalmente un mensaje a pesar de los fallos, sino que lo hace dentro de un tiempo límite especificado.

En un intervalo de tiempo cualquiera, la comunicación entre ciertos procesos puede tener éxito mientras que la comunicación entre otros procesos puede verse retrasada. Por ejemplo, el fallo de un encaminador entre dos redes puede significar que un conjunto de cuatro procesos se divida en dos parejas, de tal forma que la comunicación dentro de las parejas es posible sobre sus respectivas redes; pero la comunicación entre las parejas no es posible mientras el encaminador siga fallando. Esto se conoce como una *partición de la red* (véase la Figura 11.1). Sobre una red punto a punto como es Internet, la presencia de topologías complicadas y elecciones de encaminamiento independientes conllevan que la conectividad pueda ser *asimétrica*: la comunicación es posible entre el proceso p y el proceso q, pero no viceversa. La conectividad puede ser también *intransitiva*: la comunicación es posible desde p hasta q y desde q hasta r; pero p no puede comunicarse directamente con r. Así, nuestra suposición de fiabilidad implica que al final cualquier enlace o encaminador caído será reparado o rodeado. Sin embargo, no todos los procesos son capaces de comunicarse al mismo tiempo.

El capítulo supone, salvo que se diga lo contrario, que los procesos sólo fallan por caídas, una hipótesis que es suficientemente buena para muchos sistemas. En la Sección 11.5 consideraremos cómo tratar los casos en los que los procesos tengan fallos arbitrarios (extraños). Cualquiera que

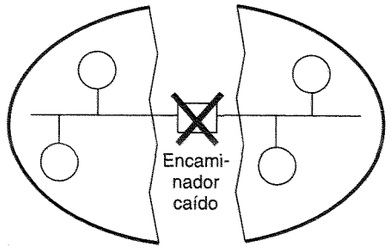


Figura 11.1. Una partición de la red.

sea el fallo, un proceso *correcto* es uno que no muestra fallos en ningún momento de la ejecución que se esté considerando. Nótese que el término corrección se aplica a la ejecución completa, no sólo a una parte de ella. Por lo tanto, un proceso que sufre un fallo por caída se dice que está «sin fallo» antes de ese momento, pero no que sea «correcto» antes de ese momento.

Uno de los problemas en el diseño de algoritmos que puedan superar las caídas de procesos es el de decidir cuándo un proceso ha caído. Un *detector de fallo* [Chandra y Toueg 1996, Stelling y otros 1998] es un servicio al que recurren los procesos para saber si un proceso determinado ha fallado. Frecuentemente, se implementa usando un objeto local en cada proceso (en el mismo computador) que ejecuta un algoritmo de detección de fallos en colaboración con sus equivalentes en los otros procesos. El objeto local a cada proceso se llama un *detector de fallo local*. Se esbozará brevemente cómo implementar detectores de fallo, pero antes hemos de concentrarnos en algunas de las propiedades de los detectores de fallos.

Un «detector» de fallos no es necesariamente exacto. La mayor parte de ellos pueden catalogarse como *detectores de fallo no fiables*. Uno de estos detectores puede generar uno de dos valores cuando se le proporcione la identidad de un proceso: *Sospechoso* o *No sospechoso*. Ambos resultados deben considerarse como indicios que pueden reflejar de forma precisa o no si el proceso ha fallado en realidad. Un resultado de *No sospechoso* significa que el detector ha recibido recientemente evidencias que le sugieren que el proceso no ha fallado; por ejemplo, se recibió recientemente un mensaje enviado por él. Pero, por supuesto, el proceso puede haber fallado desde entonces. Un resultado de *Sospechoso* debe entenderse como que el detector de fallo tiene alguna indicación de que el proceso puede haber fallado. Por ejemplo, puede ser que no haya recibido ningún mensaje por parte del proceso en un período de silencio mayor al permitido (incluso en un sistema asíncrono pueden usarse como indicios algunos límites superiores prácticos). La sospecha puede estar equivocada; por ejemplo, el proceso puede estar funcionando correctamente, pero al otro lado de una partición en la red; o puede estar funcionando más lentamente de lo esperado.

Un *detector de fallo fiable* es aquel que siempre detecta fallos en un proceso de forma exacta. Responde a las preguntas de los procesos bien con el término *No sospechoso*, que, como antes, puede ser sólo un indicio, o con el término *Fallido*. Un resultado de *Fallido* significa que el detector ha determinado que el proceso se ha caído. Recuérdese que un proceso que se ha caído permanece en ese estado, dado que por definición un proceso nunca continúa una vez se ha malogrado.

Es importante darse cuenta de que, aunque se habla de un detector de fallos actuando dentro de una colección de procesos, la respuesta que el detector proporciona al proceso es tan buena como la información de la que dispone el proceso. Además, un detector puede dar algunas veces diferentes respuestas a diferentes procesos, dado que las condiciones de comunicación varían de un proceso a otro.

Podemos implementar un detector de fallos no fiable utilizando el siguiente algoritmo. Cada proceso p manda un mensaje «p está aquí» al resto de procesos, y lo hace cada T segundos. El detector de fallos utiliza una estimación del máximo tiempo de transmisión de un mensaje de D segundos. Si el detector de fallos local en un proceso q no recibe un mensaje «p está aquí» dentro de los $T + D$ segundos desde el último, entonces informa a q que p es *Sospechoso*. Sin embargo, si posteriormente recibe un mensaje «p está aquí», entonces informa a q de que p está correctamente.

En un sistema distribuido real hay límites prácticos en los tiempos de transmisión de mensajes. Incluso los sistemas de correo abandonan tras unos pocos días, dado que es probable que los enlaces de comunicación y los encaminadores hayan sido reparados en ese tiempo. Si elegimos valores pequeños para T y D (que sumen en total 0,1 segundos, por ejemplo) entonces el detector de fallos probablemente sospechará bastantes veces de procesos que no se han caído, y una buena parte del ancho de banda se dedicará a los mensajes «p está aquí». Si elegimos un valor de timeout total grande de (por ejemplo, una semana) entonces los procesos que se han caído serán considerados frecuentemente como *No sospechosos*.

Una solución práctica para este problema es usar valores de timeout que reflejen las condiciones de retraso observadas en la red. Si un detector de fallos local recibe un mensaje «p está aquí» en 20 segundos en lugar del máximo esperado de 10 segundos, entonces tendría que cambiar el valor del tiempo para p de acuerdo con la nueva información. El detector de fallos sigue siendo no fiable y sus respuestas a las preguntas deben seguir considerándose como indicios, pero la probabilidad de que sea exacto aumenta.

En un sistema síncrono, el sistema detector antes descrito puede convertirse en fiable. Se puede elegir D de tal forma que no sea una estimación sino un límite absoluto en los tiempos de transmisión de mensajes; la ausencia de un mensaje «p está aquí» dentro de los $T + D$ segundos lleva al detector de fallos local a concluir que el proceso p se ha caído.

Los lectores pueden preguntarse si los detectores de fallo tienen alguna utilidad práctica. Por un lado, los detectores de fallos no fiables pueden sospechar de un proceso que no haya fallado (pueden ser *inexactos*); y pueden no sospechar de un proceso que de hecho haya fallado (pueden ser *incompletos*). Por otro lado, los detectores de fallos fiables requieren que el sistema sea síncrono (y muy pocos sistemas lo son en la práctica).

Se han presentado los detectores de fallos porque ayudan a pensar en la naturaleza de los fallos en un sistema distribuido. Y cualquier sistema práctico que se diseñe para soportar fallos debe detectarlos, aunque no lo haga de una forma perfecta. No obstante, parece que incluso los detectores de fallos no fiables con algunas propiedades bien definidas pueden ayudar a proporcionar soluciones prácticas al problema de la coordinación de procesos en presencia de fallos. Se volverá a este tema en la Sección 11.5.

11.2. EXCLUSIÓN MUTUA DISTRIBUIDA

Los procesos distribuidos necesitan frecuentemente coordinar sus actividades. Si una colección de procesos comparte un recurso o una colección de recursos, entonces se requiere con frecuencia la exclusión mutua para prevenir interferencias y asegurar la consistencia cuando se accede a los recursos. Éste es el problema de la *sección crítica*, conocido en el dominio de los sistemas operativos. Sin embargo, en un sistema distribuido ni las variables compartidas ni las utilidades proporcionadas por un núcleo local pueden usarse para solucionarlo de forma general. Se requiere una solución para la *exclusión mutua distribuida*: una que esté basada exclusivamente en el paso de mensajes.

En algunos casos, los recursos compartidos son gestionados por servidores que, además, proporcionan mecanismos para la exclusión mutua. El Capítulo 12 describe cómo algunos servidores sincronizan los accesos de los clientes a los recursos. Pero en algunos casos prácticos se requiere un mecanismo separado de exclusión mutua.

Considérese el caso de los usuarios que actualizan un archivo de texto. Un método sencillo de asegurar que sus actualizaciones sean consistentes es permitirles que cada vez sólo uno tenga acceso, exigiendo al editor que bloquee el archivo antes de que se hagan las actualizaciones. Los servidores de archivos NFS, descritos en el Capítulo 8, se diseñan para que no tengan estado y, por lo tanto, no admiten el bloqueo de archivos. Por esta razón, los sistemas UNIX proporcionan un servicio de bloqueo de archivos separado, implementado mediante el proceso demonio *lockd*, para gestionar las peticiones de bloqueo por parte de los clientes.

Un ejemplo, particularmente interesante, se da donde no existen servidores de forma que una colección de procesos de igual importancia debe coordinar sus accesos a recursos compartidos por ellos mismos. Esto ocurre de forma habitual en redes tales como las de tipo Ethernet y las redes inalámbricas IEEE 802.11 en modo *ad hoc*, donde las interfaces de red cooperan como procesos iguales, de tal forma que sólo un nodo transmite cada vez en el medio compartido. Otro ejemplo

sería un sistema que realiza el seguimiento del número de vacantes en un aparcamiento de vehículos con un proceso a cada entrada y salida que observe el número de vehículos que entran y salen. Cada proceso mantiene un registro del número total de vehículos dentro del aparcamiento, e informa de si está lleno o no. Los procesos han de actualizar el registro del número de vehículos de forma consistente. Hay distintas formas de conseguirlo, pero sería conveniente para estos procesos que fueran capaces de obtener exclusión mutua únicamente comunicándose entre ellos, eliminando la necesidad de un servidor aparte.

Sería útil tener a nuestra disposición un mecanismo genérico para la exclusión mutua distribuida que fuera independiente del esquema particular de gestión de recursos en cuestión. A continuación se examinarán algunos algoritmos que lo consiguen.

11.2.1. ALGORITMOS PARA LA EXCLUSIÓN MUTUA

Considérese un sistema de N procesos p_i, $i = 1, 2, ..., N$ que no comparten variables. Los procesos acceden a recursos compartidos comunes, pero lo hacen en una sección crítica. Para simplificar, se supondrá que hay solamente una sección crítica. El procedimiento para ampliar los algoritmos que presentan más de una sección crítica es sencillo.

Se supone que el sistema es asíncrono, que los procesos no fallan y que la entrega de mensajes es fiable, de tal forma que cualquier mensaje enviado finalmente es entregado intacto y exactamente una vez.

El protocolo a nivel de aplicación para ejecutar una sección crítica es como sigue:

entrar() // entrada en la sección crítica – bloquéese si es necesario
acceso_a_Recursos() // acceso a los recursos compartidos en la sección crítica
salir() // salida de la sección crítica – pueden entrar otros procesos

Nuestros requisitos esenciales para la exclusión mutua son los siguientes:

EM1: (seguridad) A lo sumo un proceso puede estar ejecutándose cada vez en la sección crítica (SC).
EM2: (pervivencia) Las peticiones para entrar y salir de la sección crítica al final son concedidas.

La condición EM2 implica la ausencia tanto de estancamiento como de inanición. Un estancamiento implicaría que dos o más procesos quedasen atascados indefinidamente intentando entrar o salir de la sección crítica, en virtud de su mutua interdependencia. Pero, incluso sin estancamiento, un algoritmo no muy logrado podría llevar a la *inanición*: el aplazamiento indefinido de la entrada de un proceso que lo ha solicitado.

La ausencia de inanición es una condición de *equidad*. Otro asunto relativo a la equidad es el orden en el cual los procesos entran en la sección crítica. No es posible ordenar la entrada en la sección crítica por el momento en el que el proceso lo solicita, dada la ausencia de relojes globales. Pero un requisito útil que se impone algunas veces para conseguir la equidad utiliza la ordenación *sucedió-antes* (véase la Sección 10.4) entre mensajes que solicitan entrada en la sección crítica:

EM3: (→ ordenación) Si una petición para entrar en la SC *ocurrió-antes* que otra, entonces la entrada a la SC se garantiza en ese orden.

Si una solución garantiza la entrada en la sección crítica en una ordenación sucedió-antes, y si todas las peticiones están relacionadas mediante el sucedió-antes, entonces no es posible para un proceso entrar en la sección crítica más de una vez mientras otro espera para entrar. Esta ordenación, además, permite a los procesos coordinar sus accesos a la sección crítica. Un proceso multihilo puede continuar con otros procesamientos mientras un hilo espera que se le garantice la entrada a una sección crítica. Durante este tiempo, podría enviar un mensaje a otro proceso, el cual, a continuación, también intentaría entrar en la sección crítica. EM3 especifica que debe garantizarse el acceso al primer proceso antes que al segundo.

A continuación se evalúa el rendimiento de los algoritmos para la exclusión mutua de acuerdo a los siguientes criterios:

- El *ancho de banda* consumido, que es proporcional al número de mensajes enviados en cada operación *entrar* y *salir*.
- El *retraso del cliente* en el que incurre un proceso en cada operación *entrar* y *salir*.
- El efecto del algoritmo sobre la capacidad de procesamiento del sistema. Ésta es la tasa promedio a la que la colección de procesos, en su totalidad, puede acceder a la sección crítica, teniendo en cuenta que es necesaria alguna comunicación entre procesos sucesivos. Se mide el efecto por medio del *retraso en la sincronización* entre un proceso que sale de la sección crítica y el siguiente proceso que entra en ella; la capacidad de procesamiento es mayor cuando el retraso en la sincronización es menor.

No se tendrá en cuenta en nuestra descripción la implementación de accesos a recursos. Sin embargo, se asumirá que los procesos cliente se comportan bien y emplean un tiempo finito accediendo a los recursos dentro de sus secciones críticas.

◇ **El algoritmo del servidor central.** La forma más simple de conseguir exclusión mutua es emplear un servidor que dé los permisos para entrar en la sección crítica. La Figura 11.2 muestra el uso de este servidor. Para entrar en una sección crítica, un proceso envía un mensaje de petición al servidor y espera una respuesta por su parte. Conceptualmente, la respuesta constituye un testigo que significa permiso para entrar en la sección crítica. Si ningún otro proceso tiene el testigo en el instante de la petición, entonces el servidor responde inmediatamente, dándoselo. Si en ese momento lo tiene otro proceso, entonces el servidor no responde sino que lo pone en una cola la petición. Cuando un proceso sale de la sección crítica envía un mensaje al servidor, devolviéndole el testigo.

Si la cola de procesos en espera no está vacía, entonces el servidor escoge la entrada más antigua en la cola, la elimina y responde al proceso correspondiente. El proceso seleccionado mantiene entonces el testigo. En la figura, se muestra una situación en la cual la petición de p_2 ha sido añadida a la cola, que ya contenía la petición de p_4. A continuación, p_3 sale de la sección crítica, y el servidor elimina la entrada de p_4 y le responde, concediéndole permiso para entrar. El proceso p_1 no requiere en ese momento entrada en la sección crítica.

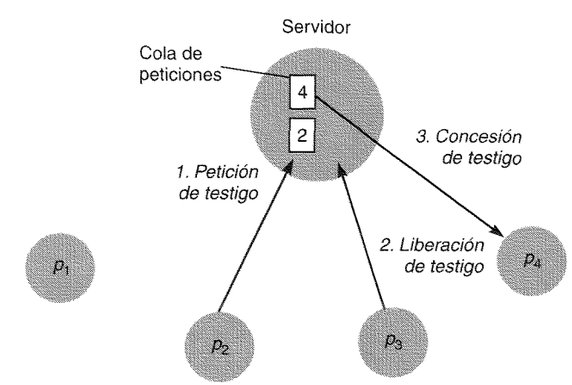


Figura 11.2. Servidor que gestiona un testigo de exclusión mutua para un conjunto de procesos.

Dada nuestra suposición de que no ocurren fallos, es fácil ver que las condiciones de seguridad y pervivencia se consiguen mediante este algoritmo. No obstante, los lectores deben observar que el algoritmo no satisface la propiedad EM3.

A continuación, se evalúa el rendimiento de este algoritmo. La entrada en la sección crítica, incluso cuando no haya un proceso ocupándola, conlleva dos mensajes (una *petición* seguida por una *concesión*) y retrasa al proceso que realiza la petición debido a la duración de este viaje de ida y vuelta. La salida de la sección crítica implica un mensaje de *liberación*. Si se supone que el paso de mensajes es asíncrono, esto no retrasa al proceso que sale.

El servidor puede convertirse en un cuello de botella para el rendimiento del sistema considerado en su totalidad. El retraso en la sincronización es el tiempo que se tarda en un viaje de ida y vuelta: un mensaje de *liberación* hacia el servidor, seguido por un mensaje de *concesión* al siguiente proceso que va a entrar en la sección crítica.

◇ **Un algoritmo basado en un anillo.** Una de las maneras más simples de conseguir la exclusión mutua entre los N procesos sin que se necesite un proceso adicional es organizarlos en un anillo lógico. Esto sólo requiere que cada proceso p_i tenga un canal de comunicación hacia el siguiente proceso en el anillo, $p_{(i+1) mod N}$. La idea se basa en conseguir la exclusión obteniendo un testigo mediante un mensaje que se pasa de un proceso a otro en una única dirección alrededor del anillo; por ejemplo, en el sentido de las agujas del reloj. La topología en anillo no tiene por qué estar relacionada con las interconexiones físicas subyacentes entre los computadores.

Si un proceso no requiere entrar en la sección crítica cuando recibe el testigo, entonces, inmediatamente, lo hace avanzar hacia su vecino. Un proceso que requiera el testigo espera hasta recibirlo y, en este caso, lo retiene. Cuando el proceso salga de la sección crítica enviará el testigo hacia el siguiente vecino.

La disposición de los procesos se muestra en la Figura 11.3. Verificar que las condiciones EM1 y EM2 se cumplen con este algoritmo es inmediato; sin embargo, también se cumple que el testigo no se obtiene necesariamente en el orden sucedió-antes (téngase en cuenta que los procesos pueden intercambiar mensajes con independencia de la rotación del testigo).

Este algoritmo consume continuamente ancho de banda de la red (excepto cuando un proceso está dentro de la sección crítica): los procesos envían mensajes alrededor del anillo incluso cuando ningún proceso requiere entrar en la sección crítica. El retraso experimentado por un proceso que está pidiendo entrar en la sección crítica está entre 0 mensajes (cuando acaba de recibir el testigo) y N mensajes (cuando acaba de pasar el testigo). Para salir de la sección crítica se requiere sola-

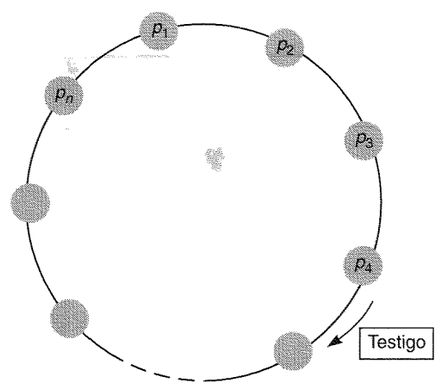


Figura 11.3. Anillo de procesos que transfieren un testigo de exclusión mutua.

En la inicialización
  estado := LIBERADA;

Para entrar en la sección crítica
  estado := BUSCADA;
  Multitransmite *petición* a todos los procesos; } *Se aplaza aquí el procesamiento de peticiones*
  T := marca temporal de la petición;
  Espera hasta que (número de respuestas recibidas = $(N - 1)$);
  estado := TOMADA;

Al recibir una petición $< T_i, p_i >$ *en el proceso* p_j $(i \neq j)$
  si (*estado* = TOMADA o (*estado* = BUSCADA y $(T, p_j) < (T_i, p_i)$))
  entonces
    pon en la cola la *petición* por parte de p_i sin responder;
  sino
    responde inmediatamente a p_i;
  finsi

Para salir de la sección crítica
  estado := LIBERADA;
  responde a cualquiera de las peticiones en la cola;

Figura 11.4. Algoritmo de Ricart y Agrawala.

mente un mensaje. El retraso en la sincronización entre la salida de un proceso de la sección crítica y la entrada del siguiente proceso está en algún punto entre 1 y N transmisiones de mensajes.

◇ **Un algoritmo que usa multidifusión y relojes lógicos.** Ricart y Agrawala [1981] desarrollaron un algoritmo para implementar la exclusión mutua entre N procesos de importancia pareja basado en la técnica de multidifusión. La idea básica es que los procesos que necesitan entrar en una sección crítica envían un mensaje de petición mediante multidifusión y pueden entrar en ella solamente cuando el resto de los procesos haya respondido al mensaje. Las condiciones bajo las cuales un proceso responde a una petición se diseñan para asegurar que se cumplan las condiciones EM1 a EM3.

Los procesos $p_1, p_2, ..., p_N$ llevan distintos identificadores numéricos. Se supone que disponen de canales de comunicación con el resto y cada proceso p_i, mantiene un reloj tipo Lamport, actualizado de acuerdo con las reglas RL1 y RL2 de la Sección 10.4. Aquellos mensajes que soliciten entrar tienen la forma $< T, p_i >$ donde T es el marca de tiempo del emisor y p_i es su identificador.

Cada proceso registra en una variable *estado* el estar fuera de la sección crítica (LIBERADA), de querer entrar (BUSCADA) o de estar en la sección crítica (TOMADA). El protocolo se muestra en la Figura 11.4.

Si un proceso solicita entrar y el estado de todos los procesos es LIBERADA, entonces todos los procesos contestarán inmediatamente al que hizo la solicitud y éste obtendrá la entrada. Si algún proceso estuviese en el estado TOMADA no responderá a las peticiones hasta que haya finalizado con la sección crítica; por lo tanto, durante ese tiempo no se concederá la entrada al proceso que realizó la solicitud. Si dos o más procesos solicitan la entrada al mismo tiempo entonces la petición que exhiba una marca temporal más baja será la que recoja las $N - 1$ respuestas, garantizando que será el siguiente en entrar. Si las dos peticiones llevan marcas temporales tipo Lamport iguales, las solicitudes se ordenan de acuerdo a los identificadores correspondientes a cada proceso. Nótese que cuando un proceso solicita la entrada, aplaza el procesamiento de peticiones por parte de otros procesos hasta que su propia solicitud haya sido enviada y se haya grabado la marca temporal T de la misma. Ésta es la razón por la que los procesos realizan decisiones consistentes cuando procesan peticiones.

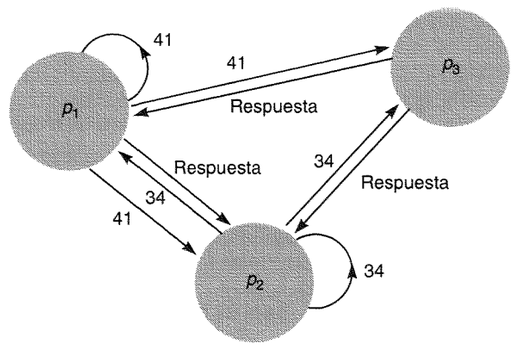


Figura 11.5. Sincronización mediante multitransmisión.

Este algoritmo cumple la propiedad de seguridad EM1. Si fuese posible que dos procesos p_i y $p_j (i \neq j)$ entrasen a la sección crítica al mismo tiempo, entonces ambos procesos deberían haberse contestado mutuamente. Pero teniendo en cuenta que los pares $<T_i, p_i>$ están totalmente ordenados, esto es imposible. Dejamos a los lectores la verificación de que el algoritmo además cumple los requisitos EM2 y EM3.

Para ilustrar el algoritmo, considérese una situación que involucra a tres procesos, p_1, p_2 y p_3 que aparecen en la Figura 11.5. Supongamos que p_3 no está interesado en entrar en la sección crítica, y que p_1 y p_2 lo solicitan de forma concurrente. La marca temporal de la petición de p_1 es 41 y la de p_2 es 34. Cuando p_3 recibe sus peticiones, responde inmediatamente. Cuando p_2 recibe la petición de p_1 se da cuenta de que su propia petición tiene una marca temporal más baja y por lo tanto no responde, manteniendo a p_1 fuera. Al mismo tiempo, p_1 encuentra que la petición de p_2 tiene una marca temporal más baja que la de su propia solicitud y responde inmediatamente. Una vez recibida esta segunda respuesta p_2 puede entrar en la sección crítica. Cuando p_2 salga de la sección crítica, responderá a la petición de p_1, garantizando así su entrada.

Conseguir la entrada en la sección crítica supone $2(N - 1)$ mensajes en este algoritmo: $N - 1$ multidifundir la solicitud, seguido de $N - 1$ respuestas. En el caso en que hubiese un hardware que diese soporte a la multidifusión sólo se requeriría un mensaje para la petición; entonces el número total de mensajes sería N. Esto hace que el algoritmo tenga un mayor coste, en términos de consumo de ancho de banda, que los algoritmos antes descritos. Sin embargo, la espera del cliente durante la solicitud de entrada es otra vez la de un viaje de ida y vuelta (si se ignora cualquier demora que pueda ocasionar el envío de mensajes de petición mediante multidifusión).

La ventaja de este algoritmo es que su retraso de sincronización es de solamente el tiempo de transmisión de un mensaje, mientras que los algoritmos anteriores incurrían en un retraso de sincronización de un viaje de ida y vuelta.

El rendimiento del algoritmo se puede mejorar. En primer lugar, nótese que el proceso que entró en último lugar a la sección crítica, y que no ha recibido peticiones para entrar, todavía sigue el protocolo, aunque pudiese decidir localmente volver a entrar. En segundo lugar, Ricart y Agrawala han refinado el protocolo de tal forma que se requieren N mensajes, en el peor de los casos, que es el más común, para conseguir la entrada sin apoyo de hardware que realice la multidifusión. Esto se describe en Raynal [1998].

◇ **Algoritmo de votación de Maekawa.** Maekawa [1985] observó que para que un proceso entrase en la sección crítica no era necesario que todos los procesos de categoría pareja a la suya le permitiesen el acceso. Los procesos sólo necesitan obtener permiso para entrar por parte de *subconjuntos* de sus pares, siempre que los subconjuntos utilizados por cualquier par de procesos se solapen. Podemos pensar en los procesos votando para que otro pueda entrar en la sección crítica. Un proceso «candidato» debe recoger suficientes votos para entrar. Los procesos en la intersección de dos conjuntos de votantes aseguran la propiedad de seguridad EM1 (que a lo sumo un proceso pueda entrar en la sección crítica) dando su voto a un único candidato.

Maekawa asoció un *conjunto de votantes* V_i con cada proceso $p_i (i = 1, 2, ..., N)$, donde $V_i \subseteq \{p_1, p_2, ..., p_N\}$. Los conjuntos V_i se eligen de tal forma que, para todo $i, j = 1, 2, ..., N$:

- $p_i \in V_i$.
- $V_i \cap V_j \neq \varnothing$ – hay al menos un miembro común a cada par de conjuntos de votantes.
- $|V_i| = K$ – para ser equitativos, cada proceso ha de tener conjuntos de votantes del mismo tamaño.
- Cada proceso p_j está contenido en M de los conjuntos de votantes V_i.

Maekawa demostró que la solución óptima, que minimiza K y permite a los procesos conseguir la exclusión mutua, tiene $K \sim \sqrt{N}$ y $M = K$ (de tal forma que cada proceso está en tantos conjuntos de votantes como elementos hay en cada uno de esos conjuntos). El cálculo de los conjuntos óptimos R_i no es trivial. De forma aproximada, una manera simple de encontrar estos conjuntos R_i tales que $|R_i| - 2\sqrt{N}$ es colocar los procesos en una matriz de $\sqrt{N}$ por $\sqrt{N}$ y hacer que V_i sea la unión de la fila y la columna que contiene a p_i.

El algoritmo de Maekawa se muestra en la Figura 11.6. Para poder entrar a la sección crítica, un proceso p_i envía mensajes de *petición* a todos los $K - 1$ miembros de V_i. p_i no puede entrar a la sección crítica hasta que no haya recibido $K - 1$ mensajes de *respuesta*. Cuando un proceso p_j en V_i recibe un mensaje de *petición* de p_i, envía un mensaje de *respuesta* inmediatamente, a no ser que su estado sea TOMADA o ya haya contestado («votado») desde que recibió el último mensaje de *liberada*. Si no es éste el caso, guarda en una cola el mensaje de petición (en el orden de llegada), pero no responde todavía. Cuando un proceso recibe un mensaje de *liberada*, elimina la cabeza de su cola de peticiones pendientes (si la cola no está vacía) y envía un mensaje de *respuesta* (un «voto»). Para dejar la sección crítica, p_i envía mensajes de *liberada* a los $K - 1$ miembros de V_i.

Este algoritmo cumple la propiedad de seguridad EM1. Si fuese posible que dos procesos p_i y p_j entrasen en la sección crítica al mismo tiempo, implicaría que los procesos en $V_i \cap V_j \neq \varnothing$ deberían haber votado por ambos a la vez. Sin embargo, el algoritmo permite a los procesos realizar, a lo sumo, un voto entre recepciones sucesivas de un mensaje *liberada*; por lo tanto, la situación es imposible.

Desgraciadamente, el algoritmo es propenso a las situaciones de estancamiento. Considérense tres procesos p_1, p_2 y p_3, con $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ y $V_3 = \{p_3, p_1\}$. Si los tres procesos solicitan entrada a la sección crítica de forma concurrente, entonces es posible que p_1 responda a p_2 pero que haga esperar a p_3; que p_2 responda a p_3 pero haga esperar a p_1; y que p_3 responda a p_1 pero haga esperar a p_2. Cada proceso ha recibido una de las dos respuestas por las que espera y ninguno puede seguir adelante.

El algoritmo puede modificarse de tal forma que no se produzcan estancamientos [Saunders 1987]. En la versión modificada del protocolo, los procesos almacenan las peticiones pendientes en orden sucedió-antes, con lo cual, también se satisface el requisito EM3.

La utilización del ancho de banda por parte del algoritmo es de $2\sqrt{N}$ mensajes por entrada en la sección crítica y de $\sqrt{N}$ mensajes por salida (asumiendo que no existan utilidades hardware para realizar la multidifusión). El número total de $3\sqrt{N}$ es superior al de $2(N - 1)$ mensajes que requiere el algoritmo de Ricart y Agrawala, siempre que $N > 4$. La espera para el cliente es la misma que la del algoritmo de Ricart y Agrawala, pero la espera en la sincronización es peor: un viaje de ida y vuelta en lugar de un tiempo de transmisión de un mensaje individual.

En la inicialización
  estado := LIBERADA;
  votado := FALSO;

Cuando p_i quiere entrar en la sección crítica
  estado := BUSCADA;
  Multitransmite *petición* a todos los procesos en $V_i - \{p_i\}$;
  Esperar hasta que (número de respuestas recibidas = $(K - 1)$);
  estado := TOMADA;

Al recibir p_j una petición de p_i ($i \neq j$)
  si (*estado* = TOMADA *o votado* = CIERTO)
  entonces
    pon en la cola la *petición* por parte de p_i sin responder;
  sino
    envía *respuesta* a p_i;
    votado := CIERTO;
  fin si

Para salir p_i de la sección crítica
  estado := LIBERADA;
  Multitransmite *liberar* a todos los procesos en $V_i - \{p_i\}$;

Al recibir en p_j una liberación por parte de p_i ($i \neq j$)
  si (la cola de peticiones no está vacía)
  entonces
    quita la cabeza de la cola – por ejemplo, p_k;
    envía *respuesta* a p_k;
    votado := CIERTO;
  sino
    votado := FALSO;
  fin si

Figura 11.6. Algoritmo de Maekawa.

◇ **Tolerancia a fallos.** Al evaluar los algoritmos anteriores con respecto a su tolerancia a fallos, las principales consideraciones a tener en cuenta serán:

- ¿Qué ocurre cuando se pierden mensajes?
- ¿Qué ocurre cuando un proceso se cae?

Ninguno de los algoritmos descritos anteriormente toleraría la pérdida de mensajes, en el caso de que los canales no fuesen fiables. El algoritmo basado en anillo no puede tolerar un fallo por caída de ningún proceso individual. Como ya se vio, el algoritmo de Maekawa puede tolerar que algunos procesos se caigan: si un proceso que se ha caído no está en un conjunto de votantes que están siendo requeridos, entonces ese fallo no afectará al resto de procesos. El algoritmo del servidor central puede soportar la caída de un proceso cliente que ni tenga ni haya pedido el testigo. El algoritmo de Ricart y Agrawala, tal y como ha quedado descrito, puede adaptarse para tolerar la rotura de un proceso haciendo que, de forma implícita, conceda todas las peticiones.

Se invita al lector a considerar cómo adaptar los algoritmos para tolerar fallos suponiendo que esté disponible un detector de fallos fiable. Incluso con un detector de fallos fiable se ha de tener cuidado para tomar en consideración un fallo en cualquier instante de tiempo (incluso durante el proceso de recuperación) y para reconstruir el estado de los procesos tras haberse detectado un fallo. Por ejemplo, en el algoritmo del servidor central, si el servidor falla entonces se debe establecer si él tiene el testigo o si lo tiene uno de los procesos cliente.

Se examinará en la Sección 11.5 el problema general de cómo los procesos deben coordinar sus acciones en presencia de fallos.

11.3. ELECCIONES

Un algoritmo para escoger un proceso único que juegue un papel específico se llama *algoritmo de elección*. Por ejemplo, en una variante de nuestro algoritmo de «servidor central» para exclusión mutua, el «servidor» se escoge entre los procesos p_i, $i = 1, 2 ..., N$ que necesitan usar la sección crítica. Se necesita un algoritmo de selección para escoger cuál de los procesos jugará el papel de servidor. Es esencial que todos los procesos estén de acuerdo en la elección. Tras ésta, si el proceso que juega el papel de servidor desea retirarse, entonces se requiere otro proceso de selección para escoger un sustituto.

Se dirá que un proceso *pide una elección* si lleva a cabo una acción que inicie una ejecución específica del algoritmo de elección. Un proceso individual no pide más de una elección cada vez, pero, en principio, los N procesos podrían pedir N elecciones concurrentes. Un proceso p_i, en cualquier instante de tiempo, es o bien un *participante* (lo que significa que está comprometido en alguna ejecución del algoritmo de elección) o bien, un *no participante* (lo que significa que no está comprometido actualmente en ninguna elección).

Un requisito importante es que la selección del proceso elegido sea única, incluso si distintos procesos piden elecciones de forma concurrente. Por ejemplo, dos procesos podrían decidir de forma independiente que un proceso coordinador ha fallado y pedir ambos elecciones.

Sin pérdida de generalidad, se requiere que el proceso elegido sea escogido como aquél con el mayor identificador. El »identificador» puede ser cualquier valor útil siempre que los identificadores sean únicos y estén totalmente ordenados. Por ejemplo, podríamos elegir el proceso con la menor carga computacional, teniendo que usar cada proceso $<1/carga, i>$ como su identificador, donde la $carga > 0$ y el índice del proceso i se usa para ordenar los identificadores con la misma carga.

Cada proceso p_i, ($i = 1, 2 ..., N$) posee una variable $elegido_i$, que contendrá el identificador del proceso elegido. Cuando el proceso se convierta en participante en una elección por primera vez, fijará la variable al valor especial «$\perp$» para señalar que no ha sido definido todavía.

Nuestros requisitos serán que durante una ejecución en particular del algoritmo:

E1: (seguridad)	Un proceso participante p_i tiene $elegido_i = \perp$ o $elegido_i = P$, donde P se elige como el proceso con el identificador mayor que no se ha caído al final de la ejecución.
E2: (pervivencia)	Todos los procesos p_i participan y, al final, fijan $elegido_i \neq \perp$; o bien se han caído.

Observe que puede haber procesos p_j que no son aún participantes y que almacenan en $elegido_j$ el identificador del proceso elegido anteriormente.

Se mide el rendimiento de un algoritmo de elección por su utilización del ancho de banda total de la red (que es proporcional al número total de mensajes enviados) y por el *tiempo de vuelta* (*turnaround*) para el algoritmo, número de veces de transmisión de mensajes serializados entre el comienzo y la finalización de una ejecución individual.

◇ **Un algoritmo de elección basado en anillo.** A continuación se presenta el algoritmo de Chang y Roberts [1979], que es apropiado para una colección de procesos dispuestos en un anillo lógico. En éste cada proceso p_i tiene un canal de comunicación con el siguiente proceso del anillo, $p_{(i+1) mod N}$, y todos los mensajes se mandan en el sentido de las agujas del reloj alrededor del anillo. Se supone que no ocurren fallos y que el sistema es asíncrono. El objetivo del algoritmo es elegir un proceso individual llamado el *coordinador*, que es el proceso con el identificador más grande.

Inicialmente, cada proceso se etiqueta como *no participante* en una elección. Cualquier proceso puede comenzar una elección. Para ello se marca a sí mismo como un *participante*, colocando

su identificador en un mensaje de *elección* y enviándolo hacia el próximo vecino en el sentido de las agujas del reloj.

Cuando un proceso recibe un mensaje de *elección*, compara el identificador que viene en el mensaje con el suyo. Si el identificador que le llega es mayor, entonces hace avanzar el mensaje hacia su vecino. Si el identificador que llega es más pequeño y el receptor es un *no participante*, entonces cambia el identificador del mensaje por el suyo y lo vuelve a enviar; si en este mismo caso, el receptor ya fuera un *participante*, no lo enviaría nuevamente. En cualquier caso, en el momento en el que se envía un mensaje de *elección*, el proceso se etiqueta a sí mismo como *participante*.

Sin embargo, si el identificador que se recibe es el del propio receptor, entonces ese identificador ha de ser el mayor y se convierte en el coordinador. El coordinador se señala a sí mismo como *no participante* una vez más y envía un mensaje de *elegido* a su vecino, anunciando su elección e incluyendo en el mensaje su identidad.

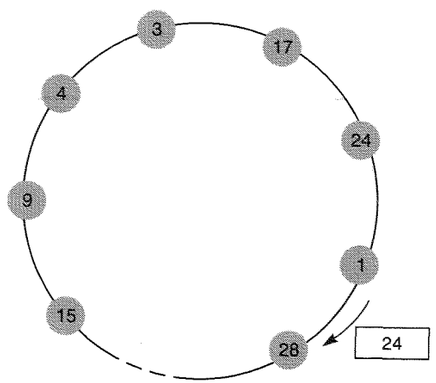
Cuando un proceso p_i recibe un mensaje de *elegido*, se marca a sí mismo como *no participante*, fija su variable $elegido_i$ al valor del identificador que hay en el mensaje y, a no ser que sea el nuevo coordinador, envía el mensaje a su vecino.

Se observa fácilmente que se cumple la condición E1. Se comparan todos los identificadores, ya que un proceso ha de recibir su propio identificador antes de enviar un mensaje de *elegido*. Para cualquier par de procesos, aquél con el identificador más grande no pasará el identificador del más pequeño. Por lo tanto, es imposible que ambos reciban sus propios identificadores de vuelta.

La condición E2 se obtiene de forma inmediata ya que se garantiza que se atraviesa el anillo (no hay fallos). Nótese cómo los estados *no participante* y *participante* se utilizan de tal manera que, si surgen mensajes cuando otro proceso comienza una elección al mismo tiempo, desaparecen tan pronto como sea posible y siempre antes de que el resultado de «ganador» de la elección haya sido anunciado.

Si un único proceso comienza una elección, entonces el caso de peor rendimiento ocurre cuando su vecino de al lado, en sentido anti-horario, tiene el identificador mayor. En ese caso, se necesitan $N - 1$ mensajes para llegar a ese vecino, que no anunciará su elección hasta que haya completado otra vuelta al anillo, dando lugar a N nuevos mensajes. Después, el mensaje *elegido* se envía N veces, totalizando $3N - 1$ mensajes. El tiempo necesario para completar la tarea es también $3N - 1$, ya que todos los mensajes se envían secuencialmente.

En la Figura 11.7 se muestra el funcionamiento de una elección basada en anillo. El mensaje de *elección* contiene 24 en ese momento, pero cuando llegue al proceso 28, éste lo reemplazará con su identificador.



Nota: La elección fue iniciada por el proceso 17. El identificador de proceso más elevado encontrado a continuación es el 24. Los procesos participantes están indicados con un tono más oscuro.

Figura 11.7. Progreso de una elección basada en anillo.

Aunque el algoritmo basado en anillo es útil para comprender las características de los algoritmos de elección en general, el hecho de que no tolere fallos hace que tenga escaso valor práctico. No obstante, si se dispone de un detector de fallos fiable es posible, en principio, rehacer el anillo cuando un proceso se cae.

◇ **El algoritmo abusón (*bully*).** El algoritmo del abusón [García-Molina 1982] permite la caída de procesos durante una elección, aunque supone que la entrega de mensajes entre procesos es fiable. A diferencia del algoritmo basado en anillo, este algoritmo supone que el sistema es síncrono, esto es, que utiliza timeouts para detectar un fallo en un proceso. Otra diferencia es que el algoritmo basado en anillo supone que los procesos tienen un conocimiento mínimo *a priori* de cada uno de los otros procesos: cada uno sabe cómo comunicarse únicamente con su vecino y ninguno conoce los identificadores de los otros procesos. Por otro lado, el algoritmo del abusón supone que cada proceso conoce qué procesos tienen identificadores mayores y que puede comunicarse con todos esos procesos.

Hay tres tipos de mensajes en este algoritmo. Un mensaje *de elección* se envía para anunciar un proceso de elección; un mensaje *de respuesta* se envía para responder a un mensaje de elección; y un mensaje *coordinador* se envía para anunciar la identidad del proceso elegido, el nuevo «coordinador». Un proceso comienza una elección cuando se da cuenta, a través de los timeouts, que el coordinador ha fallado. Además, varios procesos pueden descubrirlo de forma concurrente.

Dado que el sistema es síncrono, se puede construir un detector de fallos fiable. Existe un retraso máximo en la transmisión de un mensaje T_{trans} y un retraso máximo para el procesado de un mensaje $T_{procesado}$. Por lo tanto, podemos calcular un tiempo $T = 2T_{trans} + T_{procesado}$ que es el límite superior sobre el tiempo total transcurrido desde que se envió un mensaje a otro proceso hasta que se recibió la respuesta. Si no llega una respuesta dentro de ese tiempo T, entonces el detector de fallos local puede informar que el supuesto receptor de la petición ha fallado.

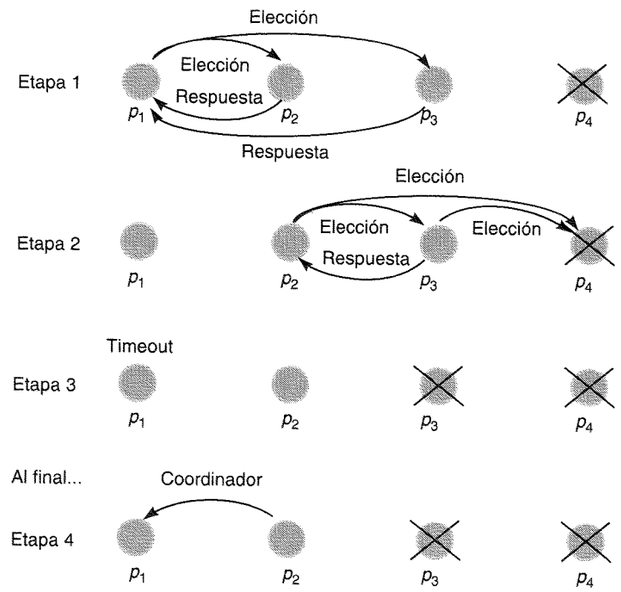
El proceso que sabe que posee el identificador más alto puede elegirse a sí mismo como el coordinador, simplemente enviando un mensaje *coordinador* a todos los procesos con identificadores más bajos. Por otro lado, un proceso con identificador más bajo comienza una elección con un mensaje *de elección* hacia aquellos procesos que tienen un identificador más alto y espera un mensaje *de respuesta*. Si no le llega ninguno dentro de un tiempo T, el proceso se considera a sí mismo el coordinador y envía un mensaje *coordinador* anunciándolo a todos los procesos con un identificador menor. Si ha recibido un mensaje *de respuesta*, el proceso espera otro período T' por que le llegue un mensaje *coordinador* procedente del nuevo coordinador. Si no llega ninguno, comienza otra elección.

Si un proceso p_i recibe un mensaje *coordinador*, fija su variable $elegido_i$ al identificador del coordinador que está contenido en el propio mensaje y trata a ese proceso como el coordinador.

Si un proceso recibe un mensaje *de elección*, devuelve un mensaje *de respuesta* y comienza otra elección a no ser que él ya haya comenzado una.

Si para sustituir a un proceso caído se lanza uno nuevo, éste comienza una elección. Si tiene el identificador de proceso más alto, entonces decidirá que es el coordinador y lo anunciará a otros procesos. Así se convertirá en el coordinador, aunque el coordinador actual siga funcionando. Ésta es la razón por la que a este algoritmo se le llama algoritmo «del abusón».

El funcionamiento del algoritmo se muestra en la Figura 11.8. Hay cuatro procesos $p_1 - p_4$. El proceso p_1 detecta el fallo del coordinador p_4 y anuncia una elección (etapa 1 en la figura). Cuando reciben el mensaje de elección proveniente de p_1, los procesos p_2 y p_3 mandan mensajes *de respuesta* a p_1 y comienzan sus propias elecciones; si bien p_3 envía un mensaje *de respuesta* a p_2, p_3 no recibe mensaje *de respuesta* del proceso fallido p_4 (etapa 2). Por lo tanto, p_3 decide que es el nuevo coordinador. Pero antes de que pueda enviar el mensaje *coordinador*, él también falla (etapa 3). Cuando el timeout T' de p_1 expira (suponemos que ocurre antes de que expire el tiempo



La elección del coordinador p_2 después del fallo de p_4 y p_3

Figura 11.8. El algoritmo del abusón.

límite de p_2) deduce la ausencia de un mensaje *coordinador* y comienza otra elección. Al final, p_2 es elegido coordinador (etapa 4).

Este algoritmo cumple claramente la condición de pervivencia E2, ya que supone que la entrega de mensajes es fiable. Y en el caso en que no se reemplace proceso alguno, entonces también cumple la condición E1. Es imposible que dos procesos decidan que ambos son el coordinador, dado que el proceso con el identificador menor descubrirá que el otro existe y cederá ante él.

Sin embargo, *no* se puede garantizar que el algoritmo cumpla la condición de seguridad E1 si los procesos que se han caído son reemplazados por procesos con el mismo identificador. Un proceso que reemplaza a otro proceso p caído puede decidir que posee el identificador más alto justo cuando otro proceso, que ha detectado la caída de p, ha decidido que posee el identificador más alto. Los dos procesos se anunciarán de forma concurrente como los coordinadores. Desgraciadamente, no existen garantías en el orden de entrega de los mensajes, y los receptores de los mensajes pueden alcanzar conclusiones diferentes sobre qué proceso es el coordinador.

Además, la condición E1 puede romperse si los valores de los tiempos límites que se han supuesto resultan ser imprecisos, o lo que es lo mismo, si el detector de fallos en los procesos no es fiable.

Volviendo al ejemplo anterior, supongamos que p_3 no había fallado pero estaba funcionando de una forma inusualmente lenta (dicho de otra forma, la suposición de que el sistema es síncrono es incorrecta) o bien que p_3 ha fallado y ha sido reemplazado. Justo cuando p_2 manda su mensaje *coordinador*, p_3 (o su sustituto) hace lo mismo. p_2 recibe el mensaje *coordinador* de p_3 cuando ya ha enviado el suyo y, por lo tanto, determina que $elegido_2 = p_3$. A continuación, debido a los retrasos variables en la transmisión de mensajes, p_1 recibe el mensaje *coordinador* de p_2 tras el mensaje de p_3 y al final decide fijar $elegido_1 = p_2$. Por lo tanto, la condición E1 no se ha cumplido.

En lo que respecta al rendimiento del algoritmo, en el mejor caso, el proceso con el segundo identificador más alto se da cuenta del fallo en el coordinador. En ese caso se puede elegir inmediatamente a sí mismo como coordinador y manda $N - 2$ mensajes de *coordinador*. El tiempo necesario para completar la tarea es de un mensaje. Sin embargo, el algoritmo del abusón requiere un número de mensajes de $O(N^2)$ en el peor caso (esto es, cuando el proceso con el identificador menor detecta en primer lugar el fallo en el coordinador). En ese caso serán $N - 1$ procesos quienes comiencen a la vez el proceso de elecciones, cada uno mandando mensajes a los procesos con los identificadores mayores.

11.4. COMUNICACIÓN POR MULTIDIFUSIÓN

La Sección 4.5.1 describía la multidifusión IP, que es una implementación de la comunicación en grupo. La comunicación en grupo, o multidifusión, requiere la presencia de coordinación y acuerdo. Su objetivo es que cada uno de los procesos de un grupo reciba los mensajes enviados al grupo, frecuentemente, con garantías de que han sido entregados. Estas garantías incluyen el acuerdo sobre el grupo de mensajes que todo proceso del grupo debería recibir y sobre el orden de entrega dentro de los miembros del grupo.

Los sistemas de comunicación en grupo son extremadamente sofisticados. Incluso la multidifusión IP, que proporciona unas garantías de entrega mínimas, requiere grandes esfuerzos de realización. Como preocupaciones principales están la eficiencia en el tiempo consumido y en el uso de ancho de banda, que suponen un reto incluso para grupos estáticos de procesos. Estos problemas se multiplican cuando los procesos pueden unirse o dejar los grupos de forma arbitraria.

Aquí se estudiará la multidifusión para grupos de procesos cuyos miembros son conocidos. El Capítulo 14 ampliará este estudio para la comunicación en grupo con todas sus variantes, incluyendo la gestión de grupos que varían de forma dinámica.

La comunicación por multidifusión ha sido el objeto de estudio de varios proyectos entre los que se incluyen el sistema V [Cheriton y Zwaenepoel 1985], Chorus [Rozier y otros 1988], Amoeba [Kaashoek y otros 1989, Kaashoek y Tanenbaum 1991], Trans/Total [Melliar-Smith y otros 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse y otros 1996], Totem [Moser y otros 1996] y Transis [Dolev y Malki 1996], junto con otros trabajos reseñables que se irán citando a lo largo de esta sección.

La característica esencial de la comunicación por multidifusión es que un proceso realiza solamente una operación *multicast* para enviar un mensaje a cada uno de los miembros de un grupo (en Java esta operación es *unSocket.send(unMensaje)*) en lugar de realizar múltiples operaciones *enviar* sobre los procesos individuales. La comunicación a *todos* los procesos de un sistema, en contraposición a un subgrupo de los mismos, se conoce como difusión (*broadcast*).

El uso de una única operación *multicast* en lugar de múltiples operaciones *enviar* implica mucho más que una ventaja para el programador. Permite una implementación eficiente y que ésta proporcione garantías de entrega más fuertes que las que serían posible obtener de otra forma.

Eficiencia: la certeza de que el mismo mensaje va a ser entregado a todos los procesos de un grupo permite a la implementación ser eficiente en el uso del ancho de banda. La implementación puede proceder de tal forma que sólo envíe el mensaje una vez sobre cada enlace de comunicación, mediante el envío de un mensaje sobre un árbol de distribución. Incluso puede valerse del hardware de la red que proporcione multidifusión allí donde esté disponible. Además dicha implementación puede también minimizar el tiempo total empleado en entregar el mensaje a todos los destinos, en lugar de transmitirlo de forma separada y en serie.

Para ver estas ventajas basta comparar la utilización del ancho de banda y el tiempo total de transmisión empleado cuando se envía el mismo mensaje desde un computador en Londres a dos computadores en la misma Ethernet en Palo Alto, en el caso (a) utilizando dos órdenes UDP *enviar*, y en el caso (b) mediante una única operación de multidifusión IP. En el primer

caso, se mandan dos copias del mensaje de forma independiente y la segunda se retrasa en función de la primera. En el segundo caso, un conjunto de encaminadores atentos a la multidifusión envía una única copia del mensaje desde Londres al encaminador en la LAN de destino. El encaminador final utiliza entonces hardware de multidifusión (proporcionado por la Ethernet) para entregar el mensaje a los dos destinos, en lugar de tener que enviarlo dos veces.

Garantías de entrega: si un proceso realiza múltiples operaciones *envía* independientes a procesos individuales, entonces la implementación no tiene medios para proporcionar garantías de entrega que afecten al grupo de procesos en su conjunto. Si el remitente falla en la mitad del proceso de envío, entonces algunos miembros del grupo recibirán el mensaje mientras que otros no lo recibirán. Además, no está definido el orden relativo entre dos mensajes entregados a dos miembros de un grupo. En el caso particular de la multidifusión IP, no se ofrecen garantías de ordenación ni de fiabilidad. Sin embargo, pueden proporcionarse mayores garantías de multidifusión y en breve definiremos algunas.

◇ **Modelo del sistema.** El sistema contiene una colección de procesos que pueden comunicarse entre ellos de forma fiable a través de canales uno-a-uno. Como se había supuesto antes, los procesos sólo pueden fallar por caída.

Dichos procesos son miembros de grupos, que son los destinos de los mensajes enviados en una operación *multicast*. Por lo general, es útil permitir que los procesos sean miembros de distintos grupos de forma simultánea; por ejemplo permitir a los procesos que reciban información de distintas fuentes mediante la inclusión en distintos grupos. Pero, para simplificar la discusión de las propiedades de ordenación, algunas veces se restringirá en cada instante la pertenencia de un proceso como máximo a un grupo.

La operación *multicast*(g, m) envía el mensaje m a todos los miembros del grupo de procesos g. A esta operación le corresponde una operación *entrega*(m) que entrega el mensaje mandado por multidifusión al proceso que ha realizado la petición. Utilizamos el término *entrega* en lugar de *recibe* para dejar claro que un mensaje multidifundido no siempre se entrega a la capa de aplicación dentro del proceso tan pronto como es recibido en el nodo del proceso. Esto será explicado brevemente cuando se discuta la semántica de la entrega por multidifusión.

Todo mensaje m porta el identificador único del proceso *emisor*(m) que lo envía y el identificador único del grupo al que se envía *grupo*(m). Se supone que los procesos no mienten sobre el origen o destino de sus mensajes.

Se dice que un grupo está *cerrado* si sólo los miembros del grupo pueden multidifundir dentro de él (véase la Figura 11.9). Un proceso dentro de un grupo cerrado se entrega a sí mismo cualquier mensaje que multidifunde al grupo. Un grupo se dice *abierto* si los procesos que no están en el grupo le pueden mandar mensajes (las categorías de «abierto» y «cerrado» también se aplican con significados análogos a las listas de correo). Los grupos cerrados de procesos son útiles, por ejemplo, entre servidores cooperantes para enviar mensajes entre ellos y que sólo ellos deberían recibir. Los grupos abiertos son útiles, por ejemplo, para informar de eventos a grupos de procesos interesados.

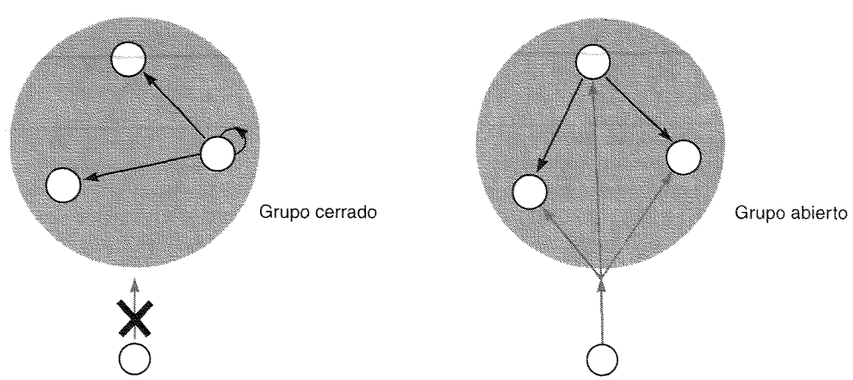


Figura 11.9. Grupos abiertos y cerrados.

Algunos algoritmos suponen que los grupos son cerrados. El mismo efecto que proporciona la apertura se puede conseguir con un grupo cerrado escogiendo a un miembro del grupo y enviándole un mensaje (uno-a-uno) para que multidifunda al resto de su grupo. Rodrigues y otros [1998] discuten la multidifusión para grupos abiertos.

11.4.1. MULTIDIFUSIÓN BÁSICA

Es útil tener a nuestra disposición una primitiva de multidifusión básica que garantice, a diferencia de la multidifusión IP, que un proceso correcto al final entregará el mensaje siempre que el sistema que realiza la multidifusión no se caiga. La primitiva se llamará *B-multicast* y su correspondiente primitiva de entrega básica se llamará *B-entrega*. Se permitirá a los procesos que pertenezcan a varios grupos y que cada mensaje se destine a un grupo en particular.

Una forma directa de implementar el *B-multicast* es utilizar una operación *envia* fiable uno-a-uno de la siguiente forma:

Para realizar *B-multicast*(g, m): para cada proceso $p \in g$, *g*, *envia*(p, m);

Al *recibir*(m) en p : *B-entrega*(m) en p.

La implementación puede usar hilos para realizar las operaciones *envía* de forma concurrente, con el fin de reducir el tiempo total empleado para entregar el mensaje. Desgraciadamente, tal implementación es susceptible de sufrir la denominada *ack-implosion* (colapso por exceso de acuses de recibo) si el número de procesos es grande. Es factible que los acuses de recibo, enviados como parte de una operación *envía* fiable, lleguen aproximadamente al mismo tiempo desde muchos procesos. Los búferes del proceso que están realizando la multidifusión se llenarán rápidamente y es probable que pierdan algún acuse de recibo. En ese caso retransmitiría el mensaje, dando lugar todavía a más acuses de recibo y un mayor desperdicio del ancho de banda. Puede construirse un servicio básico de multidifusión más práctico usando la multidifusión IP y se deja como ejercicio para el lector.

11.4.2. MULTIDIFUSIÓN FIABLE

La Sección 2.3.2 definió los canales de comunicación fiables uno-a-uno entre pares de procesos. La propiedad de seguridad requerida se denominaba *integridad*, esto es, que todo mensaje entregado es idéntico al enviado y que ningún mensaje se entrega dos veces. La propiedad de pervivencia requerida se denominaba *validez*, esto es, que todo mensaje al final sea entregado en su destino, si éste es correcto.

De forma similar a Hadzilacos y Toueg [1996], se define a continuación una *multidifusión fiable* con sus operaciones correspondientes *F-multicast* y *F-entrega*. Queda claro que sería muy conveniente disponer de propiedades análogas a la integridad y a la validez para la entrega en la operación de multidifusión. No obstante, se añade otro requisito: *todos* los procesos correctos en un grupo deben recibir un mensaje si *alguno* de ellos lo recibe. Es importante darse cuenta de que esta propiedad no existe en el algoritmo de *B-multicast* que se basa en una operación fiable de envío uno-a-uno. El emisor puede fallar en cualquier instante del *B-multicast*, por lo tanto algunos procesos pueden recibir un mensaje mientras que otros no.

Un proceso de multidifusión fiable será aquel que satisfaga las siguientes propiedades:

Integridad: un proceso correcto p entrega un mensaje m a lo sumo una vez. Además, $p \in grupo(m)$ y m fue proporcionado a la operación *multicast* por un *emisor*(m) (de la misma forma que con la comunicación uno-a-uno, los mensajes pueden distinguirse por un número de secuencia propio al emisor).

Validez: si un proceso correcto multidifunde un mensaje m, entonces al final m será entregado.

Acuerdo: si un proceso correcto entrega un mensaje m, entonces el resto de procesos correctos en el *grupo*(m) deben al final entregar el mensaje m.

Una vez definidas estas propiedades se procede a explicarlas con más detalle.

La propiedad de integridad es análoga a la de comunicación fiable uno-a-uno. La propiedad de validez garantiza la pervivencia para el emisor. Ésta podría parecer una propiedad poco habitual, ya que es asimétrica (sólo se menciona un proceso en concreto). Pero hay que darse cuenta de que las propiedades de validez y acuerdo conjuntamente permiten obtener el requisito de pervivencia global; si un proceso, *el emisor*, finalmente entrega un mensaje m entonces, puesto que los procesos correctos están de acuerdo en el conjunto de mensajes que entregan, de lo que se deduce que m será entregado, finalmente, a todos los miembros correctos del grupo.

La ventaja de expresar la condición de validez en términos de auto-entrega es la sencillez. Lo que se requiere es que el mensaje sea entregado en último caso por *algún* miembro correcto del grupo.

La condición de acuerdo está relacionada con la atomicidad, que es la propiedad de «todo o nada» aplicada a la entrega de mensajes a un grupo. Si un proceso que está enviando un mensaje mediante multidifusión se cae antes de haberlo entregado, entonces es posible que el mensaje no se entregue a ningún proceso del grupo; pero si lo ha entregado a algún proceso correcto, entonces el resto de procesos correctos lo entregarán. Muchos trabajos en la literatura existente usan el término «atómico» para incluir una condición de ordenación total, que se define a continuación.

◇ **Implementación de la multidifusión fiable sobre *B-multicast*.** La Figura 11.10 proporciona un algoritmo de multidifusión fiable con las primitivas *F-multicast* y *F-entrega*, que permiten a los procesos pertenecer a distintos grupos cerrados de forma simultánea. Para enviar un mensaje mediante *F-multicast*, un proceso envía el mensaje mediante *B-multicast* a los procesos miembros del grupo (incluido él mismo). Cuando el proceso ha sido *B-entrega*do, el receptor, a cambio, realiza *B-multicast* del mensaje al grupo (si no es el emisor original) y, a continuación, *F-entrega* el mensaje. Ya que un mensaje puede llegar más de una vez a cualquier nodo, los duplicados del mensaje se detectan y no se reenvían.

```
En la inicialización
   Recibido := {};

Para que el proceso p realice R-multicast de un mensaje m a un grupo g
   B-multicast(g, m);          // p ∈ g se incluye como destino

En B-entrega(m) a un proceso q con g = grupo(m)
   si (m ∉ Recibido)
   entonces
            Recibido := Recibido ∪ {m};
            si (q ≠ p) entonces B-multicast(g, m); fin si;
            F-entrega m;
   fin si
```

Figura 11.10. Algoritmo de multidifusión fiable.

Este algoritmo claramente satisface el requisito de validez, ya que un proceso correcto realizará en último caso una *B-entrega* a sí mismo. Además, por la propiedad de integridad de los canales de comunicación subyacentes que se usan en los envíos *B-multicast*, el algoritmo también satisface la propiedad de integridad.

El acuerdo se consigue a partir del hecho de que todo proceso correcto envía mediante *B-multicast* el mensaje a otros procesos una vez que lo ha *B-entrega*do. Si un proceso correcto no realiza la *F-entrega* del mensaje, entonces esto se debe a que nunca ha realizado la *B-entrega*. Esto, a su vez, puede deberse solamente a que ningún otro proceso correcto lo ha *B-entrega*do antes; por lo tanto, ninguno lo *B-entrega*rá.

El algoritmo de multidifusión fiable que se ha descrito es correcto en un sistema asíncrono, ya que no se han realizado suposiciones sobre las temporizaciones. Sin embargo, el algoritmo es ineficiente en la práctica, ya que cada mensaje se envía $|g|$ veces a cada proceso.

◇ **Multidifusión fiable sobre multidifusión IP.** Una alternativa para realizar *F-multicast* es combinar la multidifusión IP, acuses de recibo adheridos (*piggy backed*) (esto es, acuses de recibo adjuntos a otros mensajes), y acuses de recibo negativos. Este protocolo para *F-multicast* se basa en la observación de que la comunicación por multidifusión IP casi siempre tiene éxito. En el protocolo, los procesos no envían mensajes separados de acuse de recibo; en su lugar adhieren los reconocimentos en los mensajes que envían al grupo. Los procesos envían un mensaje de respuesta por separado sólo cuando detectan que han perdido un mensaje. La respuesta indicando la ausencia de un mensaje esperado se conoce como *acuse de recibo negativo*.

La descripción supone que los grupos son cerrados. Cada proceso p mantiene un número de secuencia S_q^p para cada grupo g al que pertenece. El número de secuencia es inicialmente cero.

Cada proceso almacena además R_g^p, el número de secuencia del último mensaje que ha sido entregado por el proceso p y que fue enviado desde el grupo g.

Cuando el proceso p emplea *F-multicast* para un mensaje al grupo g, adhiere a dicho mensaje el valor S_g^p. Además adhiere acuses de recibo sobre el mensaje que envía, de la forma $<q, R_g^p>$. Dicho acuse de recibo establece el número de secuencia del último mensaje destinado para ese grupo, que le ha sido entregado desde el proceso q desde la última vez que se multidifundió un mensaje. El proceso p que empleó la multidifusión envía a continuación el mensaje mediante multidifusión IP con su número de secuencia y con acuse de recibo adheridos para g, e incrementa S_q^p en uno.

Un proceso *F-entrega* un mensaje destinado para g con el número de secuencia S y proveniente de p si y sólo si $S = R_g^p + 1$, e incrementa R_g^p en uno inmediatamente tras la entrega. Además, retiene cualquier mensaje que no haya podido entregar en una *cola de retención* (véase la Figura 11.11); tales colas se necesitan con frecuencia para que se cumplan las garantías en la entrega de mensajes. Si, por otro lado, un mensaje que llega tiene $S \leqslant R_g^p$, entonces r ha entregado el mensaje antes y lo descarta. Si $S > R_g^p + 1$ o $R > R_g^q$, para cualquier acuse de recibo $<q, R>$ incluido en otro mensaje, entonces r ha perdido uno o más mensajes. En ese momento los solicita mediante el envío de acuses de recibo negativos. Puede enviar la petición al proceso del cual percibió la omisión o al emisor original, si es distinto. En algunas variaciones del protocolo, aquél realiza la petición mediante multidifusión a los procesos de su red vecina, por si acaso lo hubieran recibido.

La propiedad de integridad se consigue con la detección de duplicados y las propiedades subyacentes de la multidifusión IP (que utiliza sumas de comprobación para eliminar mensajes corruptos). La propiedad de validez se cumple porque la multidifusión IP tiene dicha propiedad. Por acuerdo necesitamos, primero, que un proceso siempre pueda detectar mensajes perdidos. Esto a cambio significa que siempre recibirá un mensaje posterior que le permite detectar la omisión. Como establece este protocolo simplificado, garantizamos la detección de los mensajes perdidos sólo en el caso en que se suponga que cada proceso multidifunda de forma indefinida. Segundo, la

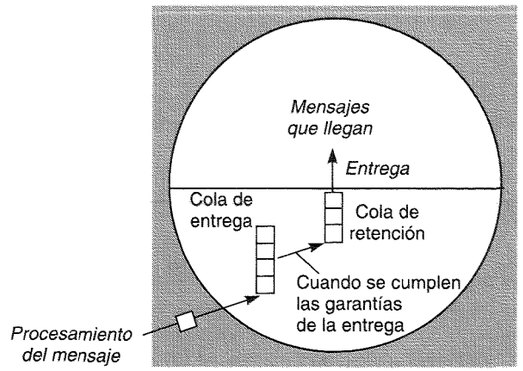


Figura 11.11. Cola de retención para la llegada de mensajes de multidifusión.

propiedad de acuerdo requiere que se garantice que haya una copia disponible para un proceso que no lo haya recibido. Por lo tanto, se supone que los procesos mantienen copias, de forma indefinida, de los mensajes que han enviado.

Ninguna de las suposiciones hechas para asegurar el acuerdo es práctica. Sin embargo, el acuerdo se consigue en la práctica en los protocolos de los cuales se deriva éste: el protocolo Psync [Peterson y otros 1989], el protocolo Trans [Melliar-Smith y otros 1990] y el protocolo de multidifusión fiable y ampliable [Floyd y cols. 1997]. Psync y Trans garantizan, además, una entrega ordenada.

◇ **Propiedades uniformes.** La definición de acuerdo antes mencionada sólo se refiere al comportamiento de los procesos *correctos*, que nunca fallan. Considérese qué ocurriría en el algoritmo de la Figura 11.10 si un proceso no es correcto y se cae tras haber *F-entrega*do un mensaje. Teniendo en cuenta que cualquier proceso que *F-entrega* el mensaje ha de haber realizado previamente un *B-multicast* del mismo, se concluye que todos los procesos correctos acabarían entregando el mensaje.

Cualquier propiedad que se cumpla tanto si los procesos son correctos como si no se denomina propiedad *uniforme*. Se define acuerdo uniforme de la siguiente forma:

> *Acuerdo uniforme*: si un proceso, correcto o con fallo, entrega un mensaje m, entonces todos los procesos correctos en el *grupo*(m) entregarán finalmente m.

El acuerdo uniforme permite que un proceso tras entregar un mensaje, y al mismo tiempo asegura que todos los procesos correctos entregarán el mensaje. Ya se discutió que el algoritmo de la Figura 11.10 satisface esta propiedad, que es más fuerte que la propiedad de acuerdo no uniforme definida con anterioridad.

El acuerdo uniforme es útil en aquellas aplicaciones donde un proceso puede realizar una acción que produzca una inconsistencia observable justo antes de caerse. Por ejemplo, considérese el caso en el que los procesos son servidores que gestionan las copias de una cuenta bancaria y que las actualizaciones en la cuenta se mandan al grupo de servidores utilizando multidifusión fiable. Si la multidifusión no satisface la propiedad de acuerdo uniforme, un cliente que acceda a un servidor justo antes de que se caiga puede observar una actualización que ningún otro servidor procesará.

Es importante resaltar que si se invierte el orden de las líneas *F-entrega m* y *si* ($q \neq p$) *entonces B-multicast*(g, m); *fin si* en la Figura 11.10, entonces el algoritmo resultante no cumple la propiedad de acuerdo uniforme.

De la misma forma que hay una versión uniforme de acuerdo, hay versiones uniformes del resto de propiedades de la multidifusión, incluyendo validez e integridad además de las propiedades de ordenación que se definen a continuación.

11.4.3. MULTIDIFUSIÓN ORDENADA

El algoritmo de multidifusión básico de la Sección 11.4.1 entrega los mensajes a los procesos en un orden arbitrario, debido a los retrasos arbitrarios asociados a las operaciones subyacentes de envío uno-a-uno. Esta falta de la propiedad de ordenación no es admisible en muchas aplicaciones. Por ejemplo, en una planta nuclear es importante que los eventos asociados a amenazas en las condiciones de seguridad y los eventos que implican acciones por parte de las unidades de control sean observados en el mismo orden por todos los procesos del sistema.

Los requisitos de ordenación más frecuentes son la ordenación total, la ordenación causal, la ordenación FIFO y las ordenaciones híbridas total-causal y total-FIFO. Para simplificar la exposición se definen a continuación estas ordenaciones bajo la suposición de que cualquier proceso pertenece a lo sumo a un grupo. Más adelante se discutirán las implicaciones de permitir que los grupos se solapen.

> *Ordenación FIFO*: si un proceso correcto realiza un *multicast*(g, m) y a continuación un *multicast*(g, m'), entonces todo proceso correcto que entregue m' ha de haber entregado previamente m.
>
> *Ordenación causal*: si *multicast*(g, m) $\rightarrow$ *multicast*(g, m'), donde $\rightarrow$ indica la relación sucedió-antes inducida por los mensajes enviados solamente entre los miembros de g, entonces cualquier proceso correcto que entregue m' habrá entregado antes m.
>
> *Ordenación total*: si un proceso correcto entrega el mensaje m antes de que entregue m', entonces cualquier otro proceso que entregue m' ha de haber entregado m antes.

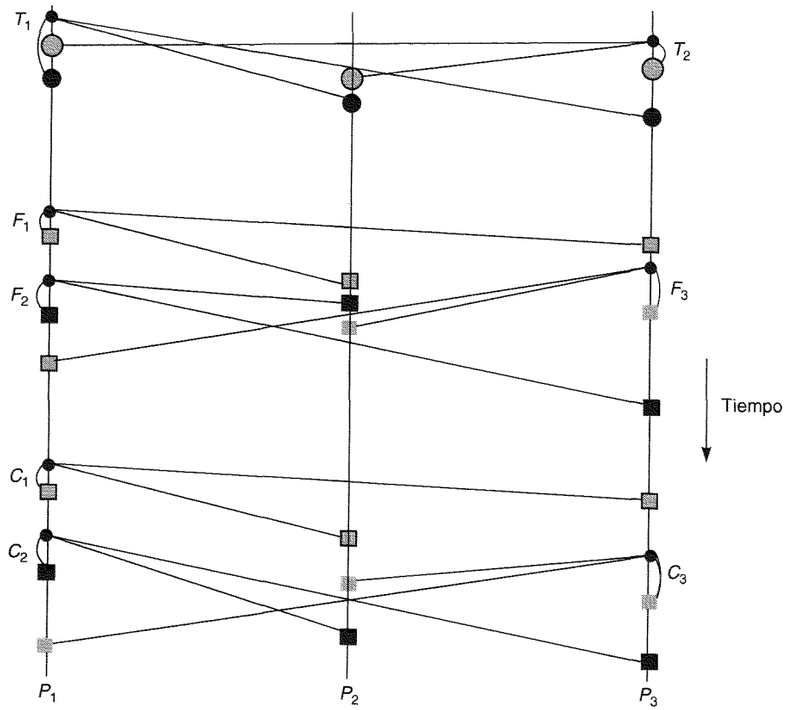
La ordenación causal implica una ordenación FIFO, ya que dos envíos *multicast* cualesquiera dentro del mismo proceso están relacionados en la forma sucedió-antes. Hay que resaltar que tanto la ordenación FIFO como la ordenación causal son ordenaciones parciales: de forma general, no todos los mensajes son enviados por el mismo proceso; de forma similar, algunas multidifusiones son concurrentes (y, por lo tanto, no ordenadas mediante sucedió-antes).

La Figura 11.12 ilustra las ordenaciones en el caso en que existan tres procesos. Un análisis en profundidad de la figura muestra que los mensajes totalmente ordenados se entregan en el orden contrario al tiempo físico en el que fueron enviados. De hecho, la definición de ordenación total permite que la entrega de mensajes se ordene de manera arbitraria, ya que el orden es el mismo en procesos diferentes. Además, dado que la ordenación total no es necesariamente ni una ordenación FIFO ni una ordenación causal, se define la ordenación híbrida *FIFO-total* como aquella en la que la entrega de mensajes obedece a ambos tipos de ordenación; de forma parecida, cuando existe una entrega de mensajes con ordenación *causal-total* se obedecen ambos tipos de ordenaciones.

Las definiciones de multidifusión ordenada ni suponen ni implican fiabilidad. Por ejemplo, el lector podría comprobar que bajo la hipótesis de ordenación total, si un proceso p entrega un mensaje m y a continuación entrega un mensaje m', un proceso correcto q puede entregar m sin necesidad de entregar m' o cualquier otro mensaje ordenado tras m.

También se pueden formar protocolos híbridos ordenados y fiables. Una multidifusión totalmente ordenada fiable suele denominarse en la literatura como un *multidifusión atómica*. De forma similar se pueden conseguir multidifusión FIFO fiable, multidifusión causal fiable y versiones fiables de los multidifusión con ordenación híbrida.

La ordenación en la entrega de mensajes por multidifusión, tal y como se verá, puede ser costosa en términos de latencia en la entrega y consumo de ancho de banda. La semántica asociada a las



Obsérvese la ordenación consistente de los mensajes con ordenación total T_1 y T_2, los mensajes relacionados por criterio FIFO. $F_1 > F_2$, y los mensajes relacionados casualmente C_1 y C_2; y en el resto de casos, entrega de mensajes con ordenación.

Figura 11.12. Ordenaciones total, FIFO y causal de mensajes de multidifusión.

ordenaciones que se han descrito puede retrasar la entrega de mensajes de forma innecesaria. Esto es, desde el punto de vista de la aplicación, un mensaje puede verse retrasado por otro del cual no depende. Por esta razón, algunos autores han propuesto sistemas de multidifusión que utilizan semánticas de los mensajes específicos de la aplicación para determinar el orden en la entrega de mensajes [Cheriton y Skeen 1993, Pedone y Schiper 1999].

◇ **El ejemplo del tablón de anuncios.** Para concretar más la semántica de la entrega de mensajes considérese una aplicación en la cual los usuarios ponen mensajes en tablones de anuncios. Cada usuario ejecuta un proceso que es una aplicación tablón de anuncios. Cada tema de discusión tiene su propio grupo de procesos. Cuando un usuario manda un mensaje al tablón, la aplicación multidifunde dicho mensaje al resto del grupo del tema en el que él o ella están interesados, de tal forma que el usuario sólo recibirá los anuncios concernientes a ese tema.

Se necesita multidifusión fiable si todo usuario ha de recibir al final cada anuncio. Los usuarios también imponen requisitos de ordenación. La Figura 11.13 muestra los anuncios tal y como le aparecen a un usuario en particular. Como mínimo, se desea tener una ordenación FIFO, de tal forma que cualquier anuncio para un usuario (por ejemplo, *A. Pérez*) se recibirá en el mismo orden y los usuarios pueden hablar de forma consistente acerca del segundo mensaje de A. Pérez.

El lector debe fijarse en que los mensajes cuyos asuntos son *Re*: *Microkernels* (25) y *Re*: *Mach* (27) aparecen tras los mensajes a los que se refieren. Por lo tanto, se necesita una multidifusión

Tablón de anuncios: *os.interesante*		
Ítem	De	Asunto
23	A. Pérez	Mach
24	G. Mayor	Microkernels
25	A. Pérez	Re: Microkernels
26	T. L. Heureux	RPC performance
27	M. Walker	Re: Mach
Fin		

Figura 11.13. Pantalla de un programa tablón de anuncios.

ordenada causalmente para garantizar estas relaciones. De otra forma, retrasos arbitrarios en los mensajes pueden ocasionar que un mensaje *Re*: *Mach* aparezca antes que el mensaje original concerniente a Mach.

Si la multidifusión estuviese totalmente ordenada, entonces la numeración en la columna de la izquierda sería consistente entre los usuarios. En ese caso, los usuarios podrían referirse sin ambigüedad al *mensaje* 24.

En la práctica, el sistema del tablón de anuncios de USENET no implementa ni ordenación causal ni total. Los costes asociados a la comunicación para conseguir estas ordenaciones a gran escala exceden ampliamente las ventajas que aportan.

◇ **Implementación de una ordenación FIFO.** Para conseguir una multidifusión con ordenación FIFO (que tendrá las operaciones *OF-multicast* y *OF-entrega*) se utilizan secuencias de números, de forma similar a como se conseguiría en una comunicación uno-a-uno. Se considerarán sólo grupos que no se solapen. Se deja como ejercicio para el lector que verifique que el protocolo de multidifusión fiable que se definió sobre la base de la multidifusión IP en la Sección 11.4.2, además garantiza la ordenación FIFO. Ahora se mostrará cómo construir multidifusión con ordenación FIFO sobre la base de cualquier multidifusión. Se usarán las variables S_g^p y R_g^q asociadas al proceso p como se hacía en el protocolo de multidifusión fiable de la Sección 11.4.2: S_g^p es un contador del número de mensajes que p ha enviado al grupo g, y para cada q, R_g^q es el número de secuencia del último mensaje que p ha entregado, partiendo del proceso q y que fue enviado al grupo g.

Cuando p quiere enviar un mensaje mediante *OF-multicast* al grupo g, adhiere en el mensaje el valor S_g^p, realiza el *B-multicast* del mensaje al grupo g e incrementa S_g^p en 1. Cuando p recibe un mensaje de q que lleva el número de secuencia S, comprueba si $S = R_g^q + 1$. Si es así, ése era el mensaje esperado por parte de q y p usa *OF-entrega*, fijando $R_g^q := S$. Si $S > R_g^q + 1$, retiene el mensaje en la cola hasta que los mensajes intermedios hayan sido entregados y $S = R_g^q + 1$.

Dado que todos los mensajes enviados por un emisor se entregan siguiendo la misma secuencia, y dado que se pospone la entrega de un mensaje hasta que se alcance su número de secuencia, se cumple claramente la condición de ordenación FIFO. Sin embargo, esto sólo ocurre si se cumple la condición de grupos que no se solapan.

En este protocolo podría usarse cualquier implementación de *B-multicast*. Además, si se usase una primitiva *F-multicast* en lugar de *B-multicast*, se obtendría una multidifusión FIFO fiable.

◇ **Implementación de la ordenación total.** La manera básica de implementar la ordenación total es asignar identificadores totalmente ordenados a los mensajes que se multidifunden de tal forma que todo proceso realice la misma ordenación basada en esos identificadores. El algoritmo de entrega sería muy similar al descrito para la ordenación FIFO, la diferencia sería que los procesos almacenarían números de secuencia específicos para cada grupo en lugar de números específicos de secuencia para cada proceso. Aquí se considerará sólo cómo ordenar totalmente mensajes

1. Algoritmo para el miembro p del grupo

En la inicialización: $r_g := 0$;

Para hacer OT-multicast de un mensaje m al grupo g
  Hacer B-multicast ($g \cup \{secuenciador(g)\}$, $<m, i>$);

En B-entrega ($<m, i>$) *con* $g = grupo(m)$
  Coloca $<m, i>$ en la cola de retención;

En B-entrega (m_{orden} <"orden", i, $S>$) *con* $g = grupo(m)$
  Espera hasta que $<m, i>$ esté en la cola de retención y $S = r_g + 1$;
  OT-entrega m; // (tras eliminarlo de la cola de retención)
  $r_g := S$;

2. Algoritmo para el secuenciador de g

En la inicialización: $s_g := 0$;

En B-entrega($<m, i>$) *con* $g = grupo(m)$
  Haz B-multicast(g, <"orden", i, $s_g>$);
  $s_g := s_g + 1$;

Figura 11.14. Ordenación total que usa un secuenciador.

que se envían a grupos que no se solapan. Las operaciones de multidifusión se denominarán *OT-multicast* y *OT-entrega*.

Se discuten fundamentalmente dos métodos para asignar los identificadores a los mensajes. El primero consiste en que un proceso denominado *secuenciador* los asigne (véase la Figura 11.14). Cuando un proceso desea enviar un mensaje m con *OT-multicast* a un grupo g adjunta un identificador único $id(m)$ al mismo. Los mensajes destinados a g se envían al secuenciador para g, *secuenciador*(g), al igual que a los miembros de g. (El secuenciador puede ser, incluso, un miembro de g.) El proceso *secuenciador*(g) guarda un número de secuencia específico para el grupo S_g, que utiliza para asignar números de forma creciente y consecutiva a los mensajes que *B-entrega*. El secuenciador anuncia la secuencia de números mediante *B-multicast* al grupo g de mensajes con el *ordenamiento* (para obtener más detalles véase la Figura 11.14).

Un mensaje permanecerá retenido en la cola indefinidamente hasta que pueda efectuarse *OT-entrega* de acuerdo con el correspondiente número de secuencia. Partiendo de que la secuencia de números está bien definida (ya que viene dada por el secuenciador), se observa que se cumple el criterio de orden total. Además, si los procesos utilizasen una variante de ordenación FIFO del *B-multicast*, entonces además de estar totalmente ordenado está también causalmente ordenado. Esta demostración se deja para el lector.

El problema obvio asociado a un esquema basado en un secuenciador es que éste puede convertirse en un cuello de botella y además es un punto de fallo crítico. Existen algoritmos que en la práctica solucionan el problema del fallo. Chang y Maxemchuk [1984] fueron los primeros en sugerir un protocolo de multidifusión que utilizaba un secuenciador (lo que denominaron *lugar del testigo*). Kaashoek y otros [1989] desarrollaron un protocolo basado en un secuenciador para el sistema Amoeba. Estos protocolos aseguran que un mensaje está en la cola de retención en $f + 1$ nodos antes de ser entregado; de esta forma se pueden tolerar f fallos. Al igual que Chang y Maxemchuck, Birman y otros [1991] también utilizan un sitio que retiene el testigo y que actúa como secuenciador. El testigo se puede pasar de unos procesos a otros, de tal forma que sólo un proceso multidifunde todos los mensajes de forma totalmente ordenada dicho proceso puede actuar como un secuenciador, ahorrando el proceso de comunicación.

El protocolo propuesto por Kaashoek y otros usa multidifusión implementada vía hardware (que está disponible, por ejemplo, en Ethernet) en lugar de usar una comunicación fiable punto-a-

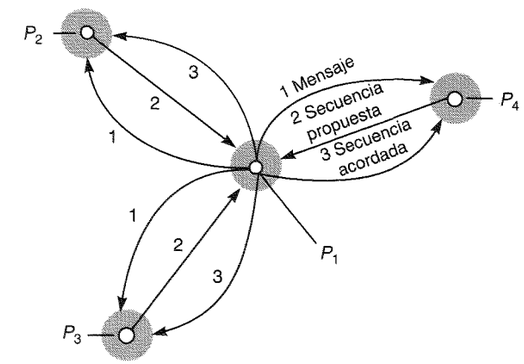


Figura 11.15. El algoritmo ISIS para ordenación total.

punto. En la versión más simple de su protocolo los procesos envían sus mensajes al secuenciador para que realice la multidifusión, y esta comunicación la realizan uno-a-uno. El secuenciador multidifunde junto con el identificador y el número de secuencia. Esto presenta la ventaja de que los otros miembros del grupo reciben sólo un mensaje mediante multidifusión; el inconveniente radica en el incremento en el uso del ancho de banda. El protocolo se encuentra descrito en su totalidad en www.cdk3.net/coordination.

El segundo método examinado para conseguir multidifusión totalmente ordenada es aquel en el que los procesos se ponen de acuerdo de una forma distribuida en la asignación de números de secuencia. En la Figura 11.15 se muestra un algoritmo simple, similar al que fue desarrollado originalmente para implementar la entrega en la multidifusión totalmente ordenada para la herramienta ISIS [Birman y Joseph 1987a]. Una vez más, un proceso envía un mensaje mediante *B-multicast* a los miembros del grupo. Éste puede ser abierto o cerrado. Los procesos que reciben los mensajes proponen números de secuencia según les van llegando y se los devuelven al emisor, que los usa para generar números de secuencia *acordados*.

Cada proceso q en el grupo g mantiene A_g^q, que es el mayor número de secuencia acordado que se ha observado hasta el momento para el grupo g y P_g^q, que es su mayor número de secuencia propuesto. El algoritmo que sigue cada proceso p para multidifundir un mensaje m al grupo g es el siguiente:

1. p usa *B-multicast* $<m, i>$ a g, donde i es el identificador único para m.
2. Cada proceso q responde al emisor p con una propuesta del número de secuencia acordado como $P_g^q := Máx(A_g^q, P_g^q) + 1$. En realidad, habría que incluir los identificadores de procesos en los valores propuestos P_g^q para asegurar el orden total, ya que de otra forma, distintos procesos podrían proponer el mismo valor para mensajes distintos; sin embargo, para facilitar la notación, no se mencionará de forma explícita. Cada proceso asigna al mensaje de forma provisional el número de secuencia propuesto y lo almacena en la cola de retención, que se ordena con el número de secuencia *menor* al comienzo.
3. p recoge todos los números de secuencia propuestos y selecciona el mayor a como el siguiente número de secuencia acordado. A continuación emplea *B-multicast* $<i, a>$ para g. Cada proceso q en g fija $A_g^q := Máx(A_g^q, a)$ y adjunta a al mensaje (que es identificado mediante i). Además reordena el mensaje en la cola de retención si el número de secuencia acordado difiere del propuesto. Cuando al mensaje al principio de la cola de retención se le haya asignado su número de secuencia acordado, se transfiere al final de la cola de entre-

ga. Por el contrario, no se transfieren aún aquellos mensajes a los que ya se les ha asignado su número de secuencia acordado, pero no están al comienzo de la cola de retención.

Si todos los procesos acuerdan el mismo conjunto de números de secuencia y los entregan en el orden correspondiente, se cumple el orden total. Está claro que los procesos correctos al final acuerdan el mismo conjunto de número de secuencia, pero hay que mostrar que éstos se incrementan de forma monotónica y que ningún proceso correcto puede anticiparse al entregar un mensaje.

Supongamos que a un mensaje m_1 se le ha asignado un número de secuencia acordado y ha alcanzado el inicio de la cola de retención. Por la forma en que se construye, un mensaje recibido tras esta etapa tiene que ser y es entregado después de m_1; tendrá un número de secuencia propuesto mayor, y por lo tanto un número de secuencia acordado mayor que m_1. De esta forma, sea m_2 cualquier otro mensaje al que no se le haya asignado su número de secuencia acordado, pero que esté en la misma cola. Se tiene que:

$$secuenciaAcordada(m_2) \geqslant secuenciaPropuesta(m_2)$$

en función del algoritmo utilizado. Y dado que m_1 está al inicio de la cola:

$$secuenciaPropuesta(m_2) > secuenciaAcordada(m_1)$$

En consecuencia,

$$secuenciaAcordada(m_2) > secuenciaAcordada(m_1)$$

y se asegura la ordenación total.

Este algoritmo tiene mayor tiempo de latencia que el *multicast* basado en secuenciador: se mandan tres mensajes en serie entre el emisor y el grupo antes de que se entregue el mensaje.

Hay que resaltar que la ordenación total elegida por este algoritmo no garantiza ordenación causal ni FIFO: dos mensajes cualesquiera se entregan en un orden total arbitrario, influido por los retrasos en las comunicaciones.

Melliar-Smith y otros [1990], García-Molina y Spauster [1991] y Hadzilacos y Toueg [1994] presentan otras aproximaciones para implementar una ordenación total.

◇ **Implementación de la ordenación causal.** La Figura 11.16 ofrece un algoritmo para grupos cerrados que no se solapen, que está basado en el propuesto por Birman y otros [1991], y en el que se introducen las operaciones de multidifusión ordenadas causalmente: *OC-multicast* y *OC-entrega*. El algoritmo tiene en cuenta la relación sucedió-antes tal y como la establecen los mensajes multidifusión. Si los procesos intercambian mensajes uno-a-uno, estos mensajes no se tienen en cuenta.

Algoritmo para un miembro del grupo p_i (i = 1, 2, ..., N)

En la inicialización
  $V_i^g[j] := 0$ (j = 1, 2, ..., N);

Para hacer OC-multicast de un mensaje m al grupo g
  $V_i^g[i] := V_i^g[i] + 1$;
  Hacer B-multicast(g, $< V^{gi}, m >$);

En B-entrega($< V_j^g, m >$) *que viene de* p_j, *con* g = *grupo*(m)
  Colocar $< V_j^g, m >$ en la cola de retención;
  Esperar hasta que $V_j^g[j] = V_i^g[j] + 1$ y $V_j^g[k] \leqslant V_i^g[k]$ ($k \neq j$);
  OC-entregar m; // tras eliminarlo de la cola de retención
  $V_i^g[j] := V_i^g[j] + 1$;

Figura 11.16. Ordenación causal que usa vectores con marcas temporales.

Cada proceso p_i (i = 1, 2, ..., N) mantiene su propio vector de marcas temporales (véase la Sección 10.4). Cada entrada asociada a la marca temporal cuenta el número de mensajes multidifundidos de cada proceso que sucedieron-antes que el próximo mensaje que va a ser enviado mediante multidifusión.

Para enviar un mensaje mediante *OC-multicast* al grupo g, el proceso suma 1 a su entrada en el vector de marcas temporales y envía el mensaje mediante *B-multicast* a g junto con su marca de tiempo.

Cuando un proceso p_i *B-entrega* un mensaje p_j, tiene que colocarlo en la cola de retención antes de realizar la *OC-entrega*: primero tiene que asegurarse de que ha entregado todos los mensajes que le precedían causalmente. Para asegurarse de ello, p_i espera hasta que a) ha entregado todo mensaje enviado anteriormente por parte de p_j y b) ha entregado cualquier mensaje que p_j hubiese entregado hasta el instante de tiempo en el que multidifundió el mensaje. Ambas condiciones pueden detectarse examinando el vector de marcas temporales, como se muestra en la Figura 11.16. Hay que tener en cuenta de que un proceso puede *OC-entregarse* inmediatamente hacia sí mismo un mensaje que él envíe con *OC-multicast*, aunque esto no se describa en la Figura 11.16.

Cada proceso actualiza su vector de marcas temporales cada vez que entrega un mensaje, para llevar la cuenta de los mensajes que le preceden causalmente. Esto lo consigue incrementando en 1 la entrada j-ésima en su marca de tiempo. Esto supone una optimización de la operación de fusión que aparece en las reglas de actualización de la Sección 10.4. Esta optimización se consigue gracias a la condición de entrega en el algoritmo de la Figura 11.16, que garantiza que sólo se incrementa la entrada j-ésima.

Sin entrar en detalles, la prueba de corrección del algoritmo sería la siguiente. Supóngase que $multicast(g, m) \rightarrow multicast(g, m')$. Sean V y V' los vectores de marcastemporales de m y m', respectivamente. Mediante inducción se prueba directamente que $V < V'$. En concreto, si un proceso p_k envía m mediante *multicast*, entonces $V[k] \leqslant V'[k]$.

Considérese qué ocurre cuando algún proceso correcto p_i realiza *B-entrega* de m' (en lugar de *OC-entregarlo*) *sin haber OC-entregado m*. Según el algoritmo, $V_i[k]$ puede incrementarse sólo cuando p_i entrega un mensaje que viene de p_k y lo incrementa en uno. Pero p_i no ha recibido m y por lo tanto $V_i[k]$ no puede aumentar más allá de $V_i[k] - 1$. En consecuencia, no es posible, para p_i realizar la *OC-entrega* de m' ya que requeriría que $V_i[k] \geqslant V'[k]$, y por consiguiente que $V_i[k] \geqslant V[k]$.

El lector debería comprobar que si se sustituye la primitiva fiable *F-multicast* en lugar de *B-multicast*, se obtendrá una multidifusión a la vez fiable y con ordenación causal.

Además, si se combina el protocolo para multidifusión causal con el protocolo basado en secuenciador para una entrega totalmente ordenada se obtiene una entrega de mensajes con ordenación total y causal. El secuenciador entrega los mensajes según el orden causal y multidifunde la secuencia de números para los mensajes según el orden en el que los recibió. Los procesos en el grupo de destino no entregan el mensaje hasta que reciben otro mensaje por parte del secuenciador con el *orden* y dicho mensaje sea el próximo en la secuencia de entrega.

Dado que el secuenciador entrega los mensajes según un orden causal y dado que el resto de procesos entregan los mensajes en el mismo orden que el secuenciador, el orden es, por lo tanto, total y causal.

◇ **Grupos que se solapan.** En las definiciones y en la semántica de los algoritmos de ordenación FIFO, causal y total, se han considerado sólo grupos que no se solapaban. Esto simplifica el problema, pero no es satisfactorio, ya que en general los procesos necesitan pertenecer a múltiples grupos que se solapan. Por ejemplo, un proceso puede estar interesado en eventos de múltiples fuentes y, consiguientemente, puede unirse al correspondiente conjunto de grupos de distribución de eventos.

Se pueden ampliar las definiciones de ordenación para cubrir ordenaciones globales [Hadzilacos y Toueg 1994], en las cuales ha de considerarse que si el mensaje m se multidifunde a g, y si el

mensaje m' se multidifunde a g', entonces ambos mensajes han de enviarse a los miembros de $g \cap g'$.

Ordenación FIFO global: si un proceso correcto realiza *multicast*(g, m), y a continuación realiza *multicast*(g', m'), entonces todo proceso correcto en $g \cap g'$ que entregue m' debe entregar m antes que m'.

Ordenación causal global: si *multicast*(g, m) $\rightarrow$ *multicast*(g', m'), donde $\rightarrow$ es la relación sucedió-antes inducida por cualquier encadenamiento de mensajes *multicast*, entonces cualquier proceso correcto en $g \cap g'$ que entregue m' debe entregar m antes que m'.

Ordenación total por parejas. si un proceso correcto entrega un mensaje m enviado a g antes de entregar m' que fue enviado a g', entonces cualquier proceso correcto en $g \cap g'$ que entregue m' ha de entregar m antes que m'.

Ordenación total global: sea «<» la relación de ordenación entre eventos asociados a entregas. Se exige que «<» cumpla la ordenación total por parejas y que sea acíclica, ya que bajo la ordenación total por parejas «<» no es acíclica por defecto.

Una forma de llevar a cabo estas ordenaciones sería multidifundir cada mensaje m al conjunto de *todos* los procesos en el sistema. Cada proceso debería rechazar o entregar el mensaje según pertenezca o no al *grupo*(m). Esta implementación sería ineficiente e insatisfactoria, ya que la multidifusión ha de involucrar al menor número de procesos posibles fuera del grupo de destino. Se han estudiado distintas alternativas en Birman y otros [1991], García-Molina y Spauster [1991], Hadzilacos y Toueg [1994], Kindberg [1995] y Rodrigues y otros [1998].

◇ **Multicast en sistemas síncronos y asíncronos.** En esta sección, hasta el momento, se han descrito algoritmos para conseguir multidifusión fiable y no ordenada, multidifusión fiable con ordenación FIFO, multidifusión fiable con ordenación causal y multidifusión con ordenación total. También se ha indicado cómo conseguir una multidifusión ordenada total y causalmente. Se deja al lector la obtención de un algoritmo para una primitiva *multicast* que garantice las ordenaciones FIFO y total. Todos los algoritmos que se han descrito funcionan correctamente en sistemas asíncronos.

Sin embargo, no se ha descrito un algoritmo que garantice que sea fiable y tenga entrega con orden total. Aunque parezca sorprendente y siendo posible en un sistema *síncrono*, es *imposible* obtener un protocolo con estas garantías en un sistema distribuido asíncrono (incluso en uno que en el que en el peor de los casos sufra una sola caída de un proceso). Se volverá a tratar este tema en la siguiente sección.

11.5. CONSENSO Y SUS PROBLEMAS RELACIONADOS

Esta sección presenta el problema del consenso [Pease y otros 1980, Lamport y otros 1982] y sus problemas asociados: los generales bizantinos y la consistencia interactiva. Estos problemas se denominarán de forma conjunta como problemas de *acuerdo*. Resumiendo, los procesos tienen problemas para ponerse de acuerdo en un valor después de que uno o más de dichos procesos haya propuesto cuál debería ser ese valor.

Por ejemplo, en el Capítulo 2 se describió la situación en la cual dos ejércitos debían decidir, de forma consistente, si atacar o retirarse. De forma parecida, se puede necesitar que todos los computadores correctos que controlan los motores de una nave espacial decidan «proseguir» o que todos decidan «abortar la operación», una vez se haya propuesto una u otra acción. Igualmente, en una transacción para realizar una transferencia de fondos de una cuenta a otra, los computadores involucrados deben acordar de forma consistente cómo actualizar los respectivos crédito y débito. En el caso de la exclusión mutua, los procesos acuerdan cuáles pueden entrar en la sección crítica.

En el caso de una elección, los procesos acuerdan cuál es el proceso elegido. En la multidifusión totalmente ordenada, los procesos acuerdan el orden de entrega de los mensajes.

Existen protocolos que solucionan de forma específica estos tipos de acuerdo. Algunos se han descrito previamente en este capítulo, y los Capítulos 12 y 13 examinarán las transacciones. No obstante, es útil considerar formas de acuerdo más genéricas buscando características y soluciones comunes.

Esta sección define el consenso de forma más precisa y lo relaciona con tres problemas asociados al acuerdo: los generales bizantinos, la consistencia interactiva y la multidifusión con ordenación total. Se procede examinando bajo qué circunstancias se pueden resolver los problemas, y se perfilan algunas soluciones. En particular, se discutirán los conocidos resultados sobre imposibilidad debidos a Fischer y otros [1985], que establecen que en un sistema asíncrono una colección de procesos que contenga un proceso que ha fallado no se puede garantizar que se alcance el consenso. Finalmente, se considerará la existencia de algoritmos que funcionan en la práctica a pesar del resultado de imposibilidad.

11.5.1. DEFINICIÓN DEL MODELO DEL SISTEMA Y DEL PROBLEMA

El modelo del sistema incluye una colección de procesos p_i (i = 1, 2, ..., N) que se comunican mediante el paso de mensajes. En muchos casos prácticos un requisito importante a considerar es que ha de alcanzarse un consenso incluso cuando hay fallos. Como antes, se supone que la comunicación es fiable, pero los procesos pueden fallar. En esta sección se considerarán fallos por caída así como procesos que fallan de formas (arbitrariamente) extrañas. Algunas veces se especificará la suposición de que pueden fallar hasta f de los N procesos, esto es, ésos exhiben algún tipo de fallo especificado, mientras que el resto permanece correcto.

Si pueden ocurrir fallos arbitrarios, ha de considerarse otro factor en la especificación del sistema y es si los procesos firman digitalmente los mensajes que envían (véase la Sección 7.4). Si los procesos lo hacen, entonces se puede limitar el daño que puede ocasionar un proceso que ha fallado. Concretamente, durante la ejecución de un algoritmo de acuerdo dicho proceso no puede hacer afirmaciones falsas sobre los valores que le ha enviado un proceso correcto. De todas formas, la importancia de la firma de mensajes quedará patente cuando se discutan soluciones al problema de los generales bizantinos. Por defecto, se asumirá que no existe dicha firma.

◇ **Definición del problema del consenso.** Para alcanzar el consenso, cada proceso p_i comienza en el estado *no decidido* y *propone* un solo valor v_i de un conjunto de posibles valores D (i = 1, 2, ..., N). Los procesos se comunican entre sí, intercambiando valores. Entonces, cada proceso fija el valor de una *variable de decisión* d_i. Al hacer esto pasa al estado *decidido*, en el cual ya no puede cambiar el valor de d_i (i = 1, 2, ..., N). La Figura 11.17 muestra a tres procesos envueltos en un algoritmo de consenso. Dos procesos proponen «proseguir» y el tercero propone «abortar», pero se cae. Los dos procesos que permanecen correctos deciden «proseguir».

Los requisitos para un algoritmo de consenso son que han de cumplirse las siguientes condiciones cada vez que se ejecute:

Terminación: finalmente, cada proceso correcto ha de fijar su variable de decisión.

Acuerdo: el valor de decisión de todos los procesos correctos es el mismo: si p_i y p_j son correctos y están en el estado *decidido*, entonces $d_i = d_j$ (i, j = 1, 2, ..., N).

Integridad: si todos los procesos correctos han propuesto el mismo valor, entonces cualquier proceso correcto en el estado *decidido* ha elegido dicho valor.

Pueden aceptarse distintas variaciones en la definición de integridad, en función de la aplicación. Por ejemplo, un tipo de integridad más débil requeriría que el valor de decisión sea igual al pro-

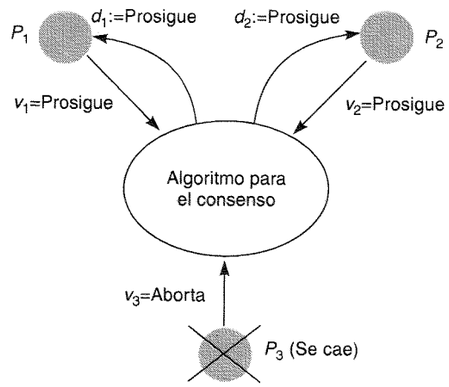


Figura 11.17. El problema del consenso para tres procesos.

puesto por algún proceso correcto, aunque no necesariamente igual para todos. De todas formas, se usará la definición dada anteriormente.

Para facilitar la comprensión de cómo obtener un algoritmo a partir de la formulación del problema, considérese un sistema en el que los procesos no pueden fallar. En ese caso es inmediato resolver el consenso. Por ejemplo, pueden reunirse todos los procesos en un grupo y que cada proceso del grupo multidifunda de modo fiable su valor propuesto al resto de los miembros del grupo. Todos los procesos esperan hasta recibir los N valores propuestos, incluido el suyo. Entonces evalúa la función *mayoría* $(v_1, v_2, ..., v_N)$, que devuelve el valor más frecuente entre sus argumentos, o el valor especial $\perp \notin D$ si no hay mayoría. Se garantiza la terminación gracias a la multidifusión fiable. Se garantizan el acuerdo y la integridad por la definición de *mayoría* y por la propiedad de integridad de la multidifusión fiable. Cada proceso recibe el mismo conjunto de valores propuestos, y cada proceso evalúa la misma función sobre esos valores. En consecuencia, han de ponerse de acuerdo y si cada proceso propone el mismo valor, entonces todos decidirán ese valor.

Hay que matizar que *mayoría* es sólo una de las posibles funciones a utilizar por los procesos para ponerse de acuerdo en los valores candidatos. Por ejemplo, si los valores están ordenados, las funciones *máximo* y *mínimo* también serían apropiadas.

Si los procesos pueden malograrse se introduce la complicación de tener que detectar los fallos y no queda claro que una ejecución del algoritmo de consenso termine. De hecho, si el sistema es asíncrono puede no hacerlo; se volverá a este punto en breve.

Si los procesos pueden fallar de formas (extrañamente) *arbitrarias*, aquellos que fallen pueden, en principio, comunicar valores aleatorios al resto. Esto puede parecer improbable en la práctica, pero no hay que descartar la posibilidad de que un proceso con un error falle de esa forma. Además, el fallo puede no ser accidental, sino el resultado de operaciones maliciosas o malévolas. Alguien podría deliberadamente forzar a un proceso a enviar diferentes valores a distintos procesos parejos con el fin de frustrar sus operaciones cuando están intentando alcanzar un consenso. En caso de que exista inconsistencia, los procesos correctos deben comparar lo que han recibido con lo que otros procesos dicen haber recibido.

◇ **El problema de los generales bizantinos.** En la definición informal del *problema de los generales bizantinos* [Lamport y otros 1982], tres o más generales han de ponerse de acuerdo en si atacar o retirarse. Uno de ellos, el comandante, cursa la orden. Los otros, los tenientes del comandante, deben decidir si atacar o retirarse. Pero uno o más de los generales puede ser un «traidor», o lo que es lo mismo, pueden fallar. Si el comandante es el traidor, manda atacar a un general y retirarse a otro. Si el traidor es un teniente, informa a uno de sus iguales que el comandante le mandó atacar mientras que a otro le dice que le ordenó retirarse.

El problema de los generales bizantinos se distingue del de consenso en que un proceso destacado proporciona un valor en el que los otros han de ponerse de acuerdo, en vez de que cada uno proponga un valor. Los requisitos son:

Terminación: al final, cada proceso correcto ha de fijar su variable de decisión.

Acuerdo: el valor de decisión de todos los procesos correctos es el mismo; si p_i y p_j son correctos y han entrado en el estado *decidido*, entonces $d_i = d_j$ $(i, j = 1, 2, ..., N)$.

Integridad: si el comandante es un proceso correcto, entonces todos los procesos correctos han de decidir el valor propuesto por el comandante.

Hay que fijarse que en el problema de los generales bizantinos la integridad implica acuerdo cuando el comandante es correcto: pero el comandante no tiene por qué estarlo.

◇ **Consistencia interactiva.** Este problema es otra variante del consenso, en el cual todo proceso propone un solo valor. El objetivo del algoritmo es que los procesos correctos se pongan de acuerdo en un *vector* de valores, uno para cada proceso. A éste se le llamará el *vector de decisión*. Por ejemplo, el objetivo para cada conjunto de procesos podría ser obtener la misma información acerca de sus estados respectivos.

Los requisitos para la consistencia interactiva son:

Terminación: al final, cada proceso correcto fija su variable de decisión.

Acuerdo: el vector de decisión de todos los procesos correctos es el mismo.

Integridad: si p_i es correcto, entonces todos los procesos correctos deciden que v_i es la i-ésima componente de su vector.

◇ **Relación del consenso con otros problemas.** Aunque lo más habitual es considerar el problema de los generales bizantinos con fallos arbitrarios en los procesos, el hecho es que cada uno de los tres problemas, el consenso, los generales bizantinos y la consistencia interactiva, tiene cabida en el contexto de los fallos arbitrarios o por caída. De forma similar, cada uno de ellos puede enmarcarse bien en sistemas síncronos o asíncronos.

Algunas veces es posible hallar una solución para un problema utilizando la solución del otro. Esta propiedad es muy útil, por un lado porque aumenta nuestra comprensión de los problemas y, por otro lado, porque reutilizando soluciones potencialmente podemos ahorrar en esfuerzo computacional y en complejidad.

Supongamos que existe una solución para el consenso (C), para los generales bizantinos (GB) y para la consistencia interactiva (CI) como sigue:

$C_i(v_1, v_2, ..., v_N)$ devuelve el valor de decisión p_i en su camino a la solución del problema del consenso, donde $v_1, v_2, ..., v_N$ son los valores propuestos por los procesos.

$GB_i(j, v)$ devuelve el valor de decisión p_i en su camino a la solución del problema de los generales bizantinos, donde p_j, el comandante, propone el valor v.

$CI_i(v_1, v_2, ..., v_N\ [j]$ devuelve el j-ésimo valor en el vector de decisión p_i en su camino a la solución del problema de consistencia interactiva, donde $v_1, v_2, ..., v_N$ son los valores propuestos por los procesos.

Las definiciones de C_i, GB_i y CI_i suponen que un proceso con fallo propone una única noción de valor, aunque haya propuesto diferentes valores a cada uno de los otros procesos. Esto es sólo un convenio: las soluciones no deben depender de esa noción de valor.

Es posible formular soluciones a partir de las soluciones de otros problemas. Se proponen tres ejemplos:

CI a partir de GB. Se crea una solución para CI a partir de GB ejecutando GB N veces, una por cada proceso p_i ($i = 1, 2, ..., N$) que actúe como comandante:

$$CI_i(v_1, v_2, ..., v_N)[j] = GB_i(j, v_j)(i, j = 1, 2, ..., N)$$

C a partir de CI. Creamos una solución para C a partir de CI ejecutando CI para producir un vector de valores en cada proceso; a continuación se aplica una función apropiada sobre los valores del vector para determinar un único valor:

$$C_i(v_1, ..., v_N) = mayoría(CI_i(v_1, ..., v_N)[1], ..., CI_i(v_1, ..., v_N)[N])$$

($i = 1, 2, ..., N$), donde *mayoría* se corresponder con su definición anterior.
GB a partir de C. Se crea una solución para GB a partir de C de la siguiente forma:

- El comandante p_j envía su valor propuesto v a sí mismo y a cada uno de los procesos restantes.
- Todos los procesos ejecutan C con los valores $v_1, v_2, ..., v_N$ que han recibido (teniendo en cuenta que p_j puede haber fallado).
- Obtienen $GB_i(j, v) = C_i(v_1, v_2, ..., v_N)(i = 1, 2, ..., N)$.

Queda como ejercicio para el lector comprobar que las condiciones de terminación, acuerdo e integridad se cumplen en cada caso. Fischer [1983] muestra los tres problemas con más detalle.

Solucionar el consenso equivale a resolver la multidifusión fiable y con ordenación total: dada una solución para uno, se puede resolver el otro. La implementación del consenso con una operación de multidifusión fiable y con ordenación total *FOT-multicast* es inmediata. Se reúnen todos los procesos en un grupo g. Para conseguir el consenso cada proceso p_i realiza *FOT-multicast*(g, v_i). Entonces, cada proceso p_i elige $d_i = m_i$, donde m_i es el *primer* valor que p_i ha *FOT-entregado*. La propiedad de terminación se deduce de la fiabilidad de la multidifusión. Las propiedades de acuerdo e integridad se deducen de la fiabilidad y ordenación total de la multidifusión. Chandra y Toueg [1996] han demostrado cómo se puede conseguir multidifusión fiable y con ordenación total a partir del consenso.

11.5.2. CONSENSO EN UN SISTEMA SÍNCRONO

Esta sección describe un algoritmo que utiliza un protocolo de multidifusión básico para resolver el problema del consenso en un sistema asíncrono. El algoritmo supone que un máximo de f de N procesos pueden sufrir fallos por caída.

Para alcanzar el consenso, cada proceso correcto recoge valores propuestos por los otros procesos. El algoritmo realiza $f + 1$ vueltas, en cada una de las cuales los procesos correctos hacen *B-multicast* de los valores entre ellos mismos. Se supone que pueden fallar un máximo de f procesos. En el peor de los casos, las f caídas ocurrieron durante las vueltas, pero el algoritmo garantiza que al final de las mismas todos los procesos correctos que han sobrevivido están en disposición de llegar a un acuerdo.

El algoritmo, que se muestra en la Figura 11.18, se basa en el propuesto por Dolev y Strong [1983] y en su presentación por parte de Attiya y Welch [1998]. La variable $Valores_i^r$ almacena el conjunto de valores conocidos para el proceso p_i al comienzo de la ronda r. Cada proceso multidifunde el conjunto de valores que no ha enviado en rondas anteriores. A continuación recoge las entregas de mensajes multidifusión similares por parte de otros procesos y guarda los nuevos valores. Aunque no se muestra en la Figura 11.18, la duración de cada ronda está delimitada por un timeout basado en el tiempo máximo que necesita un proceso correcto para realizar la multidifusión. Tras $f + 1$ rondas, cada proceso ha elegido el valor mínimo que haya recibido y su valor de decisión.


```
Algoritmo para los procesos p_i ∈ g; el algoritmo lo realiza en f + 1 rondas
En la inicialización
   Valores_i^1 := {v_i}; Valores_i^0 = {};
En la ronda r (1 ≤ r ≤ f + 1)
   Hacer B-multicast(g, Valores_i^r − Valores_i^(r−1));      // Enviar sólo valores que no hayan sido enviados
   Valores_i^(r+1) := Valores_i^r;
   mientras (se está en la ronda r)
   {
            En B-entrega(V_j) por parte de p_j hacer
               Valores_i^(r+1) := Valores_i^(r+1) ∪ V_j;
}
Tras (f + 1) rondas
   Asignar d_i = mínimo(Valores_i^(f+1))
```

Figura 11.18. El problema del consenso en un sistema síncrono.

La terminación es obvia ya que el sistema es síncrono. Para comprobar la corrección del algoritmo hay que demostrar que cada proceso llega al mismo conjunto de valores al final de la última ronda. El acuerdo y la integridad vendrán dados porque los procesos aplican la función *mínimo* a su conjunto de valores.

Supóngase, por el contrario, que dos procesos difieren en su conjunto final de valores. Sin pérdida de generalidad, algún proceso correcto p_i tiene un valor v que otro proceso correcto p_j ($i \neq j$) no tiene. La única explicación para que p_i tenga al final un valor propuesto v que p_j no tiene es que cualquier tercer proceso, por ejemplo p_k, que consiguió enviar v a p_i se cayó antes de que v pudiese ser entregada a p_j. A su vez, cualquier proceso que hubiese enviado v en la ronda anterior debería haber caído, para explicar por qué p_k tiene v en esa ronda pero p_j no lo tiene. Procediendo de esta forma hemos de suponer al menos una caída en cada una de las rondas precedentes. Sin embargo, se supuso que podían ocurrir a lo sumo f caídas y hubo $f + 1$ rondas. Hemos llegado a una contradicción.

Se concluye que *cualquier* algoritmo que quiera alcanzar el consenso a pesar de f fallos por caída requiere al menos $f + 1$ rondas de intercambio de mensajes con independencia de cómo está construido [Dolev y Strong 1983]. Este límite inferior también se cumple en el caso de fallos extraños [Fischer y Lynch 1980].

11.5.3. EL PROBLEMA DE LOS GENERALES BIZANTINOS EN UN SISTEMA SÍNCRONO

Se ha discutido previamente este problema en un sistema asíncrono. A diferencia del algoritmo para el consenso descrito en la sección anterior, se supone que los procesos pueden presentar fallos arbitrarios. Esto es, un proceso que ha fallado puede enviar un mensaje con cualquier valor en cualquier momento; y puede decidir no enviar ningún mensaje. Un máximo de f de N procesos pueden fallar. Los procesos correctos pueden determinar la ausencia de mensajes mediante un tiempo límite, pero no pueden concluir que el emisor se ha caído, ya que puede estar en silencio por un tiempo y después volver a enviar mensajes.

Se supone que los canales de comunicación entre pares de procesos son privados. Si un proceso pudiese examinar todos los mensajes enviados por otros procesos, podría detectar las inconsistencias enviadas a diferentes procesos por parte de un proceso con fallo. La suposición por defecto de fiabilidad en el canal implica que ningún proceso con fallo puede inyectar mensajes en el canal de comunicación de dos procesos correctos.

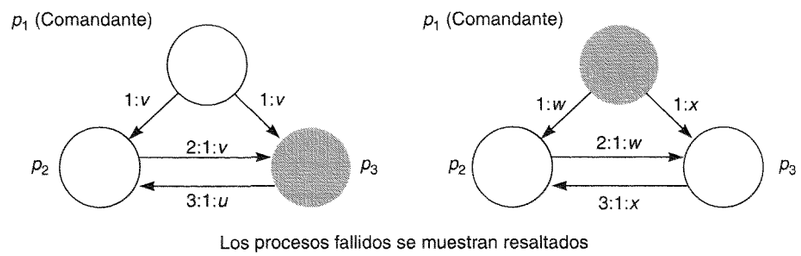


Figura 11.19. Tres generales bizantinos.

Lamport y otros [1982] consideraron el caso de tres procesos que enviaban mensajes no firmados entre ellos. Mostraron que no existe solución que garantice que se cumplan las condiciones del problema de los generales bizantinos si se permite que uno de los procesos falle. Además, generalizaron sus resultados para mostrar que no existe solución si $N \leqslant 3f$. Aquí se demostrarán estos resultados de forma resumida. Además, Lamport y otros obtuvieron un algoritmo que soluciona el problema de los generales bizantinos en un sistema síncrono si $N \geqslant 3f + 1$ para el caso de mensajes sin firma (que ellos denominaron «orales»).

◇ **Imposibilidad con tres procesos.** La Figura 11.19 muestra dos escenarios en los cuales sólo uno de los procesos ha fallado. En la configuración de la izquierda, uno de los tenientes p_3 ha fallado; en la de la derecha, ha fallado el comandante p_1. Cada escenario en la Figura 11.18 muestra dos rondas de mensajes: los valores que manda el comandante y los valores que los tenientes se mandan a continuación entre ellos. Los números permiten especificar el origen del mensaje e indicar las distintas rondas. Los «:» en los mensajes han de leerse como «dice que»; por ejemplo, «3:1:*u*» identifica el mensaje «3 dice que 1 dice que *u*».

En el escenario de la izquierda, el comandante envía correctamente el mismo valor v a cada uno de los otros procesos, y p_2 lo reproduce de forma correcta hacia p_3. Sin embargo, p_3 envía un valor $u \neq v$ a p_2. En ese momento, todo lo que p_2 conoce es que ha recibido distintos valores; no puede decir cuál fue enviado por el comandante.

En el escenario de la derecha, el comandante ha fallado y envía valores distintos a los tenientes. Una vez que p_3 ha reproducido el valor x que ha recibido, p_2 está en la misma situación que cuando p_3 falló; ha recibido dos valores distintos.

Si existe una solución, entonces p_2 está forzado a decidir el valor v, cuando el comandante es correcto, por la condición de integridad. Si se acepta que ningún algoritmo puede posiblemente distinguir entre los dos escenarios, p_2 debería elegir también el valor enviado por el comandante en el escenario de la derecha.

Si se sigue el mismo razonamiento para p_3, suponiendo que es correcto, ha de concluirse, por simetría, que p_3 también elige el valor enviado por el comandante como su valor de decisión. Pero esto contradice la condición de acuerdo (ya que el comandante envía valores distintos si ha fallado). Por lo tanto, no hay solución posible.

Hay que fijarse en que este argumento se apoya en la intuición de que nada puede hacerse para mejorar el conocimiento de un general que está correcto más allá de la primera fase, donde no puede decidir qué proceso ha fallado. Es posible comprobar la corrección de esta intuición [Pease y otros 1980]. El acuerdo bizantino *puede* ser alcanzado por tres generales, con uno de ellos fallido, si los generales firman digitalmente los mensajes.

◇ **Imposibilidad con $N \leqslant 3f$.** Pease y otros generalizaron el resultado básico de imposibilidad para tres procesos, para probar que no hay solución posible si $N \leqslant 3f$. Brevemente, el argumento

puede resumirse de la siguiente forma. Supóngase que existe una solución para $N \leqslant 3f$. Utilícese la solución de tres procesos p_1, p_2 y p_3 para simular el comportamiento de n_1, n_2 y n_3 generales, respectivamente, donde $n_1 + n_2 + n_3 = N$ y $n_1, n_2, n_3 \leqslant N/3$. Se supone, además, que uno de los tres procesos falla. Aquéllos entre p_1, p_2 y p_3 que están correctos simulan generales correctos: ellos simulan las interacciones de sus propios generales de forma interna y envían mensajes de parte de sus generales a aquellos simulados por otros procesos. Los generales simulados por procesos fallidos se suponen con fallo: los mensajes que envían como parte de la simulación pueden ser espurios. Dado que $N \leqslant 3f$ y $n_1, n_2, n_3 \leqslant N/3$, a lo sumo f generales simulados tienen fallos.

Dado que el algoritmo que ejecuta los procesos se supone correcto, la simulación termina. Los generales correctamente simulados (en los procesos correctos) están de acuerdo y satisfacen la propiedad de integridad. Pero ahora es necesario encontrar el consenso para dos de un conjunto de tres procesos: cada uno decide según el valor elegido por sus generales simulados. Esto contradice el resultado de imposibilidad para tres procesos con uno de ellos fallido.

◇ **Solución de un proceso que falla.** Por problemas de espacio no se describe completamente el algoritmo de Pease y otros que resuelve los problemas de los generales bizantinos en un sistema síncrono con $N \geqslant 3f + 1$. En su lugar, se muestra el funcionamiento del algoritmo para el caso $N \leqslant 4$, $f = 1$ y se da el ejemplo con $N = 4$, $f = 1$.

Los generales que están correctos alcanzan el acuerdo en dos rondas de mensajes:

- En la primera ronda, el comandante envía su valor a cada uno de sus tenientes.
- En la segunda ronda, cada uno de los tenientes envía el valor que ha recibido a sus iguales.

Un teniente recibe un valor de su comandante y $N - 2$ valores de sus iguales. Si el comandante ha fallado, entonces todos los tenientes están correctos y cada uno habrá recogido exactamente el conjunto de valores que les envió el comandante. En otro caso, uno de los tenientes ha fallado; cada uno de sus iguales que estaban correctos recibirá $N - 2$ copias del valor enviado por el comandante junto con un valor enviado por el teniente que ha fallado.

En ambos casos, el teniente que está correcto sólo necesita aplicar una función de mayoría simple al conjunto de valores que ha recibido. Ya que $N \geqslant 4$, $(N - 2) \geqslant 2$. Por lo tanto, la función *mayoría* ignorará cualquier valor enviado por el teniente que ha fallado y producirá el valor enviado por el comandante, si éste está correcto.

A continuación se ilustra el algoritmo que se ha esbozado para el caso de cuatro generales. La Figura 11.20 muestra dos escenarios similares a los de la Figura 11.19, pero en este caso hay cua-

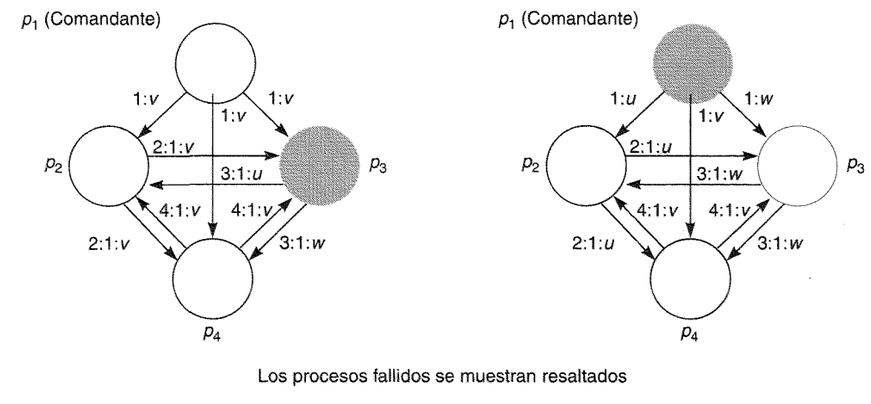


Figura 11.20. Cuatro generales bizantinos.

tro procesos y uno de ellos falla. Al igual que en la Figura 11.19, en la configuración de la izquierda ha fallado p_3, uno de los tenientes; en la de la derecha, ha fallado el comandante p_1.

En el caso de la izquierda, los dos tenientes que están correctos llegan a un acuerdo, decidiendo en función del valor del comandante:

p_2 decide por *mayoría*$(v, u, v) = v$
p_4 decide por *mayoría*$(v, v, w) = v$

En el caso de la derecha, el comandante ha fallado, pero los tres procesos correctos alcanzan un acuerdo:

p_2, p_3 y p_4 deciden por *mayoría*$(u, v, w) = \perp$ (valor especial que se da cuando no hay mayoría).

El algoritmo tiene en cuenta el hecho de que un proceso que ha fallado puede no emitir un mensaje. Si un proceso correcto no recibe un mensaje en un tiempo límite apropiado (ya que el sistema es síncrono), el proceso sigue adelante como si el proceso que ha fallado le enviase el valor $\perp$.

◇ **Discusión.** Se puede medir la eficiencia de la solución al problema de los generales bizantinos, o de cualquier otro problema de acuerdo, preguntando:

- ¿Cuántas rondas de mensajes se llevan a cabo? (Éste es el factor que determina en cuánto tiempo terminará el algoritmo.)
- ¿Cuántos mensajes se mandan y de qué tamaño? (Esto permite medir la utilización del ancho de banda y tiene efecto en el tiempo de ejecución.)

En el caso general ($f \geqslant 1$), el algoritmo de Lamport y otros para el caso de mensajes no firmados realiza $f + 1$ rondas. En cada ronda, un proceso envía a un subconjunto del resto de los procesos los valores que ha recibido en la ronda previa. El coste del algoritmo es muy elevado, ya que implica enviar $0(N^{f+1})$ mensajes.

Fischer y Linch [1982] probaron que cualquier solución determinista al consenso suponiendo que puede haber fallos extraños (y que, por lo tanto, sirve para el problema de los generales bizantinos, tal y como se mostró en la Sección 11.5.1), necesitará al menos $f + 1$ rondas de mensajes. En consecuencia, ningún algoritmo puede funcionar más rápido que el de Lamport y otros, aunque ha habido mejoras en la complejidad de los mensajes, como en el ejemplo de Garay y Moses [1993].

Distintos algoritmos, como el de Dolev y Strong [1983], sacan ventaja de los mensajes firmados. Su algoritmo también requiere $f + 1$ rondas, pero el número de mensajes enviados es solamente $0(N^2)$.

La complejidad y el coste de las soluciones sugiere que serán aplicados sólo cuando la amenaza sea grande. Si el origen de esta amenaza es un fallo en el *hardware*, entonces la posibilidad de conducta verdaderamente arbitraria es baja. Las soluciones que se basan en un conocimiento más detallado de los fallos pueden ser más eficientes [Barborak y otros 1993]. Si el origen de la amenaza son usuarios malintencionados y un sistema quisiera hacerles frente, probablemente utilizaría firmas digitales; una solución sin firmas digitales no es práctica.

11.5.4. IMPOSIBILIDAD EN SISTEMAS ASÍNCRONOS

Se han proporcionado soluciones para los problemas del consenso y de los generales bizantinos (y, por lo tanto, pueden derivarse para la consistencia interactiva) en sistemas síncronos. Y todas las soluciones necesitan que el sistema sea síncrono. Los algoritmos suponen que el intercambio de mensajes se hace por rondas, y que los procesos han de contar el *timeout* y suponer que un proceso fallido no les enviará un mensaje dentro de una ronda, ya que se excedería el retraso máximo permitido.

Fischer y otros [1985] demostraron que ningún algoritmo puede garantizar que se encuentre un consenso en un sistema asíncrono, incluso cuando sólo falle uno de los procesos. En un sistema asíncrono, los procesos pueden responder a los mensajes en tiempos arbitrarios y, en consecuencia, no se puede distinguir un proceso lento de uno que se ha caído. Su demostración, que está fuera del alcance de este libro, implica la demostración de que siempre hay alguna continuación de la ejecución de los procesos que evita que se alcance el consenso.

A partir de este resultado de Fischer y otros se deduce que no se puede garantizar una solución en un sistema asíncrono para el problema de los generales bizantinos, ni para el de la consistencia interactiva ni para la multidifusión fiable y totalmente ordenada. Si hubiese una solución, a partir de los resultados de la Sección 11.5.1, se encontraría una solución para el consenso, contradiciendo el resultado de imposibilidad.

Obsérvese el término «garantizar» en la sentencia del resultado de imposibilidad. El resultado no significa que *nunca* se pueda alcanzar el consenso distribuido en un sistema asíncrono. Permite que el consenso se alcance con una probabilidad mayor que cero, confirmándose la observación en la práctica. Por ejemplo, a pesar del hecho de que nuestros sistemas frecuentemente son asíncronos en la práctica, los sistemas de transacciones han estado llegando a consensos durante varios años.

Una aproximación para trabajar a pesar del resultado de imposibilidad es considerar a los sistemas como *parcialmente síncronos*. Estos sistemas son lo suficientemente más débiles que los sistemas síncronos como para ser útiles como modelos de sistemas en la práctica, y lo suficientemente más fuertes que los sistemas asíncronos como para que pueda resolverse en ellos el consenso [Dwork y otros 1988]. Esta aproximación va más allá del alcance de este libro; no obstante, se van a esbozar otras técnicas que permiten trabajar bajo el resultado de imposibilidad. Estas técnicas son 1) el enmascaramiento de fallos, 2) alcanzar el consenso aprovechándose de los detectores de fallos y 3) alcanzar el consenso mediante la aleatoriedad en los aspectos del comportamiento de los procesos.

◇ **Enmascaramiento de fallos.** La primera técnica consiste en evitar el resultado de imposibilidad en su conjunto mediante el enmascaramiento de los fallos que puedan ocurrir en un proceso (véase la Sección 2.3.2 para una introducción al enmascaramiento de fallos). Por ejemplo, los sistemas de transacciones utilizan el almacenamiento persistente, que sobrevive a la caída de los procesos. Si un proceso se cae, a continuación se reinicia (automáticamente o por medio de un administrador). El proceso guarda en los puntos críticos de su programa suficiente información en el almacenamiento persistente, de tal forma que si se cae y es reiniciado, encontrará suficientes datos como para continuar de forma correcta con la tarea interrumpida. Dicho de otra forma, se comportará como un proceso que está correcto, pero que de vez en cuando toma un largo tiempo largo procesar un paso.

Por supuesto, el enmascaramiento de fallos generalmente se puede aplicar durante el diseño del sistema. El Capítulo 13 tratará cómo los sistemas de transacciones aprovechan el almacenamiento persistente. El Capítulo 14 describirá cómo los fallos en los procesos también pueden ser enmascarados mediante la replicación de componentes software.

◇ **Consenso que utiliza detectores de fallos.** Otro método para evitar el resultado de imposibilidad es emplear detectores de fallos. En la práctica, algunos sistemas utilizan detectores de fallo «perfectos por diseño» para alcanzar el consenso. No obstante, ningún detector de fallo en un sistema asíncrono que funcione exclusivamente en base a paso de mensajes puede realmente ser perfecto. Sin embargo, pueden ponerse de acuerdo en *juzgar* un proceso que ha fallado si no ha respondido en un tiempo delimitado. El hecho de que un proceso no responda no significa que haya fallado, pero el resto de los procesos se comportará como si lo hubiese hecho. En estos casos de «fallo-por-silencio», los procesos descartan, a partir de ese momento, cualesquiera de los mensajes que reciban de ese proceso «fallido». Dicho de otra forma, en la práctica se ha

transformado un sistema asíncrono en uno síncrono. Ésta es la técnica usada en el sistema ISIS [Birman 1993].

Este método requiere que el detector de fallos sea exacto de forma habitual. Cuando no lo es, el sistema ha de seguir adelante sin un miembro del grupo que, de otra forma, habría contribuido potencialmente a la eficacia del sistema. Desgraciadamente, para conseguir un detector de fallos razonablemente exacto, habría que utilizar grandes *timeouts*, forzando a los procesos a esperar durante grandes espacios de tiempo (y, por lo tanto, no realizar trabajo útil en ese tiempo) para concluir que un proceso ha fallado. Otro aspecto que surge de esta aproximación es el particionamiento de la red, que se discutirá en el Capítulo 14.

Una alternativa distinta es usar detectores de fallos imperfectos y llegar al consenso permitiendo procesos sospechosos que se comporten de forma correcta en lugar de excluirlos. Chandra y Toueg [1996] analizaron las propiedades que debía tener un detector de fallos en un sistema asíncrono para poder resolver el problema del consenso. Mostraron que podía conseguirse el consenso en un sistema asíncrono, incluso cuando el detector de fallos no era fiable, si no se caían más de $N/2$ procesos y la comunicación era fiable. El tipo más débil de detector de fallos con el que se consigue se denomina *detector de fallos eventualmente débil*. Será aquel que es:

Eventualmente débilmente completo: cada proceso que ha fallado será sospechoso finalmente y de forma permanente para algún proceso correcto.

Eventualmente débilmente exacto: pasado un cierto tiempo, al menos un proceso correcto no será nunca sospechoso para cualquier otro proceso correcto.

Chandra y Toueg muestran que no se puede implementar uno de estos detectores de fallos en un sistema asíncrono utilizando exclusivamente paso de mensajes. Sin embargo, se ha descrito en la Sección 11.1 un detector de fallos basado en mensajes que adapta sus tiempos límite en función de los tiempos de respuesta observados. Si un proceso, o la conexión hasta él, es muy lenta, el valor de su tiempo límite aumentará para que apenas se produzcan sospechas falsas del proceso. En muchos sistemas reales, el algoritmo se comporta en la práctica de modo suficientemente parecido a un detector de fallos eventualmente débil.

El algoritmo de consenso de Chandra y Toueg permite que los procesos, de los que se sospechó de forma equivocada, que continúen con sus operaciones habituales, y que los procesos que sospecharon de ellos reciban sus mensajes y los procesen con normalidad. Esto complicará la vida del programador de aplicaciones, pero tiene la ventaja de que no se desperdician procesos correctos que fueron excluidos de forma errónea. Además, los tiempos límite para detectar fallos pueden fijarse de forma menos conservadora que en la aproximación ISIS.

◇ **Consenso que utiliza la aleatoriedad.** El resultado de Fischer y otros depende de qué puede considerarse como un «adversario». Éste es un «personaje» (en realidad, una colección de sucesos aleatorios) que puede aprovecharse de los fenómenos propios de las vistas asíncronas para frustrar los intentos de los procesos para alcanzar el consenso. El adversario manipula la red para retrasar los mensajes de tal forma que lleguen en un instante de tiempo erróneo y, de forma similar, ralentiza o acelera los procesos para que estén en el estado «equivocado» cuando reciban el mensaje.

La tercera técnica que solventa el resultado de imposibilidad consiste en introducir en el comportamiento del proceso un elemento de posibilidad, de tal forma que el adversario no pueda llevar a cabo su estrategia de frustración de una forma eficaz. En algunos casos no puede alcanzarse el consenso, pero este método permite a los procesos que lo alcancen en un tiempo finito *esperado*. En Canetti y Rabin [1993] se puede encontrar un algoritmo probabilístico que soluciona el consenso incluso con fallos extraños.

11.6. RESUMEN

El capítulo comenzó discutiendo la necesidad de que los procesos accedan a recursos compartidos bajo condiciones de exclusión mutua. Los servidores que gestionan los recursos compartidos no siempre incorporan la posibilidad de realizar bloqueos, y se necesita un servicio separado de exclusión mutua distribuida. Se trataron tres algoritmos para conseguir la exclusión mutua: uno que emplea un servidor central, otro basado en un anillo y el tercero basado en la multidifusión que usa relojes lógicos. Ninguno de estos mecanismos puede soportar fallos en la forma en la que fueron descritos, aunque pueden modificarse para que toleren algunos fallos.

A continuación, el capítulo consideró un algoritmo basado en anillo y el algoritmo del abusón, cuyo objetivo común es elegir exclusivamente un proceso dentro de un conjunto, incluso si tienen lugar distintos procesos de elección de forma concurrente. El algoritmo del abusón podría usarse, por ejemplo, para elegir un nuevo maestro servidor de tiempos o un nuevo servidor de bloqueos cuando el existente falla.

El capituló continuó describiendo la comunicación por multidifusión. Se discutió la multidifusión fiable, en el cual los procesos correctos se ponen de acuerdo en el conjunto de mensajes a entregar, y la multidifusión con ordenación FIFO, causal y total, en la entrega. Se dieron algoritmos para obtener multidifusión fiable y para los tres tipos de ordenación en la entrega.

Finalmente, se describieron los problemas del consenso, los generales bizantinos y la consistencia interactiva. Se establecieron las condiciones para poder solucionarlos y se mostraron las relaciones existentes entre ellos, incluyendo la relación entre el consenso y la multidifusión fiable y con ordenación total.

Pueden encontrarse soluciones en un sistema síncrono, como se describió para algunos de los problemas. De hecho, existen soluciones incluso con fallos arbitrarios. Se esbozó parte de la solución propuesta por Lamport y otros para el problema de los generales bizantinos. Aunque algoritmos más recientes tienen una menor complejidad, en principio, ninguno puede evitar las $f + 1$ rondas de dicho algoritmo, a menos que los mensajes se firmen digitalmente.

El capítulo terminó describiendo el resultado fundamental de Fischer y otros que concierne a la imposibilidad de garantizar el consenso en un sistema asíncrono. Se discutió el hecho de que, a pesar de ello, los sistemas llegan regularmente al acuerdo en ese tipo de sistemas.

EJERCICIOS

11.1. ¿Es posible implementar un detector de procesos con fallo, fiable o no, utilizando un canal de comunicación no fiable?

11.2. Si todos los procesos cliente usan una comunicación de un único hilo, ¿es relevante la condición EM3 de exclusión mutua (que especifica la entrada según ordenación sucedió-antes)?

11.3. Obténgase una fórmula para la máxima capacidad de procesamiento de un sistema de exclusión mutua en términos del retardo de sincronización.

11.4. En el algoritmo con servidor central para conseguir exclusión mutua, descríbase una situación en la cual dos peticiones no son procesadas con ordenación sucedió-antes.

11.5. Adáptese el algoritmo con servidor central para conseguir exclusión mutua para tratar el fallo por caída de cualquier cliente (en cualquier estado), suponiendo que el servidor está correctamente y que se dispone de un detector de fallos fiable. Comentar si el sistema re-

sultante es tolerante a fallos. ¿Qué podría suceder si un cliente que posee el testigo es sospechoso erróneamente de haber caído?

11.6. Dése un ejemplo de la ejecución del algoritmo basado en anillo que muestre que no se garantiza que los procesos entran en la sección crítica según la ordenación sucedió-antes.

11.7. En un cierto sistema, por lo general cada proceso entra en su sección crítica varias veces antes de que otro proceso lo pida. Explíquese por qué el algoritmo de Ricart y Agrawala para la exclusión mutua basada en anillo no es eficiente en este caso, y descríbase cómo mejorar su rendimiento. ¿Cumple la adaptación propuesta la condición de pervivencia EM2?

11.8. En el algoritmo del abusón, un proceso que se está recuperando de un fallo comienza un proceso de elección y se convierte en el nuevo coordinador si tiene un identificador mayor que el actual ocupante del cargo. ¿Es necesaria esta característica en el algoritmo?

11.9. Sugiérase cómo adaptar el algoritmo del abusón para tratar con particiones temporales en la red (lo que ocasiona una comunicación lenta) y para tratar con procesos lentos.

11.10. Desarróllese un protocolo para conseguir un servicio de multidifusión básico sobre la multidifusión IP.

11.11. ¿Cómo han de cambiarse, si es que han de hacerlo, las definiciones de integridad, acuerdo y validez para la multidifusión fiable con grupos abiertos?

11.12. Explíquese por qué invirtiendo el orden de las líneas *F-entrega m*; y *si* $(q \neq p)$ *entonces B-multicast* (q, m); *fin si* en la Figura 11.10 se consigue que el algoritmo ya no satisfaga la propiedad de acuerdo uniforme. ¿Satisface el algoritmo de multidifusión fiable basado en IP la propiedad de acuerdo uniforme?

11.13. Explíquese si el algoritmo de multidifusión fiable sobre la multidifusión IP funciona tanto para grupos abiertos como cerrados. Dado cualquier algoritmo para grupos cerrados, ¿cómo, de una forma sencilla, pueden obtenerse algoritmos para grupos abiertos?

11.14. Considérese cómo solventar las suposiciones poco prácticas que se hicieron para conseguir las propiedades de validez y acuerdo para el caso de multidifusión fiable basada en multidifusión IP. Pista: añádase una regla para borrar los mensajes retenidos cuando hayan sido entregados en todas partes; y considérese añadir un mensaje falso de «latido» que nunca se entrega a la aplicación, pero que el protocolo manda si la aplicación no tiene que enviar mensaje alguno.

11.15. Muéstrese que el algoritmo multidifusión con ordenación FIFO no funciona para grupos que se solapan, considerando los mensajes enviados por la misma fuente a dos grupos que se solapan y considerando un proceso en la intersección de ambos grupos. Adáptese el protocolo para que funcione en ese caso. Pista: los procesos han de incluir junto con los mensajes los últimos números de secuencia de mensajes enviados a *todos* los grupos.

11.16. Muéstrese por qué si la multidifusión básica usada en el algoritmo de la Figura 11.14 también cumple la ordenación FIFO, entonces la multidifusión resultante con ordenación total está también ordenada causalmente. En ese caso, ¿cualquier multidifusión que tenga ordenación FIFO y total, en consecuencia, estará ordenada causalmente?

11.17. Sugiérase cómo adaptar el protocolo de multidifusión con ordenación causal para tratar grupos que se solapen.

11.18. Cuando se discutió el algoritmo de exclusión mutua de Maekawa, se vio un ejemplo de tres subconjuntos de un conjunto de tres procesos que podrían llevar a un bloqueo mutuo.

Utilícense estos subconjuntos como grupos de destino de multidifusión para mostrar cómo una ordenación total por parejas no es necesariamente acíclica.

11.19. Constrúyase una solución a la multidifusión fiable con ordenación total en un sistema síncrono utilizando multidifusión fiable y una solución para el problema del consenso.

11.20. Se ofreció una solución para el problema del consenso a partir de una solución para la multidifusión fiable y totalmente ordenada, que involucraba la selección del primer valor a entregar. Explíquese, utilizando primeros principios, por qué en un sistema asíncrono no podemos obtener una solución utilizando un servicio de multidifusión fiable pero sin ordenación total y la función «mayoría» (ha de notarse que, si se pudiese, ¡esto estaría en contradicción el resultado de imposibilidad de Fischer y otros!) Pista: ténganse en cuenta los procesos lentos o que han fallado.

11.21. Muestre que se puede alcanzar el acuerdo bizantino por tres generales, con uno de ellos fallando, si los generales firman digitalmente sus mensajes.

12

TRANSACCIONES Y CONTROL DE CONCURRENCIA

12.1. Introducción
12.2. Transacciones
12.3. Transacciones anidadas
12.4. Bloqueos
12.5. Control optimista de la concurrencia
12.6. Ordenación por marcas de tiempo
12.7. Comparación de métodos para el control de concurrencia
12.8. Resumen

Este capítulo trata sobre la aplicación de las transacciones y el control de concurrencia a los objetos compartidos gestionados por los servidores.

Una transacción define una secuencia de operaciones que se realiza por el servidor y se garantiza por el mismo que es atómica, ya sea en presencia de múltiples usuarios e incluso de caídas del servidor. Las transacciones anidadas están estructuradas a partir de conjuntos de otras transacciones. Se utilizan particularmente en los sistemas distribuidos porque permiten concurrencia adicional.

Todos los protocolos de control de concurrencia están basados en el criterio de equivalencia en serie y se derivan de reglas para los conflictos entre operaciones. Se describen tres métodos:

- Se utilizan bloqueos para ordenar transacciones que acceden a los mismos objetos de acuerdo con el orden de llegada de sus operaciones sobre dichos objetos.
- El control optimista de concurrencia permite a las transacciones avanzar hasta que están dispuestas para consumarse, que es cuando se realiza una comprobación para ver si se han efectuado operaciones conflictivas con los objetos.
- La ordenación por marca de tiempo utiliza dichas marcas para ordenar las transacciones que acceden a los mismos objetos de acuerdo con los instantes de comienzo.

12.1. INTRODUCCIÓN

La meta de las transacciones es asegurar que todos los objetos gestionados por un servidor permanecen en un estado consistente cuando dichos objetos son accedidos por múltiples transacciones y en presencia de caídas del servidor. En el Capítulo 2 se presentó un modelo de fallo para los sistemas distribuidos. Las transacciones se ocupan de los fallos por caída de los procesos y los de omisión en la comunicación, aunque no cualquier tipo de comportamiento arbitrario (o bizantino). El modelo de fallo para transacciones se presenta en la Sección 12.1.2.

Los objetos que pueden recobrarse después de la caída de su servidor se llaman *recuperables*. En general, esto alude a los objetos gestionados por un servidor en una memoria volátil (por ejemplo, RAM) o en almacenamiento persistente (por ejemplo, disco). Incluso si los objetos se almacenan en memoria volátil, el servidor puede utilizar la memoria persistente para almacenar suficiente información para recobrar el estado de los objetos en caso que caiga el proceso servidor. Esto permite a los servidores hacer que los objetos sean recuperables. Una transacción viene especificada desde un cliente como un conjunto de operaciones sobre los objetos que se realizarán como una unidad indivisible por los servidores que gestionan dichos objetos. Los servidores deben garantizar que se realiza completamente la transacción y que los resultados se almacenan en una memoria permanente o, en el caso de una o más caídas, sus efectos se eliminan completamente. El siguiente capítulo discute temas relacionados con transacciones que implican varios servidores, en particular el cómo decidir el resultado de una transacción distribuida. Este capítulo se concentra en las cuestiones relacionadas con un único servidor. Cada transacción de un cliente es considerada también como indivisible desde el punto de vista de las transacciones de los otros clientes en el sentido de que las operaciones de una transacción no pueden observar los efectos parciales de las otras transacciones. La Sección 12.1.1 analiza la sincronización simple de acceso a objetos, y la 12.2 presenta las transacciones, que precisan de técnicas más avanzadas para prevenir la interferencia entre clientes. La Sección 12.3 discute las transacciones anidadas. Las Secciones de la 12.4 a la 12.6 discuten tres métodos de control de concurrencia para transacciones cuyas operaciones están dirigidas hacia un servidor único (bloqueos, control optimista de concurrencia, y ordenación por marca de tiempo). El Capítulo 13 discutirá cómo se extienden estos métodos para usarlos con transacciones cuyas operaciones están dirigidas hacia varios servidores.

Para explicar algunas de las consideraciones que se hacen en este capítulo, utilizamos un ejemplo bancario, como se ve en la Figura 12.1. Cada cuenta está representada por un objeto remoto cuya interfaz *Cuenta* proporciona las operaciones de hacer depósitos y realizar reintegros y para consultar y actualizar el balance. Cada sucursal del banco está representada por un objeto remoto cuya interfaz *Sucursal* proporciona operaciones para crear una nueva cuenta, para localizar una cuenta por nombre y para preguntar por el total de los fondos de la sucursal.

12.1.1. SINCRONIZACIÓN SENCILLA (SIN TRANSACCIONES)

Uno de los temas más importantes de este capítulo es que a menos que el servidor esté diseñado cuidadosamente, las operaciones realizadas en nombre de diferentes clientes pueden interferir a veces unas con otras. Dicha interferencia puede producir valores incorrectos en los objetos. En esta sección discutimos cómo pueden estar sincronizadas las operaciones de los clientes sin recurrir a las transacciones.

◇ **Operaciones atómicas en el servidor.** En los capítulos anteriores se ha visto que utilizar múltiples hilos es beneficioso para las prestaciones en muchos servidores. Hemos señalado también que el uso de hilos permite que se ejecuten concurrentemente las operaciones de varios clientes y aun accediendo, posiblemente, a los mismos objetos. Por tanto, los métodos de los objetos

deposita(cantidad)
deposita *cantidad* en la cuenta

extrae(cantidad)
saca *cantidad* de la cuenta

obténBalance() → cantidad
devuelve el balance de la cuenta

ponBalance(cantidad)
inicializa el balance de la cuenta con *cantidad*

Operaciones de la interfaz *Sucursal*

crea(nombre) → cuenta
crea una nueva cuenta con el nombre especificado

busca(nombre) → cuenta
devuelve una referencia a la cuenta con el nombre especificado

totalSucursal() → cantidad
devuelve el total de todos los balances de la sucursal

Figura 12.1. Operaciones de la interfaz *Cuenta.*

debieran estar diseñados para funcionar en un contexto multi-hilo. Por ejemplo, si los métodos *deposita* (*deposit*) y *extrae* (*withdraw*) no están diseñados para su utilización en un programa multi-hilo, es posible que las acciones de dos o más ejecuciones concurrentes del método puedan entremezclarse arbitrariamente y tener efectos extraños en las variables de instancia de los objetos *cuenta*.

El Capítulo 6 explica el uso de la palabra *synchronized* (sincronizado), que puede ser aplicado a los métodos en Java para asegurar que sólo puede acceder a un objeto un hilo cada vez. En nuestro ejemplo, la clase que implementa la interfaz *Cuenta* será capaz de declarar métodos como sincronizados. Por ejemplo:

```
public syncronized void deposita(int cantidad) throws RemoteException{
    // añade cantidad al balance de la cuenta
}
```

Si un hilo invoca un método sincronizado de un objeto, entonces el objeto es bloqueado efectivamente, y otro hilo que invoque uno de sus métodos sincronizados será bloqueado hasta que el bloqueo anterior sea liberado. Esta forma de sincronización fuerza que la ejecución de los hilos sea separada en el tiempo y asegura que las variables instancia de un único objeto sean accedidas de forma consistente. Sin sincronización, dos invocaciones *deposita* separadas podrían leer el balance antes de que ninguna lo hubiera actualizado, lo que produciría un valor incorrecto. Cualquier método que acceda a una variable instancia que pueda variar debe ser *sinchronized.*

Las operaciones que están libres de interferencia de operaciones concurrentes que se están realizando en otros hilos se llaman operaciones atómicas. El uso de métodos sincronizados en Java es una forma de conseguir operaciones atómicas. Pero en otros entornos de programación para servidores multi-hilo las operaciones en objetos necesitan disponer de operaciones atómicas con el fin de mantener sus objetos consistentes. Esto se puede conseguir mediante el uso de cualquier mecanismo de exclusión mutua, como un mútex.

◇ **Mejora de la colaboración del cliente mediante sincronización de las operaciones del servidor.** Los clientes pueden utilizar un servidor como un medio de compartir algunos re-

cursos. Esto se consigue por algunos clientes utilizando operaciones para actualizar los objetos del servidor y otros utilizando operaciones para acceder a ellos. El esquema anterior para acceso sincronizado a objetos proporciona todo lo que se precisa en muchas aplicaciones, previene que los hilos interfieran unos con otros. Sin embargo, algunas aplicaciones necesitan una forma para que los hilos se comuniquen unos con otros.

Por ejemplo, puede surgir una situación en la que la operación requerida por un cliente no puede completarse hasta que se haya realizado otra operación requerida por otro usuario.

Esto puede ocurrir cuando algunos clientes son productores y otros consumidores, los consumidores deben esperar hasta que un productor haya proporcionado algún elemento más del artículo en cuestión. Puede también ocurrir cuando los clientes comparten un recurso, pueden necesitar esperar hasta que otros clientes lo liberen. Se verá más tarde en este capítulo que se llega una situación similar cuando se utilizan los bloqueos o las marcas de tiempo para controlar la concurrencia en las transacciones.

Los métodos *wait* (espera) y *notify* (notifica) Java presentados en el Capítulo 6 permiten que los hilos se comuniquen con los otros de una manera que resuelva los problemas anteriores. Deben utilizarse dentro de los métodos sincronizados de un objeto. Un hilo llama a *wait* en un objeto para suspenderse él mismo y permitir a otro hilo ejecutar un método en ese objeto. Un hilo llama a *notify* para informar que cualquier hilo que esté esperando en el objeto que ha cambiado alguno de sus datos. El acceso a un objeto es todavía atómico cuando un hilo espera por otro: un hilo que llama a *wait* activa su bloqueo y se suspende como una acción atómica, cuando el hilo es activado después de ser notificado adquiere un nuevo bloqueo en el objeto y recupera la ejecución después del *wait*. Un hilo que llama a *notify* (desde un método sincronizado) completa la ejecución del método antes de liberar el bloqueo en el objeto.

Consideremos la implementación de un objeto *Cola* compartida con dos métodos: *primero* extrae y devuelve el primer objeto de la cola, *añade* añade un objeto determinado al final de la cola. El método *primero* comprueba primero si la cola está vacía, en cuyo caso llama a un *wait* en la cola. Si un cliente llama a *primero* cuando la cola está vacía, no conseguirá ninguna respuesta hasta que otro cliente haya añadido algo en la cola, la operación *añade* efectuará *notify* cuando haya añadido un objeto a la cola. Esto permite que uno de los hilos que están esperando continúe su ejecución y devuelva el primer objeto de la cola a su cliente. Cuando los hilos pueden sincronizar sus acciones en un objeto por medio de *wait* y *notify*, el servidor mantiene las solicitudes que no pueden ser satisfechas inmediatamente y el cliente espera por una contestación hasta que otro cliente ha producido lo que necesite.

En la última sección de bloqueos para transacciones, discutimos la implementación de un bloqueo como un objeto con operaciones sincronizadas. Cuando los clientes intentan obtener un bloqueo, se les puede hacer esperar hasta que el bloqueo es liberado por otros clientes.

Sin la capacidad para sincronizar hilos de esta forma, no se puede satisfacer a un cliente inmediatamente, por ejemplo, a un cliente que llame al primer método de una cola vacía se le dice que lo intente más tarde. Esto no es satisfactorio, porque supondrá que el cliente debe estar consultando al servidor y el servidor debe satisfacer estas consultas extras. Es también potencialmente injusto porque otros clientes pueden efectuar solicitudes antes de que el cliente que espera lo intente de nuevo.

12.1.2. MODELO DE FALLOS PARA TRANSACCIONES

Lampson [1981a] propuso un modelo de fallos para transacciones distribuidas que considera los fallos en discos, servidores y comunicación. En este modelo, se intenta que los algoritmos trabajen correctamente en presencia de fallos predecibles, pero no se hacen consideraciones sobre su comportamiento cuando ocurre un desastre. Aunque pueden existir errores, no pueden ser detectados y

considerados antes de que ocurra cualquier comportamiento incorrecto. El modelo establece lo siguiente:

- Las escrituras sobre almacenamiento permanente pueden fallar, o porque no se escribe nada o porque se escribe un valor incorrecto, por ejemplo, escribir en el bloque incorrecto es un desastre. El almacenamiento en disco puede deteriorarse. Las lecturas sobre almacenamiento permanente pueden detectar (mediante una suma de comprobación) cuando un bloque de datos es incorrecto.
- Los servidores pueden fallar ocasionalmente. Cuando un servidor averiado se reemplaza por un nuevo proceso, su memoria volátil se sitúa primeramente en un estado en el que no conoce ninguno de los valores (por ejemplo, de objetos) de antes de la ruptura. Después lleva a cabo un procedimiento de recuperación, utilizando la información del almacenamiento permanente y la obtenida de otros procesos para fijar los valores de los objetos incluyendo los relacionados con el protocolo de consumación (*commit*) en dos fases (se verá en la Sección 13.6). Cuando un procesador está fallando, se le considera caído para prevenir el envío de mensajes erróneos y la escritura de valores incorrectos en el almacenamiento permanente, esto es, para no producir fallos arbitrarios. Las rupturas se pueden producir en cualquier instante de tiempo, en particular, pueden ocurrir durante la recuperación.
- Puede existir un retardo arbitrario antes que llegue un mensaje. Un mensaje puede haberse perdido, duplicado o modificado. El receptor puede detectar mensajes modificados (mediante una suma de comprobación). Tanto a los mensajes falsificados como a los mensajes corruptos no detectados se les contempla como desastres.

El modelo de fallos para almacenamiento permanente, procesadores y comunicaciones se usó para diseñar un sistema estable cuyos componentes pueden sobrevivir a un único fallo y presentar un modelo de fallos sencillo. En particular, el *almacenamiento estable* proporcionaba una operación *write* atómica en presencia de un fallo de la operación *write* o una ruptura del proceso. Esto se consiguió replicando cada bloque en dos bloques de disco. Una operación *write* se aplicaba a los pares de bloques de disco, y en el caso de un fallo siempre estaba disponible un bloque correcto. Un *procesador estable* utilizaba almacenamiento estable para facilitar la recuperación de objetos después de una ruptura. Los errores de comunicación se enmascaraban utilizando un mecanismo de llamada a procedimientos remoto fiable.

12.2. TRANSACCIONES

En algunas situaciones, los clientes necesitan que una secuencia de solicitudes separadas al servidor sean atómicas en el sentido de:

1. Estén libres de interferencia por las operaciones que se están realizando en nombre de otros clientes concurrentes.
2. Todas las operaciones deben ser completadas con éxito o no tendrán ningún efecto si el servidor falla.

Volvemos a nuestro ejemplo bancario para ilustrar las transacciones. Un cliente, que realiza una secuencia de operaciones en una cuenta particular del banco por encargo de un usuario, realizará primero una búsqueda (*busca*) de la cuenta por nombre y después aplicará las operaciones de depositar (*deposita*), extraer (*extrae*) y obtener balance (*obténBalance*) directamente sobre la cuenta considerada. En nuestros ejemplos, utilizamos cuentas con nombres *A*, *B* y *C*. El cliente las localiza y almacena las referencias a ellas en las variables *a*, *b* y *c* de tipo cuenta (*Cuenta*). Los detalles de la búsqueda de cuentas por nombre y las declaraciones de variables se omiten de los ejemplos.

Transacción T:
a.extrae(100);
b.deposita(100);
c.extrae(200);
d.deposita(200);

Figura 12.2. Una transacción bancaria de un cliente.

La Figura 12.2 muestra un ejemplo de la transacción de un cliente que especifica una serie de acciones relacionadas en las que están implicadas las cuentas *A*, *B* y *C*. La primera acción transfiere 100$ de *A* a *B* y la segunda 200$ de *C* a *B*. Un cliente consigue una operación de transferencia realizando un reintegro en una cuenta seguido de un depósito en la otra.

Las transacciones provienen de los sistemas de gestión de bases de datos. En ese contexto, una transacción es la ejecución de un programa que accede a la base de datos. Las transacciones fueron introducidas en los sistemas distribuidos en la forma de servidores de archivos transaccionales como XDFS [Mitchell y Dion 1982]. En este contexto, de un servidor de archivos transaccional, una transacción es la ejecución de una secuencia de peticiones de operaciones sobre archivos de los clientes. En varios sistemas de investigación se han proporcionado transacciones sobre objetos distribuidos, incluyendo Argus [Liskov 1988] y Arjuna [Shrivastava y otros 1991]. En este último contexto, una transacción consiste en la ejecución de una secuencia de peticiones del cliente como, por ejemplo, las de la Figura 12.2. Desde el punto de vista del cliente, una transacción es una secuencia de operaciones que forma un único paso, transformando los datos del servidor de un estado consistente a otro.

Las transacciones pueden venir dadas como una parte del middleware. Por ejemplo, CORBA proporciona la especificación para un Servicio de Transacciones de Objetos (*Object Transaction Service*) [OMG 1977e] con interfaces IDL que permiten a las transacciones de los clientes incluir múltiples objetos en múltiples servidores. Al cliente se le proporcionan operaciones para especificar el comienzo y el final de una transacción. El cliente ORB mantiene un contexto para cada transacción, que evoluciona con cada operación en la transacción. Los objetos CORBA se pueden hacer transaccionales extendiendo sus interfaces con una llamada *TransactionalObject*.

En todos estos contextos, una transacción se aplica a objetos recuperables y está pensada para ser atómica. A menudo se la llama transacción atómica (ver la caja siguiente). Hay dos aspectos de atomicidad:

Todo o nada: Una transacción o finaliza correctamente, y los efectos de todas sus operaciones son registrados en los objetos, o (si falla o es abortada deliberadamente) no tiene ningún efecto. Este efecto *todo o nada* tiene otros dos aspectos en sí mismo:

Atomicidad de fallo: los efectos son atómicos aun en el caso de ruptura del servidor.

Durabilidad: después que una transacción ha finalizado con éxito, todos sus efectos son guardados en almacenamiento permanente. Utilizamos el término «almacenamiento permanente» para referirnos a archivos que se mantienen en disco o cualquier otro soporte permanente. Los datos guardados en disco sobrevivirán incluso en el caso de ruptura del servidor.

Aislamiento: Cada transacción debe ser realizada sin interferencia de otras transacciones, en otras palabras, los efectos intermedios de una transacción no deben ser visibles para las demás.

Para dar soporte al requisito de atomicidad de fallo y durabilidad, los objetos deben ser recuperables, cuando un proceso servidor cae inesperadamente, debido a un fallo hardware o a un error software, los cambios debidos a todas las transacciones completadas deben estar disponibles en el almacenamiento permanente de forma que cuando el servidor sea reemplazado por un nuevo proceso, se pueden recuperar los objetos para reflejar el efecto *todo o nada*. Cada vez que un servidor

abreTransacción( ) → trans;
comienza una nueva transacción y proporciona una única TID *trans*. Este identificador será utilizado en el resto de las operaciones de la transacción.

cierraTransacción(trans) → (consumado, abortado);
finaliza una transacción: la devolución de un valor *consumado* indica que la transacción se ha consumado; la devolución de un valor *abortado* indica que ella se ha abortado.

abortaTransacción(trans);
aborta la transacción.

Figura 12.3. Operaciones en la Interfaz *Coordinador*.

reconoce la finalización de una transacción del cliente, todos los cambios de la transacción en los objetos deben haber sido registrados en almacenamiento permanente.

Un servidor que soporta transacciones debe sincronizar suficientemente las operaciones para asegurar que se satisface el requisito de aislamiento. Una forma de hacerlo es realizar las transacciones secuencialmente, una cada vez en un orden arbitrario. Desafortunadamente, esta solución sería generalmente inaceptable para servidores cuyos recursos son compartidos por múltiples usuarios interactivos. En nuestro ejemplo bancario, es deseable permitir a distintos cajeros realizar transacciones en línea al mismo tiempo.

El fin de cualquier servidor que soporta transacciones es maximizar la concurrencia. Se permite, por tanto, que se ejecuten concurrentemente transacciones si el efecto de ellas fuera el mismo que si se ejecutaran secuencialmente, es decir si fueran *secuencialmente equivalentes* o *secuenciables*.

Se pueden añadir recursos de transacciones a los servidores de objetos recuperables. Cada transacción es creada y gestionada por un administrador, que implementa la interfaz *Coordinador* mostrada en la Figura 12.3. El coordinador da a cada transacción un identificador, o TID. El cliente invoca el método *abreTransacción* del coordinador para introducir una nueva transacción, se asigna un identificador de transacción, o TID, y se devuelve. Al final de una transacción, el cliente invoca el método *cierraTransacción* para indicar su fin; todos los objetos recuperables accedidos durante la transacción debieran ser guardados. Si, por alguna razón, el cliente quiere abortar la transacción, invoca el método *abortaTransacción*, y todos los efectos debieran ser eliminados.

◇ **Propiedades ACID.** Härder y Reuter [1983] sugirieron el nemotécnico *ACID* (*Atomicity Consistency Isolation Durability*) para recordar las propiedades de las transacciones de la forma siguiente:

Atomicidad: una transacción debe ser todo o nada.

Consistencia: una transacción hace pasar el sistema de un estado consistente a otro.

Aislamiento.

Durabilidad.

No hemos incluido *consistencia* en nuestra propia lista de propiedades de las transacciones porque generalmente es responsabilidad de los programadores de los servidores y los clientes el asegurar que las transacciones dejen la base de datos consistente.

Como un ejemplo de consistencia, supongamos que en el ejemplo del banco, un objeto mantiene la suma de todos los balances de las cuentas y su valor se utiliza como el resultado de *totalSucursal*. Los clientes pueden obtener la suma de los balances de todas las cuentas utilizando *totalSucursal* o llamando a *obténBlance* en cada cuenta. Para la consistencia, se debe obtener el mismo resultado de ambos métodos. Para mantener esta consistencia, las operaciones de *deposita* y *extrae* deben actualizar el objeto conservando la suma de todos los balances de las cuentas.

Con éxito	*Abortado por el cliente*	*Abortado por el servidor*	
AbreTransacción	*AbreTransacción*		*AbreTransacción*
Operación	*Operación*		*Operación*
Operación	*Operación*		*Operación*
•	•	El servidor aborta	•
•	•	la transacción →	•
Operación	*Operación*		*ERROR en la operación informado al cliente*
CierraTransacción	*AbortaTransacción*		

Figura 12.4. Historias de vida de una transacción.

Cada transacción se obtiene por cooperación entre un programa cliente, algunos objetos recuperables y un coordinador. El cliente especifica la secuencia de invocaciones sobre los objetos recuperables que forman la transacción. Para conseguir esto, el cliente envía con cada invocación el identificador de la transacción devuelto por *abreTransacción*. Una forma de hacer esto posible es incluir el TID como un argumento extra en cada operación de un objeto recuperable. Por ejemplo, en el servicio bancario la operación *deposita* podría definirse:

deposita(trans, cantidad)
deposita la *cantidad* en la cuenta para la transacción cuyo TID es *trans*

Cuando se proporcionan las transacciones como middleware, se puede pasar el TID implícitamente en cada invocación remota entre *abreTransacción* y *cierraTransacción* o *abortaTransacción*. Esto es lo que hace el servicio de transacciones de CORBA. En nuestros ejemplos no mostraremos los TID.

Así, una transacción finaliza cuando el cliente realiza una solicitud *cierraTransacción*. Si la transacción ha progresado normalmente la contestación indica que la transacción está consumada, esto constituye una garantía para el cliente de que todos los cambios solicitados en la transacción se registran permanentemente y que cualquiera de las transacciones futuras que accedan a los mismos datos verán los resultados de todos los cambios realizados durante la transacción.

Alternativamente, la transacción puede tener que abortar por alguna de distintas razones relacionadas con la naturaleza de la transacción misma, por conflictos con otra transacción o por fallo de un proceso o computador. Cuando una transacción es abortada, las partes implicadas (los objetos recuperables y el coordinador) deben asegurar que ninguno de los efectos es visible para futuras transacciones, en los objetos o en sus copias en almacenamiento permanente.

Una transacción o tiene éxito o se aborta de una de las dos maneras siguientes, el cliente la aborta (utilizando una llamada *abortaTransacción* para el servidor) o la aborta el servidor. La Figura 12.4 muestra estas tres historias de vida alternativas para las transacciones. Nos referimos a una transacción como *en fallo*, o *fallando*, en ambos casos.

◇ **Acciones de servicio relacionadas con la ruptura de un proceso.** Si un proceso servidor falla inesperadamente, se reemplazará en algún momento. El nuevo proceso servidor aborta todas las transacciones no finalizadas y usa un procedimiento de recuperación para restablecer los valores de los objetos a los valores producidos por la transacción finalizada de forma correcta más recientemente. Para tratar con un cliente que falla inesperadamente durante una transacción, los servidores pueden dar a cada transacción un tiempo de expiración y abortar cualquier transacción que no haya sido completada antes de su tiempo de expiración.

◇ **Acciones del cliente relativas a la ruptura del proceso servidor.** Si un servidor falla mientras una transacción está en progreso, el cliente será consciente de ello cuando una de las operaciones devuelve una excepción después de un *timeout* (tiempo límite de espera). Si un servidor falla y es reemplazado durante el progreso de una transacción, ésta ya no será válida y el cliente debe ser informado con una excepción para la siguiente operación. En cualquier caso, el cliente debe formular un plan, posiblemente consultando con la persona humana, para la finalización o el abandono de la tarea de la que la transacción era una parte.

12.2.1. CONTROL DE CONCURRENCIA

Esta sección ilustra dos problemas bien conocidos de transacciones concurrentes en el contexto del ejemplo bancario, el problema de las «actualizaciones perdidas» y el de las «recuperaciones inconsistentes». Esta sección muestra cómo ambos problemas pueden evitarse efectuando ejecuciones de transacciones secuencialmente equivalentes. Suponemos que cada operación *deposita*, *extrae*, *obténBalance y ponBalance* está sincronizada, es decir, sus efectos en la variable de instancia que registra el balance de la cuenta es atómica.

◇ **El problema de las actualizaciones perdidas.** Este problema se muestra mediante el siguiente par de transacciones sobre las cuentas bancarias *A*, *B* y *C*, cuyos balances iniciales son 100$, 200$ y 300$ respectivamente. La transacción *T* transfiere cierta cantidad desde la cuenta *A* a la cuenta *B*. La transacción *U* transfiere otra cantidad desde la cuenta *C* a la *B*. En ambos casos, la cantidad transferida se calcula para incrementar el balance de *B* en un 10 %. Los efectos netos sobre la cuenta *B* al ejecutar las transacciones *T* y *U* debieran hacer incrementar el balance de la cuenta *B* en un 10 % dos veces, por lo que el valor final será 242$.

Consideremos ahora los efectos de permitir que las transacciones *T* y *U* se ejecuten concurrentemente, como en la Figura 12.5. Ambas transacciones obtiene el balance de *B* como 200$ y después depositan 20$. El resultado es incorrecto, al incrementar el balance de la cuenta *B* en 20$ en lugar de 42$. Esto es un ejemplo del problema de las «actualizaciones perdidas». La actualización de *U* se pierde porque *T* escribe sin mirar. Ambas transacciones han leído el valor previo antes de escribir el nuevo valor.

En la Figura 12.5, se muestran sucesivamente las operaciones que afectan al balance de una cuenta, y el lector debe suponer que la operación de una cierta línea se ejecuta en un momento posterior al de otra que se encuentra en una línea superior.

Transacción *T*:		**Transacción *U*:**	
balance = b.obténBalance();		*balance = b.obténBalance();*	
*b.ponBalance(balance*1.1);*		*b.ponBalance(balance*1.1);*	
a.extrae(balance/10);		*c.extrae(balance/10);*	
balance = b.obténBalance();	200$		
		balance = b.obténBalance();	200$
		*b.ponBalance(balance*1.1);*	220$
*b.ponBalance(balance*1.1) ;*	220$		
a.extrae(balance/10);	80$		
		c.extrae(balance/10);	280$

Figura 12.5. El problema de las actualizaciones perdidas.

Transacción *V*: *a.extrae(100);* *b.deposita(100)*		**Transacción *W*:** *unasucursal.totalSucursal()*	
a.extrae(100);	100$	*total = a.obténBalance();*	100$
		total = total + b.obténBalance();	300$
		total = total + c.obténBalance()	
b.deposita(100)	300$	•	
		•	

Figura 12.6. El problema de las recuperaciones inconsistentes.

◇ **Recuperaciones inconsistentes.** La Figura 12.6 muestra otro ejemplo relacionado con una cuenta bancaria en la que la transacción *V* transfiere una suma desde la cuenta *A* hasta la *B* y la transacción *W* invoca el método *totalSucursal* para obtener la suma de los balances de todas las cuentas del banco. Los balances de dos cuentas bancarias, *A* y *B*, son ambos inicialmente 200$. El resultado de *totalSucursal* incluyen la suma de *A* y *B* como 300$, que es incorrecto. Esto es un ejemplo del problema de las «recuperaciones inconsistentes». Las recuperaciones de *W* son inconsistentes porque *V* había realizado sólo la parte de extracción de su transferencia mientras se calculaba la suma.

◇ **Equivalencia secuencial.** Si se sabe que cada una de las distintas transacciones tiene el efecto correcto cuando se realiza ella sola, podemos inferir que si estas transacciones se realizan una cada vez en el mismo orden, el efecto combinado también será correcto. Un solapamiento de las operaciones de las transacciones en las que el efecto combinado es el mismo que si las transacciones hubieran sido realizadas una cada vez en el mismo orden es un solapamiento secuencialmente equivalente. Cuando decimos que dos transacciones diferentes tienen el mismo efecto, significa que las operaciones devuelven los mismos valores y que las variables de instancia de los objetos tienen los mismos valores al final.

El uso de la equivalencia secuencial como un criterio para la ejecución concurrente correcta previene la ocurrencia de actualizaciones perdidas y recuperaciones inconsistentes.

El problema de las actualizaciones perdidas ocurre cuando dos transacciones leen el valor antiguo de una variable y la utilizan para calcular el nuevo valor. Esto no puede ocurrir si una transac-

Transacción *T*: *balance = b.obténBalance();* *b.ponBalance(balance*1.1);* *a.extrae(balance/10);*		**Transacción *U*:** *balance = b.obténBalance();* *b.ponBalance(balance*1.1);* *c.extrae(balance/10);*	
balance = b.obténBalance();	200$		
*b.ponBalance(balance*1.1);*	220$		
		balance = b.obténBalance();	220$
		*b.ponBalance(balance*1.1);*	242$
a.extrae(balance/10);	80$		
		c.extrae(balance/10);	278$

Figura 12.7. Un solapamiento de *T* y *U* secuencialmente equivalente.

Transacción *V*: *a.extrae(100);* *b.deposita(100)*		**Transacción *W*:** *unasucursal.totalSucursal();*	
a.extrae(100);	100$		
b.deposita(100)	300$	*total = a.obténBalance()*	100$
		total = total + b.obténBalance();	400$
		total = total + c.obténBalance();	
		...	

Figura 12.8. Un solapamiento de *V* y *W* secuencialmente equivalente.

ción se realiza antes que la otra, porque la última transacción leerá el valor escrita por la anterior. Como un solapamiento secuencialmente independiente de dos transacciones produce el mismo efecto de una secuencial, podemos resolver el problema de las actualizaciones perdidas mediante la equivalencia secuencial. La Figura 12.7 muestra dicho solapamiento en el que las operaciones que afectan a la cuenta compartida *B*, son realmente secuenciales, puesto que la transacción *T* realiza todas las operaciones antes que las realice la transacción *U*. Otro solapamiento de *T* y *U* que tenga esta propiedad es uno en el que la transacción *U* finaliza sus operaciones en la cuenta *B* antes de que comience la transacción *T*.

Vamos a considerar ahora el efecto de la equivalencia secuencial con relación al problema de las recuperaciones equivalentes, en la que la transacción *V* está transfiriendo una suma de desde la cuenta *A* a la cuenta *B* y la transacción *W* está obteniendo la suma de todos los balances (véase la Figura 12.6). El problema de las recuperaciones inconsistentes puede ocurrir cuando una transacción de recuperación se ejecuta concurrentemente con una transacción de actualización. No puede ocurrir si la transacción de recuperación se realiza antes o después de una transacción de actualización. Un solapamiento secuencialmente equivalente de una transacción de recuperación y una de actualización, por ejemplo como en la Figura 12.8, impedirá que ocurran las recuperaciones inconsistentes.

◇ **Operaciones conflictivas.** Cuando decimos que un par de operaciones tienen conflictos queremos decir que su efecto combinado depende del orden en el que se ejecutan. Para simplificar las cosas consideremos un par de operaciones *lee* y *escribe*. *Lee* accede al valor de un objeto y *escribe* cambia su valor. El *efecto* de una operación se refiere al valor colocado en el objeto por una operación *escribe* y el resultado devuelto por una operación *lee*. En la Figura 12.9 se dan las reglas de conflicto para *lee* y *escribe*.

Operaciones de diferentes transacciones		*Conflicto*	*Causa*
Lee	*Lee*	No	Porque el efecto de un par de operaciones de *lectura* no depende del orden en el que son ejecutadas
Lee	*Escribe*	Sí	Porque el efecto de una operación de *lectura* y una de *escritura* dependen del orden de su operación
Escribe	*Escribe*	Sí	Porque el efecto de un par de operaciones de *escritura* depende del orden de su ejecución

Figura 12.9. Reglas de conflicto en las operaciones *lee* y *escribe*.

Transacción *T*:	Transacción *U*:
x = lee(i)	
escribe(i, 10)	
	y = lee(j)
	escribe(j, 30)
escribe(j, 20)	
	z = lee(i)

Figura 12.10. Un solapamiento de las operaciones de las transacciones *T* y *U* no secuencialmente equivalente.

Para cualquier par de transacciones, es posible determinar el orden de pares de operaciones conflictivas sobre objetos accedidos por ambas. La equivalencia secuencial puede definirse en términos de las operaciones conflictivas como sigue:

Para que dos transacciones sean *secuencialmente equivalentes*, es necesario y suficiente que todos los pares de operaciones conflictivas de las dos transacciones se ejecuten en el mismo orden sobre los objetos a los que ambas acceden.

Consideremos como ejemplo las transacciones *T* y *U*, definidas como sigue:

T: x = lee(i); escribe(i, 10); escribe(j, 20);
U: y = lee(j); escribe(j, 30); z = lee(i);

Consideremos ahora el solapamiento de sus ejecuciones, mostrado en la Figura 12.10. Toma nota que el acceso de cada transacción a los objetos *i* y *j* es secuencial con respecto al otro, porque *T* realiza todos sus accesos a *i* antes de que lo haga *U* y *U* hace todos sus accesos a *j* antes de que lo haga *T*. Pero la ordenación no es secuencialmente equivalente, porque los pares de operaciones conflictivas no se hacen el mismo orden en ambos objetos. Las ordenaciones secuencialmente equivalentes requieren una de las dos condiciones siguientes:

1. *T* accede a *i* antes de *U* y *T* accede a *j* antes de *U*.
2. *U* accede a *i* antes de *T* y *U* accede a *j* antes de *T*.

La equivalencia secuencial se utiliza como un criterio para la obtención de protocolos de control de concurrencia. Estos protocolos intenta secuencializar las transacciones en sus accesos a los objetos. Para el control de concurrencia se utilizan normalmente tres aproximaciones alternativas: bloqueo, control de concurrencia optimista y ordenación por marca de tiempo. Sin embargo, los sistemas más pragmáticos utilizan el bloqueo, que se discutirá en la Sección 12.4. Cuando se utiliza el bloqueo, el servidor establece un bloqueo, etiquetado con el identificador de la transacción, en cada objeto justo en el momento antes de ser accedido y elimina estos bloqueos cuando la transacción ha finalizado. Mientras un objeto está bloqueado, sólo la transacción que ha realizado el bloqueo puede acceder al objeto; las otras transacciones deberán esperar hasta que el objeto sea desbloqueado o, en algunos casos, compartir el bloqueo. El uso de bloqueos puede conducir al bloqueo mutuo (*deadlock*), en el que cada transacción espera a que la otra libere los bloqueos, como, por ejemplo, cuando un par de transacciones ha bloqueado cada una un objeto bloqueado al que precisa acceder la otra. Discutiremos el problema del bloqueo mutuo y algunas soluciones para ello en la Sección 12.4.1.

El control optimista de la concurrencia se describe en la Sección 12.5. En los esquemas optimistas, una transacción avanza hasta que solicita la consumación, y antes de que se le autorice a realizar la consumación el servidor realiza una comprobación para descubrir si se han efectuado

operaciones —sobre objetos— que presentan algún conflicto con las operaciones de otras transacciones concurrentes, en cuyo caso el servidor la aborta y el cliente podrá iniciarla de nuevo. El objetivo de la comprobación es asegurar que todos los objetos son correctos.

La ordenación por marcas de tiempo se describe en la Sección 12.6. En esta ordenación, un servidor registra el tiempo más reciente de lectura y escritura en cada objeto y para cada operación, se compara la marca de tiempo de la transacción con la del objeto para determinar si puede realizarse inmediatamente, o bien retrasarse o rechazarse. Cuando se retrasa una operación, la transacción espera, cuando es rechazada; cuando es rechazada, la transacción es abortada.

Básicamente, el control de concurrencia puede ser obtenido o porque los clientes de las transacciones esperen uno por otro o reiniciando las transacciones después de que se hayan detectado conflictos entre operaciones, o bien mediante una combinación de las dos.

12.2.2. RECUPERABILIDAD DE TRANSACCIONES ABORTADAS

Los servidores deben registrar los efectos de todas las transacciones finalizadas y ninguno de los efectos de las transacciones abortadas. Deben considerar, por tanto, el hecho de que una transacción pueda ser abortada en previsión de que afecte a otras transacciones concurrentes si es el caso.

Esta sección muestra dos problemas asociados con las transacciones abortadas en el contexto del ejemplo del banco. Estos problemas se conocen como «lecturas sucias» y «escrituras prematuras», y ambas pueden aparecer en presencia de ejecuciones secuencialmente equivalentes. Estas cuestiones están relacionadas con el efecto de las operaciones sobre objetos, tales como el balance de una cuenta de banco. Para simplificar las cosas, se consideran dos categorías de operaciones, operaciones de lectura (*lectura*) y de escritura (*escritura*). En nuestras ilustraciones, *obténBalance* es una operación de lectura y *ponBalance* es una de escritura.

◇ **Lecturas sucias.** La propiedad de aislamiento de las transacciones requiere que éstas no vean el estado no finalizado de las demás. El problema de la «lectura sucia» está causado por la interacción entre una operación de lectura en una transacción y una operación temprana de escritura en otra transacción sobre el mismo objeto. Considérense las trazas mostradas en la Figura 12.11, donde *T* consigue el balance de la cuenta *A* y le añade 10$ más, a continuación *U* consigue el balance de *A* y le añade 20$ más, y las dos ejecuciones son secuencialmente equivalentes. Supongamos ahora que la transacción *T* aborta después de que *U* ha finalizado. Entonces la transacción *U* habrá visto un valor que nunca ha existido, puesto que se restaurará el valor original en *A*. En este caso se dice que la transacción *U* ha realizado una *lectura sucia*. Como ha finalizado no puede ser desecha.

Transacción *T*:		**Transacción *U*:**	
a.obténBalance()		*a.obténBalance()*	
a.ponBalance(balance + 10)		*a.ponBalance(balance + 20)*	
balance = a.obténBalance()	100$		
a.ponBalance(balance + 10)	110$		
		balance = a.obténBalance()	110$
		a.ponBalance(balance + 20)	130$
		consumar transacción	
abortar transacción			

Figura 12.11. Una lectura sucia cuando se aborta *T*.

◇ **Recuperación de transacciones.** Si una transacción (como *U*) ha finalizado después de que ha visto los efectos de una transacción que posteriormente ha sido abortada, la situación no es recuperable. Para asegurar que tales situaciones no se planteen, cualquier transacción (como *U*) que esté en peligro de tener una lectura sucia rechaza la finalización de su operación. La estrategia para la recuperación es retrasar la finalización hasta después de que haya finalizado cualquier otra transacción cuyo estado no finalizado haya sido observado. En nuestro ejemplo, *U* retrasa su finalización hasta después que se produzca la finalización de *T*. En el caso en que *T* sea abortada, *U* debe abortar también.

◇ **Abortos en cascada.** En la Figura 12.11, suponemos que la transacción *U* retrasa su consumación hasta después que *T* aborte. Como hemos dicho, *U* debe abortar también. Desafortunadamente, si algunas otras transacciones han visto los efectos debidos a *U*, deberían también ser abortadas. El aborto de estas últimas transacciones pude causar que se aborten todavía más transacciones. Tales situaciones se conocen como *abortos en cascada*. Para evitar los abortos en cascada, solo se permite a las transacciones leer objetos que fueron escritos por transacciones consumadas. Para asegurar que esto es así, debe retrasarse cualquier operación de *lectura* hasta que haya sido consumada o abortada cualquier otra transacción que haya realizado una operación de *escritura* sobre el mismo objeto. La evitación de los abortos en cascada es una condición más fuerte que la recuperabilidad.

◇ **Escrituras prematuras.** Consideremos otra implicación de la posibilidad que una transacción pueda abortar. Ésta es una relacionada con las interacciones entre operaciones de *escritura* en el mismo objeto que se realizan en diferentes transacciones. Como ilustración, consideremos dos transacciones *InicializaBalance T* y *U* sobre la cuenta *A*, como se muestra en la Figura 12.12. Antes de las transacciones el balance de la cuenta *A* era de 100$. Las dos ejecuciones son secuencialmente equivalentes, con *T* poniendo el balance a 105$ y *U* a 110$. Si la transacción *U* aborta y *T* se consuma, el balance debería ser 105$.

Algunos sistemas de bases de datos implementan la acción *aborta* restableciendo las «imágenes anteriores» de todas las *escrituras* de una transacción. En nuestro ejemplo, *A* es 100$ inicialmente, que es la «imagen anterior» de la *escritura* de *T*; de forma similar 105$ es la «imagen anterior» de la *escritura* de *U*. Por tanto si *U* aborta, conseguimos el balance correcto de 105$.

Ahora consideremos el caso en que *U* se consuma y después *T* aborta. El balance debería estar a 110$, pero la imagen anterior de la *escritura* de *T* es 100$, por lo que conseguimos el balance incorrecto de 100$. De forma similar si *T* aborta y después *U* aborta, la «imagen anterior» de la *escritura* de *U* es 105$ por lo que conseguimos el balance erróneo de 105$, el balance debería proporcionar 100$.

Para garantizar los resultados correctos en un esquema de recuperación que utiliza las imágenes anteriores, las operaciones de *escritura* deben ser retrasadas hasta que hayan sido consumadas o abortadas las transacciones anteriores que actualizaron los mismos objetos.

◇ **Ejecuciones estrictas de las transacciones.** Generalmente, se requiere que las transacciones retrasen sus operaciones de *lectura* y *escritura* lo suficiente como para impedir tanto las «lecturas sucias» como las «escrituras prematuras». Las ejecuciones de las transacciones se llamas *estrictas* si el servicio de las operaciones de *lectura* y de *escritura* sobre un objeto se retrasa hasta que todas las transacciones que previamente escribieron el objeto han sido consumadas o abortadas. La ejecución estricta de las transacciones hace cumplir la deseada propiedad de aislamiento.

Transacción *T*:		**Transacción *U*:**	
a.ponBalance(105)		*a.ponBalance(110)*	
	100$		
a.ponBalance(105)	105$		
		a.ponBalance(110)	110$

Figura 12.12. Reescritura en valores no consumados.

◇ **Versiones provisionales.** Para que un servidor de objetos recuperables participe en las transacciones, debe estar diseñado de forma que las actualizaciones de los objetos puedan ser eliminadas si y cuando la transacción aborta. Para hacer esto posible, todas las operaciones de actualización realizadas durante una transacción se hacen sobre versiones provisionales de los objetos en memoria volátil. A cada transacción se le proporciona su propio conjunto privado de versiones provisionales de cualquiera de los objetos que ella ha alterado. Todas las operaciones de actualización de una transacción almacenan valores de los objetos en el propio conjunto privado de la transacción si es posible, o fallan.

Las versiones provisionales son transferidas a los objetos sólo cuando una transacción se consuma, en cuyo caso ellas serán también registradas en memoria permanente. Esto se realiza en un único paso, durante el cual las otras transacciones son excluidas de acceder a los objetos que están siendo alterados. Cuando una transacción aborta, sus versiones provisionales se borran.

12.3. TRANSACCIONES ANIDADAS

Las transacciones anidadas extienden el modelo anterior de transacción permitiendo que las transacciones estén compuestas de otras. Por tanto desde una transacción se pueden arrancar varias transacciones, permitiendo considerarlas como módulos que pueden componerse cuando se precise.

La transacción más exterior en un conjunto de transacciones anidadas se llama la transacción de *nivel superior*. Las transacciones diferentes de la del nivel superior se llaman *subtransacciones*. Por ejemplo en la Figura 12.13, *T* es una transacción de *nivel-superior*, que inicializa un par de transacciones T_1 y T_2. La subtransacción T_1 inicializa su propio par de transacciones T_{11} y T_{12}. Del mismo modo, la transacción T_2 inicia su propia subtransacción T_{21} que inicia otra subtransacción T_{211}.

Una subtransacción se presenta como atómica a su padre con respecto a los fallos en la transacción y al acceso concurrente. Las subtransacciones del mismo nivel, como T_1 y T_2 pueden ejecutarse concurrentemente pero sus accesos a objetos comunes son secuenciados, por ejemplo mediante el esquema de bloqueo descrito en la Sección 12.4. Cada subtransacción puede fallar independientemente de su padre y otras subtransacciones. Cuando una subtransacción aborta, la

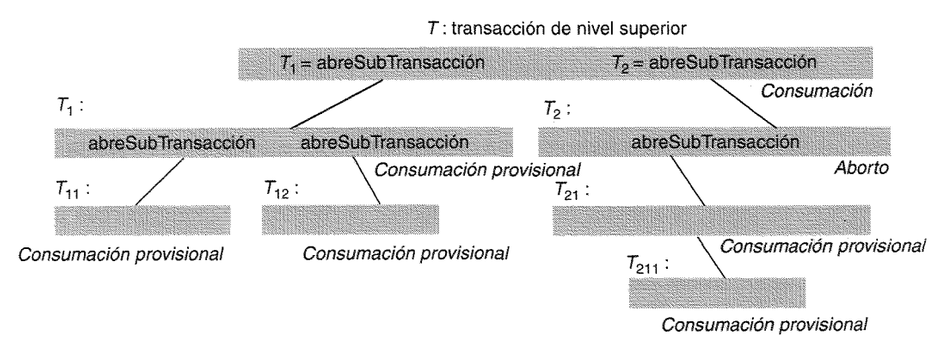


Figura 12.13. Transacciones anidadas.

transacción padre puede elegir una subtransacción alternativa para completar su tarea. Por ejemplo, una transacción para entregar un mensaje de correo a una lista de destinatarios podría estar estructurada como un conjunto de subtransacciones, cada una de las cuales entrega el mensaje a uno de los destinatarios. Si una o más subtransacciones falla, la transacción padre podría registrar el hecho y después consumarse con el resultado que todas las subtransacciones hijas con éxito se consuman. Podría empezar entonces otra transacción que intentara redirigir los mensajes que no fueron enviados la primera vez.

Cuando necesitamos distinguir nuestra forma original de transacción de una anidada, utilizamos el término transacción *plana*. Es plana porque todo su trabajo se realiza al mismo nivel entre un *abreTransacción* y un *consuma* o *aborta*, y no es posible consumar o abortar partes de ella. Las transacciones anidadas tienen las siguientes ventajas principales:

1. Las subtransacciones en un nivel se ejecutan concurrentemente con otras en el mismo nivel de la jerarquía. Esto puede permitir concurrencia adicional de una transacción. Cuando las subtransacciones se ejecutan en diferentes servidores, pueden trabajar en paralelo. Por ejemplo, consideremos la operación *totalSucursal* en nuestro ejemplo bancario. Puede implementarse invocando *obténBalance* en cada cuenta de la sucursal. Ahora bien, cada una de esas invocaciones puede realizarse como una subtransacción, en cuyo caso se podrán realizar concurrentemente. Puesto que cada una se aplica a una cuenta diferente, no habrá operaciones conflictivas entre las subtransacciones.
2. Las subtransacciones se pueden consumar o abortar independientemente. En comparación con una única transacción, un conjunto de subtransacciones anidadas es potencialmente más robusto. El ejemplo anterior de entrega de correo muestra que esto es cierto (con una transacción plana, un fallo en la transacción produciría el que la transacción fuera totalmente reiniciada). De hecho, un padre puede decidir diferentes acciones de acuerdo con si una subtransacción ha abortado o no.

Las reglas para la consumación de transacciones anidadas son bastante sutiles:

- Una transacción se puede consumar o abortar sólo después de que se han completado sus transacciones hijas.
- Cuando un subtransacción finaliza, hace una decisión independiente sobre si consumarse provisionalmente o abortar. Su decisión de abortar es final.
- Cuando un padre aborta, todas sus subtransacciones son abortadas. Por ejemplo, si T_2 aborta entonces T_{21} y T_{211} deben abortar también, incluso aunque ellas puedan haber sido consumadas provisionalmente.
- Cuando una subtransacción aborta, el padre puede decidir si abortar o no. En nuestro ejemplo, T decide consumarse aunque T_2 ha abortado.
- Si se consuman las transacciones de alto nivel, entonces todas las subtransacciones que se han consumado provisionalmente pueden consumarse también, proporcionando que ninguno de sus antecesores ha abortado. En nuestro ejemplo, la consumación de T permite que se consumen T_1, T_{11} y T_{12}, pero no T_{21} y T_{211} puesto que su padre T_2 ha abortado. Hay que considerar que los efectos de una subtransacción no son permanentes hasta que no se consuma la transacción de nivel superior.

En algunos casos, la transacción de nivel superior puede decidir abortar porque una o más de sus subtransacciones han abortado. Como un ejemplo, consideremos la siguiente transacción de *Transferencia*:

Transfiere 100$ desde *A* hasta *B*
a.deposita(100)
b.extrae(100)

Esto se puede estructurar como un par de transacciones, una para la operación de *extrae* y la otra para *deposita*. Cuando las dos subtransacciones se consuman, la transacción *Transferencia* puede consumarse también. Supongamos que una transacción *extrae* aborta cuando una cuenta tiene un descubierto. Ahora consideremos el caso en que la subtransacción *extrae* aborta y la subtransacción *deposita* se consuma; y recuerde que la consumación de una transacción hijo está condicionada a la consumación de la transacción padre. Suponemos que la transacción de alto nivel (*Transferencia*) decidirá abortar. El aborto de la transacción padre produce que las subtransacciones aborten, por lo que la operación *deposita* se aborta y todos sus efectos se eliminan.

El Servicio de Transacciones de Objetos CORBA permite tanto transacciones planas como anidadas. Las transacciones anidadas se utilizan particularmente en los sistemas distribuidos porque las transacciones hijas pueden ejecutarse concurrentemente en distintos servidores. Volveremos a este tema en el Capítulo 14. Esta forma de transacciones anidadas se debe a Moss [1985]. Se han propuesto otras variantes de transacciones anidadas con diferentes propiedades de secuencialidad; por ejemplo, ver Weikum [1991].

12.4. BLOQUEOS

Las transacciones deben planificarse de forma que sus efectos sobre datos compartidos sean secuencialmente equivalentes. Un servidor puede conseguir la equivalencia secuencial de las transacciones secuenciando el acceso a los objetos. La Figura 12.7 representaba un ejemplo de cómo se puede conseguir la equivalencia secuencial con algún grado de concurrencia. Ambas transacciones T y U acceden a la cuenta B, pero T completa su acceso antes que U comience a acceder a ella.

Un ejemplo sencillo de un mecanismo de secuenciación es el uso de bloqueos exclusivos. En este esquema de bloqueo el servidor intenta bloquear cualquier objeto que vaya a ser utilizado por cualquier operación de la transacción de un cliente. Si un cliente solicita el acceso a un objeto que ya está bloqueado por una transacción de otro cliente, la solicitud es suspendida y el cliente debe esperar hasta que el objeto es desbloqueado.

La Figura 12.14 ilustra el uso de bloqueos exclusivos. Muestra las mismas transacciones de la Figura 12.7, pero con una columna extra para cada transacción que representa el bloquo, la espera y el desbloqueo. En este ejemplo se supone que cuando las transacciones T y U comienzan los balances de las cuentas A, B y C no están todavía bloqueados. Cuando la transacción T va a utilizar la cuenta B, B se bloquea para T. Consecuentemente, cuando la transacción U va a utilizar B ésta está todavía bloqueada para T, y la transacción U espera. Cuando se consuma la transacción T, B es desbloqueado, después de lo cual se reanuda la transacción U. El uso del bloque en B serializa efectivamente el acceso a B. Hay que tener en cuenta que si, por ejemplo, T ha liberado el bloqueo en B entre sus operaciones *obténBalance* y *ponBalance*, la operación *obténBalance* de la transacción U puede solaparse entre ellas.

La equivalencia secuencial precisa que todos los accesos de una transacción a un objeto particular sean secuenciados con respecto a los accesos por otras transacciones. Todos los pares de operaciones conflictivas de dos transacciones debieran ser ejecutados en el mismo orden. Para asegurar esto, no está permitido a una transacción ningún nuevo bloqueo después que ha liberado uno. La primera fase de cada transacción se conoce como «fase de crecimiento», durante la cual se adquieren nuevos bloqueos. En la segunda fase, se liberan los bloqueos (una «fase de acortamiento»). Esto se llama *bloqueo en dos fases*.

Vimos en la Sección 12.2.2 que ya que las transacciones pueden abortar, se necesitan ejecuciones estrictas para prevenir lecturas sucias y escrituras prematuras. Bajo un régimen de ejecución estricta, una transacción que necesita leer o escribir un objeto debe ser retrasada hasta que otras transacciones que escribieron el mismo objeto sean consumadas o abortadas. Para hacer cumplir

Transacción *T*:		**Transacción *U*:**	
bal = b.obténBalance()		*bal = b.obténBalance()*	
*b.ponBalance(bal*1.1)*		*b.ponBalance(bal*1.1)*	
a.extrae(bal/10)		*c.extrae(bal/10)*	
Operaciones	Bloqueos	Operaciones	Bloqueos
abreTransacción			
bal = b.obténBalance()	bloquea *B*		
*b.ponBalan ce(bal*1.1)*		*abreTransacción*	
a.extrae(bal/10)	bloquea *A*	*bal = b.ob ténBalance()*	espera por el bloqueo en *B* de *T*
cierraTransacción	desbloquea *A*, *B*	• • •	
			bloquea *B*
		*b.ponBalance(bal*1.1)*	
		c.extrae(bal/10)	bloquea *C*
		cierraTransacción	desbloquea *B*, *C*

Figura 12.14. Las transacciones *T* y *U* con bloqueos exclusivos.

esta regla, se mantienen todos los bloqueos aplicados durante el progreso de una transacción hasta que ésta se consuma o aborta. Esto se llama *bloqueo en dos fases estricto*. La presencia de bloqueos impide que otras transacciones lean o escriban los objetos. Cuando se consuma una transacción, para asegurar la recuperabilidad, los bloqueos se deben mantener hasta que todos los objetos que ella actualizó han sido escritos en memoria permanente.

Un servidor contiene generalmente un gran número de objetos, una transacción típica accede sólo a unos pocos de ellos y es poco probable colisionar con otras transacciones en curso. La *granularidad* con la que puede aplicarse el control de la concurrencia a los objetos es un tema importante, puesto que el ámbito para el acceso concurrente a objetos en un servidor estará limitado severamente si el control de concurrencia (por ejemplo, los bloqueos) sólo se puede aplicar a todos los objetos al instante. En nuestro ejemplo bancario, si se aplican los bloqueos a todas las cuentas de usuario en una sucursal, sólo un cajero del banco podría realizar una transacción bancaria en línea cada vez, ¡una restricción difícilmente aceptable!

La porción de objetos a los que se debe secuenciar el acceso debiera ser tan pequeño como sea posible; es decir, sólo la parte implicada en cada operación solicitada por las transacciones. En nuestro ejemplo bancario, una sucursal mantiene un conjunto de cuentas, cada una de las cuales tiene un balance. Cada operación bancaria afecta a uno o más balances de cuentas. *deposita* o *extrae* afectan al balance de una cuenta y *totalSucursal* afecta al de todas ellas.

La descripción de los esquemas de control de concurrencia dados a continuación no presuponen ninguna granularidad concreta. Discutimos los protocolos del control de concurrencia que son aplicables a objetos cuyas operaciones pueden modelarse en términos de operaciones de *lectura* y *escritura* sobre los objetos. Para que los protocolos funcionen correctamente, es esencial que cada operación de *lectura* y *escritura* sea atómica en sus efectos sobre los objetos.

Los protocolos de control de concurrencia están diseñados para tratar con *conflictos* entre operaciones sobre el mismo objeto en diferentes transacciones. En este capítulo, utilizamos la noción de conflicto entre operaciones para explicar los protocolos. Las reglas de conflicto para operaciones de lectura y escritura están dadas en la Figura 12.9, que muestra qué pares de operaciones de diferentes transacciones en el mismo objeto no entran en conflicto. Por tanto, el utilizar un simple

bloqueo exclusivo tanto para las operaciones de *lectura* y como de *escritura* reduce la concurrencia más de lo necesario.

Es preferible adoptar un esquema de bloqueo que controle el acceso a cada objeto de modo que pueda haber varias transacciones concurrentes leyendo un objeto, o una única transacción escribiendo un objeto, pero no ambas. Esto se denomina normalmente el esquema *muchos lectores/un escritor*. Se utilizan dos tipos de bloqueos: *bloqueos de lectura* y *bloqueos de escritura*. Antes de realizar una operación de *lectura* en una transacción, se debe activar un bloqueo de lectura en el objeto. Antes que se realice una operación de escritura, se debe activar un bloqueo de escritura en el objeto. Cuando es imposible activar un bloqueo inmediatamente, la transacción (y el cliente) debe esperar hasta que es posible hacerlo; tenga en cuenta que nunca se rechaza una solicitud de un cliente.

Como un par de operaciones de lectura, desde transacciones diferentes, no entra en conflicto, un intento de activar un bloqueo de lectura en un objeto que presente un bloqueo de lectura siempre tendrá éxito. Todas las transacciones que leen los mismos objetos comparten su bloqueo de lectura. Por esta razón, los bloqueos de lectura se llaman, a veces, *bloqueos compartidos*.

Las reglas de conflicto de la operación nos dicen que:

1. Si una transacción *T* ha realizado ya una operación de lectura en un objeto particular, entonces una transacción concurrente *U* no debe escribir ese objeto hasta la consumación de *T*, o que aborte.
2. Si una transacción *T* ha realizado ya una operación de escritura en un objeto particular, entonces una transacción concurrente *U* no debe leer o escribir ese objeto la consumación de *T*, o que aborte.

Para hacer cumplir (1), cada solicitud de bloqueo de escritura sobre un objeto se retrasará por la presencia de un bloqueo de lectura que pertenezca a otra transacción. Para hacer cumplir (2), cada solicitud de bloqueo de lectura o escritura sobre un objeto retrasará por la presencia de un bloqueo de escritura perteneciente a otra transacción.

La Figura 12.15 muestra la compatibilidad de los bloqueos de lectura y escritura sobre cualquier objeto concreto. Las entradas en la primera columna de la tabla representan el tipo de bloqueo ya activado, si lo hay. Las entradas en la primera fila representan el tipo de bloqueo solicitado. La entrada de cada celda representa el efecto que tiene en la transacción que solicita el tipo de bloqueo, dado anteriormente, cuando el objeto ha sido ya bloqueado en otra transacción con el tipo de bloqueo de la izquierda.

Las recuperaciones inconsistentes y las actualizaciones perdidas se producen por conflictos entre las operaciones de *lectura* en una transacción y de *escritura* en otra sin la protección de un esquema de control de concurrencia como el bloqueo. Las recuperaciones inconsistentes se previenen realizando la transacción de recuperación antes de la de actualización. Si llega primero la transacción de recuperación, su bloqueo de lectura retrasará la transacción de actualización. Si llega la segunda su solicitud de bloqueo de lectura produce su retraso hasta que se haya completado la transacción de actualización.

Para un objeto			*Bloqueo solicitado*
		Lectura	*Escritura*
Bloqueo ya activado	*Ninguno*	Bien	Bien
	Lectura	Bien	Espera
	Escritura	Espera	Espera

Figura 12.15. Compatibilidad de bloqueos.

1. Cuando una operación accede a un objeto en una transacción:
   (a) Si el objeto no estaba ya bloqueado, es bloqueado y comienza la operación.
   (b) Si el objeto tiene activado un bloqueo conflictivo con otra transacción, la transacción debe esperar hasta que esté desbloqueado.
   (c) Si el objeto tiene activado un bloqueo no conflictivo de otra transacción, se comparte el bloqueo y comienza la operación.
   (d) Si el objeto ya ha sido bloqueado en la misma transacción, el bloqueo será promovido si es necesario y comienza la operación. (Donde la promoción está impedida por un bloqueo conflictivo, se utiliza la regla (b).)
2. Cuando una transacción se consuma o aborta, el servidor desbloquea todos los objetos bloqueados por la transacción.

Figura 12.16. Uso de los bloqueos en un sistema de bloqueo en dos fases estricto.

Las actualizaciones perdidas aparecen cuando dos transacciones leen un valor en un objeto y entonces usan ese valor para calcular uno nuevo. Las actualizaciones perdidas se previenen haciendo que las transacciones posteriores retrasen sus lecturas hasta que las anteriores se han completado. Esto se consigue, para cada transacción, activando un bloqueo de lectura cuando se lee un objeto y *promoviéndolo* a un bloqueo de escritura cuando se escribe sobre el mismo objeto —cuando una transacción posterior precisa un bloqueo de lectura se retrasará hasta que la transacción actual se haya completado.

Una transacción con un bloqueo de lectura compartido con otras transacciones no puede promover su bloqueo de lectura a uno de escritura, porque las posteriores entrarían en conflicto con los bloqueos de lectura mantenidos por otras transacciones. Por tanto, tal transacción debe solicitar un bloqueo de escritura y esperar a que los otros bloqueos de lectura sean liberados.

La promoción de bloqueos se refiere a la conversión de un bloqueo a otro más fuerte, es decir, a un bloqueo que es más exclusivo. La tabla de compatibilidad de bloqueos muestra qué bloqueos son más o menos exclusivos. El bloqueo de lectura permite otros bloqueos de lectura, mientras que el de escritura no. Ninguno permite otros bloqueos de escritura. Por tanto, un bloqueo de escritura es más exclusivo que uno de lectura. Los bloqueos pueden promoverse hacia bloqueos más exclusivos. No es seguro degradar un bloqueo mantenido por una transacción antes de que ésta se consume, porque el resultado será más permisivo que uno previo y puede permitir ejecuciones de otras transacciones que son inconsistentes con la equivalencia secuencial.

Las reglas para el uso de los bloqueos en una implementación de bloqueo en dos fases estricto se resumen en la Figura 12.16. Para asegurar que se siguen estas reglas, el cliente no tiene acceso a las operaciones de bloqueo o desbloqueo de partes de datos. El bloqueo se realiza cuando las solicitudes para operaciones de *lectura* y *escritura* van a ser aplicadas a los objetos recuperables, y el desbloqueo se realiza por las operaciones *consuma* o *aborta* del coordinador de transacciones.

Por ejemplo, puede utilizarse el Servicio de Control de Concurrencia de Corba [OMG 1997a] para aplicar control de concurrencia en nombre de las transacciones o para proteger objetos sin utilizar transacciones. Proporciona un medio de asociar una colección de bloqueos (llamado un *lockset*) con un recurso como un objeto recuperable. Un lockset permite adquirir o liberar bloqueos. Un método *lock* (bloqueo) del lockset adquirirá un bloqueo o se inmovilizará hasta que el bloqueo esté libre; otro método permitirá promover o liberar los bloqueos. Los locksets transaccionales permiten los mismos métodos que los locksets, pero sus métodos precisan los identificadores de transacción como argumentos. Hemos mencionado anteriormente que el servicio de transacción de CORBA etiqueta todas las solicitudes del cliente en una transacción con el identificador de transacción. Esto permite la adquisición de un bloqueo adecuado antes de que se acceda a cada uno de


```
public class Bloqueo {
    private Object objeto;              // El objeto que es protegido por el bloqueo
    private Vector propietarios;        // las TIDs de los propietarios
    private TipoBloqueo tipoBloqueo;    // el tipo actual

    public synchronized void adquiere(IDTrans trans, TipoBloqueo unTipoBloqueo ){
        while (/* otra transacción posea en bloqueo en modo conflctivo */) {
            try {
                wait();
            }catch (InterruptedException e) {/*...*/}

            }
            if (propietarios.estaVacio()) {// ningún TID posee un bloqueo
                propietarios.agregaElemento(trans);
                tipoBloqueo = unTipoBloqueo;
            }else if (/* otra transacción posee el bloqueo, lo comparte */) ){
                if (/* esta transacción no es un poseedor*/) propietarios.agregaElemento(trans);
            }else if (/* esta transacción es un poseedor pero necesita más de un bloqueo exclusivo */)
                    tipoBloqueo.promueve();
            }
        }

        public synchronized void libera(IDTrans trans){
            holders.removeElement(trans);    // elimina este poseedor
            // establece el tipo de bloqueo a ninguno
            notifyAll();
        }
}
```

Figura 12.17. La clase *Bloqueo.*

los objetos recuperables durante una transacción. El coordinador de transacciones es el responsable de liberar los bloqueos cuando una transacción se consuma o se aborta.

Las reglas dadas en la Figura 12.16 aseguran la rigurosidad, porque los bloqueos se mantienen hasta que se ha consumado o abortado una transacción. Sin embargo, no es necesario mantener un conjunto de bloqueos para asegurar su rigor. Los bloqueos de lectura no necesitan ser mantenidos hasta que llega la petición de consumar o abortar.

◇ **Implementación de bloqueos.** La concesión de los bloqueos será implementada por un objeto separado del servidor que llamamos *gestor de bloqueos*. El gestor de bloqueos mantiene un conjunto de bloqueos, por ejemplo en una tabla de dispersión (*hash*). Cada bloqueo es una instancia de la clase *Bloqueo* y está asociada con un objeto particular. La clase *Bloqueo* se representa en la Figura 12.17. Cada instancia de *Bloqueo* mantiene la siguiente información en sus variables instancia:

- El identificador del objeto bloqueado.
- Los identificadores de las transacciones que mantienen actualmente el bloqueo (los bloqueos compartidos pueden tener varios poseedores).
- Un tipo de bloqueo.

Los métodos de *Bloqueo* están sincronizados de modo que los hilos que intentan adquirir o liberar un bloqueo no interferirán con el otro. Pero además, los intentos de adquirir el bloqueo utilizan el método *wait* cuando tengan que esperar que otro hilo lo libere.

El método *adquiere* cumple las reglas dadas en las Figuras 12.15 y 12.16. Sus argumentos especifican un identificador de transacción y el tipo de bloqueo precisado por ella. Se comprueba si

```
public class GestorBloqueo {
    private Hashtable losBloqueos;

    public void ponBloqueo(Object objeto, IDTrans trans, TipoBloqueo tipoBloqueo){
        Bloqueo bloqueoEncontrado;
        synchronized (this){
                // busca el bloqueo asociado con el objeto
                // si no hay ninguno, lo crea y lo agrega a la tabla de dispersión
            }
        bloqueoEncontrado.agrega(trans, tipoBloqueo);
    }

    // sincroniza este dado que queremos eliminar todas las entradas
    public synchronized void desBloqueo(TransID trans){
        Enumeration e = losBloqueos.elements();
        while (e.hasMoreElements()){
            Bloqueo unBloqueo= (Bloqueo)(e.nextElement());
            if (/* trans is a holder of this lock*/) unBloqueo.libera(trans);
        }
    }
}
```

Figura 12.18. La clase *GestorBloqueo.*

se puede conceder la solicitud. Si otra transacción mantiene el bloqueo en un modo conflictivo, ésta invocará *wait*, que provoca que el hilo de la que lo llama sea suspendido hasta un *notify* correspondiente. Hay que tener en cuenta que el *wait* está encerrado en un *while*, porque se notifica a todos los que esperan y alguno de ellos puede no ser capaz de proseguir. Cuando, eventualmente se satisfaga la condición, el resto del método activa el bloqueo adecuadamente:

- Si ninguna otra transacción mantiene el bloqueo, agrega la transacción dada a los poseedores y establece el tipo.
- Sino, si otra transacción mantiene el bloqueo, lo comparte añadiéndola transacción dada a los poseedores (a menos que él ya sea un poseedor).
- Sino, si esta transacción es un poseedor pero está solicitando un bloqueo más exclusivo, se promueve el bloqueo.

Los argumentos del método *libera* especifican el identificador de la transacción que está liberando el bloqueo. Elimina el identificador de transacción de los poseedores, establece el tipo de bloqueo a *ninguno* y llama a *notifyAll*. El método notifica a todos los hilos esperando en el caso que haya múltiples transacciones esperando para adquirir bloqueos de lectura, en cuyo caso todos ellos serían capaces de proseguir.

En la Figura 12.18 se muestra la clase *GestorBloqueo*. Todas las solicitudes de activar bloqueos y liberarlos en beneficio de las transacciones se envían a una instancia de *GestorBloqueo*.

- Los argumentos del método *ponBloqueo* indican al objeto que una transacción dada quiere un bloqueo y el tipo de éste. Obtiene un bloqueo para ese objeto en su tabla de dispersión, o si es preciso crea una. Después invoca el método *adquiere* de ese bloqueo.
- Los argumentos del método *desBloqueo* especifican la transacción que está liberando sus bloqueos. Obtienen todos los bloqueos en la tabla de dispersión que tiene como poseedor a la transacción dada. Por cada uno, llama al método *libera*.

Algunas cuestiones de política: Observe que, cuando varios hilos *esperan* en el mismo objeto bloqueado, la semántica de *espera* garantiza que cada transacción consigue su turno. En el programa anterior, las reglas de conflicto permiten que los que mantienen un bloqueo sean varios lectores

o un único escritor. La llegada de una petición de un bloqueo de lectura está siempre garantizada a menos que el poseedor tenga un bloqueo de escritura. Se invita al lector a considerar o siguiente:

- ¿Cuál es la consecuencia para las transacciones de *escritura* de la presencia de un goteo estacionario de solicitudes para bloqueos de lectura? Piense en una alternativa de implementación.

Cuando el poseedor tiene un bloqueo de escritura, puede haber varios lectores y escritores esperando. El lector debiera considerar el efecto de *notifyAll* y pensar en una alternativa de implementación. Si un poseedor de un bloqueo de lectura intenta promover el bloqueo cuando está compartido, se inmovilizará. ¿Hay alguna solución a esta dificultad?

◇ **Reglas de bloqueo para transacciones anidadas.** El propósito de un esquema de bloqueo es serializar el acceso a objetos de modo que:

1. Cada conjunto de transacciones anidadas sea una única entidad a la que se debe impedir observar los efectos de cualquier otro conjunto de transacciones anidadas.
2. Se debe impedir que cada transacción con un conjunto de transacciones anidadas observe los efectos parciales de otras transacciones del conjunto.

La primera regla se hace cumplir disponiendo que cada bloqueo que se adquiere después de una subtransacción con éxito sea heredado por su padre cuando finaliza. los bloqueos heredados son heredados también por los ascendientes. ¡Téngase en cuenta que esta forma de herencia pasa del hijo al padre! La transacción del nivel superior hereda eventualmente todos los bloqueos que fueron adquiridos por las subtransacciones con éxito en cualquier profundidad de una transacción anidada. Esto garantiza que puedan mantenerse los bloqueos hasta que se haya consumado o abortado la transacción de nivel superior, lo que impide que los miembros de diferentes conjuntos de transacciones anidadas observen los resultados parciales de los otros.

La segunda regla se hace cumplir como sigue:

- No se permite la ejecución concurrente de las transacciones padre con las de sus hijos. Si una transacción padre tiene un bloqueo sobre un objeto, *retiene* el bloqueo durante el tiempo que su transacción hijo se está ejecutando. Esto significa que la transacción hijo adquiere temporalmente el bloqueo del padre durante su duración.
- Se permite la ejecución concurrente de las subtransacciones del mismo nivel, por lo que cuando éstas acceden a los mismos objetos, el esquema de bloqueo deberá secuenciar su acceso.

Las siguientes reglas describen la adquisición y liberación del bloqueo:

- Para que una subtransacción adquiera un bloqueo de lectura sobre un objeto, ninguna transacción activa puede tener un bloqueo de escritura sobre ese objeto, y los únicos que retienen un bloqueo de escritura son sus ascendientes.
- Para que una subtransacción adquiera un bloqueo de escritura sobre un objeto, ninguna otra transacción activa puede tener un bloqueo de lectura o escritura sobre ese objeto, y los únicos que retienen los bloqueos de lectura y escritura en ese objeto son sus ascendientes.
- Cuando se consuma una transacción, sus bloqueos son heredados por su padre, permitiendo al padre retener los bloqueos del mismo modo que el hijo.
- Cuando una subtransacción aborta, sus bloqueos son eliminados. Si el padre todavía continúa manteniendo los bloqueos puede continuar haciéndolo.

Tenga en cuenta que las subtransacciones al mismo nivel al que accede el mismo objeto realizarán turnos para adquirir los bloqueos retenidos por su padre. Esto asegura que su acceso a un objeto común es secuencializado.

Transacción *T*		Transacción *U*	
Operaciones	Bloqueos	Operaciones	Bloqueos
a.deposita(100);	Escribe bloquea *A*		
		b.deposita(200);	Escribe bloquea *B*
b.extrae(100);			
• • •	Espera por *U*	*a.extrae(200);*	Espera por *T*
	Bloqueo en *B*	• • •	Bloqueo en *A*
• • •		• • •	
• • •		• • •	

Figura 12.19. Bloqueo indefinido con bloqueos de escritura.

Como ejemplo, supongamos que las subtransacciones T_1, T_2 y T_{11} en la Figura 12.13 acceden todas a un objeto común, que no es accedido por la transacción del nivel superior *T*. Supongamos que la subtransacción T_1 es la primera que accede al objeto y adquiere un bloqueo con éxito, que le pasa a T_{11} para la duración de su ejecución, devolviéndolo cuando T_{11} finalice . Cuando T_1 finaliza su ejecución, la transacción del nivel superior *T* hereda el bloqueo, que mantiene hasta que finaliza el conjunto de transacciones anidadas. La subtransacción T_2 puede adquirir el bloqueo de *T* para la duración de su ejecución.

12.4.1. BLOQUEOS INDEFINIDOS

El uso de bloqueos puede conducir a bloqueos indefinidos. Consideremos el uso de bloqueos representado en la Figura 12.19. Puesto que los métodos *deposita* y *extrae* son atómicos, los representamos adquiriendo bloqueos de escritura, aunque en la práctica ellos leen el balance y escriben en él. Cada uno de ellos adquiere un bloqueo y se inmoviliza cuando intenta acceder a la cuenta que ha bloqueado el otro. Ésta es una situación de bloqueo indefinido: dos transacciones están esperando y cada una depende de la otra para liberar un bloqueo y poder reanudarse.

El bloqueo indefinido es una situación particularmente común cuando los clientes están implicados en un programa interactivo, una transacción en un programa interactivo puede durar un período largo de tiempo, produciendo que muchos objetos queden inmovilizados y permanezcan así, impidiendo por tanto que otros clientes los utilicen.

Tenga en cuenta que el bloqueo de sub-elementos en objetos estructurados puede ser útil para impedir conflictos y posibles situaciones de bloqueo indefinido. Por ejemplo, un día en un diario puede estar estructurado como un conjunto de períodos de tiempo, cada uno de los cuales puede

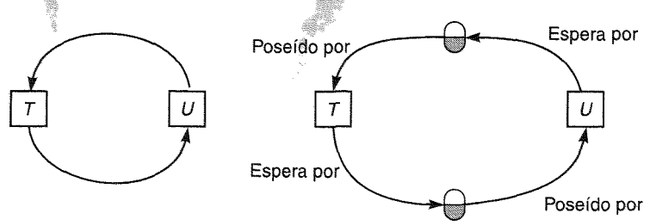


Figura 12.20. El grafo *espera por* para la Figura 12.19.

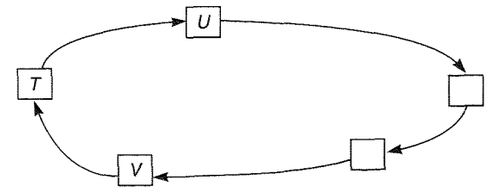


Figura 12.21. Un ciclo en el grafo *espera por*.

ser bloqueado independientemente para actualización. Los esquemas de bloqueo jerárquico son útiles si la aplicación requiere una granularidad independiente del bloqueo para diferentes operaciones. Véase la Sección 12.4.2.

◇ **Definición de bloqueo indefinido.** Un bloqueo indefinido es un estado en el que cada miembro de un grupo de transacciones está esperando por algún otro miembro para liberar un bloqueo. Se puede utilizar un *grafo espera por* para representar las relaciones de espera entre las transacciones actuales. En un grafo *espera por* los nodos representan las transacciones y los arcos representan las relaciones *espera por* entre transacciones, hay un arco del nodo *T* al nodo *U* cuando la transacción *T* está esperando que la transacción *U* libere un bloqueo. Inspeccione la Figura 12.20, que muestra el grafo *espera por* correspondiente a la situación de bloqueo indefinido ilustrada en la Figura 12.19. Recuerde que un bloqueo indefinido surge porque las transacciones *T* y *U* intentaron cada una adquirir un objeto mantenido por la otra. Por tanto *T* espera por *U* y *U* espera por *T*. La dependencia entre las transacciones es indirecta, mediante una dependencia de los objetos. El diagrama de la derecha muestra los objetos mantenidos y esperados por las transacciones *T* y *U*. Como cada transacción puede esperar sólo por un objeto, los objetos pueden ser omitidos del grafo a*espera por*, conduciendo al grafo simple de la izquierda.

Supongamos que como en la Figura 12.21, un grafo *espera por* contiene un ciclo $T \rightarrow U \rightarrow \cdots \rightarrow V \rightarrow T$, luego cada transacción está esperando por la siguiente transacción en el ciclo. Todas esas transacciones están inmovilizadas esperando a causa de los bloqueos. Ninguno de los bloqueos podrá ser liberado nunca, y las transacciones permanecerán esperando indefinidamente. Si se aborta una de las transacciones en el ciclo, se liberan sus bloqueos y el ciclo se rompe. Por ejemplo, si una transacción *T* en la Figura 12.21 se aborta, liberará un bloqueo sobre un objeto por el que está esperando *V*, y *V* no tendrá que esperar mas por *T*.

Consideremos ahora un escenario en el que tres transacciones *T*, *U* y *V* comparten un bloqueo de lectura sobre un objeto *C*, la transacción mantiene un bloqueo de escritura sobre el objeto *B* por el que está esperando la transacción *V* para obtener un bloqueo, como se muestra en la Figura 12.22. Las transacciones *T* y *W* solicitarán entonces bloqueos de escritura sobre el objeto *C* y se

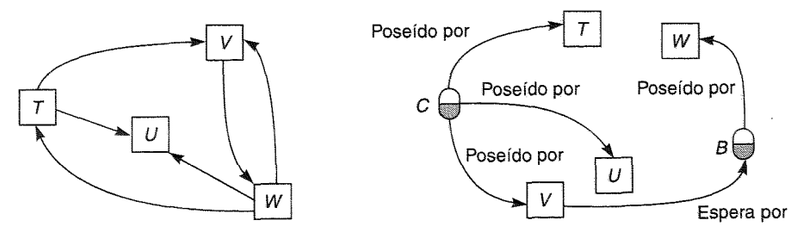


Figura 12.22. Otro grafo *espera por*.

presenta una situación de bloqueo indefinido en la que T espera por U y V, V espera por W y W espera por T, U y V, como se muestra en la izquierda de la Figura 12.22. Esto muestra que aunque cada transacción puede esperar por sólo un objeto cada vez, puede estar implicado en varios ciclos. Por ejemplo la transacción V está implicada en los ciclos: $V \rightarrow W \rightarrow T \rightarrow V$ y $V \rightarrow W \rightarrow V$.

En este ejemplo, suponemos que se aborta la transacción V. Esto liberará el bloqueo de V en C y se romperán los dos ciclos en los que está implicado V.

◇ **Prevención de bloqueos indefinidos.** Una solución es prevenir el bloqueo indefinido. Una forma aparentemente simple pero no muy buena para vencer el bloqueo indefinido es bloquear todos los objetos utilizados por una transacción cuando comienza. Esto necesitaría realizarse en un único paso atómico para impedir el bloqueo indefinido en esta etapa. Dicha transacción no puede ejecutarse en bloqueo indefinido con otras transacciones, pero restringe innecesariamente el acceso a los recursos compartidos. En suma, a veces es imposible predecir al comienzo de una transacción qué objetos serán utilizados. Éste es generalmente el caso de aplicaciones interactivas, el usuario tendría que decir con anterioridad exactamente qué objetos estaba planeando utilizar, lo que es inconcebible en aplicaciones del estilo de hojeo (*browsing*), que ofrecen a los usuarios objetos de los que no conocen nada de antemano. El bloqueo indefinido puede también impedirse solicitando los bloqueos en los objetos en un orden predefinido, pero esto puede producir un bloqueo prematuro o una reducción de la concurrencia.

◇ **Detección de bloqueos indefinidos.** Los bloqueos indefinidos pueden detectarse a través de los ciclos en el grafo *espera por*. Una vez detectado un bloqueo indefinido, para romper el ciclo se debe seleccionar un transacción para abortar.

El software responsable de la detección de bloqueos indefinidos puede ser parte del gestor de bloqueos. Debe mantener una representación del grafo *espera por* para que pueda comprobar si existen ciclos en él de tiempo en tiempo. Los arcos son añadidos y eliminados del grafo por las operaciones *ponBloqueo* y *desBloqueo* del gestor de bloqueos. En el punto ilustrado por la Figura 12.22 en la izquierda tendremos la siguiente información:

Transacción	*Esperas por la transacción*
T	U, V
V	W
W	T, U, V

Un arco $T \rightarrow U$ se añade cuando el gestor de bloqueos bloquea una solicitud desde la transacción T causada por un bloqueo en un objeto que ya está bloqueado en el marco de la transacción U. Considere, que cuando un bloqueo es compartido, se pueden añadir varios arcos. Un arco $T \rightarrow U$ es borrado cuando U libera un bloqueo por el que está esperando T y permite a T continuar. Para una discusión más detallada de la implementación de detección de bloqueos indefinidos véase el Ejercicio 12.14. Si una transacción comparte un bloqueo, el bloqueo no es liberado, pero los arcos que conducen a una transacción particular se eliminan.

La presencia de ciclos se puede comprobar cada vez que se añade un arco, o con menos frecuencia para evitar una sobrecarga innecesaria. Cuando se detecta un bloqueo indefinido, se debe elegir una de las transacciones en el ciclo y abortarla. El nodo correspondiente y los arcos involucrados se eliminan del grafo *espera por*. Esto sucederá cuando la transacción abortada tenga su bloqueos eliminados.

La elección de la transacción a abortar no es sencilla. Algunos factores que pueden ser considerados son la edad de la transacción y el número de ciclos implicados en ella.

Transacción *T*		**Transacción *U***	
Operaciones	Bloqueos	Operaciones	Bloqueos
a.deposita(100);	Escritura bloquea *A*		
		b.deposita(200);	Escritura bloquea *B*
b.extrae(100);			
• • •	Espera por *U*	*a.extrae(200);*	Espera por *T*
	Bloqueo en *B*	• • •	Bloqueo en *A*
	(Transcurre el intervalo)	• • •	
	El bloqueo de *T* en *A* llega a ser vulnerable, desbloqueo de *A*, *T* se aborta		
		a.extrae(200);	Escritura bloquea en *A*
			Desbloquea *A*, *B*

Figura 12.23. Resolución del bloqueo indefinido en la Figura 12.19.

◇ **Timeouts.** Los timeouts de bloqueo son un método para la resolución de bloqueos indefinidos que se utiliza normalmente. A cada bloqueo se le proporciona un período limitado en el que es invulnerable. Después de este tiempo es vulnerable. Supuesto que ninguna otra transacción está compitiendo por el objeto que está bloqueado, un objeto con un bloqueo vulnerable continúa bloqueado. Sin embargo, si cualquier otra transacción está esperando para acceder al objeto protegido por un bloqueo vulnerable, el bloqueo se rompe (es decir, el objeto es desbloqueado) y se reanuda la transacción que estaba esperando. La transacción cuyo bloqueo se ha roto normalmente aborta.

Hay muchos problemas con el uso de timeouts como un remedio para bloqueos indefinidos: el peor problema es que a veces se abortan transacciones debido a que sus bloqueos llegan a ser vulnerables cuando otras transacciones están esperando por ellas, aunque en realidad no hay bloqueo indefinido. En un sistema sobrecargado, el número de transacciones que superan la temporización aumentará y las transacciones que precisan un tiempo largo serán penalizadas. En resumen, es duro decidir una magnitud apropiada para un timeout. En contraste, si se utiliza la detección de bloqueo indefinido, las transacciones son abortadas porque han ocurrido bloqueos indefinidos y se ha hecho una elección para determinar qué transacción abortar.

Usando timeouts de bloqueos, podemos resolver el bloqueo indefinido en la Figura 12.19, como se muestra en la Figura 12.23, en la que el bloqueo de escritura para T en A llega a ser vulnerable después de su período de timeout. La transacción U está esperando para adquirir un bloqueo de escritura en A. Por tanto, T se aborta y libera su bloqueo en A, permitiendo a U reanudar y completar la transacción.

Cuando las transacciones acceden a objetos ubicados en varios servidores diferentes, surge la posibilidad de bloqueos indefinidos distribuidos. En un bloqueo indefinido distribuido, el grafo *espera por* puede implicar a objetos en múltiples ubicaciones. Volveremos a este tema en la Sección 13.5.

12.4.2. INCREMENTANDO LA CONCURRENCIA EN ESQUEMAS DE BLOQUEO

Incluso cuando las reglas de bloqueo están basadas en conflictos entre las operaciones de *lectura* y *escritura* y la granularidad con que se aplican será tan pequeña como sea posible, con lo que habrá posibilidades de incrementar la concurrencia. Discutiremos dos aproximaciones que se han utilizado. En la primera aproximación (bloqueo en dos versiones), la activación de bloqueos exclusivos

Para un objeto		*Bloqueo que se establece*		
		Lectura	*Escritura*	*Consumación*
Bloqueo ya establecido	*Ninguno*	Bien	Bien	Bien
	Lectura	Bien	Bien	Espera
	Escritura	Bien	Espera	—
	Consumación	Espera	Espera	—

Figura 12.24. Compatibilidad de bloqueos (letra, escritura y consumación).

se retrasa hasta que un transacción se consuma. En la segunda aproximación (bloqueos jerárquicos), se utilizan bloqueos de granularidad mixta.

◇ **Bloqueo de dos versiones.** Éste es un esquema optimista que permite que una transacción escriba versiones tentativas de objetos mientras otras transacciones leen de la versión consumada de los mismos objetos. Las operaciones de *lectura* sólo esperan si otra transacción se está consumando actualmente en el mismo objeto. Este esquema permite más concurrencia que la de los bloqueos de lectura-escritura, pero existe el riesgo que las transacciones de escritura esperen o incluso sean rechazadas cuando intentan consumarse. Las transacciones no pueden consumar sus operaciones de escritura inmediatamente si otras transacciones no completadas han leído sobre los mismos objetos. Por tanto, a las transacciones que solicitan consumarse en dicha situación se les hace esperar hasta que se han completado las transacciones de lectura. El bloqueo indefinido puede ocurrir cuando las transacciones están esperando para consumarse. Por tanto, las transacciones pueden necesitar ser abortadas cuando están esperando para consumarse, para resolver los bloqueos indefinidos.

Esta variación del bloqueo en dos fases estricto utiliza tres tipos de bloqueo: un bloqueo de lectura, uno de escritura y uno de consumación. Antes que se realice una operación de lectura, se debe activar un bloqueo de lectura en el objeto; el intento de activar un bloqueo de lectura tiene éxito a menos que el objeto haya consumado el bloqueo, en cuyo caso la transacción espera. Antes que se realice una operación de escritura de una transacción, se debe activar un bloqueo en el objeto, el intento de activar un bloqueo de escritura tendrá éxito a menos que el objeto tenga un bloqueo de escritura o de consumación, en cuyo caso la transacción espera.

Cuando el coordinador de transacciones recibe una solicitud para consumar una transacción, intenta convertir todos los bloqueos de escritura de esa transacción a bloqueos de consumación. Si alguno de los objetos tiene bloqueos de lectura pendientes, la transacción debe esperar hasta que las transacciones que han activado esos bloqueos hayan finalizado y los bloqueos se hayan liberado. La compatibilidad de bloqueos de lectura, escritura y consumación se muestra en la Figura 12.24.

Hay dos diferencias principales en las prestaciones entre el esquema de bloqueo en dos versiones y un esquema de bloqueo de lectura-escritura ordinario. Por un lado, las operaciones de *lectura* en el esquema de bloqueo en dos versiones son retrasadas sólo cuando las transacciones están siendo consumadas más bien que durante la ejecución completa de las transacciones: en la mayoría de los casos, el protocolo de consumación precisa sólo una pequeña fracción del tiempo requerido para realizar una transacción completa. Por otro lado, las operaciones de *lectura* de una transacción pueden producir un retraso en la consumación de otras transacciones.

◇ **Bloqueos jerárquicos.** En algunas aplicaciones, la granularidad adecuada para una operación no es apropiada para otra. En nuestro ejemplo bancario, la mayoría de las operaciones requieren bloqueo con granularidad de una cuenta. La operación *totalSucursal* es diferente, lee los valores de

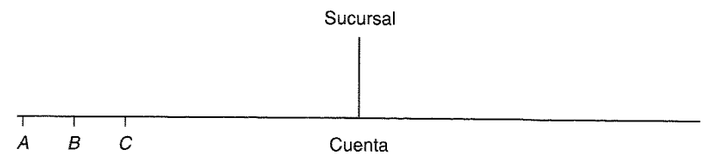


Figura 12.25. Jerarquía de bloqueos para el ejemplo bancario.

todos los balances de las cuentas y parecería requerir un bloqueo de lectura de todos ellos. Para reducir la sobrecarga de bloqueo, sería útil permitir que coexistan bloqueos de granularidad mezclada.

Gray [1978] propuso la utilización de una jerarquía de bloqueos con diferentes granularidades. En cada nivel, la activación de un bloqueo padre tiene el mismo efecto que la activación de todos los bloqueos hijo equivalentes. Esto economiza el número de los bloqueos que se deben activar. En nuestro ejemplo bancario, la sucursal es el padre y las cuentas son los hijos (ver Figura 12.25).

Se podrían utilizar bloqueos de granularidad mezclada en un sistema de dietario, en el que los datos podrían estar estructurados de modo que el dietario para una semana viene formado con una página para cada día y éstos subdivididos además en un espacio para cada hora del día, como se muestra en la Figura 12.26. La operación para ver una semana produciría la activación de un bloqueo de lectura en la parte superior de esta jerarquía, mientras que la operación de introducir un apunte produciría la activación de un bloqueo de escritura en un espacio de tiempo. El efecto de un bloqueo de lectura en una semana sería el prevenir operaciones de escritura en cualquiera de sus subestructuras, por ejemplo en los espacios de tiempo de cada día de la semana.

En el esquema de Gray, cada nodo en la jerarquía puede ser bloqueado, dando al propietario del bloqueo acceso explícito al nodo y acceso implícito a sus hijos. En nuestro ejemplo, en la Figura 12.25 un bloqueo de lectura-escritura en la sucursal bloquea implícitamente todas las cuentas para lectura-escritura. Antes de que se conceda un bloqueo de lectura-escritura a un nodo hijo, se activa una intención de bloqueo de lectura-escritura en el nodo padre y en sus antepasados (si los hubiere). La intención de bloqueo es compatible con otras intenciones de bloqueo pero entra en conflicto con los bloqueos de lectura y escritura, según las reglas habituales. La Figura 12.27 proporciona la tabla de compatibilidad para bloqueos jerárquicos. Gray propone también un tercer tipo de intención de bloqueo, uno que combina la propiedad de un bloqueo de lectura con una intención de bloqueo de escritura.

En nuestro sistema bancario, la operación *totalSucursal* solicita un bloqueo de lectura en la sucursal, que implícitamente activa bloqueos de lectura en todas las cuentas. Una operación *deposita* necesita activar un bloqueo de escritura en un balance, pero intenta primero activar una intención para un bloqueo de escritura en la sucursal. Estas reglas impiden que estas operaciones se ejecuten concurrentemente.

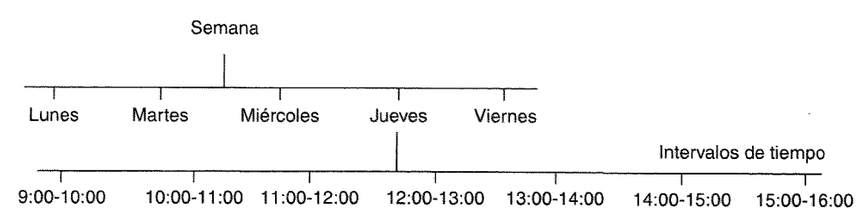


Figura 12.26. Jerarquía de bloqueos para un diario.

Para un bloqueo		*Bloqueo que se va a activar*			
		Lectura	*Escritura*	*I-Lectura*	*I-Escritura*
Bloqueo ya activado	*Ninguno*	Bien	Bien	Bien	Bien
	Lectura	Bien	Espera	Bien	Espera
	Escritura	Espera	Espera	Espera	Espera
	I-Lectura	Bien	Espera	Bien	Bien
	I-Escritura	Espera	Espera	Bien	Bien

Figura 12.27. Tabla de compatibilidad para bloqueos jerárquicos.

Los bloqueos jerárquicos tienen la ventaja de reducir el número de bloqueos cuando se precisa un bloqueo de granularidad mezclada. Las tablas de compatibilidad y las reglas para promoción de bloqueos son más complejas.

La granularidad mezclada de bloqueos podría permitir a cada transacción bloquear una porción cuyo tamaño se elige de acuerdo con las necesidades. Una transacción larga que accede a muchos objetos podría bloquear la colección total, mientras que una transacción corta puede bloquear a una granularidad más fina.

12.5. CONTROL OPTIMISTA DE LA CONCURRENCIA

Kung y Robinson [1981] identificaron un número de desventajas inherentes al bloqueo y propusieron una aproximación optimista alternativa a la secuenciación de transacciones que evita esas desventajas. Podemos resumir las desventajas del bloqueo:

- El mantenimiento del bloqueo representa una sobrecarga que no está presente en los sistemas que no soportan acceso concurrente a los datos compartidos. Incluso transacciones de *sólo lectura* (consultas), que posiblemente no afecten a la integridad de los datos, deben, en general, utilizar bloqueo para garantizar que los datos que se están leyendo no son modificados por otras transacciones al mismo tiempo.
  Por ejemplo, consideremos dos procesos clientes que están incrementando concurrentemente los valores de n objetos. Si los programas del cliente comienzan al mismo tiempo y se ejecutan aproximadamente durante la misma cantidad de tiempo, accediendo a los objetos en dos secuencias no relacionadas y utilizando una transacción separada para acceder e incrementar cada elemento, las oportunidades para que los dos programas intenten acceder al mismo objeto en el mismo tiempo son justo 1 entre n de promedio, por lo que el bloqueo sólo es necesario realmente una vez cada n transacciones.
- El uso de bloqueos puede producir un bloqueo indefinido. La prevención de bloqueos indefinidos reduce la concurrencia de forma severa, y por tanto las situaciones de bloqueo indefinido deben ser resueltas o por el uso de timeouts o por la detección de bloqueo indefinido. Ninguna de ellas es totalmente satisfactoria para uso en programas interactivos.
- Para impedir abortos en cascada, los bloqueos no pueden ser liberados hasta el final de la transacción. Esto puede reducir significativamente el potencial de concurrencia.

La aproximación alternativa propuesta por Kung y Robinson es *optimista* porque se basa en la observación de que, en la mayoría de las aplicaciones, la similitud entre las transacciones de dos clientes que acceden al mismo objeto es baja. Se permite que las transacciones procedan como si no hubiera posibilidad de conflicto con otras transacciones hasta que el cliente complete su tarea y publique una petición *cierraTransacción*. Cuando aparece un conflicto, habitualmente se abortará

alguna transacción y se necesitará reinicializar el cliente. Cada transacción presenta las siguientes fases:

Fase de trabajo: durante la fase de trabajo, cada transacción tiene una versión tentativa de cada uno de los objetos que actualiza. Ésta es una copia de la versión del objeto más recientemente consumada. El empleo de las versiones tentativas permite a la transacción abortar (sin efecto alguno sobre los objetos), tanto durante la fase de trabajo como si falla su validación debido a otras transacciones en conflicto. las operaciones de *lectura* se realizan inmediatamente: si existe una versión tentativa para la transacción, la operación de *lectura* accederá a ella, de otro modo accederá al valor más recientemente consumado del objeto. La operación de *escritura* almacena los nuevos valores de los objetos bajo versiones tentativas (las cuales son invisibles al resto de transacciones). Cuando hay varias transacciones concurrentes, podrán coexistir valores tentativos diferentes sobre el mismo objeto. Además, se almacenan dos registros para cada objeto al que se accede en la transacción: un *conjunto de lectura* que contiene los objetos leídos por la transacción; y un *conjunto de escritura* que contiene los objetos modificados por la transacción. Observe que dado que todas las operaciones de lectura se realizan sobre versiones consumadas de los objetos (o copias de ellos), no habrá lecturas sucias.

Fase de validación: cuando se recibe la solicitud *cierraTransacción*, se valida la transacción para establecer si sus operaciones en los objetos entran en conflicto o no con las operaciones en otras transacciones sobre los mismos objetos. Si la validación tiene éxito, entonces se puede consumar la transacción. Si la validación falla, se debe utilizar alguna forma de resolución de conflictos y o bien habrá que abortar la transacción actual o, en algunos casos, aquellas con las que entra en conflicto.

Fase de actualización: si una transacción está validada, todos los cambios registrados en su versiones provisionales se hacen permanentes. Las transacciones de *sólo lectura* pueden consumarse inmediatamente después de pasar la validación. Las transacciones de escritura están dispuestas a consumarse una vez que las versiones provisionales de los objetos se hayan grabado en memoria permanente.

◇ **Validación de transacciones.** La validación utiliza las reglas de conflictos de lectura-escritura para garantizar que la planificación de una transacción particular es secuencialmente equivalente con respecto a todas las demás que se *solapan* con ella, es decir, cualquier otra transacción que no se haya consumado en el instante de comienzo de la transacción considerada. Para asistir en la realización de la validación, se le asigna un número a cada transacción cuando comienza su fase de validación (esto es, cuando un cliente emite un *cierraTransacción*). Si se valida la transacción y finaliza con éxito, retiene ese número; si fallan las comprobaciones de validación y se aborta, o si las transacciones son sólo de lectura, se libera el número para su reasignación. Los números de transacciones son enteros asignados en orden de secuencia ascendente; el número de una transacción define por tanto su posición en el tiempo: una transacción siempre termina su fase de trabajo después de todas las transacciones con números más pequeños. Es decir, una transacción con el número T_i siempre precede a una transacción con número T_v si $i < j$. (Si el número de transacción se asignó al comienzo de la fase de trabajo, entonces una transacción que alcanzó el final de la fase de trabajo antes que otra con un número más pequeño tendría que esperar hasta que la anterior haya completado antes que ella pueda ser validada.)

La comprobación de validación en la transacción T_v está basada en conflictos entre pares de operaciones de la transacción T_v y T_i. Para que una transacción T_v sea secuencializable con respecto a una transacción T_i solapada con ella, sus operaciones deben ajustarse a las siguientes reglas:

T_v	T_i	*Regla*
escribe	*lee*	1. T_i no debe leer los objetos escritos por T_v.
lee	*escribe*	2. T_v no debe leer los objetos escritos por T_i.
escribe	*escribe*	3. T_i no debe escribir en los objetos escritos por T_v y T_v no debe escribir en los objetos escritos por T_i.

Como las fases de validación y actualización de una transacción son generalmente cortas en duración, comparadas con la fase de trabajo, se puede conseguir una simplificación considerando la regla de que sólo puede haber una transacción en la fase de validación y actualización cada vez. Cuando no se pueden solapar dos transacciones en la fase de actualización, se satisface la regla 3. Hay que considerar que esta restricción en las operaciones de *escritura*, junto con el hecho que no pueden ocurrir lecturas sucias, produce ejecuciones estrictas. Para impedir el solapamiento, las fases completas de validación y actualización se pueden implementar como una sección crítica por lo que sólo puede ejecutarla un cliente cada vez. Con el fin de incrementar la concurrencia, la parte de validación y actualización puede implementarse fuera de la sección crítica, pero es esencial que la asignación de números a las transacciones se realice secuencialmente. Consideramos que en cualquier instante, los números actuales de transacción son como un pseudo-reloj que marca cuándo una transacción se completa con éxito.

La validación de una transacción debe asegurar que se obedecen las reglas 1 y 2 comprobando que el solapamiento entre los objetos de pares de transacciones T_v y T_i. Existen dos formas de validación, hacia delante y hacia atrás [Härder 1984]. La validación hacia atrás realiza la validación de la transacción considerada con otras transacciones precedentes que se solapan con ella, aquellas que entraron en la fase de validación antes que ella. La validación hacia delante realiza la validación de la transacción considerada con otras transacciones posteriores, que están todavía activas.

◇ **Validación hacia atrás.** Ya que todas las operaciones de *lectura* de las transacciones anteriores que se solapan fueron realizadas antes que comenzara la validación de T_v, no pueden ser afectadas por las *escrituras* de la transacción en curso (y se satisface la regla 1). La validación de la transacción T_v comprueba si su conjunto de lectura (los objetos afectados por las operaciones de *lectura* de T_v) se solapan con cualquiera de los conjuntos de escritura de las transacciones anteriores T_i que se solapan con ella (regla 2). Si no hay ningún solapamiento, la validación falla.

Sea *nTinicial* el número de transacción más grande asignado (para alguna otra transacción consumada) en el instante en que la transacción T_v comenzó su fase de trabajo y *nTfinal* el número de transacción más grande asignado en el instante en que T_v entró en su fase de validación. El programa siguiente describe el algoritmo para la validación de T_v:

```
bolean válida = cierto;
for (int T_i = nTinicial + 1; T_i <= nTfinal; T_i++){
    if (el conjunto de lectura de T_v intersecta el conjunto de escritura de T_i) válida = falso;
}
```

La Figura 12.28 muestra transacciones que se solapan que pueden ser tenidas en cuenta en la validación de una transacción T_v. El tiempo aumenta de izquierda a derecha. Las transacciones consumadas anteriormente son T_1, T_2 y T_3. T_1 se consumó antes de que empezara T_v. T_2 y T_3 se consumaron antes de que T_v finalizara su fase de trabajo. $nTinicial + 1 = T_2$ y $nTfinal + 1 = T_3$. En la validación hacia atrás el conjunto de lectura de T_v debe ser comparado con los conjuntos de escritura de T_2 y T_3.

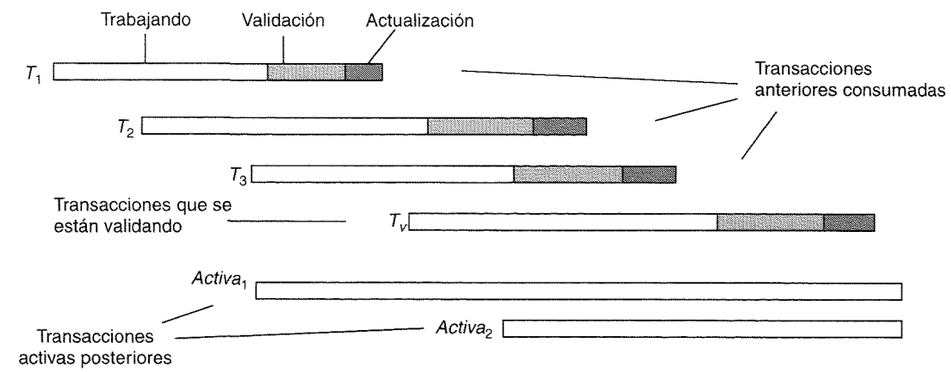


Figura 12.28. Validación de transacciones.

En la validación hacia atrás, el conjunto de lectura de la transacción que se está validando se compara con los conjuntos de escritura de otras transacciones que ya han sido consumadas. Por tanto, la única forma de resolver conflictos es abortar la transacción que está experimentando la validación.

En la validación hacia atrás, las transacciones que no tienen operaciones de *lectura* (sólo de *escritura*) no necesitan ser comprobadas.

El control optimista de concurrencia con validación hacia atrás precisa que los conjuntos de escritura de las versiones antiguas de objetos ya consumadas correspondientes a transacciones recientemente consumadas sean retenidas hasta que no haya transacciones solapadas invalidadas con las que pudieran entrar en conflicto. Cuando una transacción se valida con éxito, su número de transacción, *nTinicial* y el conjunto de escritura se registran en una lista de transacciones precedentes que es mantenida por el servicio de transacciones. Téngase en cuenta que esta lista está ordenada por número de transacción. En un entorno con transacciones largas, la retención de los conjuntos de escritura de objetos anteriores puede ser un problema. Por ejemplo, en la Figura 12.28 los conjuntos de escritura de T_1, T_2, T_3 y T_v deben ser retenidos hasta que la transacción activa $activa_1$ se completa. Hay que tener en cuenta que aunque las transacciones activas tienen identificadores de transacción, no tienen aún números de transacción.

◇ **Validación hacia delante.** En la validación hacia delante de la transacción T_v, el conjunto de escritura de T_v se compara con los conjuntos de lectura de todas las transacciones activas que se solapan, aquellas que están aún en su fase de trabajo (regla 1). La regla 2 se satisface automáticamente porque las transacciones activas no escriben hasta que no se ha completado T_v. Consideremos que las transacciones activas (consecutivas) tienen identificadores $activa_1$ a $activa_N$, el siguiente programa describe el algoritmo para la validación hacia delante de T_v.

```
bolean válida = cierto;
for (int T_id = activa_1; T_id <= activa_N; T_id++){
    if (el conjunto de escritura de T_v intersecta el conjunto de lectura de T_id) válida = falso;
```

En la Figura 12.28, el conjunto de escritura de la transacción T_v debe compararse con los conjuntos de lectura de las transacciones con identificadores $activa_1$ y $activa_2$. (La validación hacia delante debiera permitir el hecho que los conjuntos de lectura de las transacciones activas pueden cambiar durante la validación y la escritura.) Como los conjuntos de lectura de la transacción que se está validando no están incluidos en la comprobación, las transacciones de *sólo lectura* siempre pasan

la comprobación de validación. Como las transacciones que están siendo comparadas con la de validación están todavía activas, tomamos una elección de si abortar la transacción de validación o tomar alguna vía alternativa de resolver el conflicto. Arder [1984] sugiere varias estrategias alternativas:

- Aplazar la validación hasta un instante posterior cuando hayan finalizado las transacciones conflictivas. Sin embargo, no hay garantía que a la transacción que está siendo validada le vaya mejor en el futuro. Existe siempre la posibilidad de que otras transacciones conflictivas activas puedan comenzar antes que se obtenga la validación.
- Abortar todas las transacciones conflictivas activas y consumar las transacciones que se están validando.
- Abortar la transacción que se está validando. Ésta es la estrategia más sencilla pero tiene la desventaja que las futuras transacciones conflictivas pueden ser abortadas, en cuyo caso la transacción bajo validación ha abortado innecesariamente.

◇ **Comparación de la validación hacia delante y hacia atrás.** Hemos visto que la validación hacia delante permite flexibilidad en la resolución de conflictos, mientras que la validación hacia atrás sólo permite una elección: abortar la transacción que está siendo validada. En general, los conjuntos de lectura de las transacciones son mucho más grandes que los de escritura. Por tanto, la validación hacia atrás compara un conjunto posiblemente grande de lectura frente a los antiguos conjuntos de escritura, mientras que la validación hacia delante tiene la sobrecarga de almacenar los conjuntos antiguos de escritura hasta que no se necesiten más. Vemos que la validación hacia atrás tiene la sobrecarga de almacenar conjuntos de escritura antíguos hasta que ya no se necesiten. Por otra parte, la validación hacia delante tiene que considerar únicamente las nuevas transacciones que comiencen durante el proceso de validación.

◇ **Inanición (*Starvation*).** Cuando se aborta una transacción, normalmente será reiniciada por el programa del cliente. Pero en esquemas que confían en abortar y reiniciar transacciones, no hay garantía que una transacción particular pase siempre las comprobaciones de validación, por ello puede entrar en conflicto con otras transacciones en el uso de los objetos cada vez que es reiniciada. La prevención de que una transacción siempre será capaz de consumarse se llama inanición.

Las apariciones de la inanición son probablemente raras, pero un servidor que utiliza control de concurrencia optimista debe asegurar que un cliente no sufre un aborto constante de una de sus transacciones. Kung y Robinson sugieren que esto se puede conseguir si el servidor detecta una transacción que ha sido abortada varias veces. Ellos sugieren que cuando el servidor detecta tal transacción debiera dársele acceso exclusivo para usar la sección crítica, protegida por un semáforo.

12.6. ORDENACIÓN POR MARCAS DE TIEMPO

En los esquemas de control de concurrencia basados en ordenación por marcas de tiempo, cada operación en una transacción se valida cuando se lleva a cabo. Si la operación no puede ser validada, la transacción es abortada inmediatamente y puede ser reiniciada por el cliente. A cada transacción se le asigna una valor de marca de tiempo único cuando comienza. La marca de tiempo define su posición en la secuencia de tiempo de las transacciones. Las solicitudes de las transacciones pueden ser ordenadas totalmente de acuerdo con sus marcas de tiempo. La regla de ordenación básica por marca de tiempo está basada en los conflictos de operación y es muy sencilla:

- Una solicitud de una transacción para escribir un objeto es válida sólo si ese objeto fue leído y escrito por última vez por transacciones anteriores. Una petición de lectura de un objeto por una transacción es válida sólo si ese objeto fue escrito por última vez por una transacción anterior.

Esta regla supone que sólo existe una versión de cada objeto y restringe el acceso a una transacción en cada instante. Si cada transacción tiene su propia versión tentativa de cada objeto al que accede, entonces varias transacciones concurrentes pueden acceder al mismo objeto. La regla de ordenación por marca de tiempo está diseñada para asegurar que cada transacción accede a un conjunto consistente de versiones de los objetos. Debe asegurar también que las versiones tentativas de cada objeto son consumadas en el orden determinado por las marcas de tiempo de las transacciones que las realizaron. Esto se consigue haciendo que las transacciones esperen, cuando es preciso, a que las transacciones anteriores completen sus escrituras. Las operaciones de *escritura* pueden realizarse después que ha retornado la operación *cierraTransacción*, sin hacer que espere el cliente. Pero el cliente debe esperar cuando las operaciones de *lectura* deben esperar a que finalicen las transacciones anteriores. Esto no conduce a un bloqueo indefinido, puesto que las transacciones sólo esperan por las anteriores (y no puede ocurrir un ciclo en el grafo *espera por*).

Las marcas de tiempo pueden venir asignadas desde el reloj del servidor o, como en la sección anterior, puede usarse un *pseudo-tiempo* basado en un contador que se incrementa cuando se emite un valor de marca de tiempo. En el Capítulo 13 se considerará el problema de generar marcas de tiempo cuando el servicio de transacciones está distribuido y varios servidores están implicados en una transacción.

Describiremos ahora una forma de control de concurrencia basado en marca de tiempo siguiendo los métodos adoptados en el sistema SDD-1 [Bernstein y otros 1980] y descritos por Ceri y Pelaggatti [1985].

Como es normal, las operaciones de *escritura* son registradas en versiones tentativas de los objetos y son invisibles para las demás transacciones hasta que se realiza una solicitud *cierraTransacción* y la transacción se consuma. Cada objeto tiene una marca de tiempo de escritura y un conjunto de versiones tentativas, cada una de las cuales tiene una marca de tiempo de escritura asociada con ella; y un conjunto de marcas de tiempo de lectura. La marca de tiempo de escritura del objeto (consumado) es anterior que la de cualquiera de sus versiones tentativas, y el conjunto de marcas de tiempo de lectura puede representarse por su elemento con valor máximo. Cuando se acepta una operación de *escritura* de la transacción en un objeto, el servidor crea una nueva versión tentativa del mismo con la marca de tiempo de escritura colocada al valor de marca de tiempo de la transacción. Una operación de *lectura* de la transacción es dirigida a la versión con el valor máximo de la marca de tiempo de escritura menor que la de la marca de tiempo de la transacción. Cuando se acepta una operación de *lectura* sobre un objeto, la marca de tiempo de la transacción se añade a su conjunto de marcas de tiempo de lectura. Cuando se consuma una transacción, los valores de las versiones tentativas se convierten en los valores de los objetos, y las marcas de tiempo de las versiones tentativas se convierten en las de los objetos correspondientes.

En la ordenación por marcas de tiempo, cualquier solicitud por parte de un transacción para una operación de *lectura* o *escritura* en un objeto se comprueba para ver si está conforme con las reglas de conflicto de la operación. Una solicitud de la transacción actual T_c puede tener conflicto con las operaciones anteriores hechas por otras transacciones, T_i, cuyas marcas de tiempo indican que deberían ser posteriores a T_c. Estas reglas se muestran en la Figura 12.29, en la que $T_i > T_c$ significa que es posterior que T_c y $T_i < T_c$ significa que T_i es anterior a T_c.

Regla de escritura por ordenación de marca de tiempo: Combinando las reglas 1 y 2 tenemos la siguiente regla para decidir si se acepta una operación de lectura solicitada por la transacción T_c en el objeto D:

if ($T_c \geqslant$ la máxima marca de tiempo de lectura en D &&
  $T_c >$ la marca de tiempo de escritura en la versión consumada de D)
    realiza la operación de escritura en la versión tentativa de D con marca de tiempo T_c
else /* escribir es demasiado tarde */
  aborta la transacción T_c

Regla	T_c	T_i	
1.	*Escritura*	*Lectura*	T_c no debe *escribir* un objeto que haya sido *leído* por cualquier T_i donde $T_i > T_c$, esto requiere que $T_c \geqslant$ la mayor marca de tiempo de lectura del objeto.
2.	*Escritura*	*Escritura*	T_c no debe *escribir* un objeto que haya sido *escrito* por cualquier T_i donde $T_i > T_c$, esto requiere que $T_c >$ la marca de tiempo de escritura del objeto consumado.
3.	*Lectura*	*Escritura*	T_c no debe *leer* un objeto que haya sido *escrito* por cualquier T_i donde $T_i > T_c$, esto requiere que $T_c >$ la marca de tiempo de escritura del objeto consumado.

Figura 12.29. Conflictos de operación para ordenación por marcas de tiempo.

Si ya existe una versión con marca de tiempo de escritura T_c, la operación de *escritura* se dirige a ella, en otro caso se crea una nueva versión tentativa y se le da la marca de tiempo de escritura T_c. Hay que señalar que cualquier *escritura* que *llegue demasiado tarde* se aborta: es demasiado tarde en el sentido de que una transacción con una marca de tiempo posterior ya ha leído o escrito el objeto.

La Figura 12.30 ilustra la acción de una operación de *escritura* por la transacción T_3 en los casos donde $T_3 \geqslant$ máxima marca de tiempo de lectura en el objeto (las marcas de tiempo de lectura no se muestran). En los casos (a) a (c) $T_3 >$ la marca de tiempo de escritura en la versión consumada del objeto y una versión tentativa con marca de tiempo de escritura T_3 se inserta en el lugar apropiado en la lista de versiones ordenadas por las marcas de tiempo de tiempo de las transacciones. En el caso (d), $T_3 <$ marca de tiempo de escritura de la versión consumada del objeto y se aborta la transacción.

Regla de lectura por ordenación de marca de tiempo: Utilizando la regla 3 tenemos la siguiente regla para decidir si aceptar inmediatamente, esperar o rechazar una operación de *lectura* solicitada por una transacción T_c en el objeto D:

if ($T_c >$ la marca de tiempo de escritura en la versión consumada de D)
  sea $D_{seleccionada}$ la versión de D con la máxima marca de tiempo de escritura T_c
  if (se ha consumado $D_{seleccionada}$)
    realiza la operación de *lectura* en la versión $D_{seleccionada}$
  else
    espera hasta la consumación de la transacción que hizo la versión $D_{seleccionada}$, o aborte
    después reaplicar la regla de *lectura*
} else
  aborta la transacción T_c

Advierta que:

- Si la transacción T_c ha escrito su propia versión del objeto, ésta será utilizada.
- Una operación de *lectura* que llega demasiado pronto espera que se complete la anterior. Si la transacción anterior se consuma, entonces T_c leerá de su versión consumada. Si se aborta, entonces T_c repetirá la regla de lectura (y seleccionará la versión previa). Esta regla previene lecturas sucias.
- Una operación de *lectura* que llega *demasiado tarde* se aborta (es demasiado tarde en el sentido que una transacción con una marca de tiempo posterior ya ha escrito el objeto).

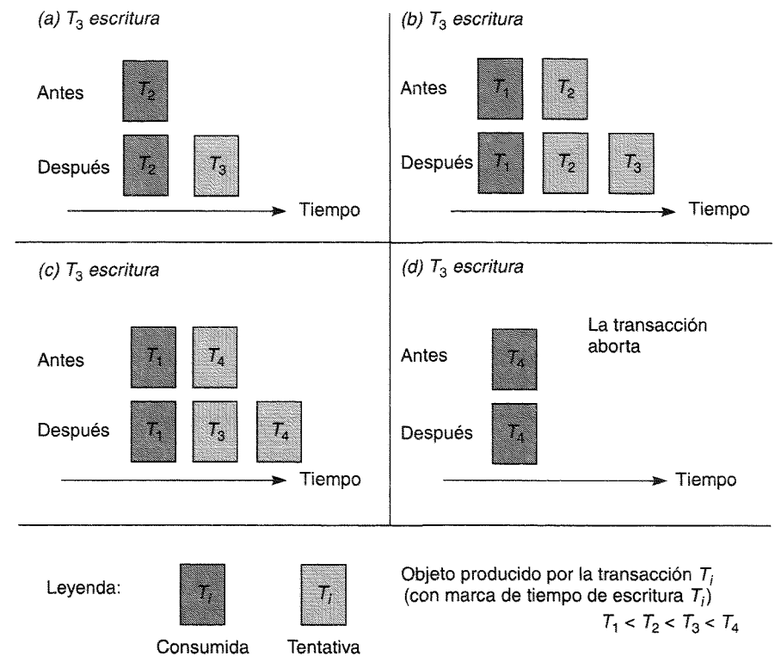


Figura 12.30. Operaciones de escritura y marcas de tiempo.

La Figura 12.31 ilustra la regla de lectura de la ordenación por marca de tiempo. Incluye cuatro casos etiquetados (a) a (d), cada uno de los cuales ilustra la acción de una operación de *lectura* por la transacción T_3. En cada caso, se selecciona una versión cuya marca de tiempo de escritura es menor o igual que T_3. Si existe tal versión, se indica con una línea. En los casos (a) y (b) la operación de *lectura* se dirige a una versión consumada, en (a) es la única versión, mientras que en (b) hay una versión tentativa que pertenece a una transacción posterior. En el caso (c) la operación de *lectura* está dirigida a una versión tentativa y debe esperar hasta que la transacción realiza la consumación o aborta. En el caso (d) no existe una versión adecuada para leer y la transacción T_3 se aborta.

Cuando un coordinador recibe una petición para consumar una transacción, siempre será capaz de hacerlo porque se comprueban todas las operaciones de las transacciones para consistencia con las transacciones anteriores antes de realizarlo. Las versiones consumadas de cada objeto deben ser creadas en el orden de la marca de tiempo. Por tanto, un coordinador necesita esperar, a veces, que se completen las transacciones anteriores antes de escribir todas las versiones consumadas de los objetos accedidos por una transacción particular, pero no necesita esperar por el cliente. Con el fin de hacer una transacción recuperable después de una ruptura, las versiones tentativas de los objetos y el hecho que la transacción se ha consumado deben ser escritos en almacenamiento permanente antes de reconocer a la solicitud del cliente para consumar la transacción.

Observe que este algoritmo de ordenación por marca de tiempo es estricto, asegura la ejecuciones estrictas de las transacciones (véase Sección 12.2). La regla de lectura por ordenación de marca de tiempo retarda una operación de lectura de una transacción en cualquier objeto hasta que todas las transacciones que han escrito previamente ese objeto han sido consumadas o abortadas. El plan

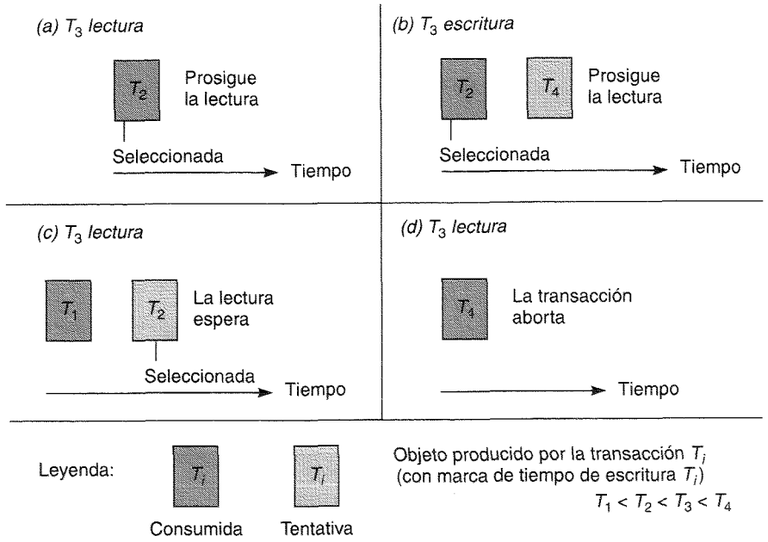


Figura 12.31. Operaciones de lectura y marcas de tiempo.

para consumar versiones en el orden asegura que la ejecución de una operación de *escritura* de la transacción en cualquier objeto se retrasa hasta que todas las transacciones que han escrito previamente ese objeto se han consumado o abortado.

En la Figura 12.32, volvemos a nuestra ilustración considerando las dos transacciones bancarias concurrentes T y U presentadas en la Figura 12.7. Las columnas encabezadas por A, B y C se refieren a información sobre cuentas con esos nombres. Cada cuenta tiene una entrada MTL que registra la marca de tiempo máxima de lectura y otra MTE que registra la marca de tiempo de escritura de cada versión, con las marcas de tiempo de versiones ya consumadas en negrita. Inicialmente, todas las cuentas tienen versiones consumadas escritas por la transacción S, y el conjunto de marcas de tiempo de lectura está vacío. Suponemos que $S < T < U$. El ejemplo muestra que cuando una transacción U está dispuesta a conseguir el balance de B esperará hasta que T finalice por lo que se puede leer el valor escrito por T si ésta se consuma.

El método de marca de tiempo escrito anteriormente no impide el bloqueo indefinido, tiende totalmente al rearranque. Existe una mejora conocida como regla de *ignorar escrituras obsoletas*. Es una modificación a la regla de escritura por ordenación de marca de tiempo:

Si una escritura es demasiado tardía puede ser ignorada en lugar de abortar la transacción, porque si hubiera llegado a tiempo sus efectos deberían haber sido sobrescritos en cualquier caso. Sin embargo, si otra transacción ha leído el objeto, la transacción con la última escritura falla debido a la marca de lectura en el ítem.

◇ **Ordenación por marca de tiempo multiversión.** En esta sección, hemos visto cómo la concurrencia proporcionada por la ordenación de marca de tiempo básica se mejora permitiendo a cada transacción escribir su propia versión tentativa de los objetos. En la ordenación de marca de tiempo multiversión, que fue presentada por Reed [1983], se mantiene una lista de versiones antiguas consumadas así como de versiones tentativas para cada objeto. Esta lista representa la historia de los valores del objeto. El beneficio de utilizar múltiples versiones está en que las operaciones de lectura que llegan demasiado tarde no necesitan ser rechazadas.

		Marcas de tiempo y versiones de objetos					
T	*U*	*A*		*B*		*C*	
		MTL	*MTE*	*MTL*	*MTE*	*MTL*	*MTE*
		{}	***S***	*{}*	***S***	*{}*	***S***
abreTransacción							
bal = b.obténBalance()				*{T}*			
	abreTransacción						
*b.ponBalance(bal*1.1)*					***S***, *T*		
	bal = b.obténBalance()						
	espera por T						
a.extrae(bal/10)	• • •		***S***, *T*				
consuma	• • •		***T***		***T***		
	bal = b.obténBalance()			*{U}*			
	*b.ponBalance(bal*1.1)*				***T***, *U*		
	c.extrae(bal/10)						***S***, *U*

Figura 12.32. Marcas de tiempo en las transacciones T y U.

Cada versión tiene una marca de tiempo de lectura que registra la marca de tiempo más grande de cualquier transacción que ha leído el objeto además de la marca de tiempo de tiempo de escritura. Como antes, cuando se acepta una operación de *escritura*, se dirige a una versión tentativa con la marca de tiempo de escritura de la transacción. Cuando se realiza una operación de *lectura* se dirige a la versión con marca de tiempo de escritura más grande menor que la marca de tiempo de la transacción. Si la marca de tiempo de la transacción es más grande que la marca de tiempo de la transacción de lectura que está siendo utilizada, la marca de tiempo de lectura se establecerá a la de la marca de tiempo de la transacción.

Cuando una lectura llega tarde, se le puede permitir leer de una versión antigua ya consumada, por lo que no hay necesidad de abortar las operaciones de *lectura* tardías. En ordenación por marcas de tiempo multiversión, las operaciones de *lectura* se permiten siempre, aunque puedan tener que *esperar* que transacciones anteriores se completen (ya consumándose o abortando), lo que asegura que las ejecuciones sean recuperables. Véase el Ejercicio 12.22 para una discusión de la posibilidad de abortos en cascada. Esto se trata con la regla 3 en las reglas de conflicto para ordenación por marcas de tiempo.

No hay conflicto entre operaciones de escritura de diferentes transacciones, porque cada transacción escribe su propia versión consumada de los objetos a los que accede. Esto elimina la regla 2 en las reglas de conflicto para ordenación por marcas de tiempo, dejándonos con:

Regla 1: T_c no debe escribir objetos que han sido leídos por cualquier T_i donde $T_i > T_c$.

Esta regla no se considerará si hay alguna versión del objeto con marca de tiempo de lectura $> T_c$, pero sólo esta versión tiene una marca de tiempo de escritura menor que o igual a T_c. (Esta escritura no puede tener ningún efecto en versiones posteriores.)

Regla de escritura en la ordenación de marca de tiempo multiversión: Como cualquier operación de *lectura* potencialmente conflictiva tendrá que ser dirigida hacia la versión más reciente de un objeto, el servidor inspecciona la versión $D_{\text{másTemprana}}$ con la marca de tiempo de escritura máxima menor o igual que T_c. Tenemos la regla siguiente para realizar una operación de *escritura* solicitada por la transacción T_c en el objeto D:

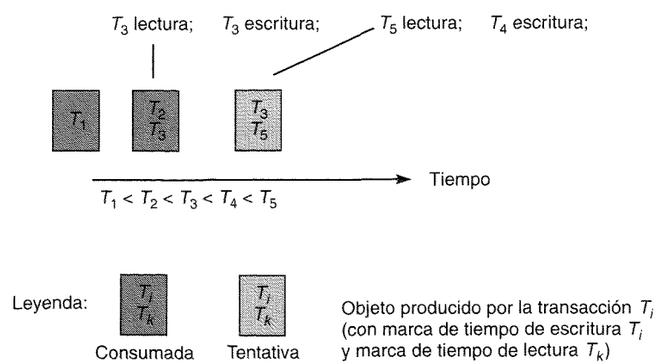


Figura 12.33. La última operación de *escritura* invalidaría una *lectura*.

if (marca de tiempo de lectura de $D_{\text{másTemprana}} \leqslant T_c$)
realiza la operación de *escritura* en una versión tentativa de D con la marca de tiempo de escritura T_c
else aborta la transacción T_c

La Figura 12.33 muestra un ejemplo donde se rechaza una *escritura*. El objeto ya tiene versiones consumadas con marcas de tiempo de escritura T_1 y T_2. El objeto recibe la siguiente secuencia de solicitudes para operaciones sobre el objeto:

T_2 *lectura*; T_3 *escritura*; T_5 *lectura*; T_4 *escritura*

1. T_3 solicita una operación de *lectura*, que pone una marca de tiempo T_3 en la versión de T_2.
2. T_3 solicita una operación de *escritura*, que hace una nueva versión tentativa con la marca de tiempo T_3.
3. T_5 solicita una operación de *lectura*, que utiliza la versión con la marca de tiempo de escritura T_3 (la marca de tiempo más alta que es menor que T_5).
4. T_4 solicita una operación de *escritura*, que es rechazada porque la marca de tiempo de lectura T_5 de la versión con marca de tiempo de escritura T_3 es más alta que T_4. (Si se permitiera la marca de tiempo de escritura de la nueva versión debería ser T_4. Si se permitiera tal versión, se debería invalidar la operación de lectura T_5, que debiera haber usado la versión con marca de tiempo T_4.)

Cuando una transacción se aborta, todas las versiones que ella creó se eliminan. Cuando una transacción se consuma, todas las versiones que ella creó se retienen, pero para controlar la utilización del espacio de almacenamiento, las versiones antiguas deberían ser borradas de cuando en cuando. Aunque supone una sobrecarga de espacio de almacenamiento, la ordenación de marca de tiempo multiversión permita una concurrencia considerable, no sufre de bloqueos indefinidos y permite siempre operaciones de *lectura*. Para más información sobre la ordenación por marca de tiempo multiversión consultar Bernstein y otros [1987].

12.7. COMPARACIÓN DE MÉTODOS PARA EL CONTROL DE CONCURRENCIA

Hemos descrito tres métodos distintos para controlar el acceso concurrente a datos compartidos: bloqueo en dos fases estricto, métodos optimistas y ordenación por marca de tiempo. Todos los métodos implican algunas sobrecargas en el tiempo y el espacio que precisan, y todos limitan de alguna forma el potencial de la operación concurrente.

El método de ordenación por marca de tiempo es similar al bloqueo en dos fases en el sentido que ambos utilizan una aproximación pesimista en la que los conflictos entre transacciones son detectadas a medida en que cada objeto es accedido. Por un lado, la ordenación por marca de tiempo decide el orden de secuenciación estáticamente, cuando comienza una transacción. Por otro lado, el bloqueo en dos fases decide el orden de secuenciación dinámicamente, de acuerdo con el orden en el que son accedidos los objetos. La ordenación por marca de tiempo y, en particular, la ordenación por marca de tiempo multiversión es mejor que el bloqueo en dos fases estricto para transacciones de sólo lectura. El bloqueo en dos fases es mejor cuando las operaciones en las transacciones son actualizaciones predominantemente.

Algunos trabajos plantean que la ordenación por marca de tiempo es beneficiosa para transacciones con operaciones predominantemente de *lectura* y que el bloqueo es beneficioso para transacciones con más *escrituras* que *lecturas* como un argumento para permitir esquemas híbridos en los que algunas transacciones utilizan ordenación por marca de tiempo y otras bloqueo para el control de concurrencia. Los lectores que estén interesados en la utilización de este tipo de métodos mezclados deben leer Bernstein y otros [1987].

Los métodos pesimistas difieren en la estrategia utilizada cuando se detecta un conflicto de acceso sobre un objeto. La ordenación por marca de tiempo aborta la transacción inmediatamente, mientras el bloqueo hace esperar a la transacción, pero con una posible penalización de aborto más tarde para impedir el bloqueo indefinido.

Cuando se utiliza el control de concurrencia optimista, se permite proceder a todas las transacciones, pero algunas son abortadas cuando ellas intentan la consumación, o en validación hacia delante las transacciones son abortadas antes. Esto consigue una operación relativamente eficiente cuando hay pocos conflictos, pero se puede repetir una cantidad sustancial de trabajo cuando se aborta una transacción.

El bloqueo se ha utilizado durante muchos años en los sistemas de bases de datos, mientras la ordenación por marca de tiempo ha sido utilizada en el sistema de bases de datos SDD-1. Ambos métodos se han utilizado en servidores de archivos.

Varios sistemas distribuidos, por ejemplo Argus [Liskov 1988] y Arjuna [lShrivastava y otros 1991], han explorado el uso de los bloqueos semánticos, ordenación por marcas de tiempo y nuevas aproximaciones para transacciones largas.

El trabajo sobre dos áreas de aplicación ha mostrado que los mecanismos anteriores de control de concurrencia no son siempre adecuados. Una de esas áreas trata con aplicaciones multiusuario en las que todos los usuarios esperan disponer de vistas comunes de los objetos que están siendo actualizados por cualquiera de ellos. Dichas aplicaciones precisan que sus datos sean atómicos en presencia de actualizaciones concurrentes y caídas del servidor, y las técnicas de transacciones parecen ofrcer una aproximación a su diseño. Sin embargo, estas aplicaciones tiene dos nuevos requisitos relacionados con el control de concurrencia: (1) el usuario precisa la notificación inmediata de los cambios realizados por otros usuarios, lo que es contrario a la idea de aislamiento, y (2) los usuarios necesitan ser capaces de acceder a los objetos antes de que otros usuarios hayan completado sus transacciones, lo que conduce al desarrollo de nuevos tipos de bloqueo que disparan acciones cuando se accede a los objetos. El trabajo en esta área ha sugerido muchos esquemas que relajan el aislamiento y proporcionan notificación de los cambios. Para una revisión de este trabajo,

véase Ellis y otros [1991]. La segunda área de aplicación está relacionado con lo que a veces se describe como aplicaciones avanzadas de bases de datos, como CAD/CAM cooperativo y sistemas de desarrollo de software. En tales aplicaciones, las transacciones duran un tiempo largo, y los usuarios trabajan en versiones independientes de los objetos que se obtienen de una base de datos común y se comprueban cuando el trabajo ha concluido. La mezcla de versiones requiere la cooperación entre los usuarios. Para una revisión de este trabajo, véase Barghouti y Kaiser [1991].

12.8. RESUMEN

Las transacciones proporcionan un medio mediante el cual los clientes pueden especificar secuencias de operaciones que son atómicas en presencia de otras transacciones concurrentes y de caídas del servidor. El primer aspecto de la atomicidad se consigue ejecutando las transacciones de forma que sus efectos sean secuencialmente equivalentes. Los efectos de las transacciones consumadas se registran en un almacenamiento permanente por lo que el servicio de transacciones se puede recuperar de fallos en los procesos. Para permitir a las transacciones la capacidad de abortar, sin tener efectos nocivos en otras transacciones, la ejecuciones deben ser estrictas (es decir, las lecturas y escrituras sobre una transacción deben ser retrasadas hasta que otra transacción que escriba sobre los mismos objetos haya consumado su operación o haya abortado). Para permitir a las transacciones la elección entre consumación o aborto, sus operaciones se realizan sobre versiones tentativas que no pueden ser accedidas por otras transacciones. Las versiones tentativas de los objetos se copian a objetos reales y se almacenan permanentemente cuando se consuma una transacción.

Las transacciones anidadas se forman estructurando unas transacciones a partir de otras. En los sistemas distribuidos se utiliza particularmente el anidamiento porque permiten la ejecución concurrente de subtransacciones en servidores separados. El anidamiento tiene también la ventaja de permitir la recuperación independiente de partes de una transacción.

Los conflictos de operación forman una base para la derivación de los protocolos de control de concurrencia. Los protocolos no sólo deben garantizar la secuenciación sino también permitir la recuperación utilizando ejecuciones estrictas para impedir los problemas asociados con el aborto de transacciones, como los abortos en cascada.

En la planificación de una operación en una transacción son posibles tres estrategias. Son (1) ejecutarla inmediatamente, (2) retrasarla, o (3) abortarla.

El bloqueo en dos fases estricto utiliza las dos primeras estrategias, recurriendo al aborto sólo en el caso de bloqueo indefinido (deadlock). Dicho bloqueo garantiza la seriabilidad ordenando las transacciones de acuerdo a cuando acceden a objetos comunes. Su principal desventaja es que pueden aparecer bloqueos indefinidos.

La ordenación por marcas de tiempo utiliza las tres estrategias para garantizar la seriabilidad, ordenando los accesos de las transacciones a los objetos de acuerdo con el instante de comienzo de las mismas. Este método no puede verse afectado de bloqueos indefinidos y es ventajoso para transacciones de sólo lectura. Sin embargo, las transacciones deben ser abortadas cuando llegan demasiado tarde. La ordenación por marcas de tiempo multiversión es particularmente efectiva.

El control optimista de concurrencia permite que se realicen transacciones sin ninguna forma de confirmación hasta que se han completado. Las transacciones son validadas antes de que se permita su consumación. La validación hacia atrás requiere el mantenimiento de múltiples conjuntos de escritura de las transacciones consumadas, mientras que la validación hacia delante debe validar las transacciones activas y tiene la ventaja que permite estrategias alternativas para resolver los conflictos. Se puede producir la muerte por inanición como consecuencia de los abortos repetidos de una transacción cuya validación falla en el control optimista de concurrencia e incluso en la ordenación por marcas de tiempo.

EJERCICIOS

12.1. La BolsadeTareas es un servicio cuya funcionalidad es proporcionar un almacén para las *descripciones de tareas*. Permite a los clientes de diferentes computadores realizar partes de una computación en paralelo. Un proceso *maestro* coloca las descripciones de las subtareas de una computación en la BolsadeTareas, y un proceso *trabajador* selecciona tareas de la BolsadeTareas y las efectúa, devolviendo los resultados a la BolsadeTareas. El *maestro* recoge los resultados y los combina para producir el resultado final.

El servicio BolsadeTareas proporciona las siguientes operaciones:

ponTarea permite a los clientes añadir descripciones de tareas a la bolsa;
tomaTarea permite a los clientes tomar descripciones de tareas de la bolsa.

Un cliente realiza una petición *tomaTarea*, cuando una tarea no está disponible pero puede estar disponible en breve. Discutir las ventajas y desventajas de las siguientes alternativas:

(i) El servidor puede responder inmediatamente, diciendo al cliente que lo intente más tarde.
(ii) Se hace que el servidor espere (y por lo tanto también el cliente) hasta que una tarea esté disponible.
(iii) Se utilizan devoluciones de llamada.

12.2. Un servidor gestiona los objetos $a_1, a_2, \ldots a_n$. El servidor proporciona dos operaciones a sus clientes:

lee(i) devuelve el valor de a_i;
escribe(i, Valor) asigna *Valor* a a_i.

Las transacciones T y U se definen de la siguiente forma:

T: *x = lee(j); y = lee(i); escribe(j, 44); escribe(i, 33);*
U: *x = lee(k); escribe(i, 55); y = lee(j); escribe(k, 66);*

Proporcione tres solapamientos serialmente equivalentes de las transacciones T y U.

12.3. Obtenga solapamientos secuencialmente equivalentes de T y U del Ejercicio 12.2 con las siguientes propiedades: (1) que sean estrictos; (2) que no sean estrictos pero no puedan producir abortos en cascada; (3) que puedan producir abortos en cascada.

12.4. La operación *crea* inserta una nueva cuenta en una sucursal. Las transacciones T y U están definidas como siguen:

T: *unaSucursal.crea(«Z»);*
U: *z.deposita(10); z.deposita(20);*

Supóngase que Z no existe todavía. Supóngase también que la operación *deposita* no hace nada si la cuenta proporcionada en el argumento no existe. Considérese el siguiente solapamiento de las transacciones T y U:

T	*U*
unaSucursal.crea(Z)	*z.deposita(10);* *z.deposita(20):*

Establecer el balance de Z después de la ejecución en este orden. ¿Son consistentes todas las ejecuciones de T y U secuencialmente equivalentes?

12.5. Un objeto recién creado como Z en el Ejercicio 12.4 se llama a veces un *fantasma*. Desde el punto de vista de la transacción U, Z no existe al principio y entonces aparece (como un fantasma). Explicar, con un ejemplo, cómo se podría producir un fantasma cuando se borra una cuenta.

12.6. Las transacciones *transferencia* T y U se definen como:

T: a.extrae(4); b.deposita(4);
U: c.extrae(3); b.deposita(3);

Supóngase que están estructuradas como pares de transacciones anidadas:

T_1: a.extrae(4); T_2: b.deposita(4);
U_1: c.extrae(3); U_2: b.deposita(3);

Compare el número de solapamientos secuencialmente equivalentes de T_1, T_2, U_1 y U_2 con el número de solapamientos secuencialmente equivalentes de T y U. Explique por qué el uso de transacciones anidadas permite un número más grande de solapamientos secuencialmente equivalentes que las no anidadas.

12.7. Considere los aspectos de recuperación de las transacciones anidadas definidas en el Ejercicio 12.6. Suponga que una transacción *reintegro* abortará si la cuenta queda en descubierto y que en este caso la transacción padre también abortará. Describa solapamientos de T_1, T_2, U_1 y U_2 secuencialmente equivalentes con las siguientes propiedades: (i) que sean estrictos; (ii) que no sean estrictos. ¿Con qué opción el criterio de ser estricto reduce la ganancia potencial de concurrencia de las transacciones anidadas?

12.8. Explique por qué la equivalencia secuencial precisa que una vez que una transacción ha liberado un bloqueo sobre un objeto, no le está permitido obtener más bloqueos.

Un servidor gestiona los objetos a_1, a_2, ... a_n. El servidor proporciona a sus clientes dos operaciones:

lee(i) devuelve el valor de a_i
escribe(i, Valor) asigna *Valor* a a_i

Las transacciones T y U están definidas de la forma siguiente:

T: *x = lee(i); escribe(j, 44);*
U: *escribe(i, 55); escribe(j, 66);*

Describa un solapamiento de las transacciones T y U en el que los bloqueos se liberan prontamente con el efecto de que el solapamiento no es secuencialmente equivalente.

12.9. Las transacciones T y U en el servidor del Ejercicio 12.8 se definen:

T: *x = lee(i); escribe(j, 44);*
U: *escribe(i, 55); escribe(j, 66);*

Los valores iniciales de a_i y a_j son 10 y 20 respectivamente. ¿Cuál de los siguientes solapamientos son secuencialmente equivalentes, y qué ocurriría con un bloqueo en dos fases?

(a)

T	*U*
x = lee(i);	
	escribe(i, 55);
escribe(j, 44);	
	escribe(j, 66);

(b)

T	*U*
x = lee(i);	
escribe(j, 44);	
	escribe(i, 55);
	escribe(j, 66);

(c)

T	*U*
	escribe(i, 55)
	escribe(j, 66);
x = lee(i);	
escribe(j, 44);	

(d)

T	*U*
	escribe(i, 55);
x = lee(i);	
	escribe(j, 66);
escribe(j, 44);	

12.10. Considere una relajación de los bloqueos en dos fases en los que las transacciones de *sólo lectura* pueden leer los bloqueos con anterioridad. ¿Debiera tener recuperaciones consistentes una transacción de sólo lectura? ¿Llegarían los objetos a ser inconsistentes? Ilustre su respuesta con las siguientes transacciones T y U en el servidor del Ejercicio 12.8:

T: *x = lee(i); y = lee(j);*
U: *escribe(i, 55); escribe(j, 66);*

en las que los valores iniciales de a_i y a_j son 10 y 20.

12.11. Las ejecuciones de transacciones son estrictas si las operaciones *lee* y *escribe* sobre un objeto son retrasadas hasta que todas las transacciones que escribieron sobre ese objeto previamente se han consumado o abortado. Explique cómo las reglas de bloqueo de la Figura 12.16 aseguran ejecuciones estrictas.

12.12. Describa cómo se podría alcanzar una situación no recuperable si los bloqueos de escritura son liberados después de la última operación de una transacción pero antes de su consumación.

12.13. Explique por qué las ejecuciones son siempre estrictas, incluso si los bloqueos de lectura son liberados después de la última operación de una transacción pero antes de su consumación. Proporcione una exposición mejorada de la regla 2 de la Figura 12.16.

12.14. Considere un esquema de detección de bloqueo indefinido para un servidor único. Describa de forma precisa cuándo se añaden y se eliminan arcos de un grafo *espera por*.

Ilustre su respuesta con respecto a las siguientes transacciones T, U y V en el servidor del Ejercicio 12.8.

T	*U*	*V*
	escribe(i, 66)	
escribe(i, 55)		
		escribe(i, 77)
	consuma	

Cuando U libera su bloqueo de escritura sobre a_i, T y V están esperando para obtener bloqueos de escritura para ellas. ¿Funciona correctamente su esquema si T (llega primero) consigue el bloqueo antes que V? Si su respuesta es «No», modifique su descripción.

12.15. Considere bloqueos jerárquicos como los representados en la Figura 12.26. ¿Qué bloqueos se deben activar cuando se asigna una cita para un intervalo de tiempo en la semana w, día d, a la hora t? ¿En qué orden se debieran activar estos bloqueos? ¿Es importante el orden en el que se liberan?

¿Qué bloqueos se deben activar cuando se llega a los intervalos de tiempo para cada día de la semana w? ¿Se puede realizar esto cuando los bloqueos para la asignación de una cita a un intervalo de tiempo ya están activados?

12.16. Considérese que se aplica el control de concurrencia optimista a las transacciones T y U definidas en el ejercicio 12.9. Supóngase que las transacciones T y U están ambas activas en el mismo instante. Describa el resultado en cada uno de los casos siguientes:

(i) La solicitud de T para su consumación llega primero y se utiliza la validación hacia atrás.
(ii) La solicitud de U para su consumación llega primero y se utiliza la validación hacia atrás.
(iii) La solicitud de T para su consumación llega primero y se utiliza la validación hacia adelante.
(iv) La solicitud de U para su consumación llega primero y se utiliza la validación hacia adelante.

En cada caso describa la secuencia en la que se realizan las operaciones de T y U, recordando que las escrituras no se realizan hasta después de la validación.

12.17. Considérese el siguiente solapamiento de las transacciones T y U:

T	*U*
abreTransacción	*abreTransacción*
y = lee(k);	
	escribe(i, 55);
	escribe(j, 66);
	consuma
x = lee(i);	
escribe(j, 44);	

El resultado del control de concurrencia optimista con validación hacia atrás es que se abortará T porque su operación *lee* tiene conflictos con la operación *escribe* de U sobre a_i, aunque los solapamientos sean secuencialmente equivalentes. Sugiera una modificación del algoritmo que solucione dichos casos.

12.18. Haga una comparación de las secuencias de operaciones de las transacciones T y U del Ejercicio 12.8 que son posibles bajo bloqueo en dos fases (Ejercicio 12.9) y bajo control de concurrencia optimista (Ejercicio 12.16).

12.19. Considere el uso de ordenación por marcas de tiempo con cada uno de los solapamientos ejemplo de las transacciones T y U en el Ejercicio 12.9. Los valores iniciales de a_i y a_j son 10 y 20, respectivamente, y las marcas de tiempo de lectura y escritura iniciales son t_0. Suponga que cada transacción se abre y obtiene una marca de tiempo justo antes de su primera operación; por ejemplo, en (a) T y U obtienen las marcas de tiempo t_1 y t_2 respectivamente, donde $t_0 < t_1 < t_2$. Describa los efectos de cada operación de T y U en orden ascendente de tiempo. Para cada operación establece lo siguiente:

(i) Si la operación puede realizarse de acuerdo con la regla de lectura o escritura.
(ii) Las marcas de tiempo asignadas a las transacciones o a los objetos.
(iii) Creación de objetos tentativos y sus valores.

¿Cuáles son los valores finales de los objetos y sus marcas de tiempo?

12.20. Repita el Ejercicio 12.19 para los siguientes solapamientos de las transacciones T y U:

T	*U*
abreTransacción	
	abreTransacción
	escribe(i, 55);
	escribe(j, 66);
x = lee(i);	
escribe(j, 44);	
	consuma

T	*U*
abreTransacción	
	abreTransacción
	escribe(i, 55);
	escribe(j, 66);
	consuma
x = lee(i);	
escribe(j, 44);	

12.21. Repita el Ejercicio 12.20 empleando la ordenación de marcas de tiempo multiversión.

12.22. En la ordenación por marcas de tiempo multiversión, las operaciones de *lectura* pueden acceder a versiones tentativas de los objetos. Dé un ejemplo para mostrar cómo pueden suceder los abortos en cascada si se permite que todas las operaciones de *lectura* se realicen inmediatamente.

12.23. ¿Cuáles son las ventajas y las desventajas de la ordenación por marcas de tiempo multiversión en comparación con la ordenación por marcas de tiempo ordinaria?

12.24. Haga una comparación de las secuencias de operaciones de las transacciones T y U del Ejercicio 12.8 que sean posibles bajo el bloqueo en dos fases (Ejercicio 12.9) y bajo control de concurrencia optimista (Ejercicio 12.16).

13

TRANSACCIONES DISTRIBUIDAS

13.1. Introducción
13.2. Transacciones distribuidas planas y anidadas
13.3. Protocolos de consumación atómica
13.4. Control de concurrencia en transacciones distribuidas
13.5. Interbloqueos distribuidos
13.6. Recuperación de transacciones
13.7. Resumen

Este capítulo presenta las transacciones distribuidas, que son aquellas que involucran a más de un servidor. Las transacciones distribuidas pueden ser planas o anidadas.

Un protocolo de consumación atómica es un procedimiento cooperativo utilizado por un conjunto de servidores involucrados en una transacción distribuida. Permite que los servidores lleguen a una decisión conjunta sobre si una transacción debe ser consumada o debe ser abortada. Este capítulo describe el protocolo de consumación en dos fases, que es el protocolo de consumación atómica más habitualmente utilizado.

La sección dedicada al control de concurrencia entre transacciones distribuidas discute cómo pueden extenderse las técnicas de control de concurrencia (mediante bloqueo, ordenación mediante marcas temporales y optimista) para su utilización en las transacciones distribuidas.

La utilización de esquemas de bloqueo puede llevar a interbloqueos distribuidos. Se discutirán los algoritmos de detección de interbloqueo distribuido.

Los servidores que proporcionan transacciones incluyen un gestor de recuperación, cuyo cometido es asegurarse de que los efectos de las transacciones sobre los objetos gestionados por un servidor puedan recuperarse cuando se le reemplaza tras un fallo. El gestor de recuperación guarda los objetos en dispositivos de almacenamiento permanente, junto con las listas de intenciones y la información acerca del estado de cada transacción.

13.1. INTRODUCCIÓN

En el Capítulo 12 se discutieron las transacciones planas y anidadas que accedían a objetos en un único servidor. En el caso general, ya sean planas o anidadas, accederán a objetos ubicados en diversos computadores distintos. Utililzamos el término *transacción distribuida* para hacer referencia a una transacción plana o anidada que accede a objetos gestionados por múltiples servidores.

Cuando una transacción distribuida llega a su fin, la propiedad de atomicidad de la transacción requiere que, todos los servidores involucrados completen la transacción o que todos aborten la transacción. Para lograr esto, uno de los servidores asume el papel de coordinador, lo que supone el asegurar el mismo resultado en todos los servidores. La forma en la que el coordinador lo consigue depende del protocolo elegido. El protocolo conocido como «protocolo de consumación en dos fases» es el más habitualmente utilizado. Este protocolo permite a los servidores que se comuniquen entre ellos para llegar a una decisión conjunta sobre si consumar la transacción o abortarla.

El control de concurrencia en transacciones distribuidas se basa en los métodos que se discutieron en el Capítulo 12. Cada servidor aplica un control de concurrencia local sobre sus propios objetos, lo que garantiza la secuenciación local de las transacciones. Las transacciones distribuidas deben secuenciarse globalmente. Cómo se consiga esto depende de que se esté utilizando control de concurrencia mediante bloqueo, ordenación mediante marcas temporales u optimista. En algunos casos, las transacciones pueden secuenciarse en cada servidor particular, pero al mismo tiempo puede darse un ciclo de dependencias entre distintos servidores, dando lugar a un interbloqueo distribuido.

La recuperación de transacciones se preocupa de asegurar que sean recuperables todos los objetos involucrados en una transacción. Además de esto, garantiza que los valores de los objetos reflejen todos los cambios producidos por las transacciones consumadas y ningún cambio producido por las transacciones abortadas.

13.2. TRANSACCIONES DISTRIBUIDAS PLANAS Y ANIDADAS

Una transacción de un cliente se hace distribuida si invoca operaciones en varios servidores diferentes. Existen dos formas distintas en las que pueden estructurarse las transacciones distribuidas: como transacciones planas y como transacciones anidadas.

En una transacción plana, un cliente realiza peticiones a más de un servidor. Por ejemplo, en la Figura 13.1(a) la transacción T es una transacción plana que invoca operaciones en objetos de los servidores X, Y y Z. Una transacción cliente plana completa cada petición antes de pasar a la siguiente. En consecuencia, cada transacción accede secuencialmente a los objetos de los servidores. Cuando los servidores utilizan bloqueos, una transacción sólo puede estar esperando por un objeto cada vez.

En una transacción anidada, la transacción de nivel superior puede abrir subtransacciones, y cada subtransacción puede abrir subtransacciones más profundas hasta cualquier profundidad de anidamiento. La Figura 13.1(b) muestra a la transacción T de un cliente que abre dos subtransacciones T_1 y T_2, que acceden a los objetos en los servidores X e Y. Las subtransacciones T_1 y T_2 abren a su vez las subtransacciones T_{11}, T_{12}, T_{21} y T_{22}, que acceden a los objetos en los servidores M, N y P. En el caso anidado, las subtransacciones del mismo nivel pueden ejecutarse concurrentemente, luegon T_1 y T_2 son concurrentes y, ya que invocan objetos en servidores distintos, pueden ejecutarse en paralelo. Las cuatro subtransacciones T_{11}, T_{12}, T_{21} y T_{22} también se ejecutan de forma concurrente.

Considérese una transacción distribuida en la cual un cliente transfiere diez dólares desde una cuenta A a otra C, y después transfiere veinte dólares desde B hasta D. Las cuentas A y B están en

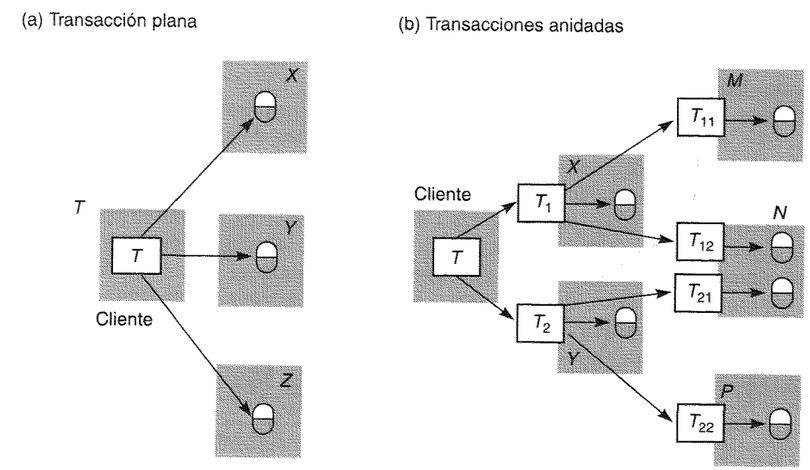


Figura 13.1. Transacciones distribuidas.

servidores separados X e Y, y las cuentas C y D están en el servidor Z. Si la transacción se estructura como un conjunto de cuatro transacciones anidadas, como se muestra en la Figura 13.2, las cuatro peticiones (dos *depósitos* y dos *extracciones*) pueden ejecutarse en paralelo, y el efecto global puede conseguirse con mejor rendimiento que el de una transacción simple en la que las cuatro operaciones se invocan secuencialmente.

13.2.1. EL COORDINADOR DE UNA TRANSACCIÓN DISTRIBUIDA

Los servidores que ejecutan peticiones como parte de una transacción distribuida necesitan poder comunicarse entre ellos para coordinar sus acciones cuando se consuma la transacción. Un cliente comienza una transacción enviando una petición de *abreTransacción* al coordinador en cualquier servidor, como se describe en la Sección 12.2. El coordinador con el que se contacta lleva a cabo

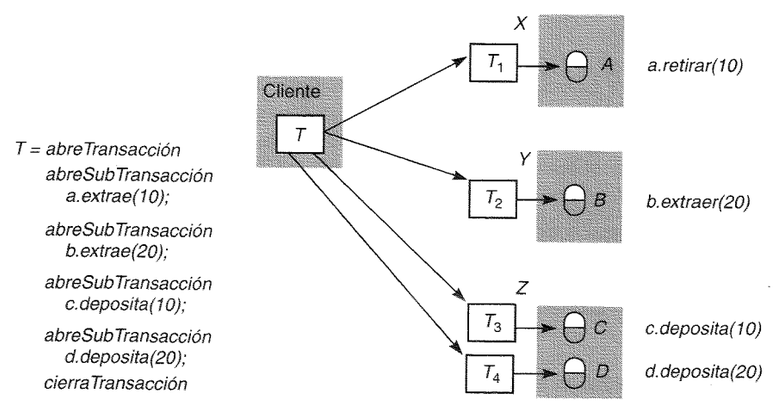


Figura 13.2. Transacción bancaria anidada.

abrirTransacción y devuelve al cliente el identificador resultante de la transacción. Los identificadores de transacciones para transacciones distribuidas han de ser únicos dentro del sistema distribuido. Una forma sencilla de obtener esto es que cada TID contenga dos partes: el identificador del servidor (por ejemplo una dirección IP) que la creó y un número único dentro del servidor.

El coordinador que abrió la transacción se convierte en el *coordinador* para la transacción distribuida y, es el responsable final de consumarla o abortarla. Cada uno de los servidores que gestione un objeto al que accede la transacción es un participante en la transacción y proporciona un objeto que llamaremos *participante*. Cada participante es responsable de seguir la pista de todos los objetos recuperables en el servidor implicado en la transacción. Los participantes son responsables de cooperar con el coordinador para sacar adelante el protocolo de consumación.

Durante el progreso de una transacción el coordinador registra una lista de referencias de participantes, y cada participante registra una referencia hacia el coordinador.

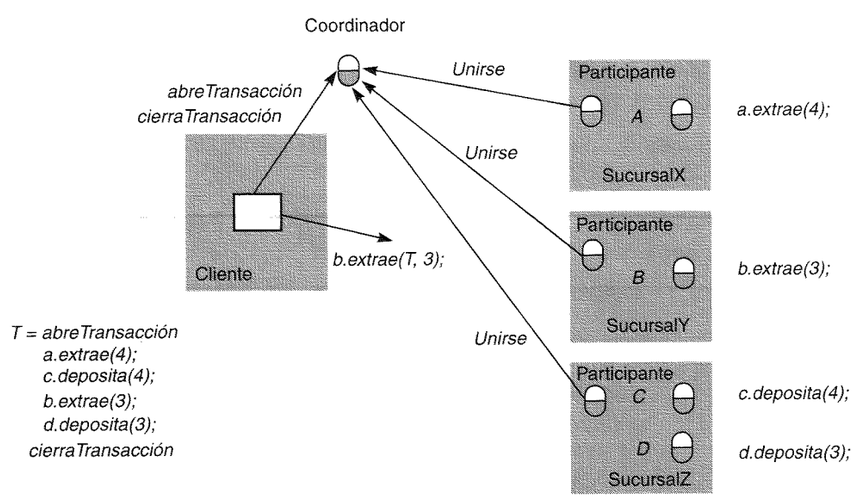
La interfaz para el *Coordinador* que se mostró en la Figura 12.3 proporciona un método adicional, *unirse*, que es cuando un participante se une a la transacción:

---

unirse(Trans, referencia hacia el participante)
Informa al coordinador que un nuevo participante se ha unido a la transacción *Trans*.

El coordinador registra al nuevo participante en su lista de participantes. El hecho de que el coordinador conozca a todos los participantes y que cada participante conozca al coordinador les permitirá recoger la información que necesitarán en el momento de la consumación.

La Figura 13.3 muestra a un cliente cuya transacción bancaria (plana) involucra las cuentas *A*, *B*, *C* y *D* en los servidores SucursalX, SucursalY y SucursalZ. La transacción del cliente, *T*, transfiere cuatro dólares desde la cuenta *A* a la cuenta *C*, y a continuación transfiere tres dólares desde la cuenta *B* hasta la cuenta *D*. La transacción descrita en la parte izquierda de la figura se expande para mostrar que *abreTransacción* y *cierraTransacción* se dirigen al coordinador, que podría estar



Nota: el coordinador está en uno de los servidores, por ejemplo SucursalX

Figura 13.3. Un ejemplo de transacción bancaria distribuida.

ubicado en uno de los servidores involucrados en la transacción. Cada servidor se muestra por medio de un *participante*, que se une a la transacción invocando al método *unirse* del coordinador. Cuando el cliente invoca uno de los métodos dentro de la transacción, por ejemplo *b.extrae(T, 3)*, el objeto que recibe la invocación (en este caso *B* en la SucursalY) informa a su objeto participante que el objeto pertenece a la transacción *T*. Si no ha informado ya al coordinador, el objeto participante utilizará la operación *unirse* para hacerlo. En este ejemplo, se muestra el identificador de la transacción siendo pasado como un argumento adicional de tal forma que el receptor puede pasárselo al coordinador. En el momento en el que el cliente llame a *cierraTransacción*, el coordinador tiene referencias de todos los participantes.

Nótese que es posible que un participante llame a *abortaTransacción* en el coordinador si, por alguna razón, es incapaz de continuar con la transacción.

13.3. PROTOCOLOS DE CONSUMACIÓN ATÓMICA

Los protocolos de consumación de transacciones fueron desarrollados a comienzos de la década de los años setenta, y el protocolo de consumación en dos fases apareció en Gray [1978]. La atomicidad de las transacciones requiere que al finalizar una transacción distribuida, bien se han llevado a cabo todas las operaciones o bien no se ha llevado a cabo ninguna. En el caso de una transacción distribuida, el cliente ha solicitado operaciones en más de un servidor. Una transacción finaliza cuando el cliente solicita que la transacción sea consumada o abortada. Una forma sencilla de completar la transacción de una forma atómica es que el coordinador comunique, a todos los participantes en la transacción, la petición de consumar o abortar, y continúe repitiendo la petición hasta que todos ellos hayan enviado un acuse de recibo indicando que la han llevado a cabo. Éste es un ejemplo de *protocolo de consumación atómica en una fase*.

Este sencillo protocolo de consumación atómica en una fase es inadecuado porque, en el caso en el que el cliente solicita una consumación, no permite a un servidor tomar una decisión unilateral de abortar la transacción. Las razones que impiden que un servidor sea capaz de consumar su parte de una transacción están relacionadas, generalmente, con temas de control de concurrencia. Por ejemplo, si se está usando un bloqueo, la resolución de un interbloqueo puede llevar a abortar una transacción sin que el cliente sea consciente de ello, a no ser que haga otra petición al servidor. Si se utiliza un control de concurrencia optimista, el fallo de la validación en un servidor podría causar que éste decida abortar la transacción. El coordinador podría no saber cuándo se ha caído un servidor y ha sido reemplazado durante el progreso de una transacción distribuida (dicho servidor necesitará abortar la transacción).

El *protocolo de consumación en dos fases* está diseñado para permitir que cualquier participante aborte su parte de la transacción. Debido al requisito de atomicidad, si una parte de la transacción es abortada, en consecuencia debe abortarse también la transacción en su totalidad. En la primera fase del protocolo, cada participante vota para que la transacción sea consumada o abortada. Una vez que un participante ha votado por la consumación de la transacción, no se le permite que la aborte. Por tanto, antes de que un participante vote para que se consume la transacción, debe asegurarse que será capaz de llevar a cabo su parte del protocolo de consumación, incluso si falla y es reemplazado en su transcurso. Se dice que un participante en una transacción está en estado *preparado* para una transacción si será finalmente capaz de consumarla. Para estar seguro de esto, cada participante guarda en un dispositivo de almacenamiento permanente todos los objetos que haya alterado durante la transacción, junto con su estado (preparado).

En la segunda fase del protocolo, todo participante en la transacción lleva a cabo la decisión conjunta. Si alguno de los participantes vota por abortar, entonces la decisión ha de ser de abortar la transacción. Si todos los participantes votan consumar, entonces la decisión es de consumar la transacción.

El problema está en estar seguro de que todos los participantes votan y que todos llegan a la misma decisión. Esto es relativamente sencillo si no ocurre ningún fallo; no obstante el protocolo debe trabajar correctamente incluso cuando algunos de los servidores fallan, se pierden mensajes o los servidores son temporalmente incapaces de comunicarse entre ellos.

◇ **Modelo de fallos para los protocolos de consumación.** La Sección 12.12 presentaba un modelo de fallo para transacciones que se aplica también a los protocolos de consumación en dos fases (o a cualquier otro protocolo de consumación). Los protocolos de consumación se diseñan para trabajar en un sistema asíncrono, en el cual pueden caer los servidores o pueden producirse pérdidas de mensajes. Se supone que un protocolo subyacente de petición-respuesta elimina los mensajes duplicados y los corruptos. No existen fallos extraños, es decir, los servidores se caen o bien obedecen los mensajes que se les envían.

El protocolo de consumación en dos fases es un ejemplo de un protocolo para alcanzar el consenso. El Capítulo 11 estableció que no puede alcanzarse el consenso en un sistema asíncrono si los procesos, de vez en cuando, fallan. Sin embargo, el protocolo de consumación en dos fases es capaz de alcanzar el consenso en estas condiciones. Esto se debe a que los fallos por caída de los procesos se enmascaran reemplazando el proceso caído por un nuevo proceso cuyo estado se establece partiendo de la información guardada en un dispositivo de almacenamiento permanente y de la información que han retenido otros procesos.

13.3.1. EL PROTOCOLO DE CONSUMACIÓN EN DOS FASES

Durante el progreso de una transacción, no existe comunicación entre el coordinador y los participantes, excepto cuando los participantes informan al coordinador de que se unen a la transacción. La petición de un cliente de consumar (o abortar) una transacción se dirige al coordinador. Si el cliente solicita *abortaTransacción*, o si la transacción es abortada por los participantes, el coordinador informará a los participantes inmediatamente. Es cuando el cliente pide al coordinador que se complete la transacción cuando el protocolo de consumación en dos fases entra en acción.

En la primera fase del protocolo de consumación en dos fases el coordinador pregunta a todos los participantes si están preparados para consumar; y en el segundo, les dice que consumen (o aborten) la transacción. Si un participante puede consumar su parte de la transacción, dirá que está de acuerdo tan pronto como haya grabado los cambios y su estado en un dispositivo de almacenamiento permanente, y esté preparado para consumar. El coordinador en una transacción distribuida

---

puedeConsumar?(trans) → *Sí / No*
Llamada desde el coordinador al participante para preguntar si puede consumar una transacción.
El participante responde con su voto.

Consuma(trans)
Llamada desde el coordinador al participante para decirle que consume su parte de una transacción.

Aborta(trans)
Llamada desde el coordinador al participante para decirle que aborte su parte de una transacción.

heConsumado(trans, participante)
Llamada desde el participante al coordinador para confirmar que ha consumado la transacción.

dameDecisión(trans) → *Sí / No*
Llamada desde el participante al coordinador para preguntar por la decisión sobre una transacción tras haber votado *Sí* aunque no ha obtenido respuesta tras cierto tiempo. Se utiliza para recuperarse de la caída de un servidor o de mensajes con retraso.

Figura 13.4. Operaciones para el protocolo de consumación en dos fases.

Fase 1 (fase de votación):

1. El coordinador envía una petición *puedeConsumar?* a cada participante en la transacción.
2. Cuando un participante recibe una petición *puedeConsumar?*, responde al coordinador con su voto (*Sí* o *No*). Antes de votar *Sí*, se prepara para consumar, guardando los objetos en un dispositivo de almacenamiento permanente. Si el voto es *No*, el participante aborta inmediatamente.

Fase 2 (finalización en función del resultado de la votación)

3. El coordinador recoge los votos (incluyendo el propio).
   (a) Si no hay fallos y todos los votos son *Sí*, el coordinador decide consumar la transacción y envía peticiones de *Consuma* a cada uno de los participantes.
   (b) En otro caso, el coordinador decide abortar la transacción y envía peticiones *Aborta* a todos los participantes que votaron *Sí*.
4. Los participantes que han votado *Sí* están esperando por una petición *Consuma* o *Aborta* por parte del coordinador. Cuando un participante recibe uno de estos mensajes, actúa en función de ellos, y en el caso de *Consuma*, realiza una llamada de *heConsumado* como confirmación hacia el coordinador.

Figura 13.5. El protocolo de consumación en dos fases.

se comunica con los participantes para llevar a cabo el protocolo de consumación en dos fases mediante las operaciones que se resumen en la Figura 13.4. Los métodos *puedeConsumar?*, *Consuma* y *Aborta* son métodos de la interfaz del participante. Los métodos *heConsumado* y *dameDecisión* están en la interfaz del coordinador.

El protocolo de consumación en dos fases, como se muestra en la Figura 13.5, consta de una fase de votación y una fase de finalización. Al final de la etapa (2), el coordinador y todos los participantes que votaron *Sí* están preparados para consumar. Al final del paso (3), la transacción ha finalizado a todos los efectos. En el paso (3a) el coordinador y los participantes se han comprometido y, por lo tanto, el coordinador puede informar al cliente de una decisión de consumar. En el paso (3b) el coordinador informa al cliente de la decisión de abortar.

En el paso (4) los participantes confirman que ellos han consumado su parte, de tal forma que el coordinador sabe en qué momento deja de ser necesaria la información que almacenaba con respecto a la transacción.

Este protocolo, aparentemente directo, podría fallar debido a la caída de uno o más de los servidores o debido a un corte de la comunicación entre los servidores. Para cubrir la posibilidad de caídas, cada servidor guarda la información relativa al protocolo de consumación en dos fases en un dispositivo de almacenamiento permanente. Esta información puede ser recuperada por un nuevo proceso que se inicie para reemplazar al servidor caído. Los aspectos de la recuperación en las transacciones distribuidas se discuten en la Sección 13.6.

El intercambio de información entre el coordinador y los participantes puede fallar cuando uno de los servidores se cae o cuando se pierden mensajes. Para evitar que los procesos se queden bloqueados de forma indefinida se utilizan tiempos límite de espera (*timeout*). Cuando ocurre un timeout dentro de un proceso, éste debe emprender una acción apropiada. Para permitir este aspecto, el protocolo incluye una acción asociada a un tiempo límite para cada paso en el cual pueda bloquearse el proceso. Estas acciones se diseñan teniendo en cuenta el hecho de que en un sistema asíncrono exceder un tiempo límite no implica, necesariamente, que el proceso haya fallado.

◇ **Acciones frente a un timeout en el protocolo de consumación en dos fases.** Existen varias fases en el protocolo en las cuales el coordinador o un participante no pueden avanzar en su parte del protocolo hasta que reciban una petición o una respuesta por parte de los otros.

En primer lugar, considérese la situación en la que un participante ha votado *Sí* y está esperando para que el coordinador le informe del resultado de la votación, indicándole que consume o

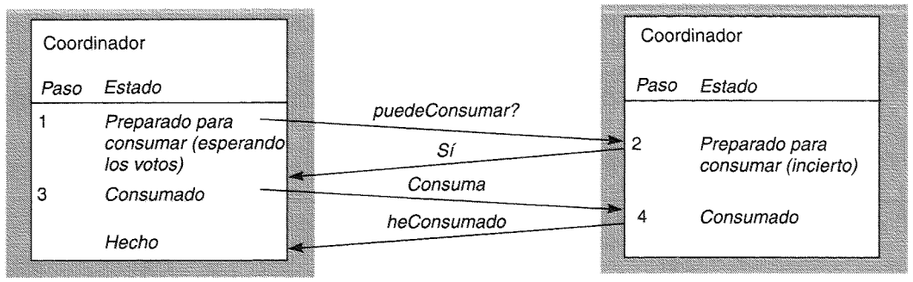


Figura 13.6. Comunicación en el protocolo de consumación en dos fases.

aborte la transacción. Véase el paso (2) en la Figura 13.6. Dicho participante está *incierto* con respecto al resultado y no puede seguir adelante hasta que obtenga el resultado de la votación por parte del coordinador. El participante no puede decidir unilateralmente qué hacer a continuación y, entretanto, los objetos utilizados por su transacción no pueden liberarse para ser utilizados por otras transacciones. El participante realiza al coordinador una petición de *dameDecisión* para determinar el resultado de la transacción. Cuando obtiene la respuesta continúa el protocolo en el paso (4) de la Figura 13.5. Si el coordinador ha fallado, el participante no será capaz de obtener una decisión hasta que el coordinador sea reemplazado, lo cual puede provocar grandes retrasos en los participantes que estén en estado incierto.

Hay otras estrategias alternativas disponibles para que los participantes obtengan una decisión cooperando entre ellos en lugar de contactar a un coordinador. Estas alternativas tienen la ventaja de que pueden utilizarse cuando ha fallado el coordinador. Para más detalles véase el Ejercicio 13.5 y el trabajo de Bernstein y otros [1987]. No obstante, incluso en un protocolo cooperativo, si todos los participantes están en estado *incierto*, serán incapaces de obtener una decisión hasta que esté disponible un coordinador o un participante con dicho conocimiento.

Otra situación en la que el participante puede sufrir retrasos es cuando han llevado a cabo todas las peticiones dentro de la transacción de su cliente, pero no ha recibido la llamada de *puedeConsumar?* por parte del coordinador. Ya que el cliente envía *cierraTransacción* al coordinador, el participante sólo puede detectar tal situación cuando se da cuenta de que no ha recibido ninguna petición asociada a una transacción durante un largo período de tiempo, por ejemplo mediante un timeout sobre un bloqueo. Dado que no se ha tomado ninguna decisión hasta ese momento, el participante puede decidir, unilateralmente, *abortar* tras cierto período de tiempo.

El coordinador puede sufrir retrasos cuando está esperando por los votos de los participantes. Dado que, en ese momento, no estaba decidido aún el destino de la transacción, tras cierto período de tiempo puede decidir abortar la transacción. En ese momento, debe anunciar su decisión de *Abortar* a todos los participantes que ya le habían enviado su voto. Algunos participantes retrasados podrían intentar votar *Sí* después de este hecho, pero se ignorarán sus votos y entrarán en el estado *incierto* que se describió antes.

◇ **Rendimiento del protocolo de consumación en dos fases.** Suponiendo que todo va bien, es decir, que no fallan ni el coordinador ni los participantes ni la comunicación entre ellos, el protocolo de consumación en dos fases que involucra a N participantes puede completarse con N mensajes de *puedeConsumar?* y sus respuestas, seguidos por N mensajes de *Consuma*. Esto es, el coste en mensajes es proporcional a $3N$, y el coste en tiempo es el de tres rondas de mensajes. En el coste estimado del protocolo no se cuentan los mensajes *heConsumado*, ya que el protocolo puede funcionar correctamente sin ellos (su papel es permitir a los servidores que borren la información obsoleta del coordinador).

En el peor de los casos, podría haber un número arbitrario de fallos en el servidor y las comunicaciones durante el protocolo de consumación en dos fases. Sin embargo, el protocolo está diseñado para tolerar una sucesión de fallos (caídas de servidores o pérdida de mensajes) y se garantiza que finaliza, de alguna forma, aunque no es posible especificar un límite de tiempo en el que se completa.

Como se apuntó en la sección de los tiempos límite, el protocolo de consumación en dos fases puede ocasionar retrasos considerables a los participantes en un estado *incierto*. Estos retrasos se dan cuando el coordinador ha fallado y no puede responder a las peticiones de *dameDecisión* por parte de los participantes. Incluso si un protocolo cooperativo permite a los participantes hacer peticiones de *dameDecisión* a otros participantes, ocurrirán retrasos si todos los participantes activos están en estado *incierto*.

Para rebajar estos retrasos se han diseñado protocolos de consumación en tres fases. Éstos son más costosos en cuanto al número de mensajes y al número de rondas necesarias para el caso normal (sin fallos). Para una descripción de los protocolos de consumación en tres fases, vea el Ejercicio 13.2 y Bernstein y otros [1987].

13.3.2. PROTOCOLO DE CONSUMACIÓN EN DOS FASES PARA TRANSACCIONES ANIDADAS

La transacción más externa en un conjunto de transacciones anidadas se denomina *transacción de nivel superior*. El resto de transacciones distintas a la de nivel superior se denominan *subtransacciones*. En la Figura 13.1(b), T es la transacción de nivel superior, T_1, T_2, T_{11}, T_{12}, T_{21} y T_{22} son subtransacciones. T_1 y T_2 son las transacciones hijas de T, a la que se refieren como madre. De forma similar, T_{11} y T_{12} son transacciones hijas de T_1, y T_{21} y T_{22} son transacciones hijas de T_2. Cada subtransacción comienza tras su madre y termina antes que ella. Así, por ejemplo, T_{11} y T_{12} comienzan antes que T_1 y terminan antes que ella.

Cuando finaliza una subtransacción, toma una decisión independiente sobre si consumarse de forma provisional o si abortar. Una consumación provisional no es lo mismo que estar preparado; simplemente, es una decisión local y no se guarda una copia en un dispositivo de almacenamiento permanente. Si, a continuación, el servidor se cae, su sustituto no será capaz de llegar a cabo la consumación provisional. Por esta razón, se requiere un protocolo de consumación en dos fases para transacciones anidadas, que permita a los servidores que han fallado, que aborten las transacciones que se hayan consumado provisionalmente.

Un coordinador para una subtransacción proporcionará una operación para abrir una subtransacción, junto con una operación que permita al coordinador de una subtransacción preguntar si su madre ya se ha consumado o ha abortado tal como se muestra en la Figura 13.7.

Un cliente comienza un conjunto de transacciones anidadas abriendo una transacción de nivel superior, mediante una operación *abreTransacción*. Dicha operación devuelve un identificador de transacción para la transacción de nivel superior. El cliente comienza una subtransacción invocando la operación *abreSubTransacción*, cuyo argumento especifica su transacción madre. La nueva

---

abreSubTransacción(trans) → subTrans
Abre una nueva subtransacción cuya madre es *trans*, y devuelve un identificador de subtransacción único.

dameEstado(trans) → consumada, abortada, provisional
Pide al coordinador que informe del estado de la transacción *trans*. Devuelve valores que representan uno de los siguientes: *consumada, abortada, provisional.*

Figura 13.7. Operaciones en el coordinador para las transacciones anidadas.

subtransacción automáticamente *se une* (unirse) a la transacción madre y se obtiene un identificador para la subtransacción.

El identificador para una subtransacción debe ser una extensión del TID de su madre y estar construido de tal forma que pueda obtenerse el identificador de la transacción madre o transacción de nivel superior a partir del identificador de la propia subtransacción. Además, todos los identificadores de las subtransacciones deberían ser globalmente únicos. El cliente consigue completar un conjunto de transacciones anidadas invocando los procedimientos *cierraTransacción* o *abortaTransacción* sobre el coordinador de la transacción de nivel superior.

Entretanto, cada una de las transacciones anidadas llevan a cabo sus operaciones. Cuando han terminado, el servidor que gestiona una subtransacción registra la información de si la subtransacción se ha consumado de forma provisional o ha abortado. Nótese que si es la madre quien aborta, entonces la subtransacción será forzada a abortar también.

Recuérdese del Capítulo 12 que una transacción madre (incluyendo a la transacción de nivel superior) puede consumarse incluso si una de sus transacciones hijas ha abortado. En dichos casos, la transacción madre estará programada para realizar diferentes acciones en función de si una subtransacción se ha consumado o ha abortado. Por ejemplo, considérese una transacción bancaria que se diseña para realizar todas las *órdenes de pago habituales* de una sucursal en un día concreto. Esta transacción se expresa mediante distintas subtransacciones *Transfiere* anidadas, cada una de las cuales consta de subtransacciones anidadas *deposita* y *extrae*. Presuponemos que cuando una cuenta está al descubierto, *extrae* abortará y su correspondiente *Transfiere* abortará. Sin embargo, no es necesario abortar todos los giros regulares porque una de las subtransacciones *Transfiere* haya abortado. En lugar de abortar, la transacción de nivel superior tomará nota de las *Transfiere* que han abortado y emprenderá las acciones oportunas.

Considérese la transacción de nivel superior T y las subtransacciones mostradas en la Figura 13.8, que está basada en la Figura 13.1(b). Cada subtransacción, bien se ha consumado provisionalmente, o bien ha abortado. Por ejemplo, T_{12} se ha consumado provisionalmente y T_{11} ha abortado, pero el destino de T_{12} depende de su madre T_1 y finalmente de la transacción de nivel superior T. Aunque T_{21} y T_{22} se han consumado provisionalmente, T_2 ha abortado y eso significa que T_{21} y T_{22} deben abortar también. Supóngase que T decide consumarse a pesar de que T_2 haya abortado, y que además T_1 decide consumarse a pesar de que T_{11} haya abortado.

Cuando se completa una transacción de nivel superior, su coordinador lleva a cabo un protocolo de consumación en dos fases. La única razón por la que una subtransacción participante no pueda completarse tras haberse consumado provisionalmente es que se haya caído. Recuerde que cuando se creó una subtransacción, ésta *se unió* a su transacción madre. Por lo tanto, el coordinador de cada transacción madre tiene una lista de sus subtransacciones hijas. Cuando una transacción anidada se consuma de forma provisional informa de su estado y del estado de sus descendientes a su madre. Cuando una transacción anidada aborta, simplemente informa de haber

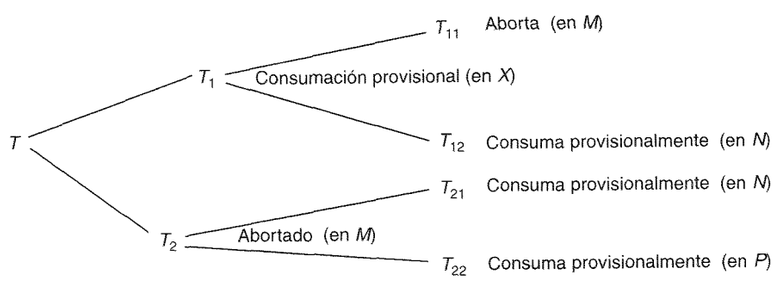


Figura 13.8. La transacción T decide si se consuma.

Coordinador de la transacción	*Transacciones hijas*	*Participante*	*Lista de consumaciones provisionales*	*Lista de abortadas*
T	T_1, T_2	Sí	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	Sí	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	No (abortada)		T_2
T_{11}		No (abortada)		T_{11}
T_{12}, T_{21}		T_{12} pero no T_{21}	T_{21}, T_{12}	
T_{22}		No (madre ha abortado)	T_{22}	

Figura 13.9. Información retenida por los coordinadores de transacciones anidadas.

abortado a su madre sin dar ninguna información acerca de sus descendientes. Al final, la transacción de nivel superior recibe una lista de todas las subtransacciones en el árbol, junto con el estado de cada una de ellas. Lo que realmente ocurre es que se eliminan de la lista los descendientes de las subtransacciones que han abortado.

En la Figura 13.9 se muestra la información que mantiene cada coordinador en el ejemplo de la Figura 13.8. Nótese que T_{12} y T_{21} comparten el mismo coordinador, ya que ambas se ejecutan en el servidor N. Cuando la subtransacción T_2 abortó, informó del hecho a su madre, T, pero no pasó información alguna acerca de sus subtransacciones T_{21} y T_{22}. Una subtransacción es *huérfana* si una de sus ascendientes ha abortado, bien de forma explícita, bien porque su coordinador se ha caído.

En nuestro ejemplo, las subtransacciones T_{21} y T_{22} son huérfanas porque sus madres abortaron sin pasar información acerca de ellos a la transacción de nivel superior. Su coordinador puede, sin embargo, realizar indagaciones acerca del estado de sus madres utilizando la operación *dameEstado*. Una subtransacción consumada provisionalmente dentro de una transacción abortada debería ser abortada, sin tener en cuenta si al final la transacción del nivel superior se consuma.

La transacción de nivel superior juega el papel de coordinador en el protocolo de consumación en dos fases, y la lista de participantes consta de los coordinadores de todas las subtransacciones en el árbol que se han consumado provisionalmente pero no tienen ascendientes que hayan abortado. Al llegar a esta fase, la lógica del programa ha determinado que la transacción de nivel superior debe intentar consumar lo que quede, a pesar de algunas subtransacciones abortadas. En la Figura 13.8, los coordinadores de T, T_1 y T_{12} son participantes y se les pedirá que voten para obtener el resultado. Si votan consumar, entonces deben *preparar* sus transacciones guardando el estado de los objetos en el dispositivo de almacenamiento permanente. Este estado se registra como perteneciente a la transacción de nivel superior de la cual formará parte. El protocolo de consumación en dos fases puede ejecutarse tanto de una forma jerárquica como de una forma plana.

La segunda fase del protocolo de consumación en dos fases es la misma que para el caso no anidado. El coordinador recoge los votos y después informa a los participantes en función del resultado. Cuando finalice, el coordinador y los participantes habrán consumado o abortado sus transacciones.

◇ **Protocolo de consumación en dos fases jerárquico.** En esta aproximación, el protocolo de consumación en dos fases se convierte en un protocolo anidado multi-nivel. El coordinador de la transacción de nivel superior se comunica con los coordinadores de las subtransacciones para las que es su madre inmediata. Envía mensajes *puedeConsumar?* a cada uno de estas últimas, las cuales, a su vez, los pasan a los coordinadores de sus transacciones hijas (y así sucesivamente en

puedeConsumar?(trans, subTrans) → *Sí / No*
Llama a un coordinador para que pregunte al coordinador de una subtransacción hija si puede consumar una subtransacción *subTrans*. El primer argumento *trans* es el identificador de la transacción del nivel superior. El participante responde con su voto *Sí / No*.

Figura 13.10. *puedeConsumar?* para el protocolo de consumación en dos fases jerárquico.

el árbol). Cada participante recoge las respuestas de sus descendientes antes de responder a sus madres. En nuestro ejemplo, T envía mensajes *puedeConsumar?* al coordinador de T_1 y luego T_1 envía mensajes *puedeConsumar?* a T_{12} preguntando acerca de los descendientes de T_1. El protocolo no incluye a los coordinadores de las transacciones tales como T_2, que ha abortado. La Figura 13.10 muestra los argumentos que son necesarios para *puedeConsumar?*. El primer argumento es el TID de la transacción de nivel superior, que será usado cuando se preparen los datos. El segundo argumento es el TID del participante que hace la llamada *puedeConsumar?*. El participante que recibe la llamada busca en su lista de transacciones cualquier transacción o subtransacción consumada provisionalmente que concuerde con el TID del segundo argumento. Por ejemplo, el coordinador de T_{12} es también el coordinador de T_{21}, ya que se ejecutan en el mismo servidor, pero cuando reciba la llamada de *puedeConsumar?* el segundo argumento será T_1 y tratará sólo con T_{12}.

Si un participante encuentra cualquier subtransacción que se ajuste al segundo argumento, prepara los objetos y responde con un voto *Sí*. Si no consigue encontrar alguna, entonces debe haberse caído desde que realizó la subtransacción y le responde con un voto *No*.

◇ **Protocolo de consumación en dos fases plano.** En esta aproximación, el coordinador de la transacción de nivel superior envía mensajes *puedeConsumar?* a los coordinadores de todas las subtransacciones en la lista de consumaciones provisionales. En nuestro ejemplo, a los coordinadores de T_1 y T_{12}. Durante el protocolo de consumación, los participantes se refieren a la transacción por su TID del nivel superior. Cada participante busca en su lista de transacciones cualquier transacción o subtransacción que se ajuste a ese TID. Por ejemplo, el coordinador de T_{12} es además el coordinador de T_{21}, ya que se ejecutan en el mismo servidor (N).

Desafortunadamente, esto no proporciona suficiente información para permitir a los participantes realizar acciones correctas, como en el caso del coordinador en el servidor N, que tiene una mezcla de subtransacciones consumadas provisionalmente o abortadas. Si al coordinador de N simplemente se le pide que consume T acabará consumando tanto T_{12} como T_{21}, porque, de acuerdo con su información local, ambos se han consumado provisionalmente. Esto es erróneo en el caso de T_{21}, ya que su madre, T_2, ha abortado. Para tener en cuenta tales casos, la operación *puedeConsumar?* en el protocolo de consumación plano tiene un segundo argumento que proporciona una lista de subtransacciones abortadas, como se muestra en la Figura 13.11. Un participante podrá consumar descendientes de la transacción del nivel superior a no ser que éstas tengan ascendientes abortados. Cuando un participante recibe una petición de *puedeConsumar?*, hace lo siguiente:

- Si el participante tiene algunas transacciones consumadas provisionalmente que sean descendientes de la transacción de nivel superior, *trans*:
  - — Comprueba que no tienen ascendientes abortados en la *listaAbortadas*. En ese caso puede prepararse para consumar (guardando la transacción y sus objetos en un dispositivo de almacenamiento permanente).
  - — Aquellas transacciones con ascendientes abortados, serán abortadas.
  - — Envía un voto *Sí* al coordinador.
- Si el participante no tiene transacciones consumadas provisionalmente que sean descendientes de la transacción de nivel superior, debe haber fallado desde que realizó la subtransacción, y envía un voto *No* al coordinador.

puedeConsumar?(trans, listaAbortadas) → *Sí / No*
Llamada desde el coordinador a un participante para preguntar si puede consumar una transacción. El participante responde con su voto *Sí / No*.

Figura 13.11. *puedeConsumar?* para el protocolo de consumación en dos fases plano.

◇ **Comparación de las dos aproximaciones.** El protocolo jerárquico tiene la ventaja de que, en cada etapa, el participante sólo necesita buscar subtransacciones de su madre inmediata, mientras que en el protocolo plano necesita disponer de la lista de transacciones abortadas para eliminar aquellas transacciones cuyos madres hubiesen abortado. Moss [1985] prefería el algoritmo plano porque permitía al coordinador de la transacción de nivel superior comunicarse directamente con todos los participantes, mientras que la variante jerárquica involucra el paso de una serie de mensajes hacia arriba y hacia abajo en el árbol en las distintas etapas.

◇ **Acciones asociadas a tiempos límite.** El protocolo de consumación en dos fases para el caso de transacciones anidadas puede ocasionar que se retrasen el coordinador o un participante en los mismos tres pasos que en la versión no anidada. Aparece un cuarto paso en el cual la subtransacción puede retrasarse. Considere las subtransacciones consumadas provisionalmente y que son hijas de subtransacciones abortadas: éstas puede que no sean informadas del resultado de la votación de la transacción. En nuestro ejemplo, T_{22} es una de estas transacciones, ésta se ha consumado provisionalmente, pero como su madre T_2 ha abortado, no se convierte en una participante. Para abordar estas situaciones, cualquier subtransacción que no haya recibido un mensaje *puedeConsumar?* indagará pasado un período de tiempo límite. La operación *dameEstado* de la Figura 13.7 permite que una subtransacción pregunte si su madre se ha completado o ha abortado. Para hacer posibles tales preguntas, los coordinadores de las subtransacciones abortadas han de sobrevivir un cierto período de tiempo. Si una subtransacción huérfana no puede contactar con su madre, al final, abortará.

13.4. CONTROL DE CONCURRENCIA EN TRANSACCIONES DISTRIBUIDAS

Cada servidor gestiona un conjunto de objetos y es responsable de asegurar que éstos mantienen su consistencia cuando se accede a ellos desde transacciones concurrentes. Por lo tanto, cada servidor es responsable de aplicar un control de concurrencia sobre sus propios objetos. Los miembros de una colección de servidores de transacciones distribuidas son responsables conjuntamente de asegurar que dichas transacciones se realizan de una forma secuencialmente equivalente.

Esto implica que si la transacción T se da antes que la transacción U en un acceso conflictivo a objetos en uno de los servidores, entonces han de estar en ese orden en todos los servidores a cuyos objetos se accede, de forma conflictiva, por T y U.

13.4.1. BLOQUEO

En una transacción distribuida, los bloqueos sobre un objeto se mantienen localmente (en el mismo servidor). El gestor de bloqueos locales, puede decidir la concesión de un bloqueo o hacer que la transacción espere. Sin embargo, no puede liberar ningún bloqueo hasta que sepa que la transacción se ha consumado o ha abortado en todos los servidores involucrados en la transacción. Cuando se utilizan bloqueos para el control de concurrencia, los objetos permanecen bloqueados y no están disponibles para otras transacciones durante el protocolo de consumación atómica, aunque una transacción abortada libere sus bloqueos tras la primera fase del protocolo.

Dado que los gestores de bloqueos en distintos servidores fijan sus bloqueos con independencia de los otros, es posible que distintos servidores impongan ordenamientos distintos sobre las transacciones. Considérese el siguiente solapamiento de las transacciones *T* y *U* en los servidores *X* e *Y*:

T	*U*
Escribe(A) en *X* bloquea *A*	
	Escribe(B) en *Y* bloquea *B*
Lee(B) en *Y* espera por *U*	
	Lee(A) en *X* espera por *T*

La transacción *T* bloquea al objeto *A* en el servidor *X* y, a continuación, la transacción *U* bloquea al objeto *B* en el servidor *Y*. Tras esto, *T* intenta acceder a *B* en el servidor *Y*, y espera dado el bloqueo de *U*. De forma similar, la transacción *U* intenta acceder a *A* en el servidor *X*, y tiene que esperar dado el bloqueo de *T*. Por lo tanto, tenemos *T* antes que *U* en un servidor y *U* antes que *T* en el otro. Estas ordenaciones distintas pueden llevar a ciclos de dependencias entre transacciones y surge una situación de interbloqueo distribuido. La detección y solución de interbloqueos distribuidos se discute en la siguiente sección de este capítulo. Cuando se detecta un interbloqueo, se aborta una transacción para resolverlo. En este caso, se informará al coordinador y se abortará la transacción en los participantes involucrados en dicha transacción.

13.4.2. CONTROL DE CONCURRENCIA CON ORDENACIÓN DE MARCAS TEMPORALES

En una transacción en un único servidor, el coordinador asigna una marca temporal única a cada transacción cuando ésta comienza. Se consigue la equivalencia secuencial consumando las versiones de los objetos en el orden de las marcas temporales de las transacciones que acceden a ellos. En las transacciones distribuidas, se requiere que cada coordinador genere marcas temporales que sean globalmente únicas. El primer coordinador al que accede la transacción le proporcionará al cliente una marca temporal para la transacción globalmente única. La marca temporal de la transacción se pasa al coordinador de cada servidor sobre cuyos objetos se realice una operación en la transacción.

Los servidores de transacciones distribuidas son responsables conjuntamente de asegurar que las transacciones se realicen de forma secuencialmente equivalente. Por ejemplo, si la versión de un objeto al que accede la transacción *U* se consuma después que la versión a la que accede *T* en otro servidor, entonces si *T* y *U* acceden al mismo objeto de la misma forma en otros servidores, deben consumarse tam bien en el mismo orden. Para conseguir la misma ordenación en todos los servidores, los coordinadores deben ponerse de acuerdo en lo referente a la ordenación de sus marcas temporales. Una marca temporal consta de una pareja *<marca temporal local, identificador del servidor>*. La ordenación acordada entre parejas de marcas temporales se basa en una comparación en la cual la parte del *identificador del servidor* es la parte menos significativa.

Se puede conseguir la misma ordenación de las transacciones en todos los servidores incluso si sus relojes locales no están sincronizados. Sin embargo, por razones de eficiencia, se requiere que las marcas temporales proporcionadas por un coordinador estén aproximadamente sincronizadas con aquellas que emiten otros coordinadores. Cuando ocurre esto, la ordenación de transacciones se corresponde, generalmente, con el orden en el que comienzan en el tiempo real. Las marcas temporales pueden mantenerse sincronizadas de una forma aproximada mediante la utilización de relojes físicos sincronizados localmente (véase el Capítulo 10).

Cuando se utiliza la ordenación de marcas temporales para el control de concurrencia, los conflictos se resuelven según se realiza cada operación. Si la resolución de un conflicto requiere que se aborte una transacción, se informará al coordinador y éste abortará la transacción en todos los participantes. Por lo tanto, toda transacción que le llegue al cliente con la petición de consumarse debería ser siempre capaz de consumarse. En consecuencia, normalmente en el protocolo de consumación en dos fases un participante llegará a un acuerdo para consumarse. La única situación en la cual un participante no llegará a un acuerdo para consumarse es que éste se haya caído durante la transacción.

13.4.3. CONTROL DE CONCURRENCIA OPTIMISTA

Recuérdese que con el control de concurrencia optimista, cada transacción se valida antes de que se le permita consumarse. Se asignan los números de transacción al comienzo de la validación y se establece la secuencia de las transacciones de acuerdo al orden de los números de transacciones. Una transacción distribuida es validada por una colección de servidores independientes, cada uno de los cuales valida las transacciones que acceden a sus propios objetos. La validación en todos los servidores tiene lugar durante la primera fase del protocolo de consumación en dos fases.

Considérense las siguientes transacciones entrelazadas *T* y *U*, que acceden a los objetos *A* y *B* en los servidores *X* e *Y*, respectivamente.

T	*U*
Lee(A) en *X*	*Lee(B)* en *Y*
Escribe(A)	*Escribe(B)*
Lee(B) en *Y*	*Lee(A)* en *X*
Escribe(B)	*Escribe(A)*

Las transacciones acceden a los objetos en el orden «*T* antes que *U*» en el servidor *X* y en el orden «*U* antes que *T*» en el servidor *Y*. Ahora supóngase que *T* y *U* comienzan la validación aproximadamente al mismo tiempo, pero el servidor *X* valida a *T* primero y el servidor *Y* valida a *U* primero. Recuérdese que la Sección 12.5 recomendaba una simplificación del protocolo de validación que establece una regla por la que sólo una transacción puede realizar, cada vez, las fases de validación y actualización. Por lo tanto, cada servidor será incapaz de validar la otra transacción hasta que la primera se haya completado. Éste es un ejemplo de interbloqueo por consumación.

Las reglas de validación de la Sección 12.5 asumía que la validación es rápida, lo cual es cierto para transacciones sobre un único servidor. Sin embargo, en una transacción distribuida, el protocolo de compromiso en dos fases puede necesitar algún tiempo y retrasar que otras transacciones entren a la validación hasta que se haya obtenido una decisión sobre la transacción actual. En el caso de transacciones distribuidas optimistas, cada servidor aplica en paralelo un protocolo de validación. Ésta es una extensión de la validación hacia delante o hacia atrás, para permitir que varias transacciones estén al mismo tiempo en la fase de validación. En esta extensión, deben comprobarse tanto la regla 3 como la regla 2 para la validación hacia atrás. Esto es, debe comprobarse el conjunto de escritura de una transacción que está siendo validada para detectar alguna superposición con el conjunto de escritura de transacciones anteriores con las que se solapen. En el trabajo de Kung y Robinson [1991] se describe la validación en paralelo.

Si se utiliza la validación en paralelo, las transacciones no sufrirán de interbloqueo por consumación. Sin embargo, si los servidores sencillamente realizan validaciones independientes, es posi-

ble que distintos servidores de una transacción distribuida puedan secuenciar el mismo conjunto de transacciones según ordenamientos diferentes; en nuestro ejemplo, podría darse *T* antes que *U* en el servidor *X* y *U* antes que *T* en el servidor *Y*.

Los servidores de transacciones distribuidas deben evitar que esto suceda. Una aproximación sería que tras cada validación local en un servidor, tuviese lugar una validación global [Ceri y Owicki 1982]. La validación global comprueba que la combinación de las ordenaciones en los servidores sea secuenciable; esto es, que la transacción que se está validando no se vea involucrada en un ciclo.

Otra aproximación consiste en que todos los servidores de una transacción concreta utilicen, al comienzo de la validación, el mismo número de transacción único globalmente [Schlageter 1982]. El coordinador del protocolo de consumación en dos fases es el responsable de generar números de transacción globalmente únicos y pasárselos a los participantes en los mensajes *puedeConsumar?*. Ya que distintos servidores pueden coordinar distintas transacciones, los servidores deben tener (como en el caso del protocolo distribuido de ordenación de marcas temporales) un orden acordado para los números de transacción que generan.

Agrawal y otros [1987] han propuesto una variación del algoritmo de Kung y Robinson que favorece las transacciones de *sólo lectura*, junto con un algoritmo que se denomina MVGV (validación generalizada multi-version (*multi-version generalized validation*). MVGV es una forma de validación en paralelo que asegura que los números de transacción reflejan un orden secuencial, pero requiere que la visibilidad de algunas transacciones se retrase una vez que se han consumado. Además, permite que se cambie el número de transacción, de tal forma que permita validar algunas transacciones que de otra forma habrían fallado. El citado trabajo también propone un algoritmo para la consumación de transacciones distribuidas. Es parecida a la propuesta de Schlageter, en cuanto que ha de encontrarse un número de transacción global. Al final de la fase de lectura, el coordinador propone un valor para dicho número de transacción global, y cada participante intenta validar sus transacciones locales utilizando ese número. No obstante, si el número propuesto de transacción global es demasiado pequeño, algunos participantes pueden no ser capaces de validar sus transacciones y negociarán con el coordinador para que incremente el número. Si no puede encontrarse un número apropiado, entonces esos participantes tendrán que abortar su transacción. Al final, si todos los participantes pueden validar sus transacciones, el coordinador habrá recibido propuestas para los números de transacción por parte de cada uno de ellos. Si pueden encontrarse números comunes, entonces la transacción será consumada.

13.5. INTERBLOQUEOS DISTRIBUIDOS

La discusión sobre interbloqueos en la Sección 12.4 mostró que pueden surgir interbloqueos dentro de un único servidor cuando se utilizan bloqueos para el control de concurrencia. Los servidores deben bien evitarlos, o bien detectarlos y resolverlos. La utilización de tiempos límite para resolver posibles interbloqueos es una aproximación tosca, ya que es difícil elegir un intervalo de tiempo límite apropiado y las transacciones serán abortadas de forma innecesaria. Con los esquemas de detección de interbloqueos, se aborta una transacción sólo cuando se ha visto involucrada en un interbloqueo. La mayoría de los esquemas de detección de interbloqueos funcionan detectando ciclos en los grafos *espera por* de las transacciones. En teoría, en un sistema distribuido en el que están involucrados múltiples servidores a los que están accediendo múltiples transacciones, puede construirse un grafo *esperar por* global a partir de los grafos *espera por* locales. Puede haber un ciclo en un grafo *espera por* global que no esté en ninguno de los grafos locales: es decir, puede existir un *interbloqueo distribuido*. Recuérdese que un grafo *espera por* es un grafo dirigido en el cual los nodos representan transacciones u objetos, y los arcos representan un objeto retenido por

U		*V*		*W*	
d.deposita(10)	bloquea *D*				
		b.deposita(10)	bloquea *B* en *Y*		
a.deposita(20)	bloquea *A* en *X*				
				c.deposita(30)	bloquea *C* en *Z*
b.extrae(30)	espera en *Y*				
		c.extrae(20)	espera en *Z*		
				a.extrae(20)	espera en *X*

Figura 13.12. Entrelazamientos de las transacciones *U, V* y *W*.

una transacción o una transacción esperando por un objeto. Existe un interbloqueo si y sólo si hay un ciclo en el grafo *espera por*.

La Figura 13.12 muestra el entrelazado de las transacciones *U*, *V* y *W* que involucran a los objetos *A* y *B* gestionados por los servidores *X* e *Y*, y los objetos *C* y *D* gestionados por el servidor *Z*.

En la Figura 13.13(a) el grafo *espera por* completo muestra que un ciclo asociado a un interbloqueo está formado por arcos alternados, que representan a una transacción esperando por un objeto y un objeto retenido por una transacción. Ya que cualquier transacción puede estar esperando sólo por un objeto cada vez, los objetos pueden dejarse fuera de los grafos *espera por*, como se muestra en la Figura 13.13(b).

La detección de un interbloqueo distribuido requiere encontrar un ciclo en el grafo *espera por* global de la transacción, que está distribuido entre los servidores que estaban involucrados en las transacciones. Los gestores de bloqueos en cada servidor pueden construir los grafos *espera por* locales, como se comentó en el Capítulo 12. En el ejemplo anterior, los grafos *espera por* locales de los servidores son:

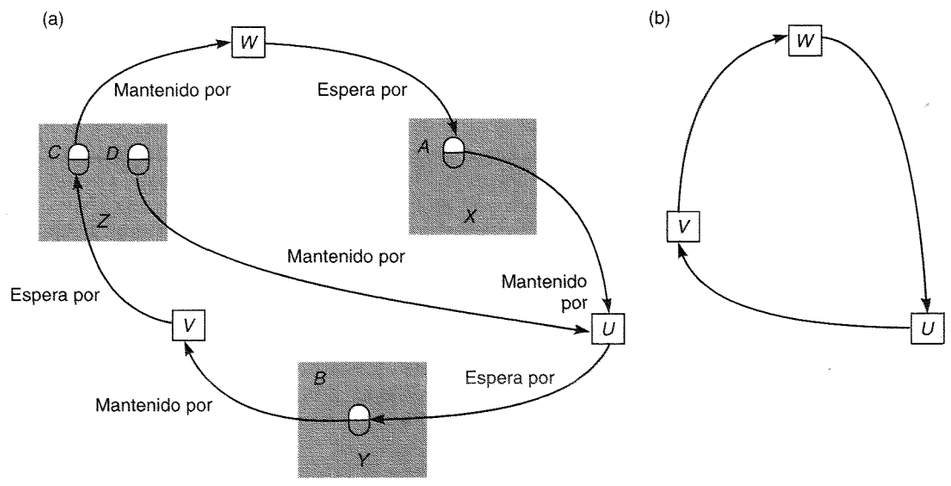


Figura 13.13. Interbloqueo distribuido.

servidor Y: $U \rightarrow V$ (se añade cuando U solicita *b.extrae(30)*)
servidor Z: $V \rightarrow W$ (se añade cuando V solicita *c.extrae(20)*)
servidor X: $W \rightarrow U$ (se añade cuando W solicita *a.extrae(20)*)

Ya que cada uno de los distintos servidores involucrados contiene una parte del grafo *espera por* global, es preciso comunicar entre servidores para encontrar ciclos en el grafo.

Una solución sencilla consiste en utilizar una detección de interbloqueos centralizada, en la cual un servidor asume el papel de detector de interbloqueos global. De vez en cuando, cada servidor envía la última copia de su grafo *espera por* local al detector de interbloqueos global, que aglutina la información de los grafos locales para construir un grafo *espera por* global. El detector de interbloqueos global comprueba la existencia de ciclos en el grafo *espera por* global. Cuando encuentra un ciclo, toma una decisión sobre cómo resolver el interbloqueo e informa a los servidores cómo han de abortarse las transacciones para resolverlo.

La detección de interbloqueos centralizados no es una buena idea porque depende de un solo servidor para llevarla a cabo. Adolece de los problemas habituales asociados con las soluciones centralizadas en sistemas distribuidos: escasa disponibilidad, falta de tolerancia a fallo e incapacidad de aumentar de tamaño. Además, el coste asociado a la transmisión frecuente de los grafos *espera por* locales es alto. Si el grafo global se construye con menos frecuencia, puede llevar más tiempo detectar los interbloqueos.

◇ **Interbloqueos fantasmas.** Un interbloqueo que se «detecta» pero que realmente no lo es se conoce como interbloqueo fantasma. En la detección de interbloqueos distribuidos, la información acerca de las relaciones *espera por* entre las transacciones se transmite de un servidor a otro. Si existe un interbloqueo, la información necesaria se recogerá, al final, en algún lugar y se detectará el ciclo. Ya que este procedimiento tarda un cierto tiempo, existe la posibilidad de que, entretanto, una de las transacciones que mantiene un bloqueo se haya liberado, en cuyo caso el bloqueo ya no existe.

Considérese el caso de un detector de interbloqueo global, que recibe los grafos *espera por* locales de los servidores X e Y, como se muestra en la Figura 13.14. Supóngase que la transacción U a continuación libera un objeto en el servidor X y solicita el que mantiene V en el servidor Y. Supóngase además que el detector global recibe el grafo local del servidor Y antes que el del servidor X. En este caso, detectaría un ciclo $T \rightarrow U \rightarrow V \rightarrow T$, aunque el arco $T \rightarrow U$ ya no existe. Éste es un ejemplo de interbloqueo fantasma.

El lector observador se habrá dado cuenta de que si las transacciones están usando bloqueos en dos fases, no pueden liberar objetos y a continuación obtener más objetos, y los ciclos asociados a interbloqueos fantasmas no pueden ocurrir de la forma en que se ha sugerido antes. Considere la situación en la cual se detecta un ciclo $T \rightarrow U \rightarrow V \rightarrow T$: o esto representa un interbloqueo, o cada una de las transacciones T, U y V deben, al final, consumarse. Es realidad, es imposible consumar cualquiera de ellas, ya que cada una está esperando por un objeto que nunca va a ser liberado.

Sin embargo, podría detectarse un interbloqueo fantasma si durante el procedimiento de detección de interbloqueos aborta una transacción en un ciclo de interbloqueo. Por ejemplo, si existe un

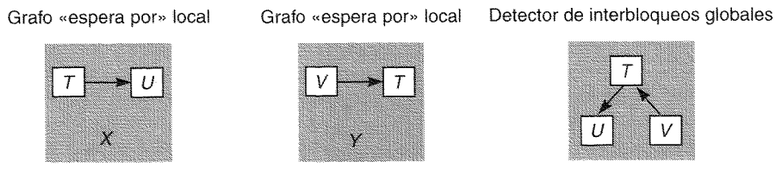


Figura 13.14. Grafos *espera por* locales y globales.

ciclo $T \rightarrow U \rightarrow V \rightarrow T$ y U aborta una vez que ha sido recogida la información relativa a U, entonces roto ya el ciclo no hay interbloqueo.

◇ **Caza de arcos.** Una aproximación distribuida para la detección de interbloqueos utiliza una técnica denominada captura de los arcos (*edge chasing*) o empuje de caminos (*path pushing*). En esta aproximación, el grafo no se construye *espera por* global, sino que cada uno de los servidores implicados posee información sobre algunos de sus arcos. Los servidores intentan encontrar ciclos mediante el envío de mensajes denominados *sondas*, que siguen los arcos de un grafo a través del sistema distribuido. Un mensaje sonda está formado por las relaciones *espera por* de transacciones que representan un camino en el grafo *espera por* global.

La pregunta es: ¿cuándo debería un servidor enviar una sonda? Considérese la situación en el servidor X en la Figura 13.13. Este servidor sólo ha añadido el arco $W \rightarrow U$ a su grafo *espera por* local y en ese momento, la transacción U está esperando para acceder al objeto B, que la transacción V retiene en el servidor Y. Este arco podría ser, posiblemente, parte de un ciclo tal como $V \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow W \rightarrow U \rightarrow V$ involucrando transacciones que utilizan objetos en otros servidores. Esto indica que hay un potencial ciclo asociado a un interbloqueo distribuido, que podría encontrarse enviando una sonda al servidor Y.

Ahora considérese la situación un poco antes, cuando el servidor Z añadió el arco $V \rightarrow W$ a su grafo local: en ese momento, W no está esperando. Por lo tanto, no sería necesario mandar la sonda.

Cada transacción distribuida comienza en un servidor (denominado el coordinador de la transacción) y se mueve a otros servidores distintos (denominados participantes en la transacción), que pueden comunicarse con el coordinador. En cualquier instante de tiempo, una transacción puede estar, bien activa, bien esperando en sólo uno de estos servidores. El coordinador es responsable de registrar si la transacción está activa o está esperando por un objeto concreto, y los participantes pueden obtener esta información desde de su coordinador. Los gestores de bloqueos informan a los coordinadores cuando las transacciones comienzan a esperar por objetos, y cuando las transacciones adquieren dichos objetos y vuelven a estar activas. Cuando se aborta una transacción para romper un interbloqueo, su coordinador informará a los participantes y se eliminarán todos sus bloqueos, con el efecto de que todos los arcos involucrados en la transacción se eliminarán de los grafos *espera por* locales.

Los algoritmos de captura de arcos tienen tres pasos: iniciación, detección y resolución.

Iniciación: cuando un servidor percibe que una transacción T comienza a buscar a otra transacción U, y ésta a su vez está esperando para acceder a un objeto en otro servidor, iniciará la detección enviando una sonda que contiene el arco $<T \rightarrow U>$ al servidor del objeto por el cual está bloqueada la transacción U. Si U está compartiendo un bloqueo, se envían sondas a todos los que poseen dicho bloqueo. Algunas veces otras transacciones pueden comenzar a compartir el bloqueo más tarde, en cuyo caso también deben enviárseles sondas a ellas.

Detección: la detección consiste en recibir sondas y decidir si se ha producido un interbloqueo o si hay que enviar nuevas sondas.

Por ejemplo, cuando un servidor de un objeto recibe la sonda $<T \rightarrow U>$ (indicando que T está esperando por la transacción U, que retiene un objeto local) comprueba si U está también esperando. Si es así, la transacción por la que espera (por ejemplo, V) se añade a la sonda (consiguiendo $<T \rightarrow U \rightarrow V>$), y si la nueva transacción (V) está esperando por otro objeto en otro sitio, se vuelve a enviar la sonda.

De esta forma, se construye un camino a través del grafo *espera-por* global añadiendo un eje cada vez. Antes de volver a enviar una sonda, el servidor comprueba si la transacción que acaba de ser añadida (por ejemplo, T) ha ocasionado un ciclo en la sonda (por ejemplo $<T \rightarrow U \rightarrow V \rightarrow T>$). Si es éste el caso, ha encontrado un ciclo en el grafo y se ha detectado un interbloqueo.

Resolución: cuando se detecta un ciclo, se aborta una transacción en el ciclo para romper el interbloqueo.

En nuestro ejemplo, los siguientes pasos describen cómo se inicia la detección de interbloqueos y cómo se envían las sondas durante la correspondiente fase de detección.

- El servidor X inicia la detección enviando la sonda $<W \rightarrow U>$ al servidor de B (Servidor Y).
- El servidor Y recibe la sonda $<W \rightarrow U>$, observa que B es retenida por V y concatena V a la sonda para producir $<W \rightarrow U \rightarrow V>$. Observa que V está esperando por C en el servidor Z. La sonda se envía al servidor Z.
- El servidor Z recibe la sonda $<W \rightarrow U \rightarrow V>$ y observa que C es retenida por W y concatena W a la sonda para producir $<W \rightarrow U \rightarrow V \rightarrow W>$.

Este camino contiene un ciclo. El servidor detecta un interbloqueo. Para romper el interbloqueo debe abortarse una de las transacciones en el ciclo. La transacción a abortar puede elegirse de acuerdo a las prioridades de las transacciones, tema que se tratará en breve.

La Figura 13.15 muestra el progreso de los mensajes sonda desde la iniciación por parte del servidor de A hasta la detección del interbloqueo por parte del servidor de C. Las sondas se muestran como flechas remarcadas, los objetos como círculos y los coordinadores de las transacciones como rectángulos. Cada sonda se muestra yendo directamente de un objeto hasta otro. En realidad, antes de que un servidor transmita una sonda a otro servidor, consulta al coordinador de la última transacción en el camino para averiguar si este último está esperando por otro objeto en otra parte. Por ejemplo, antes de que el servidor de B transmita la sonda $<W \rightarrow U \rightarrow V>$ consulta al coordinador de V para averiguar si V está esperando por C. En la mayoría de los algoritmos de captura de arcos, los servidores de los objetos envían las sondas a los coordinadores de las transacciones, que pueden, a su vez, enviarlos (si la transacción está esperando) al servidor del objeto por el cual está esperando la transacción. En nuestro ejemplo, el servidor de B transmite la sonda $<W \rightarrow U \rightarrow V>$ al coordinador de V, que a su vez la envía al servidor de C. Esto muestra que para reenviar una sonda, se necesitan dos mensajes.

El algoritmo que se acaba de describir debería encontrar cualquier interbloqueo que pueda darse, dado que las transacciones que están esperando no abortan y que no hay fallos tales como pérdida de mensajes o caída de servidores. Para entender esto, considere un ciclo asociado a un

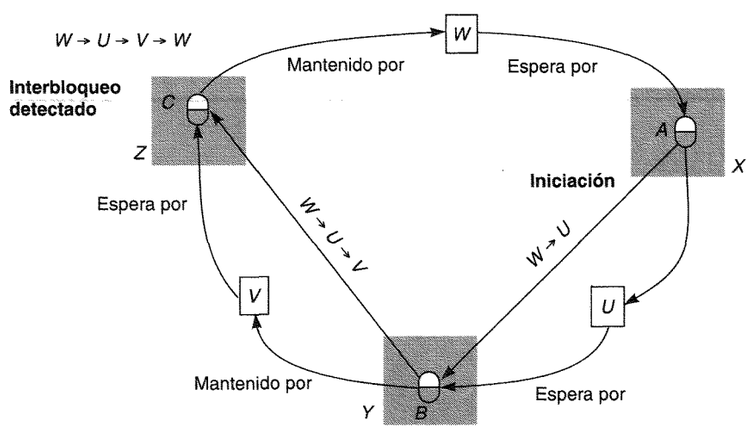


Figura 13.15. Sondas transmitidas para detectar interbloqueos.

interbloqueo en el cual la última transacción, W, comienza a esperar y se completa el ciclo. Cuando W comienza a esperar por un objeto el servidor lanza una sonda que va al servidor del objeto retenido por cada transacción por la que está esperando W. Los receptores amplían y reenvían las sondas a los servidores de los objetos solicitados por todas las transacciones en espera que pueden encontrar. De esta forma, cada transacción por la cual W está esperando, directa o indirectamente, se añadirá a la sonda, a no ser que se detecte un interbloqueo. Cuando hay un interbloqueo, W está esperando, de forma indirecta, por sí misma. Por lo tanto, la sonda volverá al objeto que retiene W.

Podría parecer que se manda un gran número de mensajes para detectar un interbloqueo. En el ejemplo de arriba, se observa que se usan dos mensajes sonda para detectar un ciclo que involucra a tres transacciones. Cada unos de los mensajes sonda consta, generalmente, de dos mensajes (del objeto al coordinador y del coordinador al objeto).

Una sonda que detecte un ciclo que involucra a N transacciones será enviado por parte de $(N - 1)$ coordinadores de transacciones a través de $(N - 1)$ servidores de los objetos, lo que requiere $2(N - 1)$ mensajes. Afortunadamente, la mayoría de los interbloqueos de deben a ciclos que contienen sólo dos transacciones, y no hay necesidad de preocuparse innecesariamente acerca del número de mensajes involucrados. Esta observación se ha obtenido partiendo de estudios en bases de datos. También se puede deducir de las probabilidades de accesos conflictivos a objetos. Véase Bernstein y otros [1987].

◇ **Prioridades de las transacciones.** En el algoritmo descrito anteriormente, cada transacción involucrada en un ciclo de un interbloqueo puede causar el inicio de una detección de un interbloqueo. El efecto asociado a que varias transacciones en un ciclo inicien la detección de interbloqueos es que pueden darse detecciones en distintos servidores del ciclo, con el resultado de que se aborte más de una transacción del ciclo.

En la Figura 13.16(a) se consideran las transacciones T, U, V y W, donde U está esperando por W y V está esperando por T. Aproximadamente al mismo tiempo, T solicita el objeto retenido por U y W solicita el objeto retenido por V. Se inician dos sondas por separado, $<T \rightarrow U>$ y $<W \rightarrow V>$, por parte de los servidores de esos objetos y están circulando hasta que dos servidores distintos detectan un interbloqueo. Véase la Figura 13.16(b), donde el ciclo es $<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$, y el apartado (c), donde el ciclo detectado es $<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$.

Para asegurarse de que sólo será abortada una de las transacciones en el ciclo, se asignan *prioridades* a las transacciones, de tal forma que estén totalmente ordenadas. Pueden utilizarse, por ejemplo, las marcas temporales como prioridades. Cuando se encuentra un ciclo asociado a un interbloqueo se aborta la transacción con la prioridad más baja. Incluso si varios servidores distintos detectan el mismo ciclo, todos tomarán la misma decisión sobre cuál debe ser la transacción a

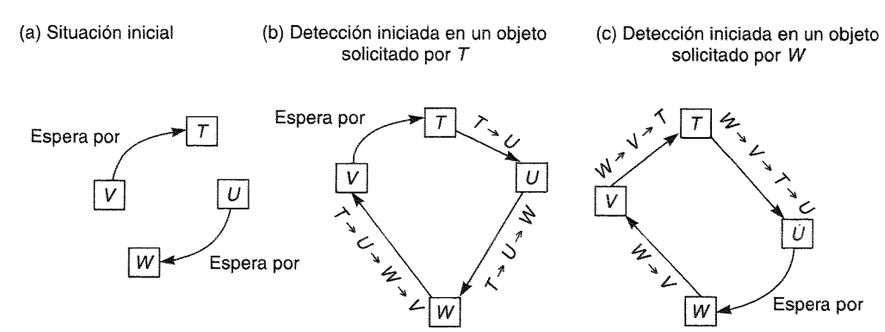


Figura 13.16. Dos sondas iniciadas.

abortar. Se escribirá $T > U$ para indicar que T tiene una prioridad mayor que U. En el ejemplo anterior, se asume que $T > U > V > W$. Por lo tanto, será abortada la transacción W cuando se detecten cualesquiera de los dos ciclos: $<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$ o $<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$.

Podría parecer que las prioridades entre transacciones podrían usarse también para reducir el número de situaciones que originan que se inicie la detección de un interbloqueo, utilizando la regla de que la detección sea iniciada sólo cuando una transacción de mayor prioridad comienza a esperar por una de prioridad más baja. En nuestro ejemplo en la Figura 13.16, dado que $T > U$ sería enviada la sonda $<T \rightarrow U>$, pero como $W < V$ no se enviaría la sonda $<W \rightarrow V>$. Si se asume que cuando una transacción comienza a esperar por otra transacción es igualmente probable que ésta tenga una mayor o menor prioridad que la transacción por la que se espera, entonces es probable que la utilización de dicha regla reduzca el número de mensajes sonda a aproximadamente la mitad.

Además, las prioridades entre transacciones podrían usarse para reducir el número de sondas que se vuelven a enviar. La idea general es que las sondas deberían viajar *cuesta abajo*, esto es, desde las transacciones con mayor prioridad hacia las transacciones con prioridades menores. Para hacer esto, los servidores utilizan la regla de que ellos no envían una sonda a quien esté reteniendo un objeto y que tenga una prioridad mayor que quien la ha iniciado. El argumento para hacer esto es que si dicho retenedor está esperando por otra transacción, entonces debe haber iniciado una detección enviando una sonda en el momento en que comenzó a esperar.

Sin embargo, existe un peligro asociado a estas mejoras aparentes. En nuestro ejemplo en la Figura 13.15, las transacciones U, V y W se ejecutan en el orden en el cual U está esperando por V, y V ya está esperando por W cuando W comienza a esperar por U. Sin las reglas de prioridad, la detección se inicia cuando W comienza a esperar y envía la sonda $<W \rightarrow U>$. Utilizando la regla de prioridad, no se enviará esta sonda, ya que $W < U$ y el interbloqueo no se detectarán.

El problema reside en que el orden en el que las transacciones empiezan a esperar puede determinar si se detectará o no un interbloqueo. El peligro mencionado con anterioridad puede evitarse utilizando un esquema en el cual los coordinadores guardan copias de todas las sondas recibidas en representación de cada transacción en una *cola de sondas*. Cuando una transacción comienza a esperar por un objeto, envía las sondas en su cola al servidor que tiene dicho objeto, el cual las propaga por rutas *cuesta abajo*.

En nuestro ejemplo de la Figura 13.15, cuando U comienza a esperar por V, el coordinador de V guardará la sonda $<U \rightarrow V>$. Véase la Figura 13.17(a). A continuación, cuando V comienza a esperar por W, el coordinador de W almacenará $<V \rightarrow W>$ y V enviará su cola de sondas $<U \rightarrow V>$ hacia W. Véase la Figura 13.17(b), en la cual la cola de sondas de W tiene $<U \rightarrow V>$ y $<V \rightarrow W>$. Cuando W comienza a esperar por A, enviará su cola de sondas $<U \rightarrow V \rightarrow W>$ al servidor de A, que además observa la nueva dependencia $W \rightarrow U$ y la combina con la información en la sonda recibida, lo que da lugar a determinar $U \rightarrow V \rightarrow W \rightarrow U$. Se ha detectado un interbloqueo.

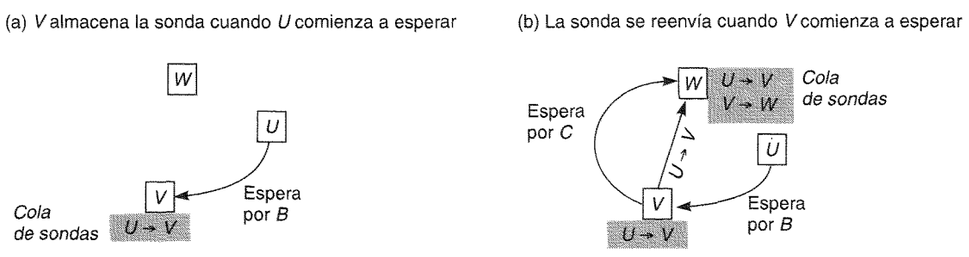


Figura 13.17. Sondas que viajan cuesta abajo.

Cuando un algoritmo requiere que se almacenen las sondas en colas de sondas, también requiere convenios para pasar las sondas a los nuevos retenedores y para descartar las sondas que se refieren a transacciones que ya se han consumado o que han abortado. Si se descartan sondas relevantes, podrían perderse detecciones de interbloqueos, y si se retienen sondas desfasadas, podrían detectarse interbloqueos falsos. Esto aumenta la complejidad de cualquier algoritmo de captura de arcos. Los lectores que estén interesados en los detalles de tales algoritmos deberían estudiar los trabajos de Sinha y Natarajan [1985] y de Choudhary y otros [1989], que presentan algoritmos para su uso con bloqueos exclusivos. Pero debe observarse que Choudhary y otros mostraron que el algoritmo de Sinha y Natarajan es incorrecto y no detecta todos los interbloqueos, y puede, incluso, informar de falsos interbloqueos. Kshemkalyani y Singhal [1991] corrigieron el algoritmo de Choudhary y otros (que no detecta todos los interbloqueos y puede informar de falsos interbloqueos) y proporcionaron una prueba de la corrección para el algoritmo corregido. En un trabajo posterior, Kshemkalyani y Singhal [1994] discuten que los interbloqueos distribuidos no están completamente comprendidos porque en un sistema distribuido no existen ni un estado ni un tiempo globales. De hecho, cualquier ciclo que se haya encontrado puede contener secciones que hayan sido grabadas en instantes de tiempo distintos. Además, los sitios pueden oír hablar de los interbloqueos, pero pueden no escuchar que han sido resueltos hasta que hayan transcurrido retrasos aleatorios. El trabajo describe los interbloqueos distribuidos en términos de los contenidos de la memoria distribuida, utilizando relaciones causales entre sucesos en sitios distintos.

13.6. RECUPERACIÓN DE TRANSACCIONES

La propiedad de atomicidad de las transacciones requiere que se reflejen en los objetos a los que han accedido los efectos de todas las transacciones que se han consumado y ninguno de los efectos de las transacciones incompletas o abortadas. Esta propiedad puede describirse en términos de dos aspectos: durabilidad o persistencia (*durability*) y atomicidad frente a fallos (*failure atomicity*). La durabilidad requiere que los objetos se guarden en dispositivos de almacenamiento permanentes y que, después, estén disponibles allí de forma indefinida. Por lo tanto, un acuse de recibo asociado a una petición de consumación por un cliente implica que todos los efectos de la transacción han sido registrados tanto en almacenamientos permanentes como en los objetos (volátiles) del servidor. La atomicidad ante fallos requiere que los efectos de las transacciones sean atómicas, incluso cuando el servidor se cae. La recuperación se ocupa de asegurar que los objetos de un servidor sean perdurables y que el servicio proporciona atomicidad ante fallos.

Aunque los servidores de archivos y los servidores de bases de datos mantienen los datos en dispositivos de almacenamiento permanente, otros tipos de servidores de objetos recuperables no necesitan hacerlo, excepto por motivos de recuperación. En este capítulo, se asume que cuando un servidor está en funcionamiento, mantiene todos sus objetos en la memoria volátil y registra sus objetos consumados en un *archivo* o *archivos de recuperación*. Por lo tanto, la recuperación consiste en restaurar el servidor a partir de los dispositivos de almacenamiento permanente y dejarlo con las últimas versiones consumadas de los objetos. Las bases de datos necesitan tratar con grandes volúmenes de datos. Éstas, generalmente, mantienen los objetos en dispositivos de almacenamiento estable en un disco, con una memoria caché en la memoria volátil.

Los dos requisitos de persistencia y de atomicidad ante fallos no son realmente independientes uno de otro, y pueden tratarse mediante un único mecanismo, el *gestor de recuperación*. Las tareas de un gestor de recuperación son:

- Guardar los objetos de todas las transacciones consumadas en dispositivos de almacenamiento permanente (en un archivo de recuperación).
- Restaurar los objetos del servidor tras una caída.

- Reorganizar el archivo de recuperación para mejorar el rendimiento de la recuperación.
- Reclamar espacio de almacenamiento (en el archivo de recuperación).

En algunos casos, se requerirá que el gestor de recuperación sea resistente a los fallos en los medios, como fallos en su archivo de recuperación tales que se perdieran algunos datos del disco, bien porque se hayan corrompido durante la caída, por degeneración aleatoria o por fallos permanentes. En tales casos, necesitamos otra copia del archivo de recuperación. Ésta puede estar en un dispositivo de almacenamiento estable, que se implementa para que sea muy improbable que falle, mediante discos en espejo (*mirror*) o copias en diferentes ubicaciones.

◇ **Lista de intenciones.** Cualquier servidor que proporcione transacciones necesita mantener el contacto con los objetos a los que acceden las transacciones de los clientes. Recuerde del Capítulo 12 que cuando un cliente inicia una transacción, el servidor con el que contacta en primer lugar genera un nuevo identificador de transacción y se lo devuelve al cliente. Cada solicitud posterior del cliente dentro de una transacción, incluyendo las peticiones *consuma* o *aborta*, incluirá como argumento el identificador de transacción. Durante el progreso de una transacción, las operaciones de actualización se aplican sobre un conjunto privado de versiones tentativas de los objetos que pertenecen a la transacción.

En cada servidor se guarda una *lista de intenciones* para cada una de sus transacciones actualmente activas (una lista de intenciones de una transacción en particular contiene una lista de las referencias y los valores de los objetos que son alterados durante la transacción). Cuando una transacción se consuma, se utiliza la lista de intenciones de esa transacción para identificar los objetos afectados. La versión consumada de cada objeto es reemplazada por la versión tentativa obtenida mediante la transacción, y se escribe el nuevo valor en el archivo de recuperación del servidor. Cuando aborta una transacción, el servidor utiliza la lista de intenciones para borrar todas las versiones tentativas de los objetos que fueron creadas por esa transacción.

Recuérdese además que una transacción distribuida ha de llevar a cabo un protocolo de consumación atómico antes de que pueda consumarse o abortarse. Nuestra discusión sobre recuperación se centra en el protocolo de consumación en dos fases, en el cual todos los participantes involucrados en una transacción en primer lugar dicen si están preparados para consumarla y después, si todos los participantes están de acuerdo, llevan a cabo las acciones reales de consumación. Si los participantes no pueden ponerse de acuerdo sobre la consumación, deben abortar la transacción.

En el momento en el que un participante dice que está preparado para consumar una transacción, su gestor de recuperación debe haber guardado en su archivo de recuperación tanto su lista de intenciones para esa transacción como los objetos de la lista de intenciones, de tal forma que después podría llevar a cabo la consumación, incluso si, entretanto, se cae.

Cuando todos los participantes implicados en una transacción acuerdan en consumarla, el coordinador informa al cliente y, a continuación, envía mensajes a los participantes para que consumen su parte de la transacción. Una vez que se ha informado al cliente de que la transacción se ha consumado, los archivos de recuperación de los servidores participantes deben contener suficiente información para asegurarse de que la transacción es consumada por todos los servidores, incluso si algunos de ellos se caen entre la preparación de la consumación y la propia consumación.

◇ **Entradas en el archivo de recuperación.** Para tratar con la recuperación de un servidor enrolado en transacciones distribuidas, se almacena en el archivo de recuperación información adicional además de los valores de los objetos. Esta información concierne al *estado* de cada transacción (ya sea *consumada*, *abortada* o *preparada* para consumarse). Además, cada objeto en el archivo de recuperación se asocia con una transacción en concreto, mediante la grabación en dicho archivo de la lista de intenciones. La Figura 13.18 muestra un resumen de los tipos de entradas incluidos en un archivo de recuperación.

Los valores del estado de la transacción relacionados con el protocolo de consumación en dos fases se discutirán en la Sección 13.6.4, cuando se trate la recuperación del protocolo de consumación en dos fases. Ahora, describiremos dos aproximaciones al empleo de los archivos de recuperación: el registro histórico (*logging*) y las versiones sombra (*shadow*).

Tipo de entrada	*Descripción de los contenidos de la entrada*
Objeto	Un valor de un objeto
Estado de la transacción	Identificador de la transacción, estado de la transacción (*preparada*, *consumada*, *abortada*), y otros valores de estado utilizados para el protocolo de consumación en dos fases.
Lista de intenciones	Identificador de la transacción y una secuencia de intenciones, cada una de las cuales consiste en <identificador de objeto>, <posición en el archivo de recuperación del valor del objeto>.

Figura 13.18. Tipos de entradas en un archivo de recuperación.

13.6.1. REGISTRO HISTÓRICO

En la técnica de registro histórico, el archivo de recuperación representa a un registro que contiene el historial de todas las transacciones realizadas por el servidor. El historial consta de los valores de los objetos, las entradas del estado de la transacción y las listas de intenciones de las transacciones. El orden de las entradas en el registro refleja el orden en el cual se prepararon, consumaron o abortaron las transacciones en el servidor. En la práctica, el archivo de recuperación contendrá una instantánea reciente de los valores de todos los objetos en el servidor, seguido de un historial de las transacciones ocurridas tras dicha instantánea.

Durante la operación normal de un servidor, se llama a su gestor de recuperación en cualquier momento en el que una transacción se prepare para consumarse, ya se consume o se aborte. Cuando el servidor está preparado para consumar una transacción, el gestor de recuperación añade al archivo de recuperación todos los objetos en su lista de intenciones, seguidos por el estado actual de la transacción (*preparada*), junto con su lista de intenciones. Cuando una transacción finalmente se consuma o aborta, el gestor de recuperación añade el correspondiente estado de la transacción a su archivo de recuperación.

Se asume que la operación añadir es atómica, en el sentido de que escribe una o más entradas completas en el archivo de recuperación. Si el servidor falla, sólo puede haber sido incompleta la última escritura. Para utilizar eficientemente el disco, pueden guardarse en un búfer varias escrituras consecutivas, y a continuación pasarlo al disco con una sola operación de escritura. Una ventaja adicional de la técnica de registro histórico es que las escrituras secuenciales en el disco son más rápidas que las escrituras en ubicaciones aleatorias.

Tras una caída, se aborta cualquier transacción que no tenga el estado *consumada* en el registro. En consecuencia, cuando una transacción se consuma, debe *forzarse* la entrada en el registro de su estado, *consumada*; es decir, ha de escribirse en el registro junto con cualquier otra entrada almacenada en el búfer.

El gestor de recuperación asocia un identificador único a cada objeto, de tal forma que puedan asociarse las versiones sucesivas de un objeto en el archivo de recuperación con los objetos del servidor. Por ejemplo, una forma perdurable de una referencia a un objeto remoto, como puede ser la referencia persistente de CORBA, actuará como identificador de un objeto.

La Figura 13.19 ilustra el mecanismo de registro para las transacciones T y U del servicio bancario en la Figura 12.7. El registro histórico se reorganizó recientemente, y las entradas a la izquierda de la línea doble representan una instantánea de los valores de A, B y C antes de que

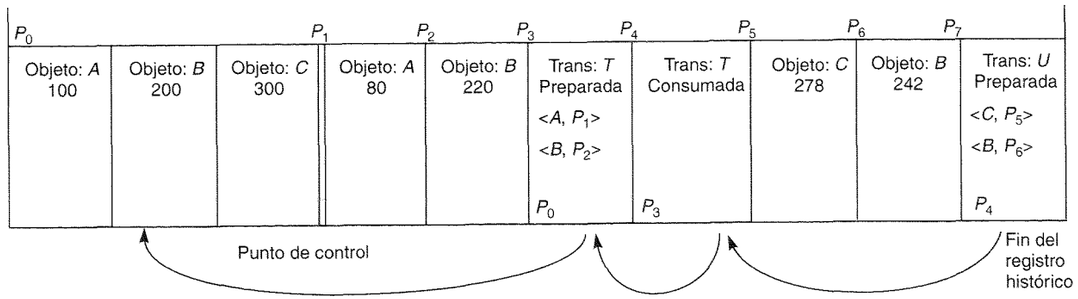


Figura 13.19. Registro para un servicio bancario.

comenzasen las transacciones T y U. En este diagrama, se usan los nombres A, B y C como identificadores únicos de objetos. Se muestra la situación en la que la transacción T se ha consumado y la transacción U se ha preparado pero no se ha consumado. Cuando la transacción T se prepara para consumarse, los valores de los objetos A y B se escriben en las posiciones P_1 y P_2 del registro, seguidas por la entrada del estado preparada de la transacción para T con su lista de intenciones ($<A, P_1>$, $<B, P_2>$). Cuando la transacción T se consuma, se pone en la posición P_4 una entrada para el estado consumada de la transacción T. A continuación, cuando la transacción U se prepara para consumarse, los valores de los objetos C y B se escriben en el registro en las posiciones P_5 y P_6, seguidas por una entrada del estado de transacción preparada para U con su lista de intenciones ($<C, P_5>$, $<B, P_6>$).

Cada entrada del estado de una transacción contiene un puntero a la posición en el archivo de recuperación de la entrada anterior de estado de la transacción, para permitir al gestor de recuperación seguir las entradas del estado de transacción en orden inverso a través del archivo de recuperación. El último puntero en la secuencia de entradas del estado de transacción apunta al punto de control.

◇ **Recuperación de objetos.** Cuando se reemplaza un servidor tras una caída, en primer lugar fija los valores iniciales por defecto de sus objetos y, a continuación, cede el control a su gestor de recuperación. Éste es responsable de restaurar los objetos del servidor de tal forma que incluyan todos los efectos de todas las transacciones consumadas realizadas en el orden correcto, y ninguno de los efectos de transacciones incompletas o abortadas.

La información más reciente acerca de las transacciones se encuentra al final del registro histórico. Existen dos aproximaciones para restaurar los datos a partir del archivo de recuperación. En la primera, el gestor de recuperación comienza al principio y restaura los valores de todos los objetos desde el punto de control más reciente. Entonces, lee los valores de cada uno de los objetos, les asocia con sus listas de intenciones y, en el caso de transacciones consumadas, reemplaza los valores de los objetos. En esta aproximación, las transacciones se repiten en el orden en el cual fueron ejecutadas, y podría haber un gran número de ellas. En la segunda aproximación, el gestor de recuperación restaurará los objetos de un servidor *leyendo en el archivo de recuperación hacia atrás*. El archivo de recuperación ha sido estructurado de tal forma que existe un puntero hacia atrás, desde una entrada del estado de la transacción a la siguiente. El gestor de recuperación utiliza las transacciones con estado consumada para restaurar aquellos objetos que no habían sido todavía restaurados. Continúa hasta que ha restaurado todos los objetos del servidor. Esto tiene la ventaja de que cada objeto sólo se restaura una vez.

Para recuperar los efectos de una transacción, un gestor de recuperación obtiene la correspondiente lista de intenciones del archivo de recuperación. La lista de intenciones contiene los identificadores y las posiciones en el archivo de recuperación de los valores de todos los objetos que se vieron afectados por la transacción.

Si el servidor falla en el punto al que se llegó en la Figura 13.19, su gestor de recuperación recuperará los objetos como sigue. Comienza en la última entrada del estado de transacción en el registro (en P_7) y concluye que la transacción U no se había consumado y sus efectos han de ser olvidados. A continuación, se mueve a la entrada anterior de estado de transacción en el registro (en P_4) y concluye que la transacción T se había consumado. Para recuperar los objetos afectados por la transacción T, se mueve a la entrada previa de estado de transacción en el registro (en P_3) y encuentra la lista de intenciones para $T(<A, P_1>, <B, P_2>)$. Entonces restaura los objetos A y B a partir de los valores en P_1 y P_2. Ya que no ha restaurado C, se mueve hacia atrás hasta P_0, que es un punto de control, y restaura C.

Para facilitar la reorganización subsiguiente del archivo de recuperación, el gestor de recuperación anota todas las transacciones preparadas que encuentra durante el proceso de restauración de los objetos del servidor. Para cada transacción preparada, añade al archivo de recuperación un estado de transacción abortada. Esto asegura que en el archivo de recuperación toda transacción, al final, se muestra bien como consumada o bien como abortada.

El servidor podría fallar de nuevo durante los procedimientos de recuperación. Es esencial que la recuperación sea idempotente en el sentido de que pueda realizarse cualquier número de veces con el mismo efecto. Esto es inmediato bajo nuestra suposición de que todos los objetos se restauran sobre la memoria volátil. En el caso de una base de datos, que guarda sus objetos en un dispositivo de almacenamiento permanente, con una memoria caché en la memoria volátil, algunos de los objetos en el dispositivo de almacenamiento permanente estarán anticuados cuando sea reemplazado el servidor tras una caída. Por lo tanto, su gestor de recuperación ha de restaurar los objetos en el dispositivo de almacenamiento permanente. Si falla durante la recuperación, los objetos parcialmente restaurados estarán ya allí. Esto hace que sea un poco más difícil conseguir la idempotencia.

◇ **Reorganización del archivo de recuperación.** Un gestor de recuperación es responsable de reorganizar su archivo de recuperación para conseguir que el proceso de recuperación sea más rápido y para reducir el espacio usado. Si no se reorganiza alguna vez el archivo de recuperación, entonces el proceso de recuperación debe buscar hacia atrás a través del archivo de recuperación hasta que haya encontrado un valor para cada uno de sus objetos. Conceptualmente, la única información que se requiere para la recuperación es una copia de las versiones consumadas de todos los objetos en el servidor. Ésta sería la forma más compacta para el archivo de recuperación. El término *establecimiento de un punto de control* (*checkpointing*) se utiliza para referirse al proceso de escribir los valores consumados actuales de los objetos de un servidor a un nuevo archivo de recuperación, junto con las entradas de los estados de las transacciones y las listas de intenciones de las transacciones que no habían sido completamente resueltas (incluyendo la información relacionada con el protocolo de consumación en dos fases). El término *punto de control* se utiliza para referirse a la información guardada por los procesos de establecimiento de puntos de control. El propósito de crear puntos de control es reducir el número de transacciones con las que tratar durante la recuperación y para aprovechar el espacio del archivo.

El establecimiento de puntos de control puede hacerse inmediatamente tras una recuperación, pero antes han de comenzar todas las nuevas transacciones. Sin embargo, puede que no se dé con mucha frecuencia la recuperación. En consecuencia, puede que sea necesario establecer de puntos de control, de vez en cuando, durante la actividad normal de un servidor. El punto de control se escribe sobre un futuro archivo de recuperación, y el archivo de recuperación actual permanece en uso hasta que el punto de control esté completo. El establecimiento de puntos de control consiste en *añadir una marca* al archivo de recuperación cuando comienza dicho proceso, escribiendo los objetos del servidor en el futuro archivo de recuperación y, a continuación, copiando (1) las entra-

das anteriores a la marca que se relacionan con transacciones no resueltas todavía, y (2) todas las entradas tras la marca en el archivo de recuperación hacia el futuro archivo de recuperación. Cuando un punto de control está completo, el futuro archivo de recuperación se convierte en el archivo de recuperación.

El sistema de recuperación puede reducir su utilización del espacio descartando el antiguo archivo de recuperación. Cuando el gestor de recuperación está llevando a cabo el proceso de recuperación, puede encontrarse un punto de control en el archivo de recuperación. Cuando ocurre esto, puede restaurar, de forma inmediata, todos los objetos destacados a partir de dicho punto de control.

13.6.2. VERSIONES SOMBRA

La técnica de *registro histórico* graba las entradas del estado de las transacciones, las listas de intenciones y todos los objetos en el mismo archivo: el registro histórico. La técnica de *versiones sombra* es una forma alternativa de organizar un archivo de recuperación. Utiliza un *mapa* para localizar versiones de los objetos del servidor en un archivo denominado *almacén de versiones*. El mapa asocia los identificadores de los objetos del servidor con las posiciones de sus versiones actuales en el almacén de versiones. Las versiones escritas por cada transacción son *sombras* de las versiones previas consumadas. Las entradas del estado de las transacciones y las listas de intenciones se tratan de forma separada. Primero se van a describir las versiones sombra.

Cuando una transacción está preparada para consumarse, cualquiera de los objetos cambiados por la transacción se añade al almacén de versiones, dejando las correspondientes versiones consumadas sin cambios. Estas nuevas versiones, todavía tentativas, se denominan versiones *sombra*. Cuando una transacción se consuma, se hace un nuevo mapa mediante la copia del viejo mapa e introduciendo las posiciones de las versiones sombreadas. Para completar el proceso de consumación, el nuevo mapa reemplaza al viejo.

Para restaurar los objetos cuando se reemplaza un servidor tras una caída, su gestor de recuperación lee el mapa y utiliza la información en el mapa para localizar los objetos en el almacén de versiones.

Esta técnica se ilustra con el mismo ejemplo que involucra a las transacciones T y U en la Figura 13.20. La primera columna en la tabla muestra el mapa antes de las transacciones T y U, cuando los saldos de las cuentas A, B y C son 100$, 200$ y 300$, respectivamente. La segunda columna muestra el mapa cuando se ha consumado la transacción T.

El almacén de versiones contiene un punto de control, seguido por las versiones, originadas por la transacción T, de A y de B en P_1 y P_2. Además contiene las versiones sombreadas de B y de C, generadas por la transacción U, en P_3 y P_4.

Mapa al inicio	*Mapa cuando T se consuma*
$A \rightarrow P_0$ $B \rightarrow P_0'$ $C \rightarrow P_0''$	$A \rightarrow P_1$ $B \rightarrow P_2$ $C \rightarrow P_0''$

Almacén de versiones

P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
100	200	300	80	220	278	242

Punto de control

Figura 13.20. Versiones sombra.

El mapa debe escribirse siempre en un lugar bien conocido (por ejemplo, al comienzo del almacén de versiones o en un archivo separado) de tal forma que pueda ser encontrado cuando el sistema necesita ser recuperado.

El cambio del viejo mapa al nuevo debe realizarse en un único paso atómico. Para conseguir esto, es esencial que se utilice para el mapa un dispositivo de almacenamiento estable, de tal forma que se garantice que sea un mapa válido incluso cuando falla una operación de escritura en archivo. El método de versiones sombra proporciona una recuperación más rápida que la del registro histórico porque las posiciones de los objetos consumados actuales se graban en el mapa, mientras que la recuperación desde un registro requiere la búsqueda a través de los objetos del registro. La técnica de registro debería ser más rápida que las versiones sombreadas durante la actividad normal del sistema. Esto se debe a que el registro requiere solamente una secuencia de operaciones añadir sobre el mismo archivo, mientras que las versiones sombra requieren una escritura sobre un dispositivo de almacenamiento estable adicional (involucrando dos bloques no relacionados del disco).

Las versiones sombra no son suficientes, por sí mismas, para un servidor que maneje transacciones distribuidas. Las entradas del estado de las transacciones y las listas de intenciones se graban en un archivo que se denomina archivo del estado de la transacción. Cada lista de intenciones representa la parte del mapa que será alterado por una transacción cuando se complete. El archivo del estado de la transacción puede, por ejemplo, organizarse como un registro histórico.

La siguiente figura muestra el mapa y el archivo del estado de la transacción para nuestro ejemplo actual, cuando T se ha consumado y U está preparada para consumarse.

Mapa
$A \rightarrow P_1$ $B \rightarrow P_2$ $C \rightarrow P_0''$

Almacenamiento estable

Archivo de estado de transacción

T	T	U
preparada $A \rightarrow P_1$ $B \rightarrow P_2$	consumada	preparada $B \rightarrow P_3$ $C \rightarrow P_4$

Existe una posibilidad de que un servidor pueda caerse entre el momento en el que se escriba el estado consumada hacia el archivo del estado de la transacción y el momento en el que se actualiza el mapa, en cuyo caso no se habrá enviado un acuse de recibo al cliente. El gestor de recuperación debe permitir esta posibilidad cuando el servidor es reemplazado tras una caída, por ejemplo comprobando si el mapa incluye los efectos de la última transacción consumada en el archivo del estado de las transacciones. Si no, entonces la última debería marcarse como abortada.

13.6.3. LA NECESIDAD DE ENTRADAS DEL ESTADO DE LA TRANSACCIÓN Y LISTA DE INTENCIONES EN UN ARCHIVO DE RECUPERACIÓN

Es posible diseñar un archivo de recuperación simple que no incluya entradas para elementos del estado de la transacción ni listas de intenciones. Esta clase de archivo de recuperación puede ser adecuada cuando todas las transacciones se dirigen a un único servidor. El uso de los elementos de estado de la transacción y de las listas de intenciones en el archivo de recuperación es esencial para un servidor pensado para participar en transacciones distribuidas. Esta aproximación también puede ser útil para servidores de transacciones no distribuidas por varias razones, incluyendo las siguientes:

- Algunos gestores de recuperación están diseñados para una pronta escritura de los objetos en el archivo de recuperación, bajo la suposición de que las transacciones normalmente se consuman.

- Si las transacciones utilizan un número elevado de objetos grandes, la necesidad de escribirlos de forma contigua en el archivo de recuperación puede complicar el diseño de un servidor. Cuando se hace referencia a los objetos desde las listas de intenciones se pueden encontrar dondequiera que estén.
- En el control de concurrencia mediante ordenación de marcas temporales, a veces, un servidor sabe que una transacción, de alguna manera, se consumará y manda un acuse de recibo al cliente (en ese momento los objetos se escriben en el archivo de recuperación, véase el Capítulo 12, para asegurar su permanencia). Sin embargo, la transacción podría tener que esperar para consumarse hasta que se hayan consumado las transacciones más tempranas. En tales situaciones, las entradas del estado de la transacción correspondiente en el archivo de recuperación estarán *esperando para consumarse* y, después, *consumada* para asegurar en el archivo de recuperación la ordenación de marcas temporales de las transacciones consumadas. Durante la recuperación, a cualquier transacción *esperando para consumarse* se le puede permitir consumarse, ya que por las que estaban esperando acaban de consumarse, o sino, han de ser abortadas debido a un fallo en el servidor.

13.6.4. RECUPERACIÓN DEL PROTOCOLO DE CONSUMACIÓN EN DOS FASES

En una transacción distribuida, cada servidor mantiene su propio archivo de recuperación. La gestión de recuperación descrita en la sección previa debe extenderse para tratar con cualquier transacción que esté efectuando el protocolo de consumación en dos fases en el momento en el que el servidor falla. Los gestores de recuperación utilizan dos nuevos valores de estado: *hecha* e *incierta*. Estos valores de estado se muestran en la Figura 13.6. Un coordinador utiliza *consumada* para indicar que el resultado de la votación es *Sí* y *hecha* para indicar que el protocolo de consumación en dos fases se ha completado. Un participante utiliza *incierta* para indicar que ha votado *Sí* pero no sabe todavía el resultado de la votación. Dos tipos adicionales de entrada permiten a un coordinador registrar una lista de participantes y a un participante registrar a su coordinador:

Tipo de entrada	*Descripción de los contenidos de la entrada*
Coordinador	Identificador de transacción, lista de participantes
Participante	Identificador de transacción, coordinador

En la fase 1 del protocolo, cuando el coordinador está preparado para consumar (y ya ha añadido una entrada del estado preparada a su archivo de recuperación) su gestor de recuperación añade una entrada *coordinador* a su archivo de recuperación. Antes de que un participante pueda votar *Sí* debe estar ya preparado para consumar (y debe haber añadido ya una entrada del estado preparada a su archivo de recuperación). Cuando vota *Sí*, su gestor de recuperación registra una entrada *participante* y añade un estado de transacción *incierta* a su archivo de recuperación como una escritura forzada. Cuando un participante vota *No* añade un estado de transacción *abortar* a su archivo de recuperación.

En la fase 2 del protocolo, el gestor de recuperación del coordinador añade a su archivo de recuperación un estado de transacción bien *consumada* o bien *abortada*, de acuerdo a la decisión. Ésta debe ser una escritura forzada. Los gestores de recuperación de los participantes añaden a sus archivos de recuperación el estado de transacción *consuma* o *aborta*, en función del mensaje recibido por parte del coordinador. Cuando un coordinador ha recibido una confirmación desde todos sus participantes, su gestor de recuperación añade un estado de la transacción *hecha* a su archivo

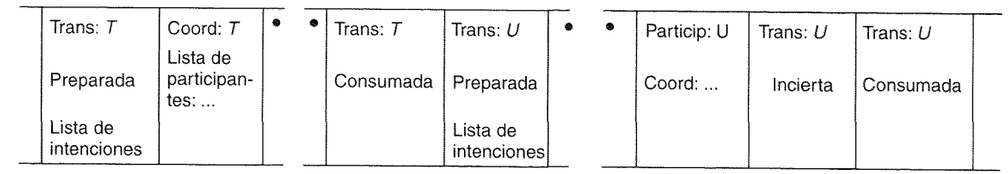


Figura 13.21. Registro con las entradas relacionadas con el protocolo de consumación en dos fases.

de recuperación (ésta no necesita ser forzada). La entrada del estado *hecha* no es parte del protocolo, pero se utiliza cuando se reorganiza el archivo de recuperación. La Figura 13.21 muestra las entradas en un registro histórico para la transacción *T* en la cual el servidor jugó el papel de coordinador, y para la transacción *U*, en la que el servidor jugó el papel de participante.

Para ambas transacciones, llega primero la entrada del estado *preparada* de la transacción. En el caso de un coordinador, es seguida por una entrada coordinador, y por una entrada del estado *consumada* de la transacción. La entrada del estado *hecha* de la transacción no se muestra en la Figura 13.21. En el caso de un participante, la entrada del estado *preparada* de la transacción es seguida por una entrada participante cuyo estado es *incierta* y, después, una entrada del estado de la transacción: *consumada* o *abortada*.

Cuando se reemplaza un servidor tras una caída, el gestor de recuperación tiene que tratar con el protocolo de consumación en dos fases además de restaurar los objetos. Para cualquier transacción donde el servidor haya jugado el papel de coordinador, debería encontrar una entrada coordinador y un conjunto de entradas del estado de la transacción. Para cualquier transacción donde el servidor haya jugado el papel de participante, debería encontrar una entrada participante y un conjunto de entradas del estado de la transacción. En ambos casos, la entrada del estado de la transacción más reciente, esto es, aquélla más cercana al final del registro, determina el estado de la transacción en el momento del fallo. La acción del gestor de recuperación con respecto al protocolo de consumación en dos fases para cualquier transacción depende de si el servidor fue el coordinador o un participante y de su estado en el momento del fallo, como se muestra en la Figura 13.22.

◇ **Reorganización de un archivo de recuperación.** Hay que tener cuidado cuando se está creando un punto de control para asegurarse de que no se eliminan del archivo de recuperación las entradas del *coordinador* asociadas a transacciones sin el estado *hecha*. Estas entradas han de retenerse hasta que todos los participantes hayan confirmado que han completado sus transacciones. Las entradas con estado *hecha* pueden descartarse. También deben retenerse las entradas de participante con estado de la transacción *incierta*.

◇ **Recuperación de transacciones anidadas.** En el caso más simple, cada subtransacción de una transacción anidada accede a un conjunto de objetos distintos diferente. Según se prepara cada participante para consumar durante el protocolo de consumación en dos fases, escribe sus objetos y listas de intenciones en el archivo de recuperación local, asociándole el identificador de transacción de la transacción del nivel superior. Aunque las transacciones anidadas utilizan una variante especial del protocolo de consumación en dos fases, el gestor de recuperación utiliza los mismos valores de estado de la transacción que para las transacciones planas.

Sin embargo, la recuperación de transacciones abortadas es compleja debido a que distintas subtransacciones, al mismo y a diferentes niveles en la jerarquía de anidamiento, pueden acceder al mismo objeto. La Sección 12.4 describe un esquema de bloqueo en el que las transacciones madre heredan bloqueos y las subtransacciones adquieren los bloqueos de sus madres. El esquema de bloqueos fuerza a las transacciones madre y a las subtransacciones a acceder a los datos de objetos

Papel	*Estado*	*Acción del gestor de recuperación*
Coordinador	*Preparada*	No se había alcanzado una decisión cuando cayó el coordinador. Envía *abortaTransacción* a todos los servidores en la lista de participantes y añade en su archivo de recuperación el estado de la transacción *abortada*. La misma acción para el estado *abortada*. Si no hay lista de participantes, los participantes, al final, llegarán a su tiempo límite y abortarán la transacción.
Coordinador	*Consumada*	Se había alcanzado una decisión de consumar antes de que el servidor fallase. Envía un mensaje *Consumar* a todos los participantes en su lista de participantes (en caso de que no lo haya hecho antes) y continúa el protocolo en dos fases en el paso 4 (véase Figura 13.5).
Participante	*Consumada*	El participante envía un mensaje *heConsumado* al coordinador (en caso de que no lo hubiese hecho antes que fallase). Esto permitirá al coordinador descartar información acerca de esta transacción en el siguiente punto de control.
Participante	*Incierta*	El participante falló antes de saber el resultado de la transacción. No puede determinar el estado de la transacción hasta que el coordinador le informe de la decisión. Enviará un mensaje *dameDecisión* al coordinador para determinar el estado de la transacción. Cuando reciba la respuesta, y en función de ella, consumará o abortará.
Participante	*Preparada*	El participante no había votado y puede abortar la transacción.
Coordinador	*Hecha*	No se requiere acción alguna.

Figura 13.22. Recuperación del protocolo de consumación en dos fases.

comunes en distintos instantes de tiempo y aseguran que los accesos por parte de subtransacciones concurrentes sobre los mismos objetos deben estar secuenciados.

Los objetos a los que se accede según las reglas de transacciones anidadas se hacen recuperables proporcionando versiones tentativas para cada subtransacción. La relación entre las versiones tentativas de un objeto utilizado por las subtransacciones de una transacción anidada es similar a la relación entre los bloqueos. Para facilitar la recuperación tras los abortos, el servidor de un objeto compartido por transacciones a múltiples niveles proporciona una pila de versiones tentativas, una por cada transacción anidada que va a ser utilizada.

Cuando la primera subtransacción de un conjunto de transacciones anidadas accede a un objeto, se le proporciona una versión tentativa que es una copia de la versión consumada actual del objeto. Ésta se considera que está en el tope de la pila, pero a no ser que alguna otra subtransacción acceda al mismo objeto, la pila no se materializará.

Cuando una de sus subtransacciones accede al mismo objeto, copia la versión de lo alto de la pila y lo vuelve a dejar. Todas las actualizaciones de las subtransacciones se aplican a la versión tentativa de lo alto de la pila. Cuando una subtransacción se consuma provisionalmente, su madre hereda la nueva versión. Para conseguir esto, tanto la versión de la subtransacción como la versión de su madre se descartan de la pila y, a continuación, la nueva versión de la subtransacción se empuja en lo alto de la pila (reemplazando efectivamente la versión de su madre). Cuando una subtransacción aborta, su versión en el tope de la pila se descarta. Al final, cuando se consuma la transacción de nivel superior, la versión de lo alto de la pila (si es que hay alguna) se convierte en la nueva versión consumada.

Por ejemplo, en la Figura 13.23, supóngase que todas las transacciones T_1, T_{11}, T_{12} y T_2 acceden al mismo objeto, A, en el orden T_1; T_{11}; T_{12}; T_2. Supóngase también que sus versiones tentativas se denominan A_1, A_{11}, A_{12} y A_2. Cuando T_1 comienza a ejecutarse, A_1 se basa en la versión

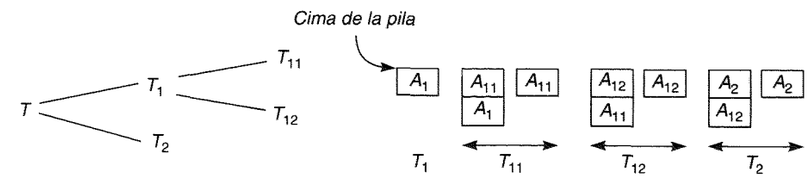


Figura 13.23. Transacciones anidadas.

consumada de A y se pone en la pila. Cuando T_{11} comienza a ejecutarse, basa su versión A_{11} en A_1 y la pone en la pila; cuando se complete, reemplaza la versión de su madre en la pila. Las transacciones T_{12} y T_2 actúan de una forma similar, dejando finalmente el resultado de T_2 en lo alto de la pila.

13.7. RESUMEN

En el caso más general, la transacción de un cliente solicitará operaciones sobre objetos en varios servidores distintos. Una transacción distribuida es aquella cuya actividad involucra a varios servidores distintos. Puede usarse una transacción con una estructura anidada para facilitar una concurrencia adicional y una consumación independiente por parte de los servidores dentro de una transacción distribuida.

La propiedad de atomicidad de las transacciones requiere que todos los servidores que participen en una transacción distribuida se consumen o aborten. Para lograr este efecto, incluso cuando los servidores se caen durante la ejecución, se diseñan protocolos de consumación atómica. El protocolo de consumación en dos fases permite que un servidor aborte de forma unilateral. Incluye acciones asociadas a tiempos límite para tratar con los retrasos ocasionados por la caída de servidores. El protocolo de consumación en dos fases puede emplear una cantidad de tiempo no delimitado para su finalización, pero se garantiza que, al final, se completará.

El control de concurrencia en las transacciones distribuidas es modular, cada servidor es responsable de la secuenciabilidad de las transacción que acceden a sus propios objetos. Sin embargo, se necesitan protocolos adicionales para garantizar que las transacciones sean globalmente secuenciables. Las transacciones distribuidas que utilizan ordenación por marcas temporales necesitan alguna forma de generar una ordenación de marcas temporales acordada entre múltiples servidores. Aquellas que utilizan control de concurrencia optimista necesitan una validación global o algún medio de forzar un ordenamiento global sobre la consumación de las transacciones.

Las transacciones distribuidas que utilizan un bloqueo en dos fases pueden padecer interbloqueos distribuidos. El objetivo de la detección de interbloqueos distribuidos es buscar ciclos en un grafo *espera por* global. Si se encuentra un ciclo, han de abortarse una o más transacciones para solucionar el interbloqueo. Una aproximación descentralizada para la detección de interbloqueos distribuidos es la captura de arcos.

Las aplicaciones basadas en transacciones tienen requisitos fuertes en lo que respecta a la persistencia y a la integridad de la información almacenada, aunque no suelan tener requisitos de respuesta inmediata en todo momento. Los protocolos de consumación atómica son la clave para las transacciones distribuidas, pero no se puede garantizar que se completen dentro de un tiempo límite concreto. Mediante los puntos de control y al registro en históricos se consigue que las transacciones sean perdurables, lo cual se utiliza para recuperarse al sustituir un servidor tras una caída. Los usuarios de un servicio de transacciones experimentarán algunas demoras durante la recupera-

ción. Aunque se asume que los servidores de transacciones distribuidas pueden fallar por caídas y que funcionan en un sistema asíncrono, son capaces de alcanzar un consenso acerca del resultado de las transacciones porque los servidores son reemplazados por nuevos procesos que pueden adquirir toda la información relevante a partir de dispositivos de almacenamiento permanente o bien de otros servidores.

EJERCICIOS

13.1. En una variante descentralizada del protocolo de consumación en dos fases, los participantes se comunican directamente entre ellos, en lugar de indirectamente a través de un coordinador. En la fase 1, el coordinador envía su voto a todos los participantes. En la fase 2, si el voto del coordinador es *No*, los participantes simplemente abortan la transacción; si es *Sí*, cada participante envía su voto al coordinador y a los otros participantes, cada uno de los cuales decide según el resultado de la votación y la llevan a cabo. Calcúlese el número de mensajes y el número de rondas que necesita el protocolo. ¿Cuáles son las ventajas o desventajas en comparación con la variante centralizada?

13.2. Un protocolo de consumación en tres fases consta de las siguientes partes:

Fase 1: es la misma que en la consumación en dos fases.

Fase 2: el coordinador colecta los votos y toma una decisión: si es *No*, *aborta* e informa a los participantes que votaron *Sí*; si la decisión es *Sí*, envía una petición *preConsuma* a todos los participantes. Los participantes que votaron *Sí* esperan por la petición de *preConsuma* o *Aborta*. Acusan el recibo de la petición *preConsuma* o llevan a cabo la petición *Aborta*.

Fase 3: el coordinador colecciona los acuses de recibo. Cuando los ha recibido todos, *Consuma* y envía los mensajes *Consuma* a los participantes. Éstos esperan por la petición *Consuma*. Cuando ésta llega, *Consuman*.

Explíquese cómo evita este protocolo el retraso de los participantes durante el período *incierto* debido a los fallos del coordinador o de otros participantes. Asúmase que la comunicación no falla.

13.3. Explíquese cómo el protocolo de consumación en dos fases para las transacciones anidadas asegura que si la transacción de nivel superior se consuma, todas las descendientes directas son consumadas o abortadas.

13.4. Dé un ejemplo de los entrelazamientos asociados a dos transacciones que son secuencialmente equivalentes en cada servidor, pero que no son globalmente secuencialmente equivalentes.

13.5. El procedimiento *dameDecisión* que se definió en la Figura 13.4 sólo lo proporcionan los coordinadores. Defínase una nueva versión de *dameDecisión* que sea proporcionado por los participantes, para ser usado por otros participantes que necesiten obtener una decisión cuando no se encuentre disponible el coordinador.

Presuponga que cualquier participante activo puede realizar una petición *dameDecisión* a cualquier otro participante activo. ¿Resuelve esto el problema de los retrasos durante el período *incierto*? Razónese la respuesta. ¿En qué punto del protocolo de consumación en dos fases debería el coordinador informar a los participantes de las identidades de los otros participantes (para permitir la comunicación)?

13.6. Extienda la definición del bloqueo en dos fases para aplicarla a las transacciones distribuidas. Explique cómo aseguran esto las transacciones distribuidas al emplear localmente el bloqueo en dos fases estricto.

13.7. Asumiendo que se usa un bloqueo en dos fases estricto, describa cómo se relacionan las acciones del protocolo de consumación en dos fases con las acciones del control de concurrencia de cada servidor individual. ¿Cómo encaja ahí la detección de interbloqueo distribuido?

13.8. Un servidor utiliza ordenación de marcas temporales para el control de concurrencia local. ¿Qué cambios deben hacerse para adaptarla y usarla en transacciones distribuidas? ¿Bajo qué condiciones podría argumentarse que el protocolo de consumación en dos fases es redundante con la ordenación de marcas temporales?

13.9. Considérese un control de concurrencia distribuida optimista en el cual cada servidor realiza validación local hacia atrás secuencialmente (esto es, sólo con una transacción en las fases de validar y actualizar de cada vez), en relación con la respuesta al Ejercicio 13.4. Describa los posibles resultados cuando las dos transacciones intentan consumarse. ¿Qué diferencia hay cuando los servidores utilizan validación en paralelo?

13.10. Un detector de interbloqueo global centralizado aloja la unión de los grafos *espera por* locales. Dé un ejemplo para explicar cómo podría detectarse un interbloqueo fantasma si una transacción que está esperando en un ciclo de interbloqueo aborta durante el procedimiento de detección del interbloqueo.

13.11. Considere el algoritmo de captura de arcos (sin prioridades). Dé ejemplos que muestren que podría detectar interbloqueos fantasma.

13.12. Un servidor gestiona los objetos $a_1, a_2, ..., a_n$. Proporciona dos operaciones a sus clientes:

Lee(i) devuelve el valor de a_i
Escribe(i, Valor) asigna *Valor* a a_i

Las transacciones *T*, *U* y *V* se definen como sigue:

T: *x = Lee(i); Escribe(j, 44);*
U: *Escribe(i, 55); Escribe(j, 66);*
V: *Escribe(k, 77); Escribe(k, 88);*

Describa la información que se escribe en el archivo de registro histórico en función de estas tres transacciones si se está usando un bloqueo en dos fases estricto, y *U* adquiere a_i y a_j antes de comenzar *T*. Describa cómo utilizaría esta información el gestor de recuperación para recuperar los efectos de *T*, *U* y *V* cuando se reemplaza el servidor tras una caída. ¿Cuál es el significado del orden de las entradas de consumación en el archivo de registros histórico?

13.13. El añadir una entrada al archivo de registro es una operación atómica, pero pueden intercalarse operaciones de adición por parte de transacciones diferentes. ¿Cómo afecta esto a la respuesta del Ejercicio 13.12?

13.14. Las transacciones *T*, *U* y *V* del Ejercicio 13.12 utilizan bloqueo en dos fases estricto y sus peticiones se intercalan como sigue:

T	U	V
x = Lee(i);		
		Escribe(k, 77);
	Escribe(i, 55);	
Escribe(j, 44)		
		Escribe(k, 88)
	Escribe(j, 66)	

Asumiendo que el gestor de recuperación añade inmediatamente al archivo de registro la entrada de datos correspondientes a cada operación *Escribe* en lugar de esperar hasta el final de la transacción, descríbase la información escrita al archivo de registro histórico en función de las transacciones *T*, *U* y *V*. ¿Afecta una escritura temprana a la corrección del procedimiento de recuperación? ¿Cuáles son las ventajas y desventajas de la escritura temprana?

13.15. Las transacciones *T* y *U* se ejecutan con control de concurrencia por ordenamiento de marcas temporales. Describa la información que se escribe en el archivo de registro histórico en función de *T* y *U*, teniendo en cuenta el hecho de que *U* tiene una marca temporal posterior a la de *T* y debe esperar por *T* para consumarse. ¿Por qué es esencial que las entradas de consumación en el archivo de registro estén ordenadas por las marcas temporales? Describa el efecto de la recuperación si el servidor se cae (i) entre las dos *Consuma* y (ii) tras ambas.

T	U
x = Lee(i);	
	Escribe(i, 55);
	Escribe(j, 66);
Escribe(j, 44);	
	Consuma
Consuma	

Cuáles son las ventajas y desventajas de la escritura temprana con ordenación por marcas temporales.

13.16. Las transacciones *T* y *U* del Ejercicio 13.15 se ejecutan con control de concurrencia optimista utilizando validación hacia atrás y reiniciando las transacciones que fallan. Describa la información escrita en el archivo de registro histórico en su nombre. ¿Por qué es esencial que las entradas de consumación en el archivo de registro histórico estén ordenadas por los números de transacción? ¿Cómo son los conjuntos de escritura de las transacciones consumadas que se representan en el archivo de registro?

13.17. Suponga que el coordinador de una transacción cae tras haber registrado la entrada de la lista de intenciones, pero antes de que haya grabado la lista de participantes o haya enviado las peticiones de *puedeConsumar?*. Describa cómo resuelven esta situación los participantes. ¿Qué hará el coordinador cuando se recupere? ¿Hubiese sido mejor registrar la entrada de la lista de participantes antes que la de la lista de intenciones?

14

REPLICACIÓN

14.1. Introducción
14.2. Modelo de sistema y comunicación en grupo
14.3. Servicios tolerantes a fallos
14.4. Servicios con alta disponibilidad
14.5. Transacciones con datos replicados
14.6. Resumen

La replicación es la clave para proporcionar alta disponibilidad y tolerancia a fallos en sistemas distribuidos. Cada vez hay más interés en la alta disponibilidad dada la tendencia hacia la computación móvil y, por consiguiente, hacia las operaciones sin conexión. La tolerancia a fallos es una preocupación constante en los servicios que se prestan en sistemas donde la seguridad es crítica y en otros tipos de sistemas importantes.

La primera parte del capítulo trata aquellos sistemas que solicitan una operación individual cada vez sobre colecciones de objetos replicados. El capítulo comienza con una descripción de los componentes de la arquitectura y un modelo de sistema para aquellos servicios que empleen replicación. Describiremos la integración de la gestión de pertenencia a grupos como parte de la comunicación en grupo, que es especialmente importante para los servicios tolerantes a fallos.

A continuación, el capítulo describe las aproximaciones para conseguir la tolerancia a fallos. Se presentan los criterios de corrección: capacidad de secuenciación y la consistencia secuencial. Posteriormente se presentan y discuten dos aproximaciones: la replicación pasiva (primario-respaldo), en la cual los clientes se comunican con una réplica distinguida, y la replicación activa, en la cual los clientes se comunican mediante multidifusión con todas las réplicas.

Se considerarán tres sistemas para proporcionar servicios con alta disponibilidad. En las arquitecturas de cotilleo y Bayou, las actualizaciones se propagan de forma diferida entre las réplicas de los datos compartidos. En Bayou, se emplea la técnica de transformación operacional para hacer cumplir la consistencia. Coda es un ejemplo de un servicio de archivos con alta disponibilidad.

El capítulo finaliza considerando las transacciones (secuencias de operaciones) sobre objetos replicados. Trata las arquitecturas de sistemas transaccionales replicados y cómo estos sistemas gestionan los fallos en los servidores y las particiones en la red.

14.1. INTRODUCCIÓN

En este capítulo se estudiará la replicación de datos: el mantenimiento de copias de datos en múltiples computadores. La replicación es la clave para la eficacia de sistemas distribuidos en la medida en que proporciona un mayor rendimiento, alta disponibilidad y tolerancia a fallos. La replicación se utiliza ampliamente. Por ejemplo, el almacenamiento en memorias caché de recursos procedentes de servidores web en los navegadores y en servidores proxy de la Web es una forma de replicación, dado que los datos que se almacenan en las memorias caché y en los servidores son réplicas de otros. El servicio de nombres de dominio, DNS, descrito en el Capítulo 9, mantiene copias de la relación *nombre a atributo* para los computadores y de él se depende en el acceso diario a servicios a todo lo largo de Internet.

La replicación es una técnica para la mejora de los *servicios*. Las motivaciones para la replicación son aumentar el rendimiento de un servicio e incrementar su disponibilidad o hacerlo tolerante a fallos.

Mejora del rendimiento: el almacenamiento de datos en caché en los clientes y servidores es hoy en día un medio conocido para mejorar el rendimiento. Por ejemplo, el Capítulo 2 señaló que las cachés de los navegadores y los servidores proxy almacenan copias de recursos web para evitar la latencia en la captura de recursos desde el servidor. Además, algunas veces los datos se replican de forma transparente entre diversos servidores origen del mismo dominio. El trabajo se reparte entre los servidores enlazando todas las direcciones IP de los servidores al nombre DNS del lugar, digamos *www.unSitioWeb.org*. Una búsqueda DNS de *www.unSitioWeb.org* devuelve como resultado una de las distintas direcciones IP de los servidores, siguiendo un turno rotatorio (véase la Sección 9.2.3). La replicación de datos inmutables es trivial: se incrementa el rendimiento con un bajo coste para el sistema. La replicación de datos cambiantes, tales como los de la Web, ocasiona sobrecargas en forma de protocolos diseñados para asegurar que los clientes reciben datos actualizados (véase la Sección 2.2.5). Por lo tanto hay límites en la efectividad en la replicación como técnica para la mejora del rendimiento.

Incremento en la disponibilidad: los usuarios requieren que los servicios tengan una alta disponibilidad. Esto es, la proporción de tiempo en la que un servicio es accesible con tiempos de respuesta razonables debería estar cercana al cien por cien. Dejando a un lado retrasos debidos a los casos pesimistas en los conflictos de control de concurrencia (bloqueo de datos), los factores relevantes para una elevada disponibilidad son:

- Fallos en el servidor.
- Particiones de la red y operación sin conexión: desconexiones en la comunicación usualmente no planificadas y que son un efecto colateral de la movilidad del usuario.

Empezando por el primero, la replicación es una técnica para el mantenimiento automático de la disponibilidad de los datos a pesar de fallos en el servidor. Si los datos se replican en dos o más servidores independientes ante fallos, entonces el software del cliente podría ser capaz de acceder a los datos en un servidor alternativo cuando el servidor por defecto falle o no esté accesible. Esto es, el porcentaje de tiempo durante el cual el *servicio* está disponible se puede incrementar replicando los datos del servidor. Si cada uno de los n servidores tiene una probabilidad independiente p de fallar o de estar inaccesible, entonces la disponibilidad de un objeto almacenado en cada uno de esos servidores es:

$$1 - \textit{probabilidad (todos los gestores han fallado o son inaccesibles)} = 1 - p^n$$

Por ejemplo, si hay una probabilidad del cinco por ciento de cualquier fallo en un servidor individual durante un período de tiempo dado y si hay dos servidores, entonces la disponibilidad es $1 - 0{,}05^2 = 1 - 0{,}0025 = 99{,}75\,\%$. Una diferencia importante entre los sistemas de caché y la replicación de servidores es que las memorias caché no mantienen necesariamente en su totalidad colecciones de objetos, como pueden ser los archivos. Por lo tanto, las memorias caché no incrementan necesariamente la disponibilidad a nivel de aplicación (un usuario podría obtener un archivo que ya se ha usado pero no otro).

Las particiones de la red (véase la Sección 11.1) y las operaciones sin conexión son el segundo factor en contra de la alta disponibilidad. Los usuarios móviles desconectan sus computadores deliberadamente o pueden inintencionadamente encontrarse sin conexión de una red inalámbrica mientras se mueven. Por ejemplo, un usuario en un tren con un portátil podría no tener acceso a la red (la conexión a la red inalámbrica podría estar interrumpida o podría no disponerse de tal posibilidad). Para poder trabajar en estas circunstancias (llamado *trabajo sin conexión* u *operación sin conexión*), el usuario se suele preparar copiando los datos frecuentemente usados, tales como los contenidos de una agenda compartida, desde su entorno habitual al portátil. Pero a menudo debe pagarse un precio por la disponibilidad durante el período de desconexión; cuando el usuario consulta o actualiza la agenda, se arriesga a leer datos que, entre tanto, otro puede haber alterado. Por ejemplo, pueden concertar una cita en un hueco que ya ha sido ocupado. El funcionamiento sin conexión sólo es factible si el usuario (o la aplicación, en su nombre) puede arreglárselas con datos anticuados y resolver más tarde cualquier conflicto que surja.

Tolerancia a fallos: datos con alta disponibilidad no equivale necesariamente a datos estrictamente correctos. Pueden no estar actualizados, por ejemplo; o dos usuarios en lados opuestos de una partición de la red podrían hacer actualizaciones que entraran en conflicto y que necesitasen ser resueltas. Por el contrario, un servicio tolerante a fallos siempre garantiza un comportamiento estrictamente correcto a pesar de un cierto número y un cierto tipo de fallos. La corrección alude a la frescura de los datos proporcionados al cliente y los efectos de las operaciones del cliente sobre los datos. Algunas veces, la corrección también se refiere a la oportunidad de las respuestas del servicio, como por ejemplo en el caso de un sistema de control de tráfico aéreo, donde se necesitan datos correctos cada poco tiempo.

La misma técnica básica usada para conseguir alta disponibilidad (de replicación de los datos y funcionalidad entre los computadores) se aplica también para conseguir tolerancia a fallos. Si un máximo de f de $f + 1$ servidores se caen, entonces, en principio, al menos uno de ellos se mantiene para proporcionar el servicio. Y si un máximo de f servidores pueden exhibir fallos complejos, entonces, en principio, un grupo de $2f + 1$ servidores pueden proporcionar un servicio correcto, ya que los servidores correctos ganarían en la votación a los servidores fallidos (quienes podrían proporcionar valores espurios). Sin embargo, la tolerancia a fallos es más sutil de lo que esta simple descripción hace pensar. El sistema debe gestionar la coordinación de sus componentes de un modo preciso para mantener las garantías de corrección en presencia de fallos, lo cual puede suceder en cualquier momento.

Un requisito común cuando los datos están replicados es la *transparencia frente la replicación*. Esto quiere decir que los clientes normalmente no deberían advertir que existen múltiples copias *físicas* de los datos. Por lo que respecta a los clientes, los datos se organizan como objetos *lógicos* individuales (u objetos) e identifican sólo un ítem en cada caso cuando requieren que se realice una operación. Además, los clientes esperan que las operaciones sólo devuelvan un conjunto de valores. Esto ha de ser así a pesar de que las operaciones puedan efectuarse de común acuerdo sobre más de una copia física.

El otro requisito general para los datos replicados, y que puede ser más o menos restrictivo según las aplicaciones, es el de la consistencia. Ésta se refiere a si las operaciones efectuadas sobre una colección de objetos replicados producen resultados que cumplan las especificaciones de corrección para esos objetos.

Vimos en el ejemplo del dietario que durante las operaciones sin conexión se podría permitir que los datos fuesen inconsistentes, al menos temporalmente. Pero cuando los clientes se encuentran conectados, no suele ser aceptable para los diferentes clientes (que usan copias físicas diferentes de los datos) el obtener resultados inconsistentes cuando realizan peticiones que afectan a los mismos objetos lógicos. Esto es, no es aceptable que los resultados incumplan los criterios de corrección de la aplicación.

A continuación examinaremos con más detalle los aspectos del diseño que surgen cuando se replican datos para conseguir alta disponibilidad y servicios tolerantes a fallo. También se examinarán algunas soluciones estándar y técnicas para tratar estos aspectos. En primer lugar, las Secciones 14.2 y 14.4 cubrirán el caso en el que los clientes hacen invocaciones individuales sobre datos compartidos. La Sección 14.2 presentará una arquitectura general para la gestión de datos replicados y presentará, como una herramienta importante, la comunicación en grupo. La comunicación en grupo es particularmente útil para conseguir tolerancia a fallos, lo que constituirá la materia de la Sección 14.3. La Sección 14.4 describirá las técnicas para conseguir alta disponibilidad, incluyendo operaciones sin conexión. Se incluyen como casos de estudio la arquitectura de cotilleo, Bayou y el sistema de archivos Coda. La Sección 14.5 examinará cómo dar soporte a las transacciones sobre datos replicados. Como se explicó en los Capítulos 12 y 13, las transacciones son secuencias de operaciones, más que operaciones individuales.

14.2. MODELO DE SISTEMA Y COMUNICACIÓN EN GRUPO

Los datos en nuestro sistema consisten en una colección de elementos a los que llamará objetos. Un *objeto* podría ser un archivo o un objeto en Java. Pero cada objeto *lógico* se implementa mediante una colección de copias *físicas* llamadas *réplicas*. Las réplicas son objetos físicos, almacenados cada uno de ellos en un computador individual, con datos y comportamientos que están vinculados con un cierto grado de consistencia dentro del funcionamiento del sistema. Las *réplicas* de un cierto objeto no son necesariamente idénticas, al menos no en cada instante de tiempo particular. Algunas réplicas pueden haber recibido actualizaciones que otras no hayan recibido.

En esta sección, se proporciona un modelo general de sistema para gestionar réplicas y a continuación, se describen sistemas de comunicación en grupo, los cuales son particularmente útiles para conseguir tolerancia a fallos mediante la replicación.

14.2.1. MODELO DEL SISTEMA

Suponemos un sistema asíncrono en el cual los procesos pueden fallar sólo por caída. Nuestra suposición por defecto será que no pueden darse particiones en la red, pero algunas veces consideraremos qué ocurriría si las hubiese. Las particiones de la red hacen más difícil la construcción de detectores de fallo, que se usan para conseguir una multidifusión fiable y totalmente ordenada.

Para generalizar, se describen los componentes de la arquitectura según los papeles que desempeñan, sin que ello implique que tengan que ser necesariamente implementados por procesos distintos (o máquinas distintas). El modelo implica réplicas mantenidas por distintos *gestores de réplicas* (véase la Figura 14.1), que son componentes que almacenan las réplicas en un cierto computador y realizan directamente operaciones sobre ellas. Este modelo general puede aplicarse en un entorno cliente-servidor, en cuyo caso el gestor de réplicas es el servidor. Algunas veces, se les llamará simplemente servidores. Igualmente, puede referirse a una aplicación y, en ese caso, los procesos de la aplicación pueden actuar tanto como clientes como gestores de réplicas. Por ejemplo, el portátil de un usuario en un tren puede contener una aplicación que actúa como un gestor de réplicas para su agenda.

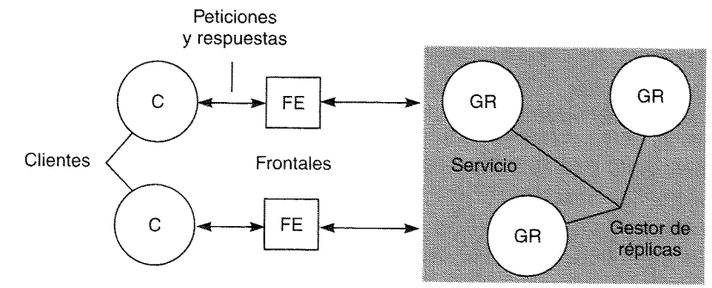


Figura 14.1. Un modelo básico de arquitectura para la gestión de datos replicados.

Siempre se exigirá que un gestor de réplicas solicite operaciones a sus réplicas con capacidad de recuperación. Esto permite suponer que una operación en un gestor de réplicas no deja resultados inconsistentes si falla durante el proceso. Algunas veces, también se exigirá a cada gestor de réplicas que sea una *máquina de estado* [Lamport 1978, Schneider 1990]. Dicho gestor de réplicas aplica operaciones atómicas (indivisibles) a sus réplicas, de tal forma que su ejecución es equivalente a realizar las operaciones siguiendo una secuencia estricta. Más aún, el estado de las réplicas es una función determinista de los estados iniciales y de la secuencia de operaciones que se les ha aplicado. Otros estímulos, como la lectura de un reloj o de un sensor adjunto, no tienen efecto sobre los valores del estado. Sin esta premisa no podrían darse las garantías de consistencia entre gestores de réplicas que acepten operaciones de actualización de forma independiente. El sistema únicamente puede determinar qué operaciones aplicar en todos los gestores de réplicas y en qué orden (no puede reproducir efectos no deterministas). La premisa implica que no sería posible, dependiendo de la arquitectura de hilos, que los servidores sean multi-hilo.

Con frecuencia un gestor de réplicas mantiene una réplica de todos los objetos y se supone que esto es así a no ser que se diga lo contrario. Aunque, en general, las réplicas de diferentes objetos pueden ser mantenidas por diferentes conjuntos de gestores de réplicas. Por ejemplo, un objeto podría ser requerido mayoritariamente por clientes de una red y otro objeto por clientes de otra red. Se gana bien poco replicándolos en los gestores en la otra red.

El conjunto de gestores de réplicas puede ser estático o dinámico. En un sistema dinámico pueden aparecer nuevos gestores de réplicas (por ejemplo, un/a segundo/a secretario/a copia una agenda en su portátil); esto no está permitido en un sistema estático. En un sistema dinámico los gestores de réplicas podrán caerse y entonces se considerará que han abandonado el sistema (aunque después sean reemplazados). En un sistema estático los gestores de réplicas no se caen (una caída implica no ejecutar *nunca* otra instrucción), sino que pueden cesar en su operación durante un período indefinido. Se volverá al tema de los fallos en la Sección 14.2.2.

El modelo general de gestión de réplicas se muestra en la Figura 14.1. Una colección de gestores de réplicas proporciona un servicio a los clientes. Los clientes perciben un servicio que les da acceso a los objetos (por ejemplo, agendas o cuentas bancarias) que, de hecho, están replicados en los gestores. Cada uno de los clientes solicita una serie de operaciones (invocaciones sobre uno o más de los objetos). Una operación implica una combinación de lecturas y actualizaciones de objetos. Las operaciones solicitadas que no incluyen actualizaciones se denominan *peticiones o solicitudes de «sólo lectura»*; las operaciones solicitadas que actualicen un objeto se denominan *peticiones de actualización* (éstas, a su vez, también pueden involucrar lecturas).

Las peticiones de cada cliente son atendidas, en primer lugar, por un componente llamado *frontal* (*front end*). El papel del frontal es comunicar por paso de mensajes con uno o más de los gestores de réplicas en lugar de forzar al cliente a hacerlo de forma explícita por sí mismo. Es el

vehículo para hacer transparente la replicación. Un frontal podría estar implementado en el espacio de direcciones del cliente o pudiera ser un proceso separado.

En general, hay cinco fases implicadas en la realización de una petición individual sobre los objetos replicados [Wiesmann y otros 2000]. Las acciones en cada fase varían en función del tipo de sistema, como se aclarará en las dos siguientes secciones. Por ejemplo, un servicio que permita operaciones sin conexión se comportará de forma distinta de uno que proporcione un servicio tolerante a fallos. Las fases son las siguientes:

El frontal realiza la petición a uno o más de los gestores de réplicas. La primera posibilidad para un frontal es comunicarse con un gestor de réplicas individual que, a su vez, se comunica con otros gestores de réplicas. La segunda posibilidad es multidifundir la petición a los gestores de réplicas.

Coordinación: los gestores de réplicas se coordinan para ejecutar la petición de forma consistente. Se ponen de acuerdo, si es necesario en esa fase, en si la petición debe realizarse (podría no realizarse en absoluto si ocurriesen fallos en este punto). Además, deciden sobre la ordenación de esta petición en relación con otras. Todos los tipos de ordenación definidos para la multidifusión en la Sección 11.4.3 también se aplican al manejo de peticiones y definimos, de nuevo, aquellas ordenaciones en este contexto:

Ordenación FIFO: si un frontal solicita r y después solicita r', entonces cualquier gestor de réplicas correcto que manipule r' debe haber manipulado antes r.

Ordenación causal: si la cuestión de la petición r *ocurrió antes* que la cuestión de la petición r', entonces cualquier gestor de réplicas correcto que manipule r' debe haber manipulado antes r.

Ordenación total: si un gestor de réplicas correcto manipula la petición r antes que la petición r', entonces cualquier gestor de réplicas correcto que manipule r' ha de haber manipulado antes r.

La mayor parte de las aplicaciones requieren ordenación FIFO. Los requisitos para la ordenación causal y la ordenación total se discuten en la siguiente sección (junto con las ordenaciones híbridas que son a la vez FIFO y total, o causal y total).

Ejecución: los gestores de réplicas ejecutan la petición, puede que de forma *tentativa*, esto es, de tal forma que pueden deshacer sus efectos posteriormente.

Acuerdo: los gestores de réplicas alcanzan un consenso, si lo hubiese, en cuanto al efecto de una petición antes de consumarla. Por ejemplo, en un sistema de transacciones los gestores de réplicas pueden acordar colectivamente abortar o consumar la transacción en este punto.

Respuesta: uno o más gestores de réplicas responden al frontal. En algunos sistemas, uno de los gestores de réplicas envía la respuesta. En otros, el frontal recibe respuestas de una colección de gestores de réplicas y selecciona o sintetiza una respuesta individual para devolvérsela al cliente. Por ejemplo, podría devolverle la primera respuesta que llegue, si el objetivo es tener una alta disponibilidad. Si el objetivo fuese la tolerancia a fallos complejos, entonces podría dar al cliente la respuesta que proporcione la mayoría de los gestores de réplicas.

Distintos sistemas pueden hacer diferentes selecciones respecto a la ordenación de las fases, así como de sus contenidos. Por ejemplo, en un sistema que soporte operaciones sin conexión, es importante dar al cliente (digamos, la aplicación del portátil del usuario) una respuesta tan pronto como sea posible. El usuario no quiere esperar hasta que el gestor de réplicas en su portátil se coordine más tarde con el gestor de réplicas en la oficina. Por el contrario, en un sistema tolerante a fallos, a un cliente no se le da respuesta hasta el final, cuando pueda garantizarse la corrección del resultado.

14.2.2. COMUNICACIÓN EN GRUPO

En la Sección 11.4 se discutió la comunicación por multidifusión, que también se conoce como comunicación en grupo porque son grupos de procesos los destinatarios de los mensajes de multidifusión. Los grupos son útiles tanto para gestionar datos replicados como para otros sistemas donde los procesos cooperen para lograr un objetivo común, recibiendo y procesando el mismo conjunto de mensajes de multidifusión. Son igualmente útiles donde los miembros del grupo consumen, de forma independiente, uno o más flujos de mensajes comunes, tales como mensajes con eventos frente a los que los procesos reaccionan de forma independiente.

La Sección 11.4 consideró que la pertenencia a los grupos estaba definida de manera estática, aunque los miembros del grupo pudieran malograrse. Sin embargo, en la práctica, los sistemas suelen requerir pertenencia dinámica: los procesos se unen y dejan el grupo según el sistema trabaja. Por ejemplo, en un servicio que maneja datos replicados, los usuarios pueden añadir o retirar un gestor de réplicas, o pudiera caerse un gestor de réplicas y, por lo tanto, necesitaría ser excluido de la operación del sistema. Una implementación completa de la comunicación en grupo incluye un *servicio de pertenencia al grupo* para gestionar la pertenencia dinámica de los grupos, además de la comunicación por multidifusión.

La gestión de la comunicación por multidifusión y la de pertenencia a grupos están fuertemente relacionadas. La Figura 14.2 muestra un sistema abierto, en el cual un proceso fuera del grupo manda al grupo un mensaje sin conocer los miembros del grupo. El servicio de comunicación en grupo ha de gestionar cambios en los miembros del grupo mientras tienen lugar multidifusiones de forma concurrente.

◇ **Papel del servicio de pertenencia a grupos.** Un servicio de pertenencia a grupos tiene cuatro tareas principales, que son las siguientes:

Proporcionar una interfaz para los cambios en la pertenencia a grupos: el servicio de pertenencia proporciona operaciones para crear y destruir grupos de procesos y para añadir o retirar un proceso de un grupo. En la mayoría de los sistemas, un proceso individual puede pertenecer a distintos grupos al mismo tiempo. Esto es así, por ejemplo, en la multidifusión IP.

Implementar un detector de fallos: el servicio incorpora un detector de fallos (véase la Sección 11.1). El servicio vigila a los miembros del grupo no sólo en el caso en el que puedan

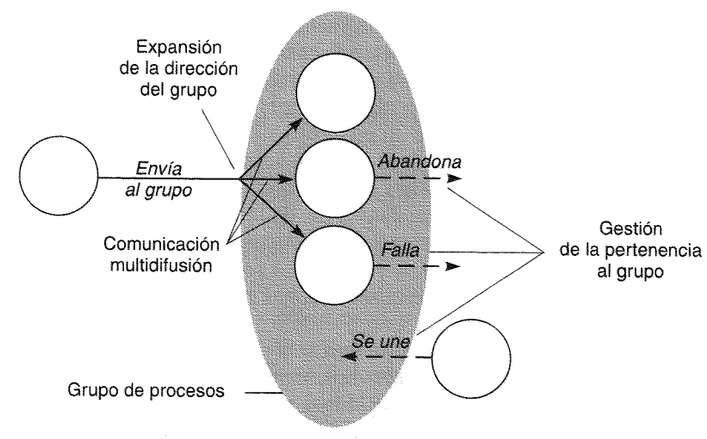


Figura 14.2. Servicios proporcionados por los grupos de procesos.

caerse sino también en el caso en el que puedan llegar a hacerse inaccesibles debido a un fallo en la comunicación. El detector marca a los procesos como *Sospechosos* o *No sospechosos*. El servicio utiliza al detector de fallos para tomar una decisión acerca de la pertenencia al grupo: excluye a un proceso si sospecha que ha fallado o se ha vuelto inaccesible.

Notificar a los miembros del grupo los cambios en la pertenencia: el servicio notifica a los miembros del grupo cuándo se añade o cuándo se excluye un proceso (debido a un fallo o cuando el proceso se retira deliberadamente del grupo).

Expandir las direcciones del grupo: cuando un proceso multidifunde un mensaje, proporciona el identificador del grupo en vez de una lista de los procesos del grupo. El servicio de gestión de pertenencia expande el identificador para obtener los miembros actuales del grupo y proceder al reparto. El servicio puede coordinar el reparto de multidifusión con cambios en los miembros mediante el control de la expansión de direcciones. Esto es, puede decidir de forma consistente dónde entregar cualquier mensaje dado, incluso aunque los miembros puedan cambiar durante la entrega. Más adelante se discutirá la comunicación *con vistas síncronas*.

Obsérvese que la multidifusión IP es un caso débil de servicio de pertenencia a un grupo, con algunas de sus propiedades pero no todas. Permite que los procesos se unan o dejen los grupos dinámicamente y realiza la expansión de direcciones, por lo que los remitentes sólo necesitan proporcionar una dirección de multidifusión IP individual como destino de un mensaje de multidifusión. Pero la multidifusión IP por sí misma no proporciona a los miembros del grupo información acerca de los miembros actuales y la entrega por multidifusión no está coordinada con los cambios de miembros.

Los sistemas que pueden adaptarse según se unan, se vayan y se rompan los procesos —en particular los sistemas tolerantes a fallos— requieren las funciones más avanzadas de detección de fallos y notificación de cambios en la pertenencia. Un servicio completo de pertenencia a grupo mantiene *vistas del grupo*, que son listas de los miembros actuales del grupo, identificados por su identificador único de proceso. La lista está ordenada, por ejemplo, de acuerdo a la secuencia en la cual los miembros se unieron al grupo. Cuando se añaden o excluyen procesos se genera una nueva vista del grupo.

Es importante comprender que un servicio de pertenencia al grupo puede excluir a un proceso de un grupo debido a que sea *Sospechoso*, incluso aunque no haya caído. Un fallo en la comunicación puede haber dejado al proceso inaccesible, aunque siguiera ejecutándose con normalidad. Un servicio de pertenencia es muy libre de excluir tales procesos. El efecto de la expulsión es que, de ahí en adelante, no se le entregarán más mensajes a este proceso. Aun más, en el caso de un grupo cerrado, si el proceso vuelve a conectarse, cualquier mensaje que intente enviar no será entregado al resto de miembros del grupo. Ese proceso ha de volver a reunirse al grupo (como una *reencarnación* de sí mismo, con un nuevo identificador) o abortar sus operaciones.

Una sospecha infundada sobre un proceso y la consecuente exclusión del mismo del grupo puede reducir la efectividad del grupo. El grupo tendrá que arreglárselas sin el rendimiento o la fiabilidad extra que potencialmente podría haber proporcionado el proceso abandonado. El reto de diseño, dejando a un lado el diseño de detectores de fallo que sean lo más precisos posible, es asegurar que un sistema basado en comunicación en grupos no se comporte *incorrectamente* si se sospecha erróneamente de un proceso.

Una consideración importante alude a cómo se ocupa el servicio de gestión de grupos de las particiones en la red. La desconexión o el fallo de componentes, tales como un encaminador en una red, puede dividir un grupo de procesos en dos o más subgrupos, de tal forma que la comunicación entre subgrupos sea imposible. Los servicios de gestión de grupos se diferencian en si son de *partición primaria* o *particionables*. En el primer caso, el servicio de gestión permite que sobreviva, a lo sumo, un subgrupo (una mayoría) de la partición; al resto de procesos se les informa que deben suspender las operaciones. Esta disposición es apropiada en casos donde los procesos gestionan datos importantes y los costes de inconsistencias entre dos o más subgrupos exceden cualquier ventaja del trabajo sin conexión.

Por otro lado, en algunas circunstancias es aceptable para dos o más subgrupos continuar operando; esto se permite en un servicio de pertenencia a grupos particionable. Por ejemplo, en una aplicación en la cual los usuarios mantienen una audio o videoconferencia para discutir algunos asuntos, puede ser aceptable para dos o más subgrupos de usuarios continuar sus discusiones independientes a pesar de la partición. Los usuarios podrán combinar sus resultados cuando la partición se restablezca y los subgrupos se conecten de nuevo.

◇ **Entrega de vistas.** Considérese la tarea de un programador escribiendo una aplicación que funcione en cada proceso de un grupo que debe enfrentarse con miembros nuevos y perdidos. El programador necesita saber que el sistema trata a cada miembro de forma consistente cuando cambian los miembros. Sería inadecuado si el programador tuviese que pedir el estado de todos los otros miembros y tomar una decisión global cuando ocurriera un cambio en la pertenencia, en lugar de ser capaz de tomar una decisión local sobre cómo responder al cambio. La vida del programador será más o menos dura en función de las garantías que se apliquen cuando el sistema entregue vistas a los miembros del grupo.

Para cada grupo g el servicio de gestión de grupos entrega a cada proceso miembro $p \in g$ una sucesión de vistas $v_0(g)$, $v_1(g)$, $v_2(g)$, etc. Una sucesión de vistas podrían ser, por ejemplo, $v_0(g) = (p)$, $v_1(g) = (p, p')$ y $v_2(g) = (p)$, esto es, p se une a un grupo vacío, posteriormente se une p' al grupo y finalmente p' lo abandona. Aunque los cambios en la pertenencia pueden ocurrir de forma concurrente, como cuando un proceso se une a un grupo justo cuanto otro se va, el sistema impone un orden en la sucesión de vistas que da a cada proceso.

Se hablará de un miembro *entregando una vista* cuando ocurra un cambio en la pertenencia y se notifique a la aplicación la nueva pertenencia, de la misma forma en que se habla de un proceso entregando un mensaje multidifundido. Y como ocurre en la entrega de multidifusión, la entrega de una vista es distinta a la recepción de una vista. Los protocolos de pertenencia a grupos mantienen las vistas propuestas en una cola de retención hasta que todos los miembros existentes se pongan de acuerdo en la entrega.

También, decimos que un evento ocurre *dentro de una vista* $v(g)$ en el proceso p si, en el momento en el que ocurra el evento, p ha entregado $v(g)$ pero todavía no ha entregado la próxima vista $v'(g)$.

Algunos requisitos básicos para la entrega de vistas son los siguientes:

Orden: **si** un proceso p entrega una vista $v(g)$ y posteriormente entrega la vista $v'(g)$, entonces ningún proceso $q \neq p$ entregará $v'(g)$ antes que $v(g)$.

Integridad: si un proceso p entrega la vista $v(g)$ entonces $p \in v(g)$.

No trivialidad: si un proceso q se une al grupo y está en contacto o puede ponerse en contacto, indefinidamente, con un proceso $p \neq q$, pasado un tiempo, q estará siempre en las vistas que entregue p. De forma similar, si el grupo se particiona y permanece particionado, entonces, al final, las vistas entregadas en cualquiera de las particiones excluirá cualquier proceso de otras particiones.

El primero de estos requisitos, de alguna forma, proporciona al programador una garantía de consistencia asegurando que los cambios en las vistas ocurren en el mismo orden en diferentes procesos. El segundo requisito es una *comprobación de consistencia*. El tercero le protege frente a soluciones triviales. Por ejemplo, un servicio de pertenencia que diga a un proceso, con independencia de su conectividad, que está en un grupo por sí mismo, no es de gran interés. La condición de no trivialidad establece que dos procesos que se hayan unido al mismo grupo podrán, al final, comunicarse indefinidamente; ambos serán considerados del mismo grupo. De forma similar, requiere que cuando se produzca una partición, el servicio de pertenencia debería reflejar finalmente la parti-

ción. La condición no establece cómo debería comportarse el servicio de pertenencia a grupos en el caso problemático de conectividad intermitente.

◇ **Comunicación en grupos con vistas síncronas.** Un sistema de comunicación en grupos con *vistas síncronas* genera garantías adicionales en lo que respecta al orden en las entregas de notificaciones de vistas con respecto al reparto de mensajes de multidifusión. La comunicación mediante vistas síncronas extiende la semántica de la multidifusión fiable, que fue descrita en el Capítulo 11, para considerar los cambios en las vistas del grupo. Con el fin de simplificar, nuestra discusión se restringirá al caso donde no puedan ocurrir particiones. Las garantías que proporciona la comunicación en grupo mediante vistas síncronas son las siguientes:

Acuerdo: los procesos correctos reparten el mismo conjunto de mensajes en cualquier vista. Dicho de otra forma, si un proceso entrega un mensaje m en una vista $v(g)$ y a continuación entrega la siguiente vista $v'(g)$, entonces todos los procesos que sobrevivan para entregar la siguiente vista $v'(g)$ (esto es, los miembros de $v(g) \cap v'(g)$) también entregan m en la vista $v(g)$.

Integridad: si un proceso p entrega un mensaje m, entonces no volverá a entregar m de nuevo. Además, $p \in grupo(m)$ y m fue proporcionado a una operación de multidifusión por el *emisor*(m) (ésta es la misma condición que en el caso de la multidifusión fiable).

Validez (*grupos cerrados*): los procesos correctos siempre entregan los mensajes que mandan. Si el sistema no es capaz de entregar un mensaje a cualquier proceso q, entonces se notifica a los procesos supervivientes mediante la entrega de una nueva vista en la que se excluye a q (inmediatamente después de la vista en la cual cualquiera de ellos entregó el mensaje). Esto es, sea p cualquier proceso correcto que entrega el mensaje m en una vista $v(g)$. Si algún proceso $q \in v(g)$ no entrega m en la vista $v(g)$, entonces la siguiente vista $v'(g)$ que entregue p tendrá que $q \notin v'(g)$.

Considérese un grupo con tres procesos p, q y r (véase la Figura 14.3). Supóngase que p envía un mensaje m mientras se está en la vista (p, q, r) pero p cae poco tiempo después de enviar m, mientras q y r siguen siendo correctos. Una posibilidad sería que p cayese antes de que m haya alcanzado cualquier otro proceso. En este caso, q y r entregan cada uno la nueva vista (q, r), pero ninguno entregará nunca m (véase la Figura 14.3a). La otra posibilidad sería que m alcance al menos a uno de los dos procesos supervivientes al caer p. Entonces, tanto q como r entregan primero m y después la vista (q, r) (véase la Figura 14.3b). No se permite que q y r entreguen primero la vista (q, r) y después m (véase la Figura 14.3c), ya que en ese caso estarían entregando un mensaje de un proceso fallido del cual habían sido informados; tampoco pueden, ninguno de los dos, entregar el mensaje y la nueva vista en órdenes distintos (véase la Figura 14.3d).

En un sistema de vistas síncronas, la entrega de una nueva vista traza una línea conceptual a lo largo del sistema, y cualquier entrega ha de ser consistente a ambos lados de esa línea. Esto permite al programador obtener conclusiones útiles acerca del conjunto de mensajes que otros procesos hayan entregado cuando entregue una nueva vista, basadas solamente en el orden local de los eventos de entrega de mensajes y de entrega de vistas.

Un ejemplo ilustrativo de la utilidad de la comunicación mediante vistas síncronas lo constituye su uso para conseguir una *transferencia de estado*: la transferencia del estado de trabajo por parte de un miembro actual de un grupo de procesos a un nuevo miembro del grupo. Por ejemplo, si los procesos son gestores de réplicas que mantienen cada uno el estado de una agenda, entonces, cuando se una un nuevo gestor de réplicas al grupo para usar esa agenda necesitará adquirir el estado actual de la agenda. Pero la agenda podría estar siendo actualizada de concurrente mientras se está capturando el estado. Es importante que el gestor de réplicas no pierda ninguno de los mensajes de actualización que no están reflejados en el estado que está adquiriendo y que no vuelva a actualizar mensajes que ya estén reflejados en el estado (a no ser que las actualizaciones sean idempotentes).

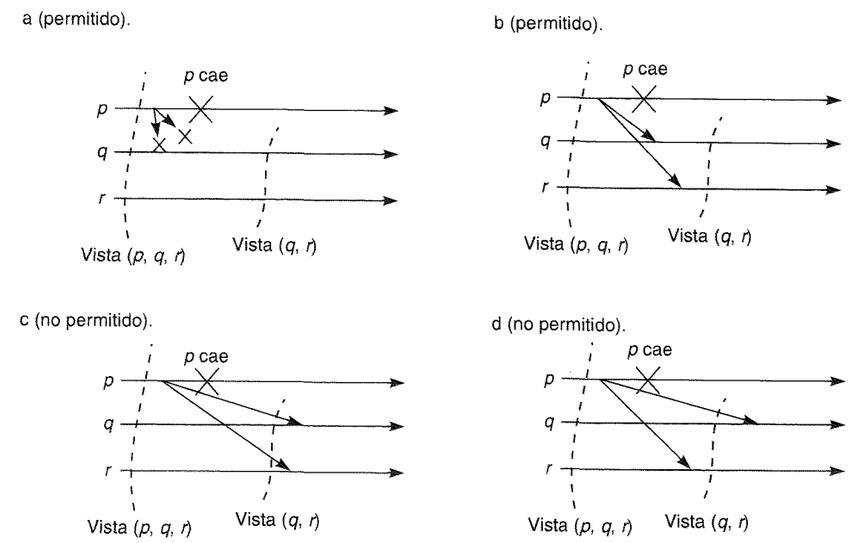


Figura 14.3. Comunicación en grupo con vista síncrona.

Para conseguir esta transferencia de estados se puede usar comunicación mediante vistas síncronas en un esquema simple como el siguiente. Al repartir la primera vista que contiene al nuevo proceso, algún proceso miembro de los ya existentes (por ejemplo el más antiguo) capta su estado, lo envía individualmente al nuevo miembro y suspende su ejecución. El resto de procesos que ya existían suspenden sus ejecuciones. Nótese que precisamente el conjunto de actualizaciones reflejadas en este estado, por definición, ha sido aplicado al resto de los miembros. Una vez recibido el estado, el nuevo proceso lo integra y multidifunde al grupo un mensaje «¡comenzad!» y, en ese momento, todos prosiguen nuevamente.

◇ **Discusión.** La noción de comunicación en grupo mediante vistas síncronas que se ha presentado es una formulación del paradigma de comunicación *virtualmente síncrona* desarrollado originalmente en el sistema ISIS [Birman 1993, Birman y otros 1991, Birman y Joseph 1987b]. En Schiper y Sandoz [1993] se describe un protocolo para conseguir la comunicación mediante vistas síncronas (o *vistas atómicas*, como ellos las denominan). Nótese que un servicio de pertenencia a grupos alcanza un consenso, pero lo hace sin despreciar el resultado de imposibilidad de Fisher y otros [1985]. Como se discutió en la Sección 11.5.4, un sistema puede burlar ese resultado usando un detector de fallos apropiado.

Schiper y Sandoz también proporcionan una versión uniforme de la comunicación mediante vistas síncronas en la cual la condición de acuerdo contempla el caso de procesos que caen. Éste es similar al acuerdo uniforme para la comunicación por multidifusión, que se describió en la Sección 11.4.2. En la versión uniforme de comunicación mediante vistas síncronas, incluso si un proceso se cae tras entregar un mensaje, todos los procesos correctos son obligados a entregar el mensaje en la misma vista. Esta garantía más fuerte se necesita, algunas veces, en aplicaciones tolerantes a fallo, ya que un proceso que ha entregado un mensaje puede haber tenido un efecto en el mundo exterior antes de caer. Por la misma razón, Hadzilacos y Toueg [1994] tratan las versiones uniformes de los protocolos de multidifusión fiables y ordenados que se describieron en el Capítulo 11.

El sistema V [Cheriton y Zwaenepoel 1985] fue el primer sistema en dar soporte a grupos de procesos. Tras ISIS, se han desarrollado los grupos de procesos con algún tipo de servicio de pertenencia a grupos en distintos sistemas, incluyendo Horus [van Renesse y otros 1996], Totem [Moser y otros 1996] y Transis [Dolev y Malki 1996].

Se han propuesto variaciones en la sincronía de las vistas para servicios de pertenencia a grupos particionables, incluyendo apoyo para aplicaciones atentas a particiones [Babaoglu y otros 1998] y para sincronía virtual extendida [Moser y otros 1994].

Finalmente, en Cristian [1991b] se discute un servicio de pertenencia a grupos para sistemas distribuidos *síncronos*.

◇ **Grupos de objetos.** Los grupos de objetos proporcionan una aproximación orientada a objetos a la computación en grupo. Un grupo de objetos es una colección de objetos (normalmente, instancias de la misma clase) que procesan el mismo conjunto de invocaciones de forma concurrente y devuelven sus respuestas. Los objetos cliente no necesitan estar al tanto de la replicación. Ellos invocan operaciones en un objeto local individual que actúa como un proxy para el grupo. El proxy utiliza un sistema de comunicación en grupo para enviar las invocaciones a los miembros del grupo de objetos.

Electra [Maffeis 1995] es un sistema conforme con CORBA que soporta grupos de objetos. Un grupo en Electra puede ser dotado de una interfaz para ser usado con cualquier aplicación conforme con CORBA. Electra fue construido originalmente sobre el sistema de comunicación en grupos Horus, al cual utiliza para gestionar la pertenencia al grupo y para multidifundir las invocaciones. En el *modo transparente*, el proxy local devuelve al objeto cliente la primera respuesta disponible. En el *modo no transparente*, el objeto cliente puede acceder a todas las respuestas devueltas por los miembros del grupo. Electra usa una extensión de la interfaz estándar ORB de CORBA, con funciones para crear y destruir grupos de objetos y para gestionar su pertenencia.

El sistema Eternal [Moser y otros 1998] y el Servicio de Grupos de Objetos (*Object Group Service*) [Guerraoui y otros 1998] también proporcionan soporte de grupos de objetos compatible con CORBA.

14.3. SERVICIOS TOLERANTES A FALLOS

En esta sección se examina cómo proporcionar un servicio que sea correcto, a pesar de la presencia de f procesos que fallan, mediante la replicación de datos y otras funcionalidades en los gestores de réplicas. Para simplificar, asumimos que la comunicación es fiable durante todo el proceso y no ocurren particiones.

Se supone que cada gestor de réplicas se comporta de acuerdo a la especificación asociada a la semántica de los objetos que gestiona, siempre que no hayan caído. Por ejemplo, una especificación de cuentas bancarias podría incluir la seguridad de que los fondos que se transfieren entre cuentas bancarias nunca pueden desaparecer, y que sólo los depósitos y las extracciones de fondos pueden afectar al balance de una cuenta en concreto.

De forma intuitiva, un servicio que esté basado en la replicación se considera correcto si es capaz de responder a pesar de los fallos y si los clientes no pueden percibir la diferencia entre el servicio que obtienen de una implementación con datos replicados y otra proporcionada por un único gestor de réplicas correcto. Es necesario tener cuidado para que se cumpla este criterio. Si no se toman precauciones, pueden surgir anomalías cuando coexistan distintos gestores de réplicas; incluso si tenemos en cuenta que sólo consideramos los efectos de operaciones individuales, no de transacciones.

Considere un sistema ingenuo de replicación, en el que dos gestores de réplicas en los computadores A y B mantienen réplicas de dos cuentas bancarias x e y. Los clientes leen y actualizan las cuentas en sus gestores de réplicas locales, pero, si éstos fallan, utilizan otros. Los gestores de réplicas propagan las actualizaciones entre ellos en segundo plano tras responder al cliente. Inicialmente, ambas cuentas tienen un balance de 0 dólares.

El cliente 1 actualiza a 1 dólar el balance de x en su gestor de réplicas local B y, a continuación, intenta actualizar el balance de y para que sea de 2 dólares, pero descubre que B ha fallado. A consecuencia de esto, el cliente 1 realiza la actualización en A en vez de en B. En ese momento, el cliente 2 lee sus balances en su gestor de réplicas local A y se encuentra que y tiene 2 dólares y que x tiene 0 dólares (la actualización de la cuenta bancaria x por parte de B no ha llegado dado que B falló). La situación se muestra a continuación, etiquetándose las operaciones por parte del computador en el cual tuvieron lugar por primera vez y donde las operaciones que están más abajo sucedieron más tarde:

Cliente 1:	Cliente 2:
$ponBalance_B(x, 1)$	
$ponBalance_A(y, 2)$	
	$obténBalance_A(y) \rightarrow 2$
	$obténBalance_A(x) \rightarrow 0$

Esta ejecución no se ajusta a una especificación dictada por el sentido común para el comportamiento de cuentas bancarias: el cliente 2 debería haber obtenido un balance de 1 dólar para x, puesto que obtiene el balance de 2 dólares para y, y el balance de y se actualizó después del de x. Este comportamiento anómalo en el caso que utiliza la replicación no se habría dado si se hubiera implementado utilizando un único servidor. Se pueden construir sistemas que gestionen objetos replicados sin los comportamientos anómalos asociados al protocolo ingenuo del ejemplo anterior. Primero, debemos entender qué se considera comportamiento correcto para un sistema replicado.

◇ **Linealizabilidad y consistencia secuencial.** Hay varios criterios de corrección para objetos replicados. Los sistemas más estrictamente correctos son *linealizables* y a esta propiedad se denomina *linealizabilidad*. De cara a comprender la linealizabilidad considere una implementación de servicio replicado con dos clientes. Sea o_{i0}, o_{i1}, o_{i2}, ... una secuencia de operaciones de lectura y actualización que el cliente i realiza en cierta ejecución. Cada operación o_{ij} de estas secuencias se especifica mediante el tipo de operación, sus argumentos y los valores que devuelve según se producen en tiempo de ejecución. Se supone que cada operación es síncrona. Esto es, los clientes aguardan a que se complete una operación antes de solicitar la siguiente.

Un único servidor que gestionase una única copia de los objetos podría secuenciar las operaciones de los clientes. En el caso de una ejecución sólo con los clientes 1 y 2, este entrelazado de las operaciones podría ser, por ejemplo, o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, ... Definimos nuestro criterio de corrección para objetos replicados haciendo referencia a un entrelazamiento *virtual* de las operaciones de los clientes, que no tiene necesariamente que ocurrir físicamente en un gestor de réplicas concreto, pero que establece la corrección de la ejecución.

Un servicio de objetos compartidos replicados se dice linealizable si *para cualquier ejecución* existe algún entrelazamiento de las series de operaciones emprendidas por cada cliente que satisface los dos siguientes criterios:

- La secuencia entrelazada de operaciones cumple la especificación de una (única) copia correcta de los objetos.
- El orden de las operaciones del entrelazamiento es consistente con los tiempos reales en los cuales ocurrieron las operaciones en la ejecución real.

Esta definición plasma la idea de que para cualquier conjunto de operaciones de los clientes existe una ejecución canónica virtual (las operaciones intercaladas a las que se refiere la definición) sobre

una única imagen virtual de los objetos compartidos. Y que cada cliente observa una vista de los objetos compartidos que es consistente con esa única imagen: esto es, los resultados de las operaciones del cliente tienen sentido según ocurran dentro del entrelazamiento.

El servicio que dio lugar a la ejecución de los clientes de las cuentas bancarias en el ejemplo anterior no es linealizable. Incluso si se ignoran los tiempos reales en los que ocurrieron las operaciones, no existe un entrelazado de las operaciones de los dos clientes que pudiera satisfacer cualquier especificación correcta de cuentas bancarias: con vistas a un examen, si la actualización de una cuenta sucedió tras otra, entonces habrá de observarse la primera actualización si es que se observa la segunda.

Observe que la linealizabilidad tiene que ver solamente con el entralazamiento de operaciones individuales y que no está pensado que sea transaccional. Una ejecución linealizable podría romper las nociones de consistencia específicas de una aplicación si no se aplica un control de concurrencia.

El requisito de capacidad de secuenciación en tiempo real es deseable en un mundo ideal, dado que capta nuestra noción de que los clientes deberían recibir información actualizada. Sin embargo, la presencia del tiempo real en la definición plantea la cuestión de la practicabilidad de la linealizabilidad, ya que no siempre se pueden sincronizar los relojes con el grado de precisión requerido. Una condición de corrección más débil es la *consistencia secuencial*, que capta un requisito esencial con respecto al orden en el cual se procesan las peticiones, sin aludir al tiempo real. Su definición mantiene el primer criterio de la definición de capacidad de secuenciación, pero modifica el segundo de la siguiente manera:

Un servicio de objetos replicados compartidos se dice que es secuencialmente consistente si *para cualquier ejecución* existe algún entrelazamiento de las series de operaciones emprendidas por los clientes que satisfaga los dos siguientes criterios:

- La secuencia de operaciones intercaladas cumple la especificación de una (única) copia correcta de los objetos.
- El orden de las operaciones en el entrelazamiento es consistente con el orden en el programa con el cual cada cliente individual ejecutó dichas operaciones.

Ha de notarse que en esta definición no hay mención de tiempos absolutos. Ni tampoco a cualquier orden *total* sobre todas las operaciones. La única noción relevante de ordenación es el orden de los eventos en cada cliente por separado, esto es, el orden en su programa. El entrelazado de las operaciones puede barajar en cualquier orden la secuencia de operaciones que provienen de un conjunto de clientes, siempre y cuando el orden de cada cliente no se viole y el resultado de cada operación sea consistente con las operaciones que la han precedido, en términos de las especificaciones de los objetos. Esto sería parecido a mezclar varios mazos de cartas de modo que las cartas se entremezclasen de forma tal que se mantuviese el orden inicial de cada baraja.

Cada servicio linealizable es, asimismo, secuencialmente consistente, ya que la ordenación por tiempo real refleja el orden dentro del programa de cada cliente. Lo contrario no se cumple. A continuación se muestra un ejemplo de ejecución para un servicio que es secuencialmente consistente pero no linealizable:

Cliente 1:	Cliente 2:
$ponBalance_B(x, 1)$	
	$obténBalance_A(y) \rightarrow 0$
	$obténBalance_A(x) \rightarrow 0$
$ponBalance_A(y, 2)$	

Esta ejecución es posible bajo una estrategia ingenua de replicación, incluso si ninguno de los computadores A o B falla menos si la actualización de x que hace el cliente 1 en B no haya alcanzado A cuando el cliente 2 lo lee. El criterio de tiempo real para la secuenciación no se satisface, ya que $dameBalance_A(x) \rightarrow 0$ sucede más tarde que $ponBalance_B(x, 1)$; sin embargo, el entrelazamiento siguiente satisface los dos criterios para la consistencia secuencial: $dameBalance_A(y) \rightarrow 0$, $dameBalance_A(x) \rightarrow 0$, $ponBalance_B(x, 1)$, $ponBalance_A(y, 1)$.

Lamport concibió ambos términos: consistencia secuencial [1979] y de capacidad de secuenciación [1986], en relación con los registros de memoria compartida (aunque él utilizó el término *atomicidad* en lugar de *linealizabilidad*). Herlihy y Wing [1990] generalizaron la idea para cubrir cualquier tipo de objetos compartidos. El Capítulo 16, que examina la memoria compartida distribuida, definirá y discutirá algunas propiedades de consistencia más débiles.

14.3.1. REPLICACIÓN PASIVA (PRIMARIO-RESPALDO)

En el modelo de replicación *pasiva* o *primario-respaldo* para la tolerancia a fallos (véase la Figura 14.4), en cada instante existe un único gestor de réplicas primario y uno o más gestores de réplicas secundarios (*respaldos* o *esclavos*). En el modelo puramente formal, los frontales se comunican exclusivamente con el gestor de réplicas primario para obtener el servicio. El gestor de réplicas primario ejecuta las operaciones y envía copias de los datos actualizados a los gestores de respaldo. Si el primario falla, uno de los secundarios se promociona para que actúe como gestor primario.

Cuando un cliente solicita que se realice una operación, la secuencia de eventos es la siguiente:

1. *Petición*: el frontal lanza la petición, que contiene un identificador único, al gestor de réplicas primario.
2. *Coordinación*: el primario acepta cada petición de forma atómica, en el orden en que las recibe. Comprueba el identificador único y, en el caso en el que ya hubiera ejecutado la petición, simplemente reenvía la respuesta.
3. *Ejecución*: el primario ejecuta la petición y guarda la respuesta.
4. *Acuerdo*: si la petición es una actualización, entonces el primario envía a todas las copias de seguridad el estado actualizado, la respuesta y el identificador único. Los respaldos envían, a su vez, un acuse de recibo.
5. *Respuesta*: el primario responde al frontal, que proporciona la respuesta de vuelta al cliente.

Obviamente, este sistema implementa la linealizabilidad si el gestor primario es correcto, dado que éste secuencia todas las operaciones sobre los objetos compartidos. Si el primario falla, entonces el sistema conserva la capacidad de secuenciación si sólo un respaldo se convierte en el nuevo primario y si la nueva configuración del sistema toma el control exactamente donde lo dejó el anterior:

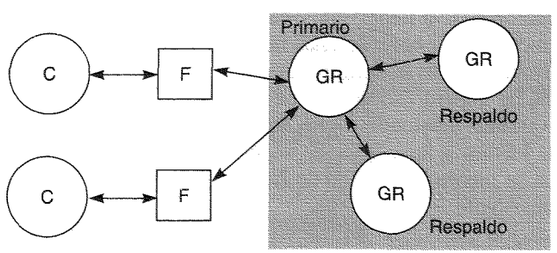


Figura 14.4. El modelo pasivo (primario-respaldo) para tolerancia a los fallos.

- El primario es reemplazado por un único respaldo (si dos clientes empezaran a utilizar dos respaldos, entonces el sistema podría comportarse de forma incorrecta).
- Los gestores de réplicas que sobreviven se ponen de acuerdo en qué operaciones se habían realizado en el momento en el que el reemplazo del primario tomó el control.

Ambos requisitos se satisfacen si los gestores de réplicas (primario y respaldos) están organizados como un grupo y si el primario utiliza comunicación en grupos con vistas síncronas para enviar las actualizaciones a los respaldos. El primero de los dos requisitos se satisface fácilmente. Cuando caiga el primario, el sistema de comunicación repartirá, en algún momento, una nueva vista a los respaldos supervivientes, donde se excluye el antiguo primario. El respaldo que reemplaza al primario puede elegirse mediante alguna función sobre esa vista. Por ejemplo, los respaldos pueden elegir como sustituto al primer miembro de esa vista. Dicho respaldo puede registrarse como primario bajo un nombre de servicio al que los clientes consultan cuando sospechan que el primario ha fallado (o cuando soliciten el servicio por primera vez).

También se satisface el segundo requisito, gracias a la propiedad de ordenamiento de la sincronización de vistas y al uso de los identificadores almacenados para detectar peticiones repetidas. La semántica de las vistas síncronas garantiza que o bien todas las copias de seguridad o ninguna de ellas entregarán una actualización concreta antes de entregar la nueva vista. Así, el nuevo primario y los respaldos supervivientes están de acuerdo en si cierta actualización de un cliente ha sido o no procesada.

Considere un frontal que no haya recibido una respuesta. El frontal retransmite la petición a cualquier respaldo que haya tomado el control como primario. El primario puede haber caído en cualquier instante de la operación. Si cayó antes de la etapa de acuerdo (4), entonces los gestores de réplicas supervivientes no habrán podido procesar la petición. Si cayó durante la etapa de acuerdo, las réplicas pueden haber procesado la petición. Si cayó tras esta etapa, entonces seguro que la habrán procesado. Sin embargo, el nuevo primario no tiene que saber en qué etapa estaba el antiguo primario cuando cayó. Cuando recibe una petición, prosigue a partir de la etapa 2 mencionada anteriormente. Gracias a la sincronía de vistas, no es necesario consultar con los respaldos, puesto que todos han procesado el mismo conjunto de mensajes.

◇ **Discusión sobre la replicación pasiva.** El modelo primario-respaldo, puede ser usado incluso donde los gestores de réplicas primarios se comporten de una forma *no determinista*, por ejemplo debido a la utilización de operaciones multihilo. Dado que el gestor primario comunica el estado actualizado resultante de las operaciones, en lugar de especificar las operaciones en sí mismas, los respaldos registran sumisamente el estado determinado únicamente por las acciones del primario.

Para sobrevivir a un máximo de f caídas de procesos, un sistema de replicación pasiva requiere $f + 1$ gestores de réplica (tales sistemas no toleran fallos extraños —bizantinos—). El frontal necesita poca funcionalidad para lograr la tolerancia a fallos. Necesita ser capaz de encontrar al nuevo primario cuando no responda el primario actual.

La replicación pasiva tiene la desventaja de producir sobrecargas relativamente grandes. La comunicación mediante vistas síncronas necesita varias rondas de comunicación por multidifusión, y si falla el primario todavía se introduce mayor latencia mientras el sistema de comunicación en grupo se pone de acuerdo sobre una nueva vista y la entrega.

En una variante del modelo que aquí se presenta, los clientes pueden ser capaces de enviar peticiones de lectura a las copias de seguridad, descargando así de trabajo al gestor primario. En consecuencia, se pierde la garantía de linealizabilidad, pero los clientes reciben un servicio secuencialmente consistente.

La replicación pasiva se utiliza en el sistema de archivos replicados Harp [Liskov y otros 1991]. El Servicio de Información en Red de Sun (*Network Information Service*, NIS, antiguamente *Yellow Pages*) emplea replicación pasiva para obtener una alta disponibilidad y un buen rendimiento, aunque con garantías más débiles que la consistencia secuencial. Las garantías de consistencia más débiles son todavía satisfactorias para muchos fines, tales como el almacenamiento de ciertos tipos de registros de administración del sistema. Los datos replicados se actualizan en un servidor maestro y se propaga desde ahí a los servidores esclavos utilizando una comunicación *uno a uno* (en lugar de la comunicación en grupo). Para recuperar información, los clientes pueden comunicarse bien con un servidor maestro, bien con uno esclavo. En NIS, sin embargo, los clientes no pueden solicitar actualizaciones: las actualizaciones se hacen vía archivos maestros.

14.3.2. REPLICACIÓN ACTIVA

En el modelo de replicación *activa* para tolerancia a fallos (véase la Figura 14.5), los gestores de réplicas son máquinas de estado que desempeñan papeles equivalentes y se organizan como un grupo. Los frontales multidifunden sus peticiones al grupo de gestores de réplicas y todos los gestores de réplicas procesan la petición de forma independiente, pero idéntica, y responden. Si cae cualquier gestor de réplicas, no tiene necesariamente impacto en las prestaciones del servicio, puesto que los restantes gestores de réplicas continúan respondiendo de la forma habitual. Veremos que la replicación activa puede tolerar fallos bizantinos (extraños), puesto que el frontal puede recopilar y comparar las respuestas que recibe.

Bajo replicación activa, la secuencia de eventos que se desarrolla cuando un cliente solicita que se efectúe una operación es como sigue:

1. *Petición*: el frontal adjunta un identificador único a la petición y la multidifunde al grupo de gestores de réplicas, usando una primitiva de multidifusión fiable y totalmente ordenado. Se asume que, en el peor de los casos, el frontal puede fallar por caída. Además, no emitirá la siguiente petición hasta que haya recibido una respuesta.
2. *Coordinación*: el sistema de comunicación en grupo reparte la petición a cada gestor de réplicas correcto en el mismo orden (total).
3. *Ejecución*: cada gestor de réplicas ejecuta la petición. Puesto que son máquinas de estado, y las peticiones se entregan en el mismo orden total, todos los gestores de réplicas correctos procesan idénticamente la petición. La respuesta contiene el identificador único de la petición del cliente.
4. *Acuerdo*: no se necesita fase de acuerdo, debido a la semántica de reparto de la multidifusión.
5. *Respuesta*: cada gestor de réplicas envía su respuesta al frontal. El número de respuestas que recoge el frontal depende de las suposiciones de fallo y del algoritmo de multidifusión.

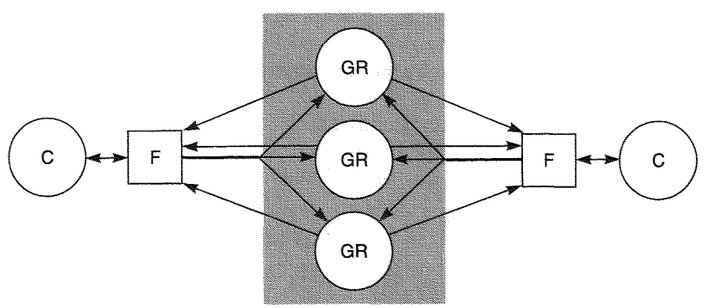


Figura 14.5. Replicación activa.

Por ejemplo, si el objetivo es tolerar sólo fallos por caída y la multidifusión satisface las propiedades de ordenación y acuerdo uniformes, entonces el frontal pasa al cliente la primera respuesta que llegue y descarta el resto (puede distinguir éstas de las respuestas a otras peticiones examinando el identificador asociado a la respuesta).

Este sistema consigue la consistencia secuencial. Todos los gestores de réplicas correctos procesan la misma secuencia de peticiones. La fiabilidad de la multidifusión asegura que todo gestor de réplicas correcto procesa el mismo conjunto de peticiones, y el orden total asegura que las procesan en el mismo orden. Dado que son máquinas de estados, todos finalizan en el mismo estado tras cada petición. Las peticiones de cada frontal son servidas en orden FIFO (porque el frontal aguarda por una respuesta antes de hacer la siguiente petición), que es lo mismo que *el orden del programa*. Esto asegura la consistencia secuencial.

Si los clientes no se comunican con otros clientes mientras esperan por respuestas a sus peticiones, entonces éstas se procesan en orden *sucedió antes*. Si los clientes disponen de multi-hilo y pueden comunicarse con otros mientras esperan respuestas por parte del servicio, entonces para garantizar el procesado de peticiones con ordenación *sucedió antes* debería reemplazarse la multidifusión por una que esté ordenada a la vez total y causalmente.

El sistema de replicación activa no consigue linealizabilidad. Esto se debe a que el orden total, en el cual los gestores de réplicas procesan las peticiones, no es necesariamente el mismo que el orden impuesto por el tiempo real en el que los clientes hicieron sus peticiones. Schneider [1990] describe cómo, en un sistema síncrono con relojes sincronizados aproximadamente, el orden total en el que los gestores de réplicas procesan las peticiones puede basarse en el orden de las marcas temporales físicas que proporcionarían los frontales junto a sus peticiones. Esto no garantiza la linealizabilidad, puesto que las marcas temporales no son perfectamente precisas; pero se aproximan a ella.

◇ **Discusión sobre la replicación activa.** Hemos supuesto una solución para la multidifusión totalmente ordenada y fiable. Como se apuntó en el Capítulo 11, resolver la multidifusión fiable y totalmente ordenada es equivalente a resolver el problema del consenso. A su vez, esto requiere que el sistema sea síncrono o que se utilice una técnica como los detectores de fallos en un sistema asíncrono, para soslayar el resultado de imposibilidad de Fischer y otros [1985].

Algunas soluciones al problema del consenso, como la de Canetti y Rabin [1993], funcionan incluso bajo la suposición de fallos extraños. A partir de esa solución y, por lo tanto, la solución a la multidifusión fiable y totalmente ordenada, el sistema de replicación activa puede enmascarar hasta f fallos extraños, siempre que el servicio incorpore como mínimo $2f + 1$ gestores de réplicas. Cada frontal espera a recoger $f + 1$ respuestas idénticas y entrega la respuesta al cliente. Descarta otras respuestas a la misma petición. Para estar totalmente seguros de qué respuesta está realmente asociada con qué petición se requiere que los gestores de réplicas firmen digitalmente sus respuestas (dado un comportamiento bizantino).

Sería posible relajar las especificaciones del sistema que se ha descrito. Primero, hemos asumido que todas las actualizaciones sobre los objetos replicados compartidos deben de suceder en el mismo orden. Sin embargo, en la práctica, algunas operaciones pueden conmutarse: esto es, que el efecto de dos operaciones realizadas en el orden o_1; o_2 sea el mismo que en el orden inverso o_2; o_1. Por ejemplo, cualesquiera dos operaciones de *sólo lectura* (por parte de diferentes clientes) conmutan; y cualesquiera dos operaciones que no hacen lecturas sino que actualizan objetos distintos también conmutan. Un sistema de replicación activa debe ser capaz de explotar el conocimiento sobre conmutatividad para evitar el gasto de ordenar todas las peticiones. En el Capítulo 11, se apuntó que algunos autores han propuesto semánticas de ordenación de multidifusión específicas de la aplicación [Cheriton y Skeen 1993, Pedone y Schiper 1999].

Finalmente, los frontales pueden enviar peticiones de *sólo lectura* únicamente a gestores de réplicas individuales. Haciendo esto, pierden la tolerancia a fallos que proporciona el multidifundir las peticiones, pero el servicio mantiene la consistencia secuencial. Además, el frontal puede fácilmente enmascarar el fallo de un gestor de réplicas en esta situación, simplemente enviando la petición de *sólo lectura* a otro gestor de réplicas.

14.4. SERVICIOS CON ALTA DISPONIBILIDAD

En lo que resta de capítulo, se considerará cómo aplicar las técnicas de replicación para conseguir una alta disponibilidad en los servicios. Nuestro énfasis está ahora en proporcionar a los clientes el acceso al servicio (con tiempos de respuesta razonables) durante el mayor tiempo posible, incluso si algunos resultados no cumplen la consistencia secuencial. Por ejemplo, el usuario en el tren al comienzo del capítulo podría admitir trabajar con inconsistencias temporales entre las copias de los datos, tales como agendas, si puede continuar trabajando mientras está desconectado y puede corregir los problemas después.

En la Sección 14.3, se vio que los sistemas tolerantes a fallos transmiten de modo *acuciante* (*eager*) las actualizaciones a los gestores de réplicas: todos los gestores de réplicas correctos reciben las actualizaciones lo antes posible y llegan a un acuerdo colectivo antes de devolver el control al cliente. Este comportamiento no es deseable para operaciones con alta disponibilidad. En su lugar, el sistema debería proporcionar un nivel de servicio aceptable utilizando un conjunto mínimo e gestores de réplicas conectados al cliente. Y debería minimizar el tiempo que el cliente está obligado a esperar mientras los gestores de réplicas coordinan sus actividades. Generalmente, los grados de consistencia más débiles requieren un menor grado de acuerdo y permiten que los datos compartidos estén disponibles durante más tiempo.

A continuación se examinan distintos sistemas que proporcionan servicios con alta disponibilidad: sistemas de cotilleo, Bayou y Coda.

14.4.1. LA ARQUITECTURA COTILLA

Ladin y otros [1992] desarrollaron lo que en adelante se llamará la *arquitectura cotilla*, o *de cotilleo*, como un marco de referencia para implementar servicios con alta disponibilidad mediante la replicación de datos en puntos cercanos a los lugares donde los grupos de clientes los necesitan. El nombre refleja el hecho de que los gestores de réplicas intercambian periódicamente mensajes de *cotilleo* para transmitir las actualizaciones que ha recibido cada uno por parte de sus clientes (véase la Figura 14.6). La arquitectura está basada en un trabajo anterior en bases de datos de Fischer y Michael [1982] y Wuu y Bernstein [1984]. Puede usarse, por ejemplo, para un tablón de anuncios electrónico altamente disponible o un servicio de dietario.

Un servicio cotilla proporciona dos tipos básicos de operaciones: *preguntas* que son operaciones de *sólo lectura* y *actualizaciones* que modifican pero no leen el estado (ésta es una definición más restrictiva que la que se venía usando). Una característica importante es que los frontales envían preguntas y actualizaciones a cualquier gestor de réplicas que elijan (cualquiera que esté disponible y proporcione tiempos de respuesta razonables). El sistema garantiza dos cosas, incluso aunque los gestores de réplicas no puedan comunicarse temporalmente unos con otros:

Cada cliente consigue un servicio consistente a lo largo del tiempo: en respuesta a una pregunta, los gestores de réplicas siempre proporcionan al cliente datos que reflejan al menos las actualizaciones que hubiese observado hasta ese momento. Esto es así incluso aunque los clientes puedan comunicarse con distintos gestores de réplicas en instantes de tiempo diferentes y, por lo tanto, en principio, podrían comunicarse con un gestor de réplicas que estuviese *menos avanzado* que con otro con el que se comunicaron antes.

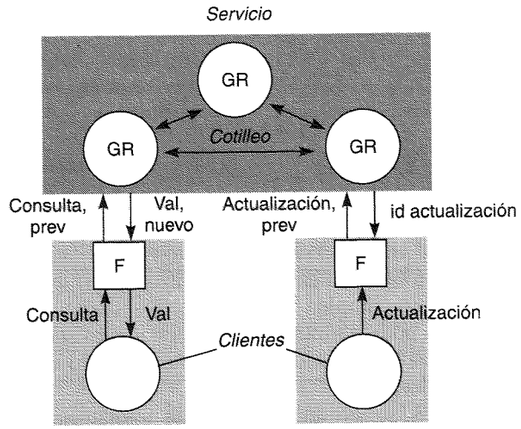


Figura 14.6. Operaciones de consulta y actualización en un servicio cotilla.

Consistencia relajada entre réplicas: todos los gestores de réplicas, en algún momento, recibirán todas las actualizaciones y las aplican con garantías de ordenación que hacen a las réplicas lo suficientemente parecidas como para cumplir las necesidades de la aplicación. Es importante darse cuenta de que aunque la arquitectura de cotilleo pueda usarse para obtener consistencia secuencial, está pensada inicialmente para proporcionar garantías de consistencia más débiles. Dos clientes pueden observar réplicas distintas aunque las réplicas incluyan el mismo conjunto de actualizaciones; y un cliente pueda observar datos anticuados.

Para proporcionar la consistencia relajada, la arquitectura cotilla proporciona actualización con orden *causal*, tal y como se definió en la Sección 14.2.1. Además, aporta garantías de ordenación más fuertes en la forma de orden *forzado* (total y causal) e *inmediato*. Las actualizaciones *inmediatamente ordenadas* se aplican en un orden consistente en relación con *cualquier* otra actualización en todos los gestores de réplicas, ya sean las otras ordenaciones de actualización causales, forzadas o inmediatas. Se proporciona el *orden inmediata* además del *orden forzado* porque una actualización con *orden forzado* y una actualización con *orden causal* que no estén relacionadas mediante la relación *sucedió antes* pueden ser aplicadas en distintos órdenes en diferentes gestores de réplicas.

Se deja al diseñador de la aplicación la elección de qué ordenación utilizar, y refleja un compromiso entre la consistencia y los costes de la operación. Las actualizaciones causales son considerablemente menos costosas que las otras y es de esperar que se utilicen siempre que sea posible. Obsérvese que las preguntas, que pueden ser satisfechas por cualquier gestor de réplicas individual se ejecutan siempre en orden causal con respecto a otras operaciones.

Considere una aplicación tablón de anuncios electrónico, en el cual un programa cliente (que incorpora al frontal) se ejecuta en el computador del usuario y se comunica con un gestor de réplicas local. El cliente envía los anuncios del usuario al gestor de réplicas local y éste envía los nuevos anuncios mediante cotilleos a otros gestores de réplicas. Los lectores de los tablones de anuncios perciben listas de anuncios ligeramente atrasadas, pero esto generalmente no es importante si el retraso es del orden de minutos u horas en lugar de días. Se puede utilizar un orden causal para depositar un anuncio. Esto, en general, significaría que los anuncios podrían aparecer con distintas ordenaciones en gestores de réplicas diferentes, pero que, por ejemplo, un anuncio cuyo asunto es *Re*: *naranjas* siempre se anunciará tras el mensaje sobre *naranjas* al cual se refiere. La ordenación forzada podría utilizarse para añadir un nuevo suscriptor al tablón de anuncios, de tal forma que

exista un registro no ambiguo del orden en el cual los usuarios se unieron al grupo. La ordenación inmediata podría utilizarse para retirar a un usuario de una lista de suscripción a un tablón de anuncios, de tal forma que los mensajes no puedan ser retirados por el usuario mediante algún gestor de réplicas lento una vez que la operación de eliminación haya terminado.

El frontal para un servicio cotilla manipula las operaciones que el cliente realiza utilizando una API específica de la aplicación y las convierte en operaciones de cotilleo. Por lo general, las operaciones de un cliente pueden bien leer el estado replicado, bien modificarlo o ambas cosas. Ya que en las actualizaciones cotillas únicamente se modifica el estado, el frontal convierte una operación que lee y modifica el estado en una pregunta y una actualización separadas.

En términos de nuestro modelo básico de replicación, se esboza, a continuación, cómo un servicio cotilla procesa las operaciones de pregunta y actualización:

1. *Petición*: El frontal normalmente envía peticiones sólo a un gestor de réplicas de cada vez. No obstante, un frontal se comunicará con un gestor de réplicas diferente cuando el que utiliza normalmente falla o se vuelve inaccesible, e incluso puede intentar uno o más si el gestor habitual se encuentra muy cargado. Los frontales y, por lo tanto, los clientes, pueden bloquearse en operaciones de pregunta. Por otro lado, la decisión por defecto para operaciones de actualización consiste en volver al cliente tan pronto como la operación haya sido pasada al frontal; a continuación, éste propaga la operación en segundo plano. Alternativamente, y para incrementar la fiabilidad, se podría impedir al cliente continuar hasta que la actualización haya sido entregada a $f + 1$ gestores de réplicas y, por lo tanto, puede entregarse en cualquier destino a pesar de un máximo de f fallos.
2. *Respuesta a una actualización*: Si la petición es una actualización, entonces el gestor de réplicas responde tan pronto como haya recibido la actualización.
3. *Coordinación*: El gestor de réplicas que recibe una petición no la procesa hasta que pueda aplicar la petición de acuerdo con las restricciones de ordenación que se requieran. Esto puede involucrar la recepción de actualizaciones por parte de otros gestores de réplicas mediante mensajes de cotilleo. No se realiza ninguna otra coordinación entre los gestores de réplicas.
4. *Ejecución*: El gestor de réplicas ejecuta la petición.
5. *Respuesta a la pregunta*: Si la petición es una pregunta, el gestor de réplicas se responde en este momento.
6. *Acuerdo*: Los gestores de réplicas se actualizan entre ellos intercambiando *mensajes de cotilleo*, que contienen las actualizaciones más recientes que hayan recibido. Se dice que se actualizan unos a otros de una forma *perezosa*, ya que pueden intercambiarse los mensajes cotillas sólo ocasionalmente tras recoger varias actualizaciones; o cuando un gestor de réplicas se da cuenta de que ha perdido una actualización enviada a uno de sus iguales y que la necesita para procesar una petición.

A continuación describiremos el sistema cotilla con más detalle. Comenzaremos considerando las estructuras de datos y marcas temporales que mantienen los frontales y los gestores de réplicas para preservar las garantías de ordenamiento en las actualizaciones. A continuación, a partir de estos, explicaremos cómo los gestores de réplicas procesan las preguntas y las actualizaciones. La mayor parte del procesamiento de los vectores de marcas temporales necesarias para mantener las actualizaciones causales es parecido al del algoritmo de multidifusión causal de la Sección 11.4.3.

◇ **Las marcas temporales de la versión del frontal.** Para controlar la ordenación en el procesamiento de operaciones, cada frontal guarda un vector de marcas temporales que refleja la versión de los últimos valores de los datos a los que accedió (y a los que, por lo tanto, accedió el cliente). Estas marcas temporales, denominadas *prev* en la Figura 14.6, contienen una entrada para cada gestor de réplicas. El frontal los envía en cada mensaje de petición a un gestor de réplicas junto con una descripción de la operación de pregunta o actualización. Cuando un gestor de

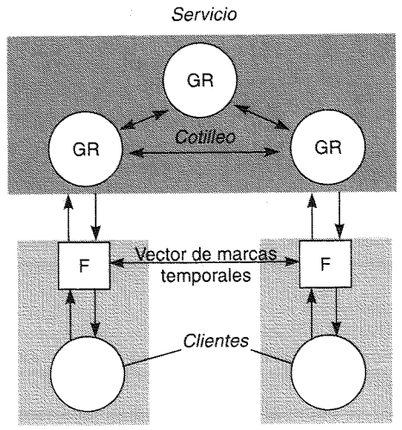


Figura 14.7. Los frontales propagan sus marcas temporales siempre que los clientes se comunican.

réplicas devuelve un valor como resultado de una operación de pregunta, proporciona un nuevo vector de marcas temporales (*nuevo* en la Figura 14.6), ya que las réplicas pueden haber sido actualizadas desde la última operación. De forma similar, una operación de actualización devuelve un vector de marcas temporales (*id actualización* en la Figura 14.6), que es único para la actualización. Cada vector de marcas temporales devuelto se fusiona con el vector de marcas temporales anterior existente en el frontal para registrar la versión de los datos replicados que han sido observados por el cliente (véase la Sección 10.4 para una definición de la fusión de vectores de marcas temporales).

Los clientes intercambian datos accediendo al mismo servicio cotilla y comunicándose directamente entre ellos. Dado que la comunicación *cliente a cliente* puede, además, dar lugar a relaciones causales entre las operaciones que se aplican al servicio, esta comunicación también se hace a través de los frontales de los clientes. De esta forma, los frontales pueden adherir sus vectores de marcas temporales sobre los mensajes hacia otros clientes. Los receptores los fusionan con sus propias marcas temporales para que puedan inferirse las relaciones causales de forma correcta. La situación se muestra en la Figura 14.7.

◇ **Estado del gestor de réplicas.** Con independencia de la aplicación, un gestor de réplicas contiene los siguientes componentes de estado principales (Figura 14.8):

Valor: éste es el valor del estado de la aplicación tal y como lo mantiene el gestor de réplicas. Cada gestor de réplicas es una máquina de estado que empieza con un valor inicial especificado y que es, de ahí en adelante, exclusivamente el resultado de aplicar las operaciones de actualización a ese estado.

Valores de las marcas temporales: éste es el vector de marcas temporales que representa los tiempos de las actualizaciones que se reflejan en el componente valor. Contiene una entrada para cada gestor de réplicas. Se actualiza siempre que se realiza una operación de actualización sobre el componente valor.

Registro histórico de actualizaciones: todas las operaciones de actualización se graban en este registro tan pronto como se reciben. Un gestor de réplicas mantiene las actualizaciones en un registro histórico por una de dos razones. La primera es que el gestor de réplicas no puede todavía aplicar la actualización porque no es *estable*. Una actualización estable es aquella que puede aplicarse consistentemente con sus garantías de ordenación (causal, forzada o inmediata). Una actualización que no es aún estable debe ser retenida y no procesarse aún. La segunda razón para mantener una actualización en el registro histórico es que, aunque la actualización se haya vuelto estable y haya sido aplicada al valor, el gestor de réplicas no ha recibido confirmación de que esta actualización haya sido recibida por el resto de los gestores de réplicas. Mientras tanto, el gestor propaga la actualización en mensajes de cotilleo.

Marcas temporales de la réplica: este vector de marcas temporales representa aquellas actualizaciones que han sido aceptadas por el gestor de réplicas, esto es, colocadas en el registro histórico del gestor. Generalmente, difiere de los valores de las marcas temporales, por supuesto, pues no todas las actualizaciones del registro histórico son estables.

Tabla de operaciones ejecutadas: la misma actualización podría llegar a cierto gestor de réplicas desde un frontal y en mensajes de cotilleo desde otros gestores de réplicas. Para evitar que una actualización se aplique dos veces se mantiene la tabla de *operaciones ejecutadas*, que contiene los identificadores únicos de las actualizaciones que han sido aplicadas al componente valor y que fueron proporcionados por el frontal. Los gestores de réplicas comprueban esta tabla antes de añadir una actualización al registro histórico.

Tabla de marcas temporales: esta tabla contiene un vector de marcas temporales para cada uno de los otros gestores de réplicas, que contiene las marcas temporales que han llegado de ellos en los mensajes de cotilleo. Los gestores de réplicas utilizan la tabla para establecer cuándo se ha aplicado una actualización en todos los gestores de réplicas.

Los gestores de réplicas se numeran 0, 1, 2, ... y el i-ésimo elemento de un vector de marcas temporales mantenido por el gestor de réplicas i corresponde al número de actualizaciones hechas recibidas por i de otros frontales; y el j-ésimo componente ($j \neq i$) corresponde al número de actualizaciones recibidas por j y propagadas hasta i en mensajes cotillas. Por ejemplo, en un sistema cotilla con tres gestores un valor de marca temporal de (2, 4, 5) en el gestor 0 representaría el hecho de que el valor en la posición 0 refleja las dos primeras actualizaciones aceptadas desde los frontales en el gestor 0, las cuatro primeras en el gestor 1 y las cinco primeras en el gestor 2. A continuación se examina con más detalle cómo se usan las marcas temporales para asegurar la ordenación.

◇ **Operaciones pregunta.** La operación más simple que se considera es la pregunta. Recuérdese que una petición de pregunta q contiene una descripción de la operación y una marca temporal $q.prev$ enviadas por el frontal. El último dato refleja la última versión del valor que el frontal ha leído o enviado como una actualización. Por lo tanto, la tarea del gestor de réplicas consiste en devolver un valor que sea, por lo menos, tan reciente como éste. Si *valorMT* es el valor de la marca temporal de la réplica, entonces q puede aplicarse sobre el valor de la réplica si:

$$q.prev \leqslant valorMT$$

El gestor de réplicas guarda q en una lista de operaciones pregunta pendientes (esto es, una cola de retención) hasta que se cumpla esta condición. Bien puede esperar por las actualizaciones extraviadas que, en último término, deberían llegar en mensajes cotillas; o bien puede solicitar las actualizaciones a los gestores de réplicas aludidos. Por ejemplo, si *valorMT* es (2,5,5) y $q.prev$ es (2,4,6) se puede observar que sólo se ha extraviado una actualización, por parte del gestor de réplicas 2 (para que el frontal que envió q haya visto esta actualización, que el gestor de réplicas no ha visto, debe haber contactado previamente con un gestor de réplicas diferente).

Cuando pueda hacerse la pregunta, el gestor de réplicas devolverá *valorMT* al frontal como la marca temporal *nueva*, que aparece en la Figura 14.6. A continuación, el frontal fusiona ésta con su marca temporal: *frontalMT*: = *fusiona*(*frontalMT*, *nueva*). La actualización en el gestor de réplicas 1, que el frontal no había visto antes de la pregunta en el ejemplo anterior ($q.prev$ tiene un 4

donde el gestor de réplicas tiene un 5), se reflejará en la actualización de *frontalMT* (y, dependiendo de la pregunta, posiblemente, en el valor que se devuelva).

◇ **Procesamiento de operaciones de actualización en orden causal.** Un frontal envía una petición de actualización a uno o más gestores de réplicas. Cada petición de actualización *a* contiene la especificación de dicha actualización (su tipo y parámetros) *a.op*, la marca temporal del frontal *a.prev* y un identificador único *a.id* generado por el frontal. Si éste envía, a distintos gestores de réplicas, la misma petición *a*, utilizará en todos los casos el mismo identificador para *a* para que no sea procesada como varias peticiones idénticas.

Cuando un gestor de réplicas *i* recibe una petición de actualización de un frontal, comprueba que no la haya procesado ya, buscando el identificador de la operación en la tabla de operaciones ejecutadas y en las fichas del registro histórico. El gestor de réplicas descarta la actualización si ya la había visto; en otro caso, incrementa en uno el *i*-ésimo elemento en la marca temporal de su réplica para llevar la cuenta del número de actualizaciones que ha recibido directamente de los frontales. A continuación, el gestor de réplicas asigna a la petición de actualización *a* un único vector de marcas temporales (en un proceso que será explicado en breve), y se coloca una nueva entrada en el registro del gestor de réplicas para esa actualización. Si *mt* es la marca temporal única que el gestor de réplicas asigna a la actualización, entonces el registro de la actualización, que se guarda en el registro histórico, se construye como una tupla que tiene la siguiente forma:

$$fichaRegistro := \langle i, mt, a.op, a.prev, a.id \rangle$$

El gestor de réplicas *i* obtiene la marca temporal *mt* a partir de *a.prev* reemplazando el *i*-ésimo elemento de *a.prev* por el *i*-ésimo elemento de su marca temporal de réplica (que acaba de ser incrementada). Esta acción hace que *mt* sea única, asegurando de esta forma que todos los componentes del sistema registrarán correctamente si han observado o no la actualización. Los elementos restantes en *mt* se copian de *a.prev*, ya que son estos valores mandados por el frontal los que deben usarse para determinar cuándo la actualización es estable. En ese momento, inmediatamente el gestor de réplicas devuelve *mt* al frontal, que lo fusiona con la marca temporal que ya tenía. Obsérvese que un frontal puede enviar su actualización a varios gestores de réplicas y, consecuentemente, recibiendo de vuelta distintas marcas temporales, que deben fusionarse en su marca temporal.

La condición de estabilidad para una actualización *a* es similar a la de las preguntas:

$$a.prev \leqslant valorMT$$

Esta condición establece que todas las actualizaciones de las cuales depende esta actualización (esto es, todas las actualizaciones que han sido observadas por el frontal que generó la actualización) han sido ya aplicadas al valor. Si esta condición no se cumple en el momento en el que se envía la actualización, será comprobada de nuevo cuando lleguen los mensajes de cotilleo. Cuando se ha cumplido la condición de estabilidad para un registro de actualización *r*, el gestor de réplicas aplica la actualización al valor y actualiza tanto el valor de la marca temporal como la tabla de operaciones ejecutadas *ejecutadas*:

$$valor := aplica(valor,\ r.a.op)$$
$$valorMT := fusiona(valorMT,\ r.mt)$$
$$ejecutadas := ejecutadas \cup \{r.a.id\}$$

La primera de estas tres instrucciones representa la aplicación de la actualización al valor. En la segunda instrucción, la marca temporal de la actualización se fusiona con la del valor. En la tercera, el identificador de operación de la actualización se añade al conjunto de identificadores de operaciones que han sido ejecutadas, lo cual se utiliza para detectar peticiones de operaciones repetidas.

◇ **Operaciones de actualización forzadas e inmediatas.** Las actualizaciones forzadas e inmediatas requieren un tratamiento especial. Recuérdese que las actualizaciones forzadas están ordenadas tanto total como causalmente. El método de ordenación básico para actualizaciones forzadas consiste en asignar un único número de secuencia, que es añadido a las marcas temporales asociadas a las actualizaciones, y procesarlas de acuerdo con este número de secuencia. Como ya se explicó en el Capítulo 11, un método general para generar números de secuencia consiste en utilizar un proceso secuenciador individual. Sin embargo, la dependencia de un proceso individual es inadecuada en el contexto de un servicio de alta disponibilidad. La solución consiste en designar como secuenciador a un *gestor de réplicas primario* en cualquier instante, pero también asegurar que otro gestor de réplicas pueda ser elegido y pueda éste asumir sus funciones de forma consistente tan pronto como el gestor primario falle. Lo que se requiere para la mayoría de los gestores de réplicas (incluyendo al primario) es registrar qué actualización es la siguiente en la secuencia, antes de que se aplique la operación. Por lo tanto, ya que la mayoría de los gestores de réplicas sobrevivirán al fallo, la decisión sobre la ordenación podrá ser cumplida por el nuevo gestor primario, que será elegido entre los gestores de réplicas supervivientes.

Las actualizaciones inmediatas se ordenan con respecto a las actualizaciones forzadas mediante el gestor de réplicas primario que las ordena en esa secuencia. El gestor primario también determina qué actualizaciones causales se considera que han precedido a una actualización inmediata. Esto lo consigue comunicándose y sincronizándose con los otros gestores de réplicas para alcanzar un acuerdo sobre este tema. En Ladin y otros [1992] pueden encontrarse más detalles a este respecto.

◇ **Mensajes de cotilleo.** Los gestores de réplicas envían mensajes de cotilleo a otros gestores conteniendo información relativa a una o más actualizaciones, de tal forma que puedan mantener su estado al día. Un gestor de réplicas utiliza las entradas en su tabla de marcas temporales para estimar qué actualizaciones no han sido recibidas por algún otro gestor de réplicas (esto es una estimación, ya que el gestor de réplicas puede que no haya recibido todavía esas actualizaciones).

Un mensaje de cotilleo *m*, enviado por el gestor de réplicas origen, consta de dos elementos: su registro histórico *m.log* y la marca temporal de su réplica *m.mt* (véase la Figura 14.8). El gestor de réplicas que reciba un mensaje cotilla tiene tres tareas principales:

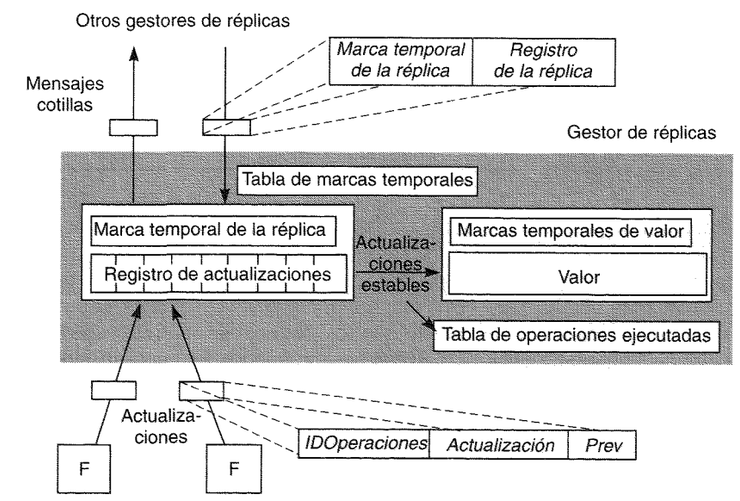


Figura 14.8. Un gestor de réplicas cotilla mostrando sus componentes principales de estado.

- Fusionar el registro histórico que le llega con el suyo (ya que puede contener actualizaciones que el receptor no había observado).
- Aplicar cualquier actualización que se haya convertido en estable y que no haya sido ejecutada antes (a su vez, algunas actualizaciones estables en el registro que llega puede que hagan estables algunas actualizaciones pendientes).
- Eliminar entradas en el registro histórico y entradas de la tabla de operaciones ejecutadas, cuando se sepa que dichas actualizaciones se han realizado en todas partes y que no existe peligro de repeticiones. Ésta es una tarea importante, ya que si no se borrasen entradas de ambas tablas, éstas podrían crecer de forma ilimitada.

El proceso de fusionar el registro histórico contenido en un mensaje recibido con el registro del receptor es inmediato. Sea *réplicaMT* la marca temporal de la réplica del receptor. Se añade cada entrada *r* del registro en *m.log* al registro histórico del receptor a no ser que $r.mt \leqslant$ *réplicaMT*, en cuyo caso ya estará en el registro o ya fue aplicada la actualización y después descartada.

El gestor de réplicas fusiona la marca temporal del mensaje de cotilleo entrante con su propia marca temporal de réplica *réplicaMT*, de forma que se corresponda con las nuevas incorporaciones al registro:

$$réplicaMT := fusiona(réplicaMT,\ m.mt)$$

Cuando se hayan fusionado los nuevos registros de actualizaciones en el registro histórico, el gestor de réplicas colecciona el conjunto E de actualizaciones en el registro que en ese momento sean estables. Éstas se pueden aplicar al valor, pero se ha de tener cuidado en el orden en el que se aplicarán para que se cumpla la relación *sucedió antes*. El gestor de réplicas ordena las actualizaciones dentro de dicho conjunto según el orden parcial «$\leqslant$» entre vectores de marcas temporales. Éste aplica entonces las actualizaciones en este orden: primero las menores. Esto es, cada $r \in E$ se aplica sólo cuando no haya un $s \in E$ tal que $s.prev < r.prev$.

A continuación el gestor de réplicas busca en el registro las entradas que puedan descartarse. Si el mensaje cotilla fue enviado por el gestor de réplicas j y si la *tablaMT* es la tabla de las marcas temporales de los gestores de réplicas, entonces el gestor de réplicas establece:

$$tablaMT[j] := m.mt$$

El gestor de réplicas puede ahora descartar cualquier entrada para actualización r en el registro histórico que haya sido recibido en todas partes. Esto es, si c es el gestor de réplicas que ha creado la entrada en el registro, entonces se requiere para todos los gestores de réplicas i que:

$$tablaMT[i][c] \geqslant r.mt[c]$$

La arquitectura cotilla también define cómo los gestores de réplicas pueden descartar entradas en la tabla de operaciones ejecutadas. Es importante no descartar estas entradas demasiado pronto ya que una operación muy retrasada podría aplicarse erróneamente dos veces. Ladin y otros [1992] proporcionan más detalles del esquema. En esencia, los frontales emiten acuses de recibo a las respuestas a sus actualizaciones, por lo que los gestores de réplicas saben cuándo un frontal terminará de enviar la actualización. A partir de ese momento, asumen un retraso máximo en la propagación de las actualizaciones.

◇ **Propagación de las actualizaciones.** La arquitectura cotilla no especifica cuándo intercambian mensajes cotillas los gestores de réplicas, ni cómo un gestor de réplicas escoge a otro para enviarle un mensaje cotilla. Se necesita una estrategia robusta de propagación de actualizaciones para que todos los gestores de réplicas reciban todas las actualizaciones en un período de tiempo aceptable.

El tiempo que tardan todos los gestores de réplicas en recibir una actualización determinada depende de tres factores:

- La frecuencia y la duración de las particiones de la red.
- La frecuencia con las que los gestores de réplicas envían mensajes cotillas.
- La política de selección de un socio con el cual intercambiar mensajes cotillas.

El primer factor está más allá del control del sistema, aunque los usuarios pueden determinar, hasta cierto punto, la frecuencia con la que trabajan sin conexión.

La frecuencia deseada de intercambio de mensajes cotillas puede sintonizarse en función de la aplicación. Considérese un sistema de tablón de anuncios compartido entre varios sitios. Parece innecesario que cada ítem se comunique inmediatamente a todos los sitios. Pero, ¿qué sucedería si sólo se intercambiasen los mensajes cotillas tras largos períodos, digamos una vez al día? Si sólo se usan actualizaciones causales, entonces es muy posible que los clientes en cada sitio tengan sus propios debates consistentes sobre el mismo tablón de anuncios, sin ser conscientes de las discusiones en los otros sitios. Entonces, todos los debates se fusionarían, por ejemplo a medianoche; pero es probable que los debates sobre el mismo tema se vuelvan incongruentes, cuando habría sido preferible para ellos tenerse en cuenta los unos a los otros. Un período de tiempo de minutos u horas, para el intercambio de mensajes de cotilleo, hubiera sido más apropiado para este caso.

Hay varios tipos de política de selección de socios. Golding y Long [1993] consideran las políticas *aleatorias*, *deterministas* y *topológicas* para su *protocolo anti-entrópico con marcas temporales*, que utiliza un esquema de propagación de actualizaciones de estilo cotilla.

Las políticas aleatorias escogen al azar un socio, pero con probabilidades ponderadas que favorecen a algunos socios frente a otros: por ejemplo, socios cercanos frente a socios lejanos. Golding y Long se dieron cuenta de que tal política funcionaba sorprendentemente bien en simulación. Las políticas deterministas utilizan una expresión que es función del estado del gestor de réplicas para escoger un socio. Por ejemplo, un gestor de réplicas puede examinar su tabla de marcas temporales y escoger al gestor de réplicas que parezca más divergente en cuanto a las actualizaciones que ha recibido.

Las políticas topológicas organizan a los gestores de réplicas en grafos fijos. Una posibilidad es una malla: los gestores de réplicas envían mensajes cotillas a los cuatro gestores de réplicas con los que están conectados. Otra posibilidad es organizar a los gestores de réplicas en un círculo, pasando cada uno de ellos un mensaje cotilla sólo a su vecino (por ejemplo, en el sentido de las agujas del reloj), de tal modo que las actualizaciones que llegan de cada gestor de réplicas finalmente atraviesan el círculo. Hay muchas otras topologías posibles, incluyendo los árboles.

Distintas políticas de selección de socios, entre las que se incluyen las anteriores, valoran la cantidad de comunicación frente a mayores latencias de transmisión y la posibilidad de que un único fallo afecte a otros gestores de réplicas. En la práctica, la elección depende de la importancia relativa de estos factores. Por ejemplo, la topología circular produce relativamente poca comunicación pero está sujeta a mayores latencias en las transmisiones ya que los mensajes cotillas han de pasar, generalmente, por distintos gestores de réplicas. Aún más, si un gestor de réplicas falla, el círculo no funciona y necesita ser reconfigurado. Por el contrario, la política de selección aleatoria no es susceptible a los fallos pero puede producir tiempos de propagación de las actualizaciones más variables.

◇ **Discusión sobre la arquitectura cotilla.** La arquitectura cotilla tiene como objetivo alcanzar una alta disponibilidad en los servicios. Se puede decir en su favor que los clientes pueden seguir obteniendo un servicio incluso cuando están en una partición aislados del resto de la red, siempre que, al menos, un gestor de réplicas continúe funcionando en esa partición. No obstante, esta disponibilidad se consigue a cambio de proporcionar garantías de consistencia relajadas. Para el caso de objetos como cuentas bancarias, donde se requiere consistencia secuencial, una arquitec-

tura cotilla no será mejor que el sistema tolerante a fallos de la Sección 14.3 y proporcionará el servicio sólo en la partición mayoritaria.

La forma perezosa en la que propaga las actualizaciones hace que este sistema no sea apropiado para la actualización de réplicas con requisitos próximos al tiempo real, como cuando los usuarios participan en una conferencia en *tiempo real* y actualizan un documento compartido. En ese caso sería más apropiado un sistema basado en multidifusión.

Otro tema es la capacidad de aumentar de tamaño de un sistema cotilla. En la medida que el número de gestores de réplicas crece también aumenta el número de mensajes que han de transmitirse y el tamaño de los vectores de marcas temporales. Si un cliente realiza una pregunta, normalmente se necesitarán dos mensajes (entre el frontal y el gestor). Si un cliente realiza una operación de actualización causal y si cada uno de los R gestores de réplicas normalmente recoge C actualizaciones dentro de un mensaje cotilla, entonces el número de mensajes que se intercambian es $2 + (R - 1)/C$. El primer término representa la comunicación entre el frontal y el gestor de réplicas y el segundo es la parte de la actualización de un mensaje cotilla enviado a los otros gestores de réplicas. Aumentando C se mejora el número de mensajes. Pero eso también empeora las latencias de entrega, porque el gestor de réplicas espera hasta que lleguen más actualizaciones antes de propagarlas.

Una aproximación para conseguir que los sistemas basados en cotilleo puedan aumentar de tamaño es hacer que la mayoría de las réplicas sean de *sólo lectura*. En otras palabras, estas réplicas se actualizan a través de mensajes cotillas, pero no reciben actualizaciones directamente de los frontales. Esta organización es útil, potencialmente, donde la proporción *actualización/pregunta* es pequeña. Las réplicas de *sólo lectura* pueden ser situadas próximas a los grupos de clientes y las actualizaciones pueden ser servidas por un número relativamente pequeño de gestores de réplicas. El tráfico cotilla se reduce, ya que las réplicas de *sólo lectura* no propagan mensajes de cotilleo. Y los vectores de marcas temporales sólo contienen entradas para réplicas actualizables.

14.4.2. EL SISTEMA BAYOU Y LA APROXIMACIÓN DE LA TRANSFORMACIÓN OPERACIONAL

El sistema Bayou [Terry y otros 1995, Petersen y otros 1997] proporciona replicación de datos con alta disponibilidad con garantías más débiles que la consistencia secuencial, como la arquitectura cotilla y el protocolo anti-entrópico con marcas temporales. Como en esos sistemas, los gestores de réplicas Bayou pueden hacer frente a una conectividad variable, mediante el intercambio de actualizaciones por parejas, en lo que los diseñadores también denominan un protocolo anti-entropía. Sin embargo, el sistema Bayou adopta una aproximación marcadamente diferente, en la que se posibilitan técnicas de detección y resolución de conflictos específicas del dominio.

Considérese a un usuario que necesita actualizar una agenda mientras trabaja sin conexión. Si se requiere consistencia estricta, entonces, en la arquitectura cotilla las actualizaciones tendrían que realizar una operación forzada (totalmente ordenada). Pero entonces, sólo los usuarios de una partición mayoritaria podrían actualizar la agenda. Así, el acceso de los usuarios a la agenda podría verse limitado independientemente de si ellos de hecho necesiten hacer actualizaciones que rompieran la integridad de la agenda. Los usuarios que quieran rellenar un hueco con una cita que no dé lugar a un conflicto serán tratados de la misma forma que aquellos usuarios que involuntariamente puedan haber reservado por duplicado un hueco en su agenda.

Por el contrario, en el sistema Bayou los usuarios, en el tren y en el trabajo, podrían hacer cualquier actualización que quisieran. Todas las actualizaciones se aplican y se registran en cualquier gestor de réplicas que se encuentren. Cuando las actualizaciones recibidas en dos gestores de réplicas cualesquiera se fusionan durante un intercambio anti-entropía, los gestores de réplicas detectan y resuelven los conflictos. Se puede aplicar cualquier criterio de conflicto específico del dominio entre las operaciones. Por ejemplo, si una ejecutiva y su secretario han añadido citas en el mismo hueco, entonces un sistema Bayou lo detecta después de que la ejecutiva haya reconectado su portátil. Además, el sistema resuelve el conflicto según una política específica del dominio. En este caso, podría, por ejemplo, confirmar la cita de la ejecutiva y quitar la reserva del secretario en el hueco. Tal efecto, en el que una o más de un conjunto de operaciones conflictivas se deshacen o se alteran para resolver dichos conflictos, se denomina *transformación operacional*.

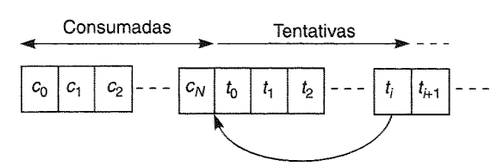
El estado que el sistema Bayou replica se guarda en forma de base de datos, permitiendo preguntas y actualizaciones (que podrían insertar, modificar o borrar ítems en la base de datos). Aunque aquí no se insistirá en este aspecto, una actualización Bayou es un caso especial de transacción. Consiste en una operación individual, una llamada a un *procedimiento prefijado*, que afecta a varios objetos dentro de cada gestor de réplicas, pero que es llevado a cabo con las garantías ACID. El sistema Bayou puede deshacer y rehacer actualizaciones de la base de datos según proceda la ejecución.

La garantía que proporciona el sistema Bayou es que, al final, cada gestor de réplicas recibe el mismo conjunto de actualizaciones y finalmente aplica esas actualizaciones de modo que las bases de datos de los gestores de réplicas sean idénticas. En la práctica podría haber un flujo continuo de actualizaciones y las bases de datos no lleguen a ser nunca idénticas; pero llegarían a serlo si cesasen las actualizaciones.

◇ **Actualizaciones tentativas y consumadas.** Las actualizaciones se marcan como *tentativas* al principio de aplicarse a la base de datos. Bayou se encarga de que las actualizaciones tentativas se dispongan finalmente en un orden canónico y se marquen como *consumadas*. Mientras las actualizaciones sean tentativas, el sistema puede deshacerlas y reaplicarlas según produzcan un estado consistente. Una vez consumadas, se mantiene su aplicación en el orden asignado. En la práctica, el orden de consumación puede obtenerse designando a algún gestor de réplicas como gestor de réplicas *primario*. Como es normal, éste decide el orden de consumación como aquel en el que recibe las actualizaciones, y propaga esta información de ordenación a los otros gestores de réplicas. Para el primario, los usuarios puede elegir, por ejemplo, una máquina rápida que esté usualmente disponible; igualmente, podría ser el gestor de réplicas en el portátil de la ejecutiva, si las actualizaciones de esa usuaria tienen prioridad.

En cada momento, el estado de la base de datos de la réplica se obtiene a partir de una secuencia (posiblemente vacía) de actualizaciones consumadas, seguidas por una secuencia (posiblemente vacía) de actualizaciones tentativas. Cuando llega la siguiente actualización consumada, o si una de las actualizaciones tentativas que se aplicó se convierte en la siguiente actualización consumada, entonces debe tener lugar una reordenación de las actualizaciones. En la Figura 14.9, t_i pasa a estar consumada. Todas las actualizaciones tentativas posteriores a c_N deben ser desechas; después t_i se aplica tras c_N y desde t_0 hasta t_{i-1} y t_{i+1}, etc. se vuelven a aplicar tras t_i.

◇ **Procedimientos de chequeo de consistencia y de fusión.** Una actualización puede entrar en conflicto con alguna otra operación que ya se haya aplicado. Debido a esta posibilidad, toda



La actualización tentativa t_i se convierte en la siguiente actualización consumada y se inserta tras la última actualización consumada c_N.

Figura 14.9. Actualizaciones consumadas y tentativas en el sistema Bayou.

actualización Bayou contiene un *chequeo de consistencia* y un *procedimiento de fusión* además de la especificación de la operación (su tipo y sus parámetros). Todos estos componentes de la actualización son específicos del dominio.

Un gestor de réplicas invoca al procedimiento de *chequeo de consistencia* antes de aplicar la operación. Éste comprueba si podría ocurrir un conflicto si se aplicara la actualización y para hacerlo podría comprobar cualquier parte de la base de datos. Por ejemplo, considérese el caso de una reserva para una cita en una agenda. En el caso más simple, el chequeo de consistencia comprobará si existe un conflicto *escritura-escritura*: esto es, si otro cliente ya ha rellenado el hueco requerido. Pero también podría comprobar si hay un conflicto lectura-escritura. Por ejemplo, podría comprobar que el hueco deseado esté vacío y que el número de citas de ese día sea menor que seis.

Si el chequeo de consistencia indica que hay un conflicto, entonces Bayou invoca el procedimiento de fusión de la operación. Ese procedimiento altera la operación que se aplicará de tal forma que consiga algo similar a lo que se buscaba, pero evitando el conflicto. Por ejemplo, en el caso de la agenda el procedimiento de fusión podría elegir otro hueco en una hora próxima a esa en su lugar o, como se mencionó anteriormente, podría utilizar un simple esquema de prioridades para decidir qué cita es más importante y la impone. El procedimiento de fusión puede no encontrar una alteración adecuada de la operación, en cuyo caso el sistema indica un error. No obstante, el efecto de un procedimiento de fusión es determinista. (Los gestores de réplica de Bayou son máquinas de estado.)

◇ **Discusión.** El sistema Bayou se diferencia de los otros sistemas de replicación que hemos tratado en que la replicación no es transparente para la aplicación. Se explota el conocimiento de la semántica de la aplicación para incrementar la disponibilidad de los datos, al mismo tiempo que mantiene un estado replicado que es lo que podríamos denominar *finalmente consistente secuencialmente*.

Las desventajas de esta aproximación son, primero, el aumento de la complejidad para el programador de la aplicación, que debe proporcionar procedimientos de chequeo de consistencia y de fusión. Ambos pueden ser difíciles de producir, dado un gran número de posibles conflictos que necesitan ser detectados y resueltos. La segunda desventaja es el aumento de la complejidad para el usuario. No sólo se espera que los usuarios manejen datos leídos mientras aún son tentativos, sino que además han de asumir que la operación que especifiquen puede llegar a verse alterada. Por ejemplo, el usuario reservó un hueco en la agenda, para darse cuenta sólo después que la reserva ha *saltado* a un hueco cercano. Es muy importante que el usuario reciba claras indicaciones de qué datos son tentativos y cuáles están consolidados.

La aproximación de transformación operacional utilizada por Bayou aparece especialmente en sistemas de soporte de trabajo cooperativo apoyado en computadores (*computer supported cooperative working*, CSCW), donde pueden ocurrir conflictos entre las actualizaciones de usuarios que están separados geográficamente [Kindberg y otros 1996], [Sun y Ellis, 1998]. La aproximación está limitada, en la práctica, a aplicaciones donde los conflictos sean relativamente escasos, donde la semántica subyacente a los datos sea relativamente simple y donde los usuarios puedan tratar con información tentativa.

14.4.3. EL SISTEMA DE ARCHIVOS CODA

El sistema de archivos Coda es un descendiente de AFS (véase la Sección 8.4) que intenta solucionar distintos requisitos que el AFS no cumple, en concreto proporcionar alta disponibilidad a pesar de que se realicen operaciones sin conexión. El sistema se desarrolló dentro de un proyecto de investigación emprendido por Satyanarayanan y sus colaboradores en la Universidad Carnegie-Mellon (CMU) [Satyanarayanan y otros 1990; Kistler y Satyanarayanan 1992]. Los requisitos de diseño para Coda provenían de la experiencia con AFS en la CMU y otros lugares, donde se había empleado en sistemas distribuidos a gran escala en redes de comunicaciones de área tanto local como amplia.

Mientras que el funcionamiento y la facilidad de administración de AFS se consideró satisfactoria bajo las condiciones de uso en la UCM, se intuyó que la forma limitada de replicación ofrecida por AFS (restringida a volúmenes de *sólo lectura*) se convertiría en un factor limitante a una cierta escala, especialmente para acceder a archivos ampliamente compartidos tales como los tableros de anuncios electrónicos y otras bases de datos de grandes sistemas.

Además, había espacio para la mejora de la disponibilidad del servicio ofrecido por AFS. Las dificultades más comunes con las que se encontraban los usuarios de AFS surgían del fallo (o de la interrupción programada) de servidores y componentes de red. El tamaño del sistema en la CMU era tal que ocurrían fallos en el servicio algunas veces al día, que podían causar molestias a muchos usuarios durante períodos de tiempo que iban desde unos pocos minutos a varias horas.

Finalmente, aparecía en esos momentos un uso de los computadores que AFS no podía proporcionar: el uso móvil de los computadores portátiles. Esto hizo incluir el requisito de que debían estar disponibles todos los archivos necesarios para que un usuario continuase su trabajo mientras estaba desconectado de la red, sin que tuviese que recurrir a métodos manuales para gestionar la ubicación de los archivos.

El sistema Coda intenta cumplir estos tres requisitos bajo el encabezamiento general de *disponibilidad constante de los datos*. Su propósito era que los usuarios se beneficiasen de un repositorio de archivos compartidos a la vez que pudieran depender por completo de recursos locales cuando el repositorio estuviera parcial o totalmente inaccesible. Además de estos propósitos, el sistema Coda mantenía los objetivos originales de AFS en lo que respecta a la capacidad de crecimiento y la emulación de la semántica de los archivos en UNIX.

A diferencia de AFS, donde los volúmenes de lectura-escritura se almacenan sólo en un servidor, el diseño de Coda se basa en la replicación de volúmenes de archivos para conseguir una mayor capacidad de procesamiento de las operaciones de accesos a archivos y un mayor grado de tolerancia a fallos. Además, el sistema Coda se basa en la ampliación de un mecanismo utilizado en AFS para el mantenimiento en caché de copias de los archivos dentro de los computadores de los clientes, para permitir a esos computadores operar cuando no están conectados a la red.

Veremos que Coda es como Bayou (véase la Sección 14.4.2) en cuanto que mantiene una estrategia optimista. Esto es, permite que los usuarios actualicen datos mientras el sistema está particionado, basándose en que los conflictos son relativamente improbables y que pueden solucionarse si llegaran a ocurrir. Detecta conflictos como Bayou, pero a diferencia de él, realiza el chequeo con independencia de la semántica de los datos almacenados en los archivos. Y también, a diferencia de Bayou, sólo proporciona un apoyo limitado a la resolución de los casos donde las réplicas entran en conflicto.

◇ **La arquitectura Coda.** El sistema Coda, adoptando la terminología AFS, ejecuta lo que denomina procesos *Venus* en los computadores de los clientes y procesos *Vice* en los computadores del servidor de archivos. Los procesos Vice son los que aquí se han denominado gestores de réplicas. Los procesos Venus son un híbrido entre los frontales y los gestores de réplicas. Realizan la tarea del frontal de ocultar la implementación del servicio a los procesos locales de los clientes, pero desde el momento en el que manejan una copia local caché de los archivos, también son gestores de réplicas, aunque de un tipo distinto al de los procesos Vice.

El conjunto de servidores que mantiene las réplicas de un volumen de archivos se conoce como el *grupo de almacenamiento del volumen* (*volume storage group*, VSG). En cualquier instante, un cliente que desee abrir un archivo en dicho volumen puede acceder a algún subconjunto del VSG, conocido como el grupo disponible de almacenamiento del volumen (*available* VSG, AVSG). Los

miembros del AVSG varían según los servidores estén accesibles o se vuelvan inaccesibles debido a fallos en la red o en los servidores.

Normalmente, el acceso a archivos del sistema Coda se realiza de una forma similar a AFS, mediante el almacenamiento en caché de copias de los archivos que son proporcionadas a los computadores de los clientes desde cualquiera de los servidores en el actual AVSG. Como en AFS, a los clientes se les notifican los cambios por medio de un mecanismo de *promesa de devolución de llamada*, pero que ahora depende de un mecanismo adicional para la distribución de actualizaciones a cada réplica. Al *cerrar*, las copias de los archivos modificados se difunden en paralelo a todos los servidores del AVSG.

En el sistema Coda se dice que se trabaja en operación sin conexión cuando el AVSG está vacío. Esto puede deberse a fallos en la red o en los servidores, o puede ser consecuencia de una desconexión deliberada por parte del computador del cliente, como es en el caso de un portátil. Para operar eficazmente sin conexión es necesaria la presencia en la memoria caché del computador del cliente de *todos* los archivos que serán necesarios para que el trabajo del usuario pueda realizarse. Para conseguir esto, el usuario ha de cooperar con el sistema Coda para generar una lista de los archivos que deberían mantenerse en la memoria caché. Se proporciona una herramienta que almacena un histórico de la lista de los archivos utilizados mientras el cliente estaba conectado y que se usa como base para predecir la utilización en la fase sin conexión.

Es un principio del diseño del sistema Coda que las copias de los archivos que residan en los servidores sean más fiables que aquellas que residan en las memorias caché de los computadores de los clientes. Aunque podría ser lógicamente posible construir un sistema de archivos que se basase por completo en copias de los archivos en la memoria caché de los computadores de los clientes, es improbable que se consiguiese un servicio satisfactorio. Los servidores del sistema Coda existen para proporcionar la calidad necesaria al servicio. Las copias de los archivos que residen en las memorias caché de los computadores de los clientes se consideran como útiles siempre que su vigencia pueda ser periódicamente revalidada frente a las copias residentes en los servidores. En el caso de operaciones sin conexión, la reválida se produce cuando termina la desconexión y los archivos en la memoria caché se armonizan con los de los servidores. En el peor caso, esto puede requerir alguna intervención manual para resolver inconsistencias o conflictos.

◇ **La estrategia de replicación.** La estrategia de replicación del sistema Coda es optimista, ya que permite la modificación de archivos cuando la red está particionada o durante una operación sin conexión. Se apoya en la inclusión en cada versión de un archivo de un *vector de versiones Coda* (*Coda version vector*, CVV). Un CVV es un vector de marcas temporales con un elemento para cada servidor en el VSG relevante. Cada elemento del CVV es una estimación del número de modificaciones realizadas en la versión del archivo que está en el servidor correspondiente. El propósito del CVV es proporcionar suficiente información acerca de la historia de las actualizaciones de cada réplica del archivo, para permitir que se detecten conflictos potenciales y que éstos se notifiquen de cara a una intervención manual, y para que se actualicen automáticamente las réplicas anticuadas.

Si el CVV en uno de los sitios es mayor o igual que sus correspondientes CVV en los otros sitios (la Sección 10.4 definió el significado de $v_1 \geqslant v_2$ para los vectores de marcas temporales v_1 y v_2), entonces no hay conflicto. Las réplicas más antiguas (con marcas temporales estrictamente menores) añadirán todas las actualizaciones en una réplica más nueva y podrán ser automáticamente actualizadas a partir de ella.

Cuando no se da ese caso, que es cuando no se cumple que $v_1 \geqslant v_2$ ni $v_2 \geqslant v_1$ para dos CVV, entonces hay un conflicto: cada réplica refleja al menos una actualización que la otra no refleja. En general, el sistema Coda no resuelve los conflictos de forma automática. El archivo se señala como *inoperable* y se informa del conflicto al propietario del archivo.

Cuando se cierra un archivo modificado, el proceso Venus en el cliente envía un mensaje de actualización a cada sitio en el AVSG actual; el mensaje contiene el CVV actual y los nuevos

contenidos del archivo. El proceso Vice en cada sitio comprueba el CVV y, si es mayor que el que guarda en ese momento, almacena los nuevos contenidos para ese archivo y devuelve un acuse de recibo positivo. A continuación, el proceso Venus calcula el nuevo CVV incrementando los contadores de modificaciones para los servidores que respondieron de forma positiva al mensaje de actualización y distribuye el nuevo CVV entre los miembros del AVSG.

Dado que el mensaje se envía sólo a los miembros del AVSG y no al VSG, los servidores que no estén en el AVSG actual no reciben el nuevo CVV. Por lo tanto, todo CVV contendrá un contador de modificaciones exacto para el servidor local, pero los contadores para los servidores no locales serán, en general, más bajos, ya que serán actualizados sólo cuando el servidor reciba un mensaje de actualización.

El recuadro siguiente contiene un ejemplo que ilustra el uso de los CVV para gestionar la actualización de un archivo replicado en tres sitios. En Satyanarayanan y otros [1990] se pueden encontrar más detalles del uso de CVV para la gestión de actualizaciones. El concepto de CVV está basado en las técnicas de replicación utilizadas en el sitema Locus [Popek y Walker 1985].

En el modo de operación normal, el comportamiento del sistema Coda es similar al de AFS. La pérdida de una memoria caché es transparente al usuario y sólo impone un peor rendimiento. Las ventajas que se derivan de la replicación de varios o de todos los volúmenes de archivos en múltiples servidores son:

- Los archivos en un volumen replicado se mantienen accesibles para cualquier cliente que pueda acceder, al menos, a una de las réplicas.
- El rendimiento del sistema puede mejorarse compartiendo algo de la carga de las peticiones de servicio sobre un volumen replicado entre todos los servidores que mantengan réplicas.

En la operación sin conexión (cuando el cliente no puede acceder a ninguno de los servidores de un volumen), la pérdida de una memoria caché impide que se pueda continuar y se suspende la computación hasta que se recupere la conexión o el usuario aborte el proceso. Por lo tanto, es importante que se cargue la memoria caché antes de que comience la operación sin conexión de tal forma que se evite la pérdida de dicha memoria caché.

En resumen, comparado con AFS, Coda aumenta la disponibilidad gracias tanto a la replicación de archivos entre todos los servidores como a la posibilidad de que los clientes operen enteramente a partir de sus cachés. Ambos métodos dependen del uso de una estrategia optimista para la detección de conflictos en la actualización en presencia de particiones en la red. Ambos mecanismos son complementarios e independientes. Por ejemplo, un usuario puede aprovechar los beneficios de la operación sin conexión incluso aunque los volúmenes de archivos requeridos estén almacenados en un único servidor.

Ejemplo: Considérese una secuencia de modificaciones a un archivo F en un volumen que está replicado en tres servidores: S_1, S_2 y S_3. El VSG para F es $\{S_1, S_2, S_3\}$. F es modificado, aproximadamente al mismo tiempo, por dos clientes: C_1 y C_2. Debido a un fallo en la red, C_1 puede acceder sólo a S_1 y S_2 (el AVSG de C_1 es $\{S_1, S_2\}$) y C_2 puede acceder sólo a S_3 (el AVSG de C_2 es $\{S_3\}$).

1. Inicialmente, los CVV para F en los tres servidores son iguales, por ejemplo [1, 1, 1].
2. C_1 ejecuta un proceso que abre F, lo modifica y después lo cierra. El proceso Venus en C_1 difunde un mensaje de actualización a su AVSG, $\{S_1, S_2\}$, obteniendo al final una nueva versión de F y un CVV [2,2,1] en S_1 y S_2, pero no en S_3.
3. Mientras tanto, C_2 ejecuta dos procesos, cada uno de los cuales abre F, lo modifican y posteriormente lo cierran. El proceso Venus en C_2 difunde tras cada modificación un ensaje de actualización a su AVSG, $\{S_3\}$; al final, se obtiene una nueva versión de F y un CVV de [1,1,3] en S_3.

4. Algún tiempo después, se repara el fallo en la red, y C_2 realiza un chequeo rutinario para comprobar si los miembros no accesibles de su VSG se han vuelto accesibles (más tarde se describirá el proceso mediante el cual se realizan estas comprobaciones) y descubre que S_1 y S_2 lo están. Modifica su AVSG a $\{S_1, S_2, S_3\}$ para el volumen que contiene a F y solicita los CVV para F por parte de todos los miembros del nuevo AVSG. Cuando los recibe, C_2 descubre que S_1 y S_2 cada uno tiene [2,2,1] mientras que S_3 tiene [1,1,3]. Esto representa un *conflicto* que requiere una intervención manual para actualizar F, de tal forma que se minimice la pérdida de información de actualización.

Por otro lado, considérese un escenario parecido, pero más simple, que siga la misma secuencia de eventos que se acaba de describir, pero omitiendo el ítem (3), de tal forma que F no es modificado por C_2. Por lo tanto, el CVV en S_3 no ha cambiado y sigue siendo [1,1,1]; cuando se repara la red, C_2 descubre que los CVVs en S_1 y S_2 ([2,2,1]) *dominan* al de S_3. La versión del archivo en S_1 o en S_2 debería reemplazar la de S_3.

◇ **Semántica de actualización.** Las garantías de vigencia ofrecidas por el sistema Coda cuando un cliente abre un archivo son más débiles que las de AFS, dada la estrategia de actualización optimista. El servidor único S al que se refieren las garantías de vigencia en AFS se reemplazan por un conjunto de servidores $\bar{S}$ (el VSG del archivo), y el cliente C puede acceder a un subconjunto de servidores $\bar{s}$ (el AVSG del archivo que puede ver C).

Dicho de manera informal, la garantía que ofrece un *open* con éxito en el sistema Coda consiste en proporcionar la copia más reciente de F por parte del actual AVSG y, en el caso en el que no haya servidores disponibles, se utiliza una copia local de F, si está disponible, almacenada en la memoria caché. Un *close* con éxito garantiza que el archivo ha sido propagado al conjunto de servidores actualmente accesibles o, en el caso de que no los hubiera, el archivo es marcado para que se propague a la menor oportunidad.

Se puede hacer una definición más precisa de estas garantías, teniendo en cuenta el efecto de las devoluciones de llamada perdidas, utilizando una extensión de la notación usada para AFS. En cada definición, excepto la última, hay dos casos: el primero, comenzando por $\bar{s} \neq \emptyset$, que se refiere a todas las situaciones en las cuales el AVSG no está vacío; y el segundo, que trata la operación sin conexión:

tras un *open* con éxito: $(\bar{s} \neq \emptyset$ *y* $(latest(F, \bar{s}, 0)$ *o* $(latest(F, \bar{s}, T)$ *y* $lostCallback(\bar{s}, T)$ *y* $inCache(F))))$ *o* $(\bar{s} = \emptyset$ *y* $inCaché(F))$

tras un *open* fallido: $(\bar{s} \neq \emptyset$ *y* $conflict(F, \bar{s}))$ *o* $(\bar{s} = \emptyset$ y $\neg\, inCaché(F))$

tras un *close* con éxito: $(\bar{s} \neq \emptyset$ *y* $updated(F, \bar{s}))$ *o* $(\bar{s} = \emptyset)$

tras un *close* fallido: $\bar{s} \neq \emptyset$ *y* $conflict(F, \bar{s})$

Este modelo asume que el sistema es síncrono: T es el mayor tiempo durante el cual un cliente puede permanecer sin estar advertido de una actualización en otra parte de un archivo que esté en su caché; $latest(F, \bar{s}, T)$ denota el hecho de que el valor actual de F en C fue el último entre todos los servidores en $\bar{s}$ en algún instante dentro de los últimos T segundos y que no hubo conflictos entre las copias de F hasta aquel instante; $lostCallback(\bar{s}, T)$ significa que se envió una devolución de llamada desde algún miembro de $\bar{s}$ en los últimos T segundos y no fue recibida por C; por último, $conflict(F, \bar{s})$ indica que los valores de F en algunos servidores de $\bar{s}$ están en ese momento en conflicto.

◇ **Acceso a las réplicas.** La estrategia usada en *open* y *close* para acceder a las réplicas de un archivo es una variante de la aproximación *uno lee/todos escriben*. En *open*, si no hay una copia del archivo en la caché local el cliente identifica, para ese archivo, un servidor preferido dentro del AVSG. Dicho servidor preferido puede elegirse de forma aleatoria o basándose en criterios de rendimiento tales como la proximidad física o la carga del servidor. El cliente solicita al servidor preferido una copia de los atributos y el contenido del archivo, y cuando los recibe, contacta con el resto de los miembros del AVSG para verificar que dicha copia es la última versión disponible. Si no es así, uno de los miembros del AVSG con la última versión pasa a ser el sitio preferido, se vuelven a traer los contenidos del archivo y se notifica a los miembros del AVSG que algunos miembros poseen réplicas anticuadas. Cuando se ha terminado de traer los contenidos, se establece en el servidor preferido una promesa de devolución de llamada.

Cuando se cierra un archivo en un cliente tras una modificación, sus atributos y contenidos se transmiten en paralelo a todos los miembros del AVSG utilizando un protocolo de llamada a procedimiento remoto multidifundido. Esto maximiza la probabilidad de que cada sitio de replicación para un archivo tenga una versión actualizada en todo momento. Pero no lo garantiza, porque el AVSG no incluye, necesariamente, a todos los miembros de un VSG. Al mismo tiempo minimiza la carga en el servidor cediendo a los clientes la responsabilidad de propagar los cambios a los sitios de replicación en el caso normal (los servidores sólo se ven implicados cuando se descubre una réplica atrasada en una operación *open*).

Ya que el mantenimiento en todos los miembros de un AVSG del estado de devolución de llamada sería caro, la promesa de devolución de llamada se mantiene solamente en el servidor. Pero esto presenta un nuevo problema: el servidor preferido para un cliente no tiene que estar en el AVSG de otro cliente. Si es éste el caso, una actualización por parte del segundo cliente no ocasionará una devolución de llamada sobre el primer cliente. En la siguiente subsección se tratará la solución adoptada para este problema.

◇ **Coherencia caché.** Las garantías de vigencia del sistema Coda, expuestas anteriormente, implican que el proceso Venus en cada cliente debe detectar los siguientes eventos dentro de los T segundos siguientes a que hayan ocurrido:

- Aumento del tamaño de un AVSG (debido a que un servidor que antes estaba inaccesible se ha vuelto accesible).
- Reducción del tamaño de un AVSG (debido a que un servidor se vuelve inaccesible).
- Pérdida de un evento de devolución de llamada.

Para conseguir esto, Venus envía un mensaje de sondeo cada T segundos a todos los servidores en los VSG de los archivos que tiene en su memoria caché. Sólo se recibirán respuestas por parte de los servidores accesibles. Si Venus recibe una respuesta de un servidor previamente inaccesible aumenta de tamaño su correspondiente AVSG y retira las promesas de volver a llamar de cualquier archivo que esté manteniendo del volumen en cuestión. Esto se hace debido a que la copia en la memoria caché puede que ya no sea la última versión disponible en el nuevo AVSG.

Si no recibe una respuesta por parte de un servidor previamente accesible, Venus reduce el tamaño del correspondiente AVSG. No se requieren cambios en las promesas de devolución de llamada a no ser que la reducción esté causada por la pérdida de un servidor preferido, en cuyo caso se retiran de ese servidor todas las promesas de devolución de llamada. Si una de las respuestas indica que un mensaje de devolución de llamada fue enviado pero no recibido, se retira la promesa de devolución de llamada asociada al archivo correspondiente.

Resta abordar el problema mencionado anteriormente, de actualizaciones perdidas por un servidor porque no está en el AVSG de otro cliente que realiza una actualización. Para tratar este caso, en respuesta a cada mensaje sonda, a Venus se le manda un *vector de versiones del volumen* (*volume CVV*). El CVV del volumen contiene un resumen de los CVV de todos los archivos del volumen. Si Venus detecta cualquier desajuste entre los CVV del volumen, quiere decir que algunos miembros del AVSG deben tener versiones de algunos archivos que no están al día. Aunque los archivos desfasados pueden no ser los mismos que los que están en su memoria caché local, Venus realiza una suposición pesimista y retira las promesas de devolución de llamada de todos los archivos que mantenga del volumen en cuestión.

Observe que Venus sólo sondea a los servidores en los VSG de los archivos para los cuales mantiene copias en su memoria caché, y que un simple mensaje sonda sirve para actualizar los AVSG y comprobar las devoluciones de llamada para todos los archivos en un volumen. Esto, combinado con un valor relativamente grande de T (del orden de 10 minutos en la implementación experimental), significa que los mensajes sonda no son un obstáculo para la capacidad de crecimiento de un sistema Coda hacia un gran número de servidores y redes de área amplia.

◇ **Operación sin conexión.** Durante las desconexiones breves, tales como las que se deben a interrupciones inesperadas en el servicio, la política normalmente adoptada por Venus, de reemplazo de la memoria caché menos recientemente usada, puede ser suficiente para evitar pérdidas de memorias caché en los volúmenes desconectados. Pero no es probable que un cliente pueda operar en modo desconectado durante períodos de tiempo largos sin generar referencias a archivos o directorios que no estén en la memoria caché a menos que se adopte una política diferente.

Por lo tanto, el sistema Coda permite a los usuarios especificar una lista priorizada de archivos y directorios que Venus debería esforzarse por retener en la memoria caché. Los objetos en el nivel más alto se identifican como *pegadizos* y deben ser retenidos en la memoria caché en todo momento. Si el disco local tiene tamaño suficiente como para acomodarlos a todos, el usuario se asegura que permanecerán accesibles. Ya que a menudo es difícil saber exactamente qué accesos a los archivos serán generados por cualquier secuencia de acciones del usuario, se proporciona una herramienta que permite al usuario agrupar una secuencia de acciones; Venus observa las referencias a los archivos generadas por la secuencia y los señala con una prioridad determinada.

Cuando termina la operación sin conexión empieza un proceso de *reintegración*. Para cada archivo o directorio en la memoria caché que fue modificado, creado o borrado durante la operación sin conexión, Venus ejecuta una secuencia de operaciones de actualización para hacer las réplicas del AVSG idénticas a la copia en la memoria caché. La reintegración prosigue de forma descendente desde la raíz de cada volumen en la memoria caché.

Se pueden detectar conflictos durante la reintegración debido a las actualizaciones por parte de otros clientes en las réplicas del AVSG. Cuando esto ocurre, la copia en la memoria caché se almacena en una ubicación temporal en el servidor y se informa al usuario que inició la reintegración. Esta aproximación se basa en la filosofía de diseño adoptada en el sistema Coda, que asigna prioridad a las réplicas en el servidor sobre las copias en la memoria caché. Las copias temporales se almacenan en un *covolumen*, que se asocia a cada volumen en un servidor. Los covolúmenes se parecen a los directorios *lost+found* que se encuentra en los sistemas UNIX convencionales. Reflejan justo a aquellas partes de la estructura del directorio de archivos que se necesitan para guardar los datos temporales. Se requiere poco almacenamiento adicional, ya que los covolúmenes están casi vacíos.

◇ **Rendimiento.** Satyanarayanan y otros [1990] comparan el rendimiento de Coda frente a AFS bajo cargas de referencia diseñadas para simular poblaciones de usuarios en un rango de cinco a cincuenta usuarios AFS típicos.

Sin replicación, no hay diferencias significativas entre el rendimiento del AFS y el del sistema Coda. Con la triple replicación, el tiempo que le lleva al sistema Coda ejecutar una carga de referencia equivalente a cinco usuarios típicos sobrepasa sólo en un 5 % la del AFS sin replicación. Sin embargo, con la triple replicación y una carga equivalente a 50 usuarios, el tiempo necesario para completar la carga de referencia se incrementa hasta un 70 %, mientras que el incremento para el AFS sin replicación es de sólo un 16 %. Esta diferencia se atribuye sólo en parte a los costes operacionales asociados con la replicación (se dice que las diferencias en la sintonización de la implementación explican parte de las diferencias en el rendimiento).

◇ **Discusión.** Anteriormente se apuntó que Coda es similar a Bayou en que también emplea una aproximación optimista para obtener una alta disponibilidad (aunque difieren en otras cosas, siendo una de las más importantes que uno gestiona archivos y el otro bases de datos). Hemos descrito también cómo Coda utiliza los CVV para comprobar la existencia de conflictos, sin tener en cuenta la semántica de los datos archivados en los archivos. La aproximación puede detectar conflictos potenciales *escritura-escritura* pero no conflictos *lectura-escritura*. Éstos son conflictos *potenciales escritura-escritura* porque al nivel de la semántica de la aplicación podría no haber conflictos reales: los clientes podrían haber actualizado diferentes objetos en el archivo de un modo compatible y sería posible realizar una fusión automática simple.

La aproximación global de Coda a la detección de conflictos independientes de la semántica y su resolución manual es sensata en muchos casos, especialmente en aplicaciones que requieran del juicio humano o en sistemas donde no hay conocimiento de la semántica de los datos.

Los directorios son un caso especial en Coda. Algunas veces es posible mantener automáticamente la integridad de esos objetos clave a través de la resolución de los conflictos, ya que sus semánticas son relativamente simples. Los únicos cambios que pueden hacerse en los directorios son la inserción o supresión de entradas en el directorio. Coda incorpora su propio método para solucionar el caso de los directorios. Tiene el mismo efecto que la aproximación de transformación operacional de Bayou, pero Coda fusiona el estado de los directorios en conflicto directamente, ya que no tiene un registro histórico de las operaciones que han efectuado los clientes.

14.5. TRANSACCIONES CON DATOS REPLICADOS

Hasta el momento, en este capítulo se han considerado sistemas en los cuales los clientes solicitan, de cada vez, operaciones individuales sobre conjuntos de objetos replicados. Los Capítulos 12 y 13 explicaban que las transacciones son *secuencias* de una o más operaciones, aplicadas de tal manera que se respetan las propiedades ACID. Al igual que con los sistemas de la Sección 14.4, los objetos en los sistemas transaccionales podrían replicarse para incrementar tanto su disponibilidad como su rendimiento.

Desde el punto de vista de un cliente, una transacción sobre objetos replicados debería parecer idéntica a una con objetos no replicados. En un sistema sin replicación las transacciones parece que se realizan, una de cada vez, siguiendo algún orden. Esto se consigue asegurando un entrelazado secuencial equivalente de las transacciones de los clientes. El efecto de las transacciones realizadas por los clientes sobre objetos replicados debería ser el mismo que si se hubiesen realizado una por una sobre un único conjunto de objetos. A esta propiedad se la denomina *secuenciabilidad de una copia*. Es similar a la consistencia secuencial, pero no hay que confundirlas. La consistencia secuencial considera ejecuciones válidas sin ninguna noción de agregar las operaciones del cliente como transacciones.

Cada gestor de réplicas proporciona control de concurrencia y capacidad de recuperación de sus propios objetos. En esta sección, se asumirá que se utiliza bloqueo en dos etapas para el control de concurrencia.

La recuperación se complica por el hecho de que un gestor de réplicas fallido es un miembro de una colección y que los otros miembros continúan proporcionando el servicio durante el tiempo

que aquél no está disponible. Cuando un gestor de réplicas se recupera de un fallo, utiliza la información que obtiene de los otros gestores de réplicas para restaurar sus propios objetos a sus valores actuales, teniendo en cuenta todos los cambios que han ocurrido durante el tiempo que no estuvo disponible.

En esta sección se presenta primero la arquitectura para realizar transacciones con datos replicados. Las preguntas que atañen a la estructura son: si la petición de un cliente puede enviarse a cualquiera de los gestores de réplicas, cuántos gestores de réplicas se necesitan para que se complete una operación de forma satisfactoria, si un gestor de réplicas con el que contacte un cliente puede aplazar el paso de peticiones hasta que se consume una transacción y cómo llevar a cabo un protocolo de consumación en dos fases.

La implementación de la secuenciabilidad de *una copia* se ilustra mediante *uno lee/todos escriben* (que es un esquema simple de replicación en el cual las operaciones *lee* son realizadas por un único gestor de réplicas y las operaciones *escribe* son realizadas por todos los gestores).

A continuación, en la sección se discuten los problemas de implementar esquemas de replicación cuando hay servidores que se caen y se recuperan. Se introduce la replicación de copias disponibles, una variante del esquema de replicación *uno lee/todos escriben* en el que las operaciones *lee* son realizadas por un único gestor de réplicas y las operaciones *escribe* son realizadas por todos aquellos gestores que están disponibles.

Finalmente, la sección presenta tres esquemas de replicación que funcionan correctamente cuando la colección de gestores de réplicas se divide en subgrupos a causa de una partición en la red:

- *Copias disponibles con validación*: la replicación de copias disponibles se aplica en cada partición y cuando ésta se repara, se aplica un procedimiento de validación y se trata cualquier inconsistencia.
- *Consenso con quórum*: un subgrupo debe tener un quórum (lo que significa que tiene suficientes miembros) para que se le permita continuar proporcionando un servicio cuando se produce una partición. Cuando ésta es reparada, y cuando un gestor de réplicas se reinicia tras un fallo, los gestores de réplicas consiguen actualizar sus objetos por medio de procedimientos de recuperación.
- *Partición virtual*: una combinación de consenso con quórum y copias disponibles. Si una partición virtual tiene quórum puede usar replicación de las copias disponibles.

14.5.1. ARQUITECTURAS PARA TRANSACCIONES REPLICADAS

Al igual que con la serie de sistemas que ya se han considerado en secciones anteriores, un frontal podría tanto multidifundir las peticiones de un cliente a grupos de gestores de réplicas o bien podría enviar cada petición a un gestor de réplicas individual, el cual es entonces responsable del procesamiento de la petición y de la respuesta al cliente. Wiesmann y otros [2000] y Schiper y Raynal [1996] consideran el caso de peticiones por multidifusión que no se tratará aquí. A partir de aquí, se asumirá que un frontal envía las peticiones de un cliente a un gestor del grupo de gestores de réplicas de un objeto lógico. En la aproximación mediante *copia primaria*, todos los frontales se comunican con un gestor de réplicas distinguido como *primario* para realizar una operación; ese gestor de réplicas mantiene actualizadas las copias de respaldo. De forma alternativa, los frontales podrían comunicarse con cualquier gestor de réplicas para realizar una operación, pero en ese caso la coordinación entre los gestores de réplicas es, en consecuencia, más compleja.

El gestor de réplicas que recibe una petición para realizar una operación sobre un objeto concreto es responsable de conseguir la cooperación de otros gestores de réplicas en el grupo que tienen copias de ese objeto. Los diferentes esquemas de replicación tienen reglas distintas relativas

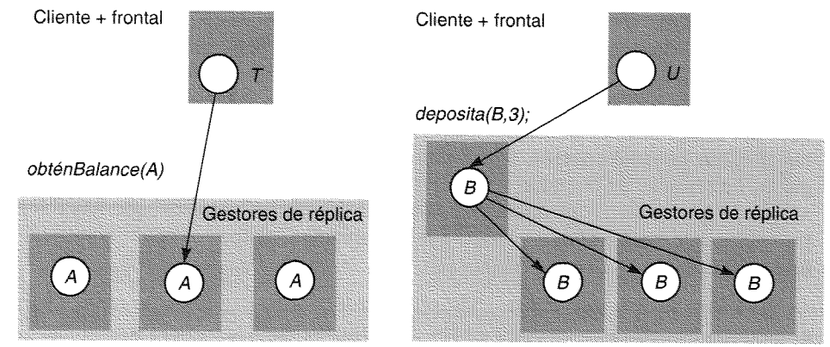


Figura 14.10. Transacciones sobre datos replicados.

a cuántos gestores de réplicas de un grupo son necesarios para completar con éxito una operación. Por ejemplo, en el esquema *uno lee/todos escriben*, una petición *lee* puede ser realizada por un único gestor de réplicas, mientras que una petición *escribe* debe ser llevada a cabo por todos los gestores de réplicas en el grupo, como se muestra en la Figura 14.10 (puede haber diferentes números de réplicas de los distintos objetos). Los esquemas de consenso con quórum se diseñan para reducir el número de gestores de réplicas que deben realizar operaciones de actualización, pero a cambio incrementan el número de gestores de réplicas requeridos para realizar las operaciones de *sólo lectura*.

Otro asunto a tratar es si el gestor de réplicas contactado por un frontal debería aplazar el paso de peticiones de actualización a otros gestores de réplica dentro del grupo hasta que la transacción se haya consumado (conocida como aproximación *perezosa* a la propagación de actualizaciones); o, por el contrario, si los gestores de réplicas deberían enviar cada petición de actualización a todos los gestores de réplicas necesarios dentro de la transacción antes de consumarla (lo que se conoce como la aproximación *ansiosa* o *acuciante*). La aproximación perezosa es una alternativa atractiva ya que reduce la cantidad de comunicación entre los gestores de réplicas que tiene lugar antes de responder al cliente que está haciendo la actualización. Sin embargo, también hay que considerar el control de concurrencia. Algunas veces, la aproximación perezosa se utiliza con la replicación de copia primaria (véase más adelante), donde un único gestor de réplicas primario secuencia las transacciones. No obstante, si varias transacciones distintas pueden intentar acceder a los mismos objetos en distintos gestores de réplicas dentro de un grupo entonces, para asegurar que las transacciones son secuenciadas correctamente en todos los gestores de réplicas del grupo, cada gestor de réplicas necesita conocer las peticiones atendidas por los otros. En este caso, la aproximación ansiosa es la única viable.

◇ **Protocolo de consumación en dos fases.** Este protocolo se convierte en un protocolo de consumación en dos fases anidado en dos niveles. Al igual que antes, el coordinador de una transacción se comunica con los trabajadores. Pero, si bien el coordinador o uno de los trabajadores es un gestor de réplicas, se comunicará con los otros gestores de réplicas a los cuales les pasó peticiones durante la transacción.

Esto es, en la primera fase, el coordinador envía el mensaje *puedoConsumar?* a los trabajadores, que lo pasan a los otros gestores de réplicas y recogen sus respuestas antes de responder al coordinador. En la segunda fase, el coordinador envía la solicitud *Consuma* o *Aborta*, que se pasa a los miembros de los grupos de los gestores de réplicas.

◇ **Replicación de copia primaria.** La replicación de copia primaria puede utilizarse en el contexto de transacciones. En este esquema, todas las peticiones de los clientes (sean o no de *sólo lectura*) se dirigen a un único gestor de réplica primario (véase la Figura 14.4). Para la replicación de copia primaria, se aplica el control de concurrencia sobre el primario. Para consumar una transacción, el primario se comunica con los gestores de réplicas que son copia de seguridad y a continuación, en la aproximación ansiosa, responde al cliente. Este método de replicación permite a un gestor de réplicas de respaldo reemplazar al primario de forma consistente si éste falla. En la aproximación perezosa, el primario responde a los frontales antes de que haya actualizado sus copias primarias. En ese caso, un respaldo que reemplace a un frontal caído no tendrá, necesariamente, el último estado de la base de datos.

◇ **Uno lee/todos escriben.** Usaremos este simple esquema de replicación para ilustrar cómo el bloqueo en dos fases en cada gestor de réplicas puede ser usado para conseguir secuenciabilidad de *una copia*, donde los frontales puede comunicarse con cualquier gestor de réplicas. Toda operación de *escribe* debe realizarse en todos los gestores de réplicas, cada uno de los cuales establece un bloqueo de escritura sobre el objeto al que afecta la operación. Cada operación de *lee* se realiza por un único gestor de réplicas, que establece un bloqueo de lectura sobre el objeto al que afecta la operación.

Considere parejas de operaciones de transacciones distintas sobre el mismo objeto: cualquier par de operaciones *escribe* requerirán bloqueos que entran en conflicto en todos los gestores de réplicas; una operación *lee* y una operación de *escribe* requerirán bloqueos que entren en conflicto en un gestor de réplicas individual. De esta forma se consigue la capacidad de secuenciación de *una copia*.

14.5.2. REPLICACIÓN DE COPIAS DISPONIBLES

El esquema de replicación simple *uno lee/todos escriben* no es realista, dado que no puede salir adelante si uno de los gestores de réplicas no está disponible, bien porque se ha caído o bien por un fallo en la comunicación. El esquema de copias disponibles se diseña para permitir que algunos de los gestores de réplicas no estén disponibles durante un tiempo. La estrategia consiste en que una petición *lee* por parte de un cliente sobre un objeto lógico puede ser realizada por cualquier gestor de réplicas disponible, pero la petición de actualización por parte de un cliente debe ser realizada por todos los gestores de réplicas disponibles en el grupo que tengan copias del objeto. La idea de *miembros disponibles de un grupo de gestores de réplicas* es similar al grupo disponible de almacenamiento del volumen de Coda que se describió en la Sección 14.4.3.

En un caso normal, las peticiones del cliente son recibidas y realizadas por un gestor de réplicas en funcionamiento. Las peticiones *lee* pueden ser realizadas por parte del gestor de réplicas que las recibe. Las peticiones *escribe* son realizadas por el gestor de réplicas que las recibe y por todos los gestores de réplicas disponibles en el grupo. Por ejemplo, en la Figura 14.11, la operación de *obténBalance* de la transacción *T* es realizada por *X*, mientras que la operación *deposita* de *T* es realizada por *M*, *N* y *P*. El control de concurrencia en cada gestor de réplicas afecta a las operaciones que se realizan localmente. Por ejemplo, en *X*, la transacción *T* ha leído *A* y, por lo tanto, a la transacción *U* no se le permite actualizar *A* con la operación *deposita* hasta que la transacción *T* se haya completado. Mientras no cambie el conjunto de gestores de réplicas disponibles, el control de concurrencia local la secuenciabilidad de *una copia* de la misma forma que en la replicación *uno lee/todos escriben*. Desafortunadamente, no sucede esto si uno de los gestores de réplicas falla o se recupera mientras se están realizando las transacciones que entran en conflicto.

◇ **Fallo de un gestor de réplicas.** Se asume que los gestores de réplicas sufren fallos benignos por caída. Sin embargo, un gestor de réplicas caído es reemplazado por un nuevo proceso, que

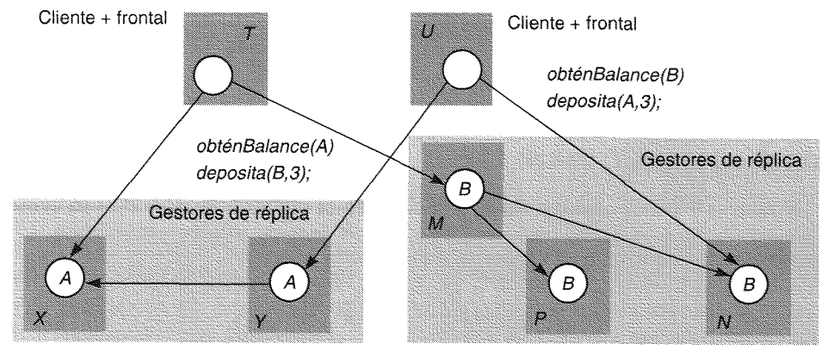


Figura 14.11. Copias disponibles.

recupera el estado consumado de los objetos desde un archivo de recuperación. Los frontales utilizan timeouts para decidir que un gestor de réplicas no está actualmente disponible. Cuando un cliente hace una petición a un gestor de réplicas que se ha caído, el frontal contempla un timeout y reintenta la petición en otro gestor de réplicas del grupo. Si la petición es recibida por un gestor de réplicas en el cual el objeto está desfasado porque el gestor de réplicas no se ha recuperado totalmente de un fallo, éste rechaza la petición y el frontal reintenta la petición en otro gestor de réplicas del grupo.

La secuenciabilidad de *una copia* requiere que las caídas y recuperaciones sean secuenciadas con respecto a las transacciones. Una transacción observa que ha ocurrido un fallo una vez que ha finalizado o antes de que comience, en función de si puede acceder a un objeto o no. La secuenciabilidad de *una copia* no se consigue cuando transacciones distintas realizan observaciones sobre fallos que entran en conflicto.

Considérese el caso de la Figura 14.11, donde el gestor de réplicas *X* falla justo después de que la transacción *T* haya realizado la operación *obténBalance*, y el gestor de réplicas *N* falla justo después de que *U* haya realizado la misma operación. Supóngase que ambos gestores de réplicas fallan antes de que *T* y *U* hayan realizado las operaciones de *deposita*. Esto implica que el *deposita* de *T* se realizará desde los gestores de réplicas *M* y *P*, y que el *deposita* de *U* será realizado en el gestor de réplicas *Y*. Desafortunadamente, el control de concurrencia sobre *A* en el gestor de réplicas *X* no evita que la transacción *U* actualice *A* en el gestor de réplicas *Y*. Ni el control de concurrencia sobre *B* en el gestor de réplicas *N* evita que la transacción *T* actualice *B* en los gestores de réplicas *M* y *P*.

Esto va en contra del requisito de secuenciabilidad de *una copia*. Si estas operaciones tuviesen que ser realizadas sobre copias individuales de los objetos, podrían ponerse en serie bien como la transacción *T* antes que la *U* o como la transacción *U* antes que la *T*. Esto aseguraría que una de las transacciones leerá el valor fijado por la otra. El control de concurrencia local sobre copias de objetos no es suficiente para asegurar la secuenciabilidad de *una copia* en el esquema de replicación de copias disponibles.

Ya que las operaciones *escribe* se dirigen a todas las copias disponibles, el control de concurrencia local asegura que las escrituras que entren en conflicto sobre un conjunto están secuenciadas. Por el contrario, una operación *lee* por parte de una transacción y una *escribe* por parte de otra pueden no afectar necesariamente a la misma copia de un objeto. Por lo tanto, el esquema requiere control de concurrencia adicional para evitar las dependencias que formen un ciclo, entre una operación *lee* en una transacción y una operación *escribe* de otra. Tales dependencias no pueden surgir si los fallos y las recuperaciones de las réplicas de objetos se secuencian con respecto a las transacciones.

◇ **Validación local.** El procedimiento adicional de control de concurrencia se conoce como validación local. El procedimiento de validación local se diseña para asegurar que no parezca ocurrir ningún evento asociado a un fallo o a una recuperación durante el desarrollo de una transacción. En nuestro ejemplo, ya que *T* ha leído de un objeto en *X*, el fallo en *X* debe ocurrir tras *T*. De forma similar, ya que *T* observa el fallo de *N* cuando intenta actualizar el objeto, el fallo de *N* debe producirse antes que *T*. Esto es:

N falla → *T* lee el objeto *A* en *X*; *T* escribe en el objeto *B* en *M* y *P* → se consuma *T* → *X* falla

También se puede argumentar para la transacción *U* que:

X falla → *U* lee el objeto *B* en *N*; *U* escribe en el objeto *A* en *Y* → se consuma *U* → *N* falla

El procedimiento de validación local asegura que no pueden ocurrir secuencias incompatibles tales. Antes de que se consume una transacción se comprueba cualquier fallo (y recuperación) de gestores de réplicas de objetos a los que ha accedido. En el ejemplo, la transacción *T* comprobaría que *N* está aún indispuesto y que *X*, *M* y *P* están todavía disponibles. Si fuese ése el caso, puede consumarse *T*. Esto implica que *X* falla tras ser validado por *T* y antes de ser validado *U*. Dicho de otra forma, la validación de *U* ocurre tras la validación de *T*. La validación de *U* falla porque *N* ya ha fallado.

En el momento en que una transacción ha observado un fallo, el procedimiento de validación local intenta comunicarse con los gestores de réplicas fallidos para asegurarse de que no se han recuperado. La otra parte de la validación local, que está comprobando que los gestores de réplicas no han fallado mientras se accedía a los objetos, puede combinarse con el protocolo de consumación en dos fases.

Los algoritmos para copias disponibles no pueden utilizarse en entornos en los cuales los gestores de réplicas que están operando son incapaces de comunicarse entre ellos.

14.5.3. PARTICIONES EN LA RED

Los esquemas de replicación han de tener en cuenta la posibilidad de que se produzcan particiones en la red. Una partición en la red separa un grupo de gestores de réplicas en dos o más subgrupos, de tal forma que los miembros de cada subgrupo pueden comunicarse entre ellos, pero los miembros de distintos subgrupos no pueden comunicarse. Por ejemplo, en la Figura 14.12 los gestores de réplicas que reciben la petición *deposita* no pueden enviarla a los gestores de réplicas que reciben la petición *extrae*.

Los esquemas de replicación se diseñan bajo la suposición de que las particiones serán finalmente reparadas. Por lo tanto, los gestores de réplicas dentro de una partición individual deben asegurarse de que cualesquiera peticiones que ejecuten durante una partición no harán inconsistentes al conjunto de réplicas cuando se repare la partición.

Davidson y otros [1985] discuten muchas aproximaciones distintas, que categorizan bien como optimistas o bien como pesimistas en lo que respecta a la probabilidad de que ocurran inconsistencias. Los esquemas optimistas no limitan la disponibilidad durante una partición, mientras que los esquemas pesimistas sí que lo hacen.

Los esquemas optimistas permiten las actualizaciones en todas las particiones, lo que puede llevar a inconsistencias entre particiones, que han de resolverse cuando se repare la partición. Un ejemplo de esta aproximación es una variante del algoritmo de copias disponibles en el cual se permiten las actualizaciones dentro de las particiones y, una vez que la partición haya sido reparada, se validan dichas actualizaciones (se aborta cualquier actualización que rompa el criterio de capacidad de secuenciación para *una copia*).

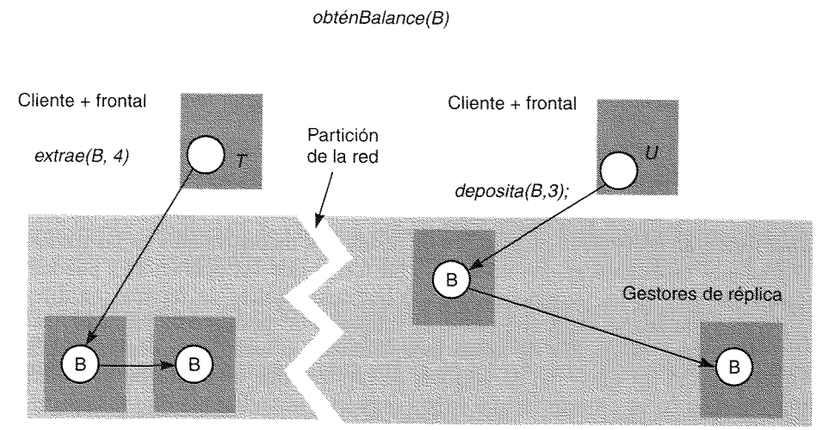


Figura 14.12. Partición de la red.

La aproximación pesimista limita la disponibilidad incluso cuando no hay particiones, pero evita las inconsistencias que ocurren mientras hay particiones. Cuando se repara la partición, todo lo que se necesita hacer es actualizar las copias de los objetos. La aproximación de consenso con quórum es pesimista. Permite actualizaciones en una partición que tenga la mayoría de sus gestores de réplicas y propaga las actualizaciones a los otros gestores de réplicas cuando se repara la partición.

14.5.4. COPIAS DISPONIBLES CON VALIDACIÓN

El algoritmo de copias disponibles se aplica dentro de cada partición. Esta aproximación optimista mantiene el nivel normal de disponibilidad para las operaciones *lee*, incluso durante las particiones. Cuando ésta se repara, se validan las transacciones que pudiesen estar en conflicto y que tuvieron lugar dentro de las particiones separadas. Si falla la validación, deben darse algunos pasos para salvar las inconsistencias. Si no hubiese habido partición, se retrasaría o abortaría una de las transacciones de un par que haya entrado en conflicto. Desafortunadamente, como hubo una partición, al par de transacciones en conflicto se les ha permitido consumarse en particiones diferentes. La única elección posible tras este suceso es que se aborte una de ellas. Esto requiere hacer cambios en los objetos y, en algunos casos, compensar sus efectos en el mundo real, como por ejemplo gestionar descubiertos en cuentas bancarias. La aproximación optimista sólo es factible en aplicaciones donde pueden realizarse dichas acciones compensatorias.

Se pueden utilizar vectores de versiones para validar conflictos entre pares de operaciones *escribe*. Dichos vectores se usan en el sistema de archivos Coda y se describen en la Sección 14.4.3. La aproximación no puede detectar conflictos de lectura-escritura pero funcionan bien en los sistemas de archivos, donde las transacciones tienden a acceder a un único archivo y los conflictos de lectura-escritura no son importantes. No es adecuada para las aplicaciones tales como el ejemplo del banco que se comentó antes, donde sí que son importantes dichos conflictos.

Davidson [1984] utilizó los *grafos de precedencia* para detectar inconsistencias entre particiones. Cada partición mantiene un registro histórico de los objetos afectados por las operaciones *lee* y *escribe* de las transacciones. Este registro se utiliza para construir un grafo de precedencia cuyos nodos son las transacciones y cuyos arcos representan los conflictos entre las operaciones *lee* y

escribe de las transacciones. Dicho grafo no contendrá ciclos, ya que se ha aplicado control de concurrencia dentro de la partición. El procedimiento de validación toma los grafos de precedencia de las particiones, y añade arcos, que representan conflictos, entre transacciones en particiones distintas. Si el grafo resultante contiene ciclos, entonces falla la validación.

14.5.5. MÉTODOS DE CONSENSO CON QUÓRUM

Una forma de prevenir la producción de resultados inconsistentes en las transacciones en diferentes particiones es establecer una norma, de tal forma que las operaciones puedan llevarse a cabo sólo dentro de una de las particiones. Puesto que los gestores de réplicas en diferentes particiones no pueden comunicarse unos con otros, el subgrupo de gestores de réplicas dentro de cada partición debe ser capaz de decidir de modo independiente si están autorizados a efectuar las operaciones. Un quórum es un subgrupo de gestores de réplicas que está autorizado a efectuar las operaciones. Por ejemplo, si el criterio es que exista mayoría, un subgrupo que tiene la mayoría de los miembros del grupo formaría un quórum ya que ningún otro subgrupo podría tener la mayoría.

En los esquemas de replicación con consenso con quórum, una operación de actualización de un objeto lógico podría ser completada con éxito por un subgrupo de su grupo de gestores de réplicas. Por lo tanto, los otros miembros del grupo tendrán copias desfasadas del objeto. Los números de versión o las marcas temporales se podrían usar para determinar si las copias están actualizadas. Si se usan versiones, el estado inicial de un objeto es la primera versión, y tras cada cambio, tenemos una nueva versión. Cada copia de un objeto tiene un número de versión, pero sólo las copias que están actualizadas tienen el número de versión actual, mientras que las copias desfasadas tienen números de versión anteriores. Las operaciones deberían aplicarse sólo a aquellas copias que tengan el número de versión actual.

Gifford [1979a] desarrolló un esquema de replicación de archivos en el cual se asigna un número de *votos* a cada copia física en un gestor de réplicas de un archivo lógico individual. Cada voto puede considerarse como un peso relacionado con la conveniencia de usar una copia en particular. Cada operación *lee* debe obtener primero un quórum de lectura de L votos antes de que pueda proseguir la lectura de cualquier copia actualizada. Y cada operación *escribe* debe obtener un quórum de escritura de E votos antes de que pueda proseguir con una operación de actualización. Para un grupo de gestores de réplicas E y L se fijan de la siguiente forma:

$E >$ la mitad de los votos totales
$L + E >$ número total de votos para el grupo

Esto garantiza que cualquier pareja de subgrupos, consistentes en un quórum de lectura y en un quórum de escritura o dos quorum de escritura, deben contener copias comunes. Por lo tanto, si hay una partición no es posible realizar operaciones que entren en conflicto sobre la misma copia, aunque en diferentes particiones.

Para realizar una operación *lee* se recoge un quórum de lectura, haciendo las suficientes indagaciones sobre los números de versión como para encontrar un conjunto de copias cuya suma de votos no sea menor que L. No todas estas copias tienen por qué estar actualizadas. Ya que cada quórum de lectura se solapa con cada quórum de escritura, cada quórum de lectura estará seguro de que incluye una copia actual. La operación de lectura puede aplicarse a cualquier copia actualizada.

Para efectuar una operación *escribe* se recoge un quórum de escritura, haciendo las suficientes indagaciones sobre los números de versión como para encontrar un conjunto de gestores de réplicas con copias actualizadas, cuya suma de votos no sea menor que E. Si no hay suficientes copias actualizadas, entonces un archivo no actual se reemplaza por una copia del archivo actual, para que se establezca un quórum. Las actualizaciones especificadas en la operación *escribe* son entonces aplicadas por cada gestor de réplicas en el quórum de escritura, se incrementa el número de versión y se informa al cliente de la finalización de la escritura.

Los archivos en los restantes gestores de réplicas disponibles se actualizan a continuación realizando la operación *escribe* como una tarea en segundo plano. Cualquier gestor de réplicas, cuya copia del archivo tenga un número de versión más antiguo que el usado por el quórum de escritura, actualizará dicha copia reemplazando el archivo completo por una obtenida de un gestor de réplicas actualizado.

En el esquema de replicación de Gifford, puede utilizarse el bloqueo de lectura/escritura en dos fases para el control de concurrencia. La indagación preliminar sobre el número de versión que se hace para obtener el quórum de lectura, L, provoca que se establezcan bloqueos de lectura en cada gestor de réplicas contactado. Cuando se aplica una operación *escribe* al quórum de escritura, E, se establece un bloqueo de escritura en cada uno de los gestores de réplicas involucrados (los bloqueos se aplican con la misma granularidad que la de los números de versión). Los bloqueos aseguran la secuenciabilidad de *una copia* ya que cualquier quórum de lectura se solapa con algún quórum de escritura y dos quórum de escritura cualesquiera se solaparían.

◇ **Configurabilidad de los grupos de gestores de réplicas.** Una propiedad importante del algoritmo de votación ponderada es que los grupos de gestores de réplicas puedan ser configurados para proporcionar características distintas de rendimiento y fiabilidad. Una vez que se establece la fiabilidad y el rendimiento generales de un grupo de gestores de réplicas mediante su configuración de votos, la fiabilidad y el rendimiento de las operaciones *escribe* pueden aumentarse disminuyendo E y, de forma similar, disminuyendo L para las operaciones de *lee*.

El algoritmo puede, además, permitir la utilización de copias de archivos en los servidores o en discos locales en los computadores de los clientes. Estas copias locales se consideran como *representantes débiles* y se les asignan cero votos. Esto asegura que no se les incluirá en ningún quórum. Una operación *lee* puede realizarse en cualquier copia actualizada una vez que se ha obtenido un quórum de lectura. En consecuencia, puede llevarse a cabo una operación *lee* en la copia local del archivo si ésta está al día. Los representantes débiles pueden utilizarse para acelerar las operaciones *lee*.

◇ **Un ejemplo del trabajo de Gifford.** Gifford proporciona tres ejemplos que muestran el abanico de propiedades que pueden conseguirse asignando pesos a los distintos gestores de réplicas en un grupo y asignando L y E de forma apropiada. A continuación se reproducen los ejemplos de Gifford, que se basan en la tabla próxima. Las probabilidades de bloqueo dan una indicación de la probabilidad de que no pueda obtenerse un quórum cuando se realiza una petición *lee* o *escribe*. Se calculan suponiendo que hay una probabilidad de 0,01 de que cualquier gestor de réplicas no se encuentre disponible en el momento de la petición.

El ejemplo 1 está configurado para un archivo con una tasa alta de *lectura/escritura* en una aplicación con distintos representantes débiles y un único gestor de réplicas. La replicación se utiliza para mejorar el rendimiento del sistema, no su fiabilidad. Existe en la red local un gestor de réplicas al que se puede acceder en 75 milisegundos. Dos clientes han decidido tener representantes débiles en sus discos locales, a los que pueden acceder en 65 milisegundos, produciendo una menor latencia y menor tráfico en la red.

El ejemplo 2 está configurado para un archivo con una tasa moderada de *lectura/escritura*, al que se accede fundamentalmente desde una red local. Al gestor de réplicas en la red local se le asignan dos votos y a los gestores de réplicas en las redes remotas se les asigna un voto a cada uno. Las lecturas pueden satisfacerse utilizando sólo el gestor de réplicas local, pero las escrituras deben acceder al gestor de réplicas local y a uno de los gestores de réplicas remotos. El archivo permanecerá disponible en el modo de *sólo lectura* si falla el gestor de réplicas local. Los clientes podrían crear representantes débiles para tener una menor latencia en las lecturas.

		Ejemplo 1	*Ejemplo 2*	*Ejemplo 3*
Latencia (milisegundos)	Réplica 1 Réplica 2 Réplica 3	75 65 65	75 100 750	75 750 750
Configuración de votación	Réplica 1 Réplica 2 Réplica 3	1 0 0	2 1 1	1 1 1
Tamaños de quórum	*L* *E*	1 1	2 3	1 3
Rendimiento derivado de la colección de archivos:				
Lectura	Latencia Probabilidad de bloqueo	65 0,01	75 0,0002	75 0,000001
Escritura	Latencia Probabilidad de bloqueo	75 0,01	100 0,0101	750 0,03

El ejemplo 3 está configurado para un archivo con una tasa de *lectura/escritura* muy alta, tal como un directorio del sistema en un entorno de *tres gestores de réplicas*. Los clientes pueden leer de cualquier gestor de réplicas, y la probabilidad de que el archivo no esté disponible es pequeña. Las actualizaciones deben aplicarse a todas las copias. Una vez más, los clientes podrían crear representantes débiles en sus máquinas locales para tener una latencia de lectura menor.

La principal desventaja del consenso con quórum es que el rendimiento de las operaciones *lee* se reduce por la necesidad de recoger un quórum de lectura por parte de *L* gestores de réplicas.

Herlihy [1986] propuso una extensión del método de consenso con quórum para tipos de datos abstractos. El método permite que se tenga en cuenta la semántica de las operaciones y, en consecuencia, incrementar la disponibilidad de los objetos. El método de Herlihy utiliza marcas temporales en lugar de números de versiones. Esto tiene la ventaja de que no es necesario hacer indagaciones acerca de los números de versiones para obtener un nuevo número de versión antes de realizar una operación de escritura. Según Herlihy, la principal ventaja es que el uso del conocimiento de la semántica puede incrementar el número de elecciones para un quórum.

14.5.6. EL ALGORITMO DE LA PARTICIÓN VIRTUAL

Este algoritmo, que fue propuesto por El Abbadi y otros [1985], combina la aproximación del consenso con quórum con el algoritmo de copias disponibles. La técnica de consenso con quórum funciona perfectamente en presencia de particiones, pero la de copias disponibles es menos costosa para las operaciones *lee*. Una *partición virtual* es una abstracción de una partición real y contiene un conjunto de gestores de réplicas. Nótese que el término *partición de la red* se refiere a la barrera que coloca a los gestores de réplicas en distintas partes, mientras que el término *partición virtual* se refiere a las partes en sí mismas. Aunque no están conectadas mediante multidifusión, las

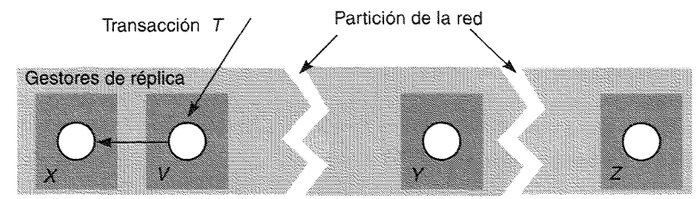


Figura 14.13. Dos particiones en la red.

particiones virtuales son similares a las vistas de grupo, que se presentaron en la Sección 14.2.2. Una transacción puede operar en una partición virtual si contiene suficientes gestores de réplicas para tener un quórum de lectura y un quórum de escritura para los objetos a los que accede. En este caso, la transacción utiliza el algoritmo de copias disponibles. Éste tiene la ventaja de que las operaciones de *lee* siempre necesitan acceder solamente a una copia individual de un objeto y puede mejorar el rendimiento eligiendo la copia *más cercana*. Si un gestor de réplicas falla y la partición virtual cambia a lo largo de una transacción, dicha transacción aborta. Esto asegura la secuenciabilidad de *una copia* porque todas las transacciones que sobreviven son capaces de observar en el mismo orden los fallos y las recuperaciones de los gestores de réplicas.

Cuandoquiera que un miembro de una partición virtual detecta que no puede acceder a uno de los otros miembros (por ejemplo, cuando no reciben acuse de recibo de una operación *escribe*), intenta crear una nueva partición virtual con una vista para obtener una partición virtual con quórum de lectura y escritura.

Supongamos, por ejemplo, que se tienen cuatro gestores de réplicas *V*, *X*, *Y* y *Z*, cada uno de los cuales tiene un voto, y que los quórum de lectura y escritura son $L = 2$ y $E = 3$. Inicialmente, todos los gestores pueden contactar unos con otros. Mientras se mantengan en contacto, pueden usar el algoritmo de las copias disponibles. Por ejemplo, una transacción *T* que consta de una operación *lee* seguida de *escribe* efectuará *lee* en un único gestor de réplicas (por ejemplo, *V*) y la operación de *escribe* en los cuatro.

Supóngase que la transacción *T* comienza realizando su *lee* en *V* en un instante en el que *V* está todavía en contacto con *X*, *Y* y *Z*. Ahora, supóngase que se produce una partición en la red, como en la Figura 14.13, en la cual *V* y *X* están en una parte e *Y* y *Z* están en otra diferente. Entonces, cuando la transacción *T* intenta aplicar su *escribe*, *V* se dará cuenta de que no puede contactar con *Y* y *Z*.

Cuando un gestor de réplicas no puede contactar con otros gestores con los que previamente podía contactar, sigue intentándolo hasta que puede crear una nueva partición virtual. Por ejemplo, *V* seguirá intentando contactar con *Y* y *Z* hasta que una o ambas contesten, como por ejemplo en la Figura 14.14 cuando se puede acceder a *Y*. El grupo de gestores de réplicas *V*, *X* e *Y* conforman una partición virtual, ya que son suficientes para formar quórum de lectura y escritura.

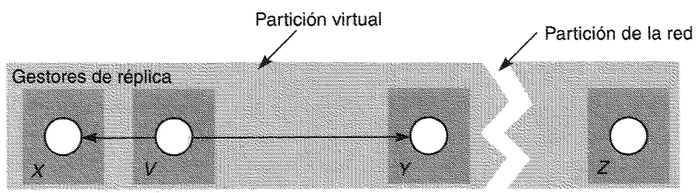


Figura 14.14. Partición virtual.

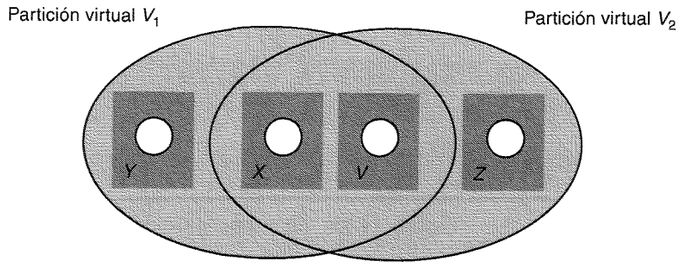


Figura 14.15. Dos particiones virtuales que se solapan.

Cuando se crea una nueva partición virtual durante una transacción que ha realizado una operación en uno de los gestores de réplicas (como la transacción *T*), la transacción debe ser abortada. Además, las réplicas dentro de una nueva partición virtual deben actualizarse copiándolas desde otras réplicas. Los números de versión pueden utilizarse como en el algoritmo de Gifford para determinar qué copias están actualizadas. Es esencial que todas las réplicas estén actualizadas porque las operaciones de *lee* se realizan en cualquier réplica individual.

◇ **Implementación de las particiones virtuales.** Una partición virtual tiene un tiempo de creación, un conjunto de miembros potenciales y un conjunto de miembros reales. Los tiempos de creación son marcas temporales lógicas. Los miembros reales de una partición virtual en particular comparten la misma idea en lo que se refiera a su tiempo de creación y a su pertenencia (una *vista* compartida de los gestores de réplicas con los cuales se pueden comunicar). Por ejemplo, en la Figura 14.14 los miembros potenciales son *V*, *X*, *Y*, *Z* y los miembros reales son *V*, *X* e *Y*.

La creación de una nueva partición virtual se consigue mediante un protocolo cooperativo efectuado por aquellos miembros potenciales accesibles por los gestores de réplicas que lo inician. Varios gestores de réplicas pueden intentar crear una nueva partición virtual simultáneamente. Por ejemplo, supóngase que los gestores de réplicas *Y* y *Z*, que se muestran en la Figura 14.13, continúan haciendo intentos de contactar con los otros, y pasado un rato se repara parcialmente la partición de la red, de tal modo que *Y* no puede comunicarse con *Z*, pero los dos grupos *V*, *X*, *Y* y *V*, *X*, *Z* pueden comunicarse entre ellos. Entonces, existe el riesgo de que pudieran crearse dos particiones virtuales solapadas, tales como V_1 y V_2 que se muestran en la Figura 14.15.

Considérese el efecto de ejecutar diferentes transacciones en dos particiones virtuales. La operación *lee* de la transacción en *V*, *X*, *Y* podría aplicarse en el gestor de réplicas *Y*, en cuyo caso su bloqueo de lectura no entrará en conflicto con bloqueos de lectura fijados por una operación *escribe* por parte de una transacción en la otra partición virtual. Las particiones virtuales solapadas van en contra de la secuenciabilidad de *una copia*.

La intención del protocolo es crear nuevas particiones virtuales de modo consistente, incluso si se producen particiones reales durante la ejecución del protocolo. El protocolo de creación de nuevas particiones virtuales tiene dos fases, como se muestra en la Figura 14.16.

Un gestor de réplica que contesta *Sí* en la Fase 1 no pertenece a una partición virtual hasta que reciba el correspondiente mensaje de *Confirmación* en la Fase 2.

En nuestro ejemplo anterior, los gestores de réplicas *Y* y *Z* que se muestran en el Figura 14.13 intentan crear, cada uno, una partición virtual, y el que tenga la marca temporal lógica más grande será el que al final lo consiga.

Éste es un método eficaz cuando no se producen particiones habitualmente. Cada transacción utiliza el algoritmo de copias disponibles dentro de una partición virtual.

Fase 1:

- El iniciador manda una petición *Unirse* a cada miembro potencial. El argumento de *Unirse* es una marca temporal lógica propuesta para la nueva partición virtual.
- Cuando un gestor de réplicas recibe una petición de *Unirse* compara la marca temporal lógica propuesta con la de su partición virtual actual.
  - — Si la marca temporal lógica propuesta es mayor accede a la unión y contesta *Sí*.
  - — Si es menor, rechaza la unión y contesta *No*.

Fase 2:

- Si el iniciador ha recibido el número suficiente de respuestas *Sí* como para tener quórum para *escritura* y de *lectura*, puede completar la creación de la nueva partición virtual enviando mensajes de *Confirmación* a los sitios que acceden a unirse. Se envían como argumentos la marca temporal de creación y la lista de los miembros actuales.
- Los gestores de réplicas que reciben el mensaje de *Confirmación* se unen a la nueva partición virtual y graban su marca temporal de creación y la lista de los miembros actuales.

Figura 14.16. Creación de una partición virtual.

14.6. RESUMEN

La replicación de objetos es un medio importante de conseguir servicios con buen rendimiento, alta disponibilidad y tolerancia a fallos en sistemas distribuidos. Se han descrito arquitecturas para servicios en los cuales los gestores de réplicas almacenan las réplicas de los objetos, y en los cuales los frontales hacen transparente la replicación. Los clientes, frontales y los gestores de réplicas pueden ser procesos separados o residir en el mismo espacio de direcciones.

El capítulo comenzó describiendo un modelo de sistema en el cual cada objeto lógico se implementa por medio de un conjunto de réplicas físicas. Frecuentemente, las actualizaciones de estas réplicas pueden realizarse de forma apropiada mediante la comunicación en grupo. Se extendió el concepto de comunicación en grupo para incluir los servicios de pertenencia a grupo y la comunicación mediante vistas síncronas.

Se definieron la linealizabilidad y la consistencia secuencial como criterios de corrección para los servicios tolerantes a fallos. Estos criterios expresan cómo los servicios han de proporcionar el equivalente a una única imagen del conjunto de objetos lógicos, incluso cuando dichos objetos se replican. El criterio con más significado práctico es la consistencia secuencial.

En la replicación pasiva (primario-respaldo), la tolerancia a fallos se consigue dirigiendo todas las peticiones a través de gestores de réplicas distinguidos y disponiendo de un gestor de réplicas de respaldo que asumirá el papel si otro falla. En la replicación activa, todos los gestores de réplicas procesan todas las peticiones de forma independiente. Ambas formas de replicación puede implementarse de forma conveniente utilizando la comunicación en grupo.

A continuación se estudiaron los servicios con alta disponibilidad. Tanto el sistema cotilla como el de Bayou permiten que sus clientes realicen actualizaciones sobre las réplicas locales mientras existe una partición. En ambos sistemas, los gestores de réplicas se intercambian actualizaciones cuando vuelven a encontrarse conectados. El sistema cotilla proporciona su mayor disponibilidad a costa de una consistencia causal relajada. El Bayou proporciona unas mayores garantías de consistencia eventuales, utilizando una detección automática de conflictos y la técnica de transformación operacional para resolver dichos conflictos. Coda es un sistema de archivos con alta disponibilidad, que utiliza vectores de versiones para detectar actualizaciones que potencialmente pueden entrar en conflicto.

Finalmente, se consideró las prestaciones de las transacciones en presencia de datos replicados. Para este caso existen tanto arquitecturas primario-respaldo y arquitecturas en las cuales los frontales pueden comunicarse con cualquier gestor de réplicas. Se discutió cómo los sistemas de transacciones permiten los fallos en los gestores de réplicas y las particiones en la red. Las técnicas de copias disponibles, quórum con consenso y particiones virtuales permiten que las operaciones dentro de las transacciones prosigan en algunas circunstancias en las cuales no se pueden alcanzar todos los gestores de réplicas.

EJERCICIOS

14.1. Tres computadores proporcionan conjuntamente un servicio replicado. Los fabricantes aseguran que cada computador tiene un tiempo medio entre fallos de cinco días, y que un fallo típico necesita cuatro horas para repararse. ¿Cuál es la disponibilidad del servicio replicado?

14.2. Explique por qué un servidor multi-hilo podría no calificarse como una máquina de estado.

14.3. En un juego multi-usuario, los jugadores mueven las figuras sobre una escena común. El estado del juego se replica en las estaciones de trabajo y en un servidor, que contiene servicios que controlan el juego en su globalidad, tales como la detección de colisiones. Las actualizaciones se multidifunden a todas las réplicas.

(i) Las figuras pueden arrojarse proyectiles entre ellas, y cada impacto debilita, por un tiempo limitado, al infortunado receptor. ¿Qué tipo de ordenación de actualizaciones se requiere aquí? Pista: considérense los eventos *arroja*, *colisiona* y *revive*.

(ii) El juego incorpora dispositivos mágicos que pueden ser escogidos por un jugador para ayudarle. ¿Qué tipo de ordenación debería aplicarse a la operación de toma-el-dispositivo?

14.4. Un encaminador, que separa al proceso p de otros dos q y r, falla inmediatamente después de que p haya iniciado la multidifusión del mensaje m. Si el sistema de comunicación en grupo utiliza vistas síncronas, explíquese que le ocurre a continuación a p.

14.5. Se te entrega un sistema de comunicación en grupo con una operación de multidifusión totalmente ordenada, y un detector de fallos. ¿Es posible obtener comunicación en grupo con vistas síncronas a partir de estos componentes solamente?

14.6. Una operación de multidifusión *ordenada-sinc* es aquella cuya semántica de ordenación en la entrega es la misma que la de la entrega de vistas en un sistema de comunicación en grupo con vistas síncronas. En un servicio de *chismes*, las operaciones sobre los *chismes* están ordenadas causalmente. El servicio incluye listas de usuarios que son capaces de realizar operaciones sobre cada *chisme* en particular. Explíquese por qué la eliminación de un usuario de una lista debería ser una operación ordenada-sinc.

14.7. ¿A qué consideración sobre consistencia da lugar la transferencia de estados?

14.8. Una operación X sobre un objeto o provoca que o invoque una operación sobre otro objeto o'. Ahora se propone replicar a o pero no a o'. Explíquese la dificultad que surge asociada a las invocaciones sobre o', y proponga una solución.

14.9. Explique la diferencia entre linealizabilidad y consistencia secuencial, y por qué, en general, es más fácil de implementar la última.

14.10. Explique por qué el permitir que los respaldos procesen operaciones de lectura en un sistema de replicación pasiva lleva a ejecuciones consistentes secuencialmente en lugar de linealizables.

14.11. ¿Podría utilizarse la arquitectura cotilla para un juego de computador distribuido como el que se describió en el Ejercicio 14.3?

14.12. En la arquitectura cotilla, ¿por qué un gestor de réplicas necesita mantener tanto una marca temporal de la *réplica como una marca temporal del «valor»*?

14.13. En un sistema cotilla, un frontal tiene un vector de marcas temporales (3, 5, 7), que representa los datos que ha recibido por parte de los miembros de un grupo de tres gestores de réplicas. Los tres gestores de réplicas tienen como vectores de marcas temporales (5, 2, 8), (4, 5, 6) y (4, 5, 8), respectivamente. ¿Qué gestor (o gestores) de réplicas podrían satisfacer, inmediatamente, una pregunta por parte de un frontal y cuál sería la marca temporal resultante en el frontal? ¿Cuál podría incorporar una actualización, inmediatamente, por parte del frontal?

14.14. Explíquese por qué haciendo que algunos gestores de réplicas sean de *sólo lectura* se mejora el rendimiento de la arquitectura cotilla.

14.15. Escriba un pseudocódigo para los procedimientos de chequeo de consistencia y fusión (tal y como los usa Bayou) apropiados para una simple aplicación de reserva de habitaciones.

14.16. En Coda, ¿por qué es necesario, algunas veces, que los usuarios intervengan manualmente en el proceso de actualización de copias de un archivo en múltiples servidores?

14.17. Diseñe un esquema para integrar dos réplicas de un directorio de un sistema de archivos que fueron sometidas a actualizaciones separadas durante la operación sin conexión. Utilice o bien una aproximación de transformación operacional como en Bayou, o proponga una solución para el sistema Coda.

14.18. Se aplica replicación por copias disponibles a los ítems A y B con réplicas A_x, A_y y B_m, B_n. Las transacci-ones T y U se definen como:

$$T\text{: } Lee(A);\ Escribe(B, 44). \quad U\text{: } Lee(B);\ Escribe(A, 55),$$

Muestre un entrelazado de T y U, asumiendo que se aplican bloqueos en dos fases a las réplicas. Explique por qué los bloqueos por sí solos no pueden asegurar la secuenciabilidad de *una copia* si una de las réplicas fallas durante la progresión de T y U. Explique, con referencia a este ejemplo, cómo la validación local asegura la secuenciabilidad sobre una copia.

14.19. En los servidores X, Y y Z, los cuales mantienen réplicas de los datos A y B, se usa la replicación con consenso con quórum de Gifford. Los valores iniciales de todas las réplicas de A y B son 100, y los votos para A y B son 1 en cada servidor X, Y y Z. También se tiene $L = E = 2$, tanto para A como para B. Un cliente lee el valor de A y entonces lo escribe en B.

(i) En el momento en el que el cliente realiza estas operaciones, una partición separa los servidores X e Y de Z. Descríbase los quórum obtenidos y las operaciones que tienen lugar si el cliente puede acceder a los servidores X e Y.

(ii) Descríbase los quórum obtenidos y las operaciones que tienen lugar si el cliente puede acceder sólo al servidor Z.

(iii) La partición se repara y a continuación se produce otra partición, de tal modo que X y Z son separadas de Y. Descríbanse los quórum obtenidos y las operaciones que tienen lugar si el cliente puede acceder a los servidores X y Z.

15

SISTEMAS MULTIMEDIA DISTRIBUIDOS

15.1. Introducción
15.2. Características de los datos multimedia
15.3. Gestión de la calidad de servicio
15.4. Gestión de recursos
15.5. Adaptación de caudales
15.6. Caso de estudio: el servidor de vídeo Tiger
15.7. Resumen

Las aplicaciones multimedia generan y consumen caudales de datos continuos en tiempo real. Éstos contienen grandes cantidades de audio, vídeo y otros elementos de datos dependientes del tiempo, y resulta esencial el procesamiento y la entrega a tiempo de los elementos individuales de datos (muestras de audio, marcos de vídeo). Los elementos entregados tarde no tienen interés y son desechados normalmente.

Una especificación de un caudal multimedia se expresa en términos de valores aceptables para la tasa a la que los datos pasan desde la fuente al destino (el ancho de banda), el retardo en la entrega de cada elemento (latencia) y la tasa a la que se pierden o se desechan elementos. La latencia es particularmente importante en las aplicaciones interactivas. En las aplicaciones multimedia a menudo resulta aceptable un grado pequeño de pérdida de datos de los caudales multimedia ya que las aplicaciones pueden volver a sincronizarse con los elementos que siguen a aquellos perdidos.

La reserva y la planificación (*scheduling*) de los recursos pensadas para satisfacer las necesidades tanto de las aplicaciones multimedia como las otras se denomina gestión de la calidad de servicio. Parámetros importantes son la reserva de la capacidad de procesamiento, el ancho de banda de la red y la memoria (para el almacenamiento de los elementos de datos que son entregados temprano).

Todos ellos son reservados en respuesta a solicitudes de calidad de servicio hechas por las aplicaciones. Una solicitud de calidad de servicio resuelta con éxito garantiza una calidad de servicio a la aplicación y produce la reserva y la consiguiente planificación de los recursos solicitados.

Este capítulo está basado en gran parte en un trabajo escrito por Ralf Herrtwich [1995], y queremos agradecerle el permiso para utilizar su material.

15.1. INTRODUCCIÓN

Los computadores modernos pueden manejar caudales de datos continuos (caudales), dependientes del tiempo como audio y vídeo digital. Esta capacidad ha conducido al desarrollo de aplicaciones multimedia distribuidas como bibliotecas de vídeo en red, telefonía sobre Internet y videoconferencia. Dichas aplicaciones son viables con las actuales redes y sistemas de propósito general, a pesar de que a menudo la calidad del audio y del vídeo resultante esté lejos de ser satisfactoria. Las aplicaciones más exigentes como la videoconferencia a gran escala, la televisión digital, y los sistemas de vigilancia están más allá de las capacidades de las redes y de los sistemas distribuidos actuales.

Las aplicaciones multimedia demandan la entrega a tiempo a los usuarios de caudales de datos multimedia. Los caudales de audio y vídeo se generan y se consumen en tiempo real, la entrega a tiempo de los elementos individuales (muestras de audio y marcos de vídeo) es esencial para la integridad de la aplicación. En resumen, los sistemas multimedia son sistemas de tiempo real: deben ejecutar tareas y entregar sus resultados de acuerdo con una planificación que es determinada externamente. El grado en el que esto se consigue por el sistema subyacente es conocido como la *calidad de servicio* (*quality of service*, QoS) de que disfruta una aplicación.

Aunque los problemas de diseño de los sistemas de tiempo real han sido estudiados antes de la llegada de los sistemas multimedia, y muchos sistemas de tiempo real han sido desarrollado con éxito (véase, por ejemplo, Kopetz y Verissimo [1993]), generalmente no han sido integrados en sistemas operativos y redes de propósito más general. La naturaleza de las tareas ejecutadas por los sistemas de tiempo real existentes, como los aeronáuticos, el control del tráfico aéreo, el control de procesos de fabricación y la conmutación telefónica, difiere de aquellas ejecutadas en las aplicaciones multimedia. Las primeras tratan generalmente con cantidades de datos pequeñas y en algunos casos tienen *tiempos límite estrictos*, pero el fallo en el cumplimento de cualquiera de sus tiempos límite de entrega pueden acarrear consecuencias serias, e incluso desastrosas. En tales casos, la solución adoptada ha sido sobredimensionar los recursos de cómputo y reservarlos con una planificación fija que garantice que se cumplen siempre lo requisitos del peor caso.

La reserva y la planificación de los recursos diseñadas para responder a las necesidades de las aplicaciones multimedia y otras similares se denomina *gestión de la calidad de servicio* (*QoS*). La mayoría de los sistemas operativos y de las redes no incluyen las capacidades de gestión de la QoS requeridas por las aplicaciones multimedia.

Las consecuencias de un fallo en el cumplimiento de los tiempos límite en las aplicaciones multimedia pueden ser serias, especialmente en entornos comerciales como servicios de vídeo bajo demanda, aplicaciones de conferencias de negocios y medicina remota, pero los requisitos difieren significativamente de los de las otras aplicaciones de tiempo real:

- Las aplicaciones multimedia son, a menudo, altamente distribuidas y operan sobre entornos de computación distribuida de propósito general. Compiten, por tanto, con otras aplicaciones distribuidas por el ancho de banda de la red y por los recursos de computación de las estaciones de trabajo de los usuarios y de los servidores.
- Los requisitos de recursos de las aplicaciones multimedia son dinámicos. Una videoconferencia puede necesitar más o menos ancho de banda dependiendo del aumento o de la disminución del número de conferenciantes. El uso de los recursos de cómputo en cada estación de trabajo de usuario también varía, ya que cambia, por ejemplo, el número de caudales de vídeo que debe mostrar. Las aplicaciones multimedia pueden suponer otras cargas variables o intermitentes. Por ejemplo, la celebración de una clase multimedia puede incluir una actividad de simulación con un uso intensivo del procesador.
- A menudo los usuarios desean equilibrar los costes en recursos de las aplicaciones multimedia con otras actividades. Pueden querer reducir sus peticiones de ancho de banda para el

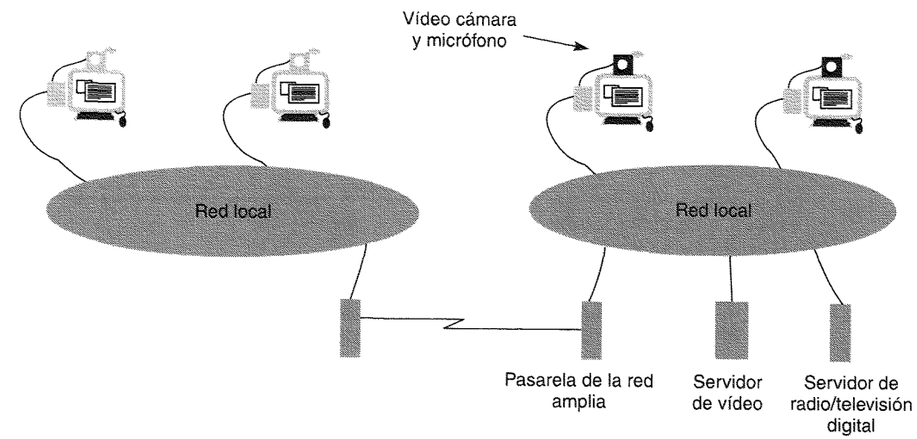


Figura 15.1. Un sistema distribuido multimedia.

vídeo en una aplicación de conferencia para permitir que se realice una conversación de audio separada, o pueden querer seguir programando o editando textos mientras están participando en la conferencia.

Los sistemas de gestión de la QoS están pensados para responder a todas estas necesidades, gestionando de forma dinámica los recursos disponibles y variando las reservas en respuesta a los cambios en la demanda y las prioridades de los usuarios. Un sistema de gestión de la QoS debe gestionar todos los recursos de cómputo y de comunicación necesarios para adquirir, procesar y transmitir caudales de datos multimedia, especialmente donde los recursos son compartidos entre diferentes aplicaciones.

La Figura 15.1 ilustra un sistema distribuido multimedia típico capaz de soportar una variedad de aplicaciones, como conferencias, acceso a secuencias almacenadas de vídeo y difusión de radio y televisión digitales. Los recursos requeridos para la gestión de esta QoS incluyen ancho de banda de la red, ciclos de procesador y capacidad de memoria. También hay que considerar el ancho de banda de disco en el servidor de vídeo. Adoptaremos el término genérico de *ancho de banda de los recursos* para referirnos a la capacidad de cualquier recurso hardware (red, procesador central, subsistema de disco) para transmitir o procesar datos multimedia.

En un sistema distribuido abierto, las aplicaciones multimedia pueden ser iniciadas y utilizadas sin anuncio previo. Pueden coexistir varias aplicaciones en la misma red e incluso en la misma estación de trabajo. Por lo tanto, la necesidad de la gestión de la QoS surge independientemente de la *cantidad total* de ancho de banda de los recursos o de la capacidad de la memoria de un sistema. Se necesita gestionar la QoS para *garantizar* que las aplicaciones serán capaces de obtener la cantidad de recursos necesaria en los momentos requeridos, incluso cuando otras aplicaciones estén compitiendo por esos recursos.

Se han desplegado algunas aplicaciones multimedia incluso en los computadores y redes actuales sin QoS y basadas en el principio del mejor esfuerzo. Entre ellas están:

Multimedia basado en web: estas aplicaciones proporcionan acceso según el mejor esfuerzo a caudales de audio y vídeo publicados en el Web. Han tenido éxito cuando existe poca o ninguna sincronización de los caudales de datos entre diferentes localizaciones. Sus prestaciones están restringidas por el limitado ancho de banda y por las latencias variables que se dan en las redes actuales y por la imposibilidad de los sistemas operativos actuales para soportar una pla-

nificación de tiempo real de los recursos. En el caso del audio y de las secuencias de audio y de vídeo de baja calidad, la utilización extensiva de almacenamiento en el destino para suavizar las variaciones en el ancho de banda y en la latencia hace que se puedan reproducir secuencias de vídeo de forma continua y sin sobresaltos, aunque existan un retardo desde el origen al destino de hasta varios segundos.

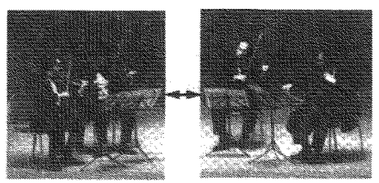
Telefonía de red y conferencias de audio: esta aplicación tiene unos requisitos de ancho de banda relativamente bajos, especialmente cuando se aplican técnicas de compresión eficientes. Aunque la naturaleza interactiva de la misma implica tiempos de ida y vuelta pequeños, algo que no siempre se puede conseguir.

Servicios de vídeo bajo demanda: éstos proporcionan vídeo en formato digital desde grandes sistemas de almacenamiento hasta la herramienta de visualización del usuario. Resultan satisfactorios cuando existe suficiente ancho de banda dedicado, y tanto el servidor como el cliente son computadores dedicados. También emplean una cantidad considerable de almacenamiento en el destino.

Las aplicaciones altamente interactivas plantean problemas mucho más graves. Muchas aplicaciones multimedia son cooperativas (involucran varios usuarios) y sincronizadas (requieren que las actividades de los usuarios estén coordenadas). Éstas abarcan un ancho espectro de contextos y escenarios de aplicación. Por ejemplo:

- Una simple videoconferencia involucra dos o más usuarios, cada uno utilizando una estación de trabajo equipada con una cámara de vídeo digital, un micrófono y posibilidades de salida de audio y vídeo. Existe software disponible para proporcionar sistemas de teleconferencia simples (por ejemplo: *CUSeeMe* [Dorcey 1995, www.cuseeme.com], *NetMeeting* [www.microsoft.com III]), pero sus prestaciones están severamente limitadas por los entornos de computación y de comunicaciones actuales.

- Posibilidades de ensayo y de ejecución que permite a músicos en diferentes ubicaciones tocar juntos [Konstantas y otros 1997]. Ésta es una aplicación multimedia particularmente demandante ya que las restricciones de sincronización son muy exigentes.



Las aplicaciones como éstas requieren:

Conmutación con baja latencia: retardos de ida y vuelta < 100 milisegundos, de modo que la interacción entre los usuarios parezca que está sincronizada.

Estado de sincronización distribuida: si un usuario detiene un vídeo en un determinado marco, los otros usuarios deberían ver el vídeo parado en el mismo marco.

Sincronización de medios: todos los participantes en una actuación musical deberían escuchar la ejecución aproximadamente a la vez (Konstantas y otros [1997] identifica como requisito de sincronización un intervalo de 50 milisegundos). La banda sonora y el caudal de vídeo deberían mantener la *sincronización de labios*, por ejemplo, para un usuario haciendo comentarios en una reproducción de vídeo o en una sesión distribuida de *karaoke*.

Sincronización externa: en conferencias y en otras aplicaciones cooperativas, pueden existir datos activos en distintos formatos, tales como animaciones generadas por computador, datos CAD, pizarras electrónicas y documentos compartidos. Las actualizaciones de éstos deben ser distribuidas y suceder de forma que parezcan casi sincronizadas con los caudales multimedia dependientes del tiempo.

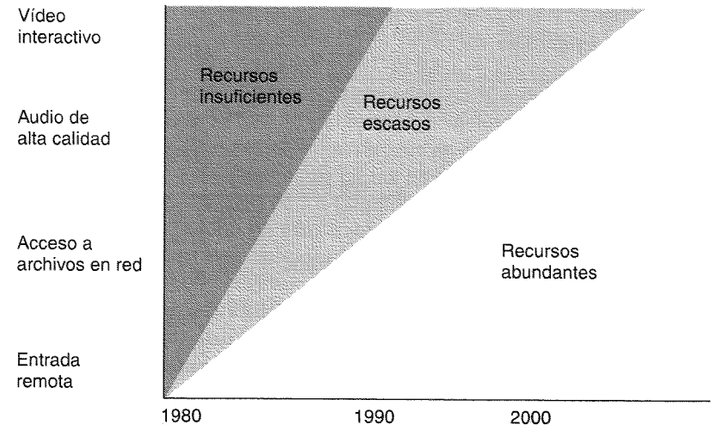


Figura 15.2. La ventana de la escasez para los recursos de cómputo y de comunicación.

Tales aplicaciones se ejecutarán satisfactoriamente sólo en sistemas que incluyan esquemas de gestión de la QoS rigurosos.

◇ **La ventana de la escasez.** Muchos de los computadores modernos proporcionan alguna capacidad para manejar datos multimedia, pero los recursos necesarios son muy limitados. Muchos sistemas limitan la cantidad y la calidad de los caudales que soportan, especialmente cuando se trata de trabajar con grandes caudales de audio y vídeo. Esta situación ha sido descrita como *la ventana de la escasez* [Anderson y otros 1990b]. Siempre que una cierta clase de aplicaciones cae dentro de esta ventana, un sistema necesita reservar y planificar sus recursos cuidadosamente para proporcionar el servicio deseado (véase la Figura 15.2). Antes de alcanzar la ventana de la escasez, un sistema tiene recursos insuficientes para ejecutar sus aplicaciones relevantes. Esta era la situación de las aplicaciones multimedia en la primera mitad de los años ochenta. Una vez que una clase de aplicaciones ha dejado la ventana de la escasez, las prestaciones del sistema son suficientes para proporcionar el servicio incluso bajo circunstancias adversas y sin mecanismos de adecuación.

Los avances en las prestaciones de los sistemas se han utilizado también para mejorar la calidad de los datos multimedia hasta incluir tasas de marcos mayores y mejor resolución en los caudales de vídeo o para soportar más medios de forma concurrente, por ejemplo en un sistema de videoconferencia. Pero las aplicaciones más demandantes, incluidas la realidad virtual y la manipulación de caudales en tiempo real (*efectos especiales*) pueden extender la ventana de la escasez casi indefinidamente.

En la Sección 15.2 revisamos las características de los datos multimedia. La Sección 15.3 describe diferentes aproximaciones a la reserva de recursos escasos de modo que se consiga una QoS, y la Sección 15.4 discute métodos para planificar esos recursos escasos. La Sección 15.5 discute métodos para optimizar el caudal de datos en los sistemas multimedia. La Sección 15.6 describe el servidor de vídeo Tiger, un sistema escalable de bajo coste para la entrega de secuencias almacenadas de vídeo a un gran número de usuarios concurrentemente.

15.2. CARACTERÍSTICAS DE LOS DATOS MULTIMEDIA

Nos hemos referido a los datos de audio y vídeo como continuos y dependientes del tiempo. ¿Cómo podemos definir sus características de forma más precisa? El término *continuo* se refiere a

	Tasa de datos (aproximados)	*Muestra o marco* *Tamaño*	*Frecuencia*
Conversación telefónica	64 kbps	8 bits	8.000/segundo
Sonido calidad CD	1,4 Mbps	16 bits	44.000/segundo
Vídeo TV estándar (sin comprimir)	120 Mbps	hasta 640 × 480 píxeles × 16 bits	24/segundo
Vídeo TV estándar (comprimido MPEG-1)	1,5 Mbps	variable	24/segundo
Vídeo HDTV (sin comprimir)	1.000-3.000 Mbps	hasta 1.920 × 1.080 píxeles × 24 bits	24-60/segundo
Vídeo HDTV (comprimido MPEG-2)	10-30 Mbps	variable	24-60/segundo

Figura 15.3. Características de caudales multimedia típicos.

la visión de los datos desde el punto de vista del usuario. Internamente, los medios continuos son representados como secuencias de valores discretos que se reemplazan unos a otros en el tiempo. Por ejemplo, el valor de una matriz de imagen es reemplazado 25 veces por segundo para dar la impresión de una escena en movimiento con una calidad de televisión; un valor de la amplitud de sonido se varía 8.000 veces por segundo para conseguir una calidad telefónica.

Los caudales multimedia se dice que son *dependientes del tiempo* (o *isocronos*) porque los elementos temporales de datos en los caudales de audio y vídeo definen la semántica o *contenido* del caudal. Los tiempos a los que los valores son reproducidos o grabados afectan a la validez de los datos. Por lo tanto los sistemas que soportan aplicaciones multimedia necesitan preservar la sincronización cuando manejan datos continuos.

Los caudales multimedia son a menudo voluminosos. Por lo tanto los sistemas que soportan aplicaciones multimedia necesitan mover datos con un rendimiento mayor que los sistemas convencionales. La Figura 15.3 muestra algunas tasas de datos y valores de frecuencias de muestreo/marcos típicos. Hay que hacer notar que en algún caso los requisitos de ancho de banda de recursos son muy grandes. Esto es especialmente verdad en el caso de vídeo de una calidad razonable. Por ejemplo, un caudal de vídeo de televisión estándar requiere más de 120 Mbps lo que excede la capacidad de una red Ethernet de 100 Mbps. También se ponen a prueba las capacidades de la CPU; un programa que copie o aplique una transformación simple a cada marco de un caudal de vídeo de calidad televisión estándar requiere al menos el 10 % de la capacidad de una CPU de un PC de 400 MHz. Las cifras para los caudales de televisión de alta definición son incluso mayores, y en muchas aplicaciones, como videoconferencia, se tienen que gestionar múltiples caudales de audio y vídeo concurrentemente. La utilización de representaciones comprimidas es, por lo tanto, esencial, aunque transformaciones como la mezcla de vídeos sean difíciles de llevar a cabo con caudales comprimidos.

La compresión puede reducir los requisitos de ancho de banda en un factor entre 10 y 100, pero los requisitos de tiempo de los datos continuos no se ven alterados. Hay una intensa actividad de investigación y estandarización dirigida a producir representaciones y métodos de compresión de propósito general eficientes para caudales de datos multimedia. Este trabajo ha producido varios formatos de datos comprimidos, como GIF, TIFF y JPEG para imágenes estáticas, y MPEG-1, MPEG-2 y MPEG-4 para secuencias de vídeo. No vamos a entrar en detalles aquí; existen otras fuentes como Buford [1994] y Gibbs y Tsichritzis [1994], que proporcionan revisiones de los tipos,

representaciones y estándares de medios, y la página web de Gibbs y Szentivanyi [multimedia index] es una fuente útil de referencias a documentación sobre los estándares multimedia actuales.

Aunque el uso de audio y vídeo comprimido disminuye las necesidades de ancho de banda de las redes de comunicación, impone cargas adicionales en los recursos de procesamiento tanto en origen como en el destino. Esto se ha proporcionado, a menudo, con el uso de hardware de propósito especial para procesar y entregar información audio y vídeo, los codificadores/decodificadores (*codecs*) de vídeo y audio que se encuentran en las tarjetas de vídeo en los computadores personales. Pero el incremento en la potencia de los computadores personales y las arquitecturas multiprocesador hacen probable que mucho de este trabajo se pueda realizar utilizando filtros software de codificación y decodificación. Esta aproximación aporta más flexibilidad, con una mejor adaptación a formatos específicos de aplicación, lógicas de aplicación de propósito especial y el manejo simultáneo de varios caudales multimedia.

El método de compresión utilizado por el formato de vídeo MPEG es asimétrico, con un algoritmo de compresión complejo y otro sencillo de descompresión. Esto tiende a facilitar su uso en conferencias, donde la compresión es realizada a menudo por un *codec* (codificador descodificador) hardware pero la descompresión de los diferentes caudales que llegan a los computadores de los usuarios se realiza por software, haciendo posible que el número de participantes en la conferencia pueda variar sin depender del número de codecs presentes en cada computador de usuario.

15.3. GESTIÓN DE LA CALIDAD DE SERVICIO

Cuando las aplicaciones multimedia corren sobre redes de computadores personales, compiten por los recursos en las estaciones de trabajo que ejecutan esas aplicaciones (ciclos de procesador, ciclos de bus, capacidad de búferes) y en las redes (enlaces de transmisión, conmutadores, pasarelas). Las estaciones de trabajo y las redes pueden tener que soportar varias aplicaciones multimedia y convencionales a la vez. Existe competencia entre las aplicaciones multimedia y las convencionales, entre diferentes aplicaciones multimedia e incluso entre los caudales multimedia de las aplicaciones individuales.

El uso concurrente de los recursos físicos por una variedad de tareas hace tiempo que es posible con los sistemas operativos multitarea y las redes compartidas. En los sistemas operativos multitarea, el procesador centrar es reservado para las tareas individuales (o procesos) según un esquema de planificación de ronda (*round robin*) u otro que reparta los recursos de procesamiento según el principio del *mejor esfuerzo* entre todas las tareas que compiten por el procesador central.

Las redes están diseñadas para hacer posible que sean intercalados mensajes de diferentes fuentes, de modo que existan muchos canales de comunicación virtuales sobre los mismos canales físicos. La tecnología predominante en las redes de área local, Ethernet, gestiona el medio de transmisión compartido según el *mejor esfuerzo*. Cualquier nodo puede utilizar el medio de transmisión cuando está en silencio. Pero se pueden producir colisiones de paquetes, y cuando esto es así, los nodos emisores esperan una cantidad de tiempo aleatorio para evitar la repetición de la colisión. Las colisiones casi son inevitables cuando la red está muy cargada, y este esquema no proporciona garantías respecto al ancho de banda o la latencia en esas condiciones.

La característica clave de estos esquemas de reserva de recursos es que gestionan los incrementos de la demanda asignando los recursos disponibles durante menos tiempo a las tareas en competencia. El método de ronda y otros basados en el mejor esfuerzo para compartir los ciclos del procesador y el ancho de banda de la red no pueden satisfacer los requisitos de las aplicaciones multimedia. Como hemos visto, el procesamiento y la transmisión de los caudales multimedia a tiempo es crucial. La entrega con retraso no tiene valor. Para conseguir la entrega a tiempo, las

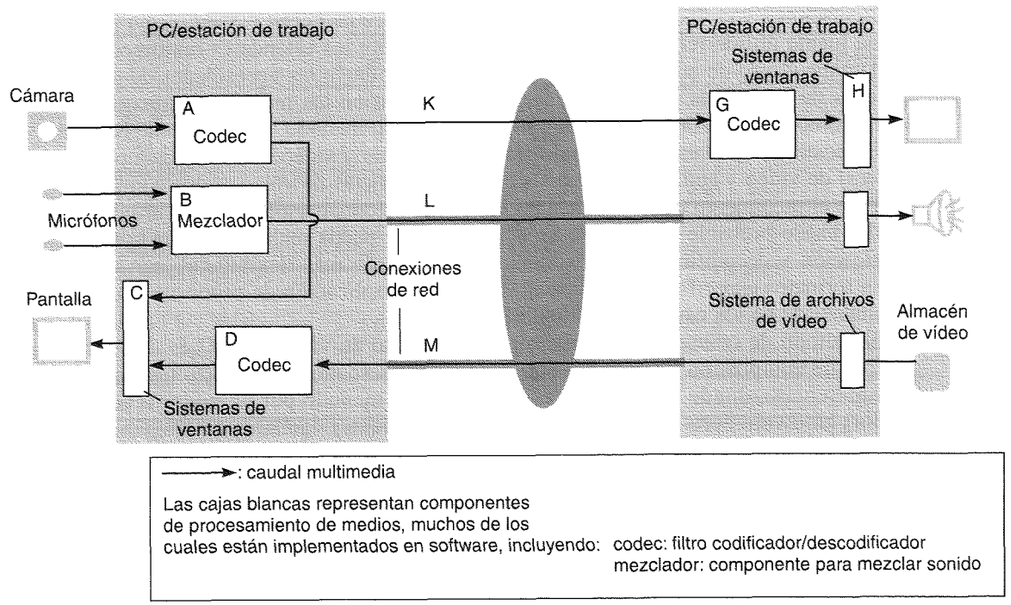


Figura 15.4. Componentes de la infraestructura típica para aplicaciones multimedia.

aplicaciones necesitan garantizar que los recursos necesarios serán reservados y planificados en los instantes oportunos.

La gestión y la reserva de los recursos que proporciona tales garantías se denomina *gestión de la calidad de servicio*. La Figura 15.4 muestra los componentes de la infraestructura de una aplicación de conferencia simple ejecutándose sobre dos computadores personales, utilizando compresión de datos y conversión de formatos. Las cajas blancas representan componentes software cuyos requisitos de recursos pueden afectar a la calidad de servicio de la aplicación.

La figura muestra la arquitectura abstracta más utilizada en el software multimedia, en la que caudales de elementos de datos (marcos de vídeo, muestras de sonido) que fluyen continuamente es procesado por una colección de procesos y transferido entre los procesos por conexiones entre procesos. Los procesos producen, transforman y consumen caudales de datos multimedia continuos. Las conexiones enlazan los procesos en una secuencia entre una *fuente* de elementos de datos a un *destino*, en el cual son reproducidos o consumidos. Las conexiones entre los procesos se pueden implementar mediante conexiones de red o mediante transferencias en memoria cuando los procesos residen en la misma máquina. Para que los elementos de datos multimedia lleguen a tiempo a su destino, cada proceso debe reservar el tiempo de CPU, la capacidad de memoria y el ancho de banda adecuados para ejecutar sus tareas asignadas y debe ser planificado para utilizar esos recursos de forma lo suficientemente frecuente como para poder entregar a tiempo los datos en su caudal al siguiente proceso.

En la Figura 15.5 mostramos los requisitos de recursos de los principales componentes software y de las conexiones de red de la Figura 15.4 (observe que las letras relacionan los componentes en estas dos figuras). Obviamente, los recursos requeridos sólo pueden ser garantizados si existe un componente responsable de la reserva y de la planificación de esos recursos. Nos referiremos a ese componente como el *gestor de la calidad de servicio*.

	Componente		*Ancho de banda*	*Latencia*	*Tasa de pérdidas*	*Recursos requeridos*
	Cámara	Salida:	10 marcos/segundo, vídeo calidad baja 640 × 480 × 16 bits	—	Cero	—
A	Codec	Entrada:	10 marcos/segundo, vídeo calidad baja	Interactiva	Baja	10 ms CPU cada 100 ms;
		Salida:	caudal MPEG-1			10 Mbytes RAM
B	Mezclador	Entrada:	audio 2 × 44 kbps	Interactiva	Muy baja	1 ms CPU cada 100 ms;
		Salida:	audio 1 × 44 kbps			1 Mbytes RAM
H	Sistema	Entrada:	varias	Interactiva	Baja	5 ms CPU cada 100 ms;
	de ventanas	Salida:	búfer de 50 marcos/segundo			5 Mbytes RAM
K	Conexión de red	Entrada/Salida:	caudal MPEG-1, aprox. 1,5 Mbps	Interactiva	Baja	1,5 Mbps, protocolo de caudales con pocas pérdidas
L	Conexión de red	Entrada/Salida:	audio 44 kbps	Interactivo	Muy baja	44 kbps, protocolo de caudales con pocas pérdidas

Figura 15.5. Especificación de la QoS para los componentes de la aplicación mostrada en la Figura 15.4.

La Figura 15.6 muestra las responsabilidades del gestor de la QoS en forma de diagrama de caudal. En las siguientes dos subsecciones describimos las dos principales subtareas del gestor de la QoS:

Negociación de la calidad de servicio: la aplicación indica sus requisitos de recursos al gestor de la QoS. El gestor de la QoS evalúa la posibilidad de satisfacer los requisitos a partir de la base de datos de recursos disponibles y de los recursos comprometidos actualmente, dando una respuesta positiva o negativa. Si es negativa, la aplicación puede ser reconfigurada para utilizar menos recursos y el proceso se repite.

Control de admisión: si el resultado de la evaluación de los recursos ha sido positivo, se reservan los recursos requeridos y se da a la aplicación un *contrato de recursos*, que establece los recursos que han sido reservados. El contrato incluye un límite de tiempo. Entonces la aplicación está autorizada para ejecutarse. Si cambia sus requisitos de recursos debe notificarlo al gestor de la QoS. Si los requisitos disminuyen, los recursos liberados pasan a la base de datos de recursos disponibles. Si se incrementan, se establece una nueva ronda de negociación y de control de admisión.

En lo que resta de sección describimos con detalle las técnicas para llevar a cabo estas subtareas. Por supuesto, mientras una aplicación se está ejecutando, existe la necesidad de ajuste fino en la planificación de los recursos como el tiempo de procesador y el ancho de banda de red para asegurarse que los procesos de tiempo real reciben a tiempo los recursos reservados. Las técnicas para conseguir esto se describen en la Sección 15.4.

15.3.1. NEGOCIACIÓN DE LA CALIDAD DE SERVICIO

Para negociar la QoS entre una aplicación y el sistema que la soporta, una aplicación debe especificar sus necesidades de QoS al gestor de la QoS. Esto se realiza mediante la transmisión de un conjunto de parámetros. Tres parámetros son de principal interés cuando se trata de procesar y transportar caudales de datos multimedia: *ancho de banda*, *latencia* y *tasa de pérdidas*.

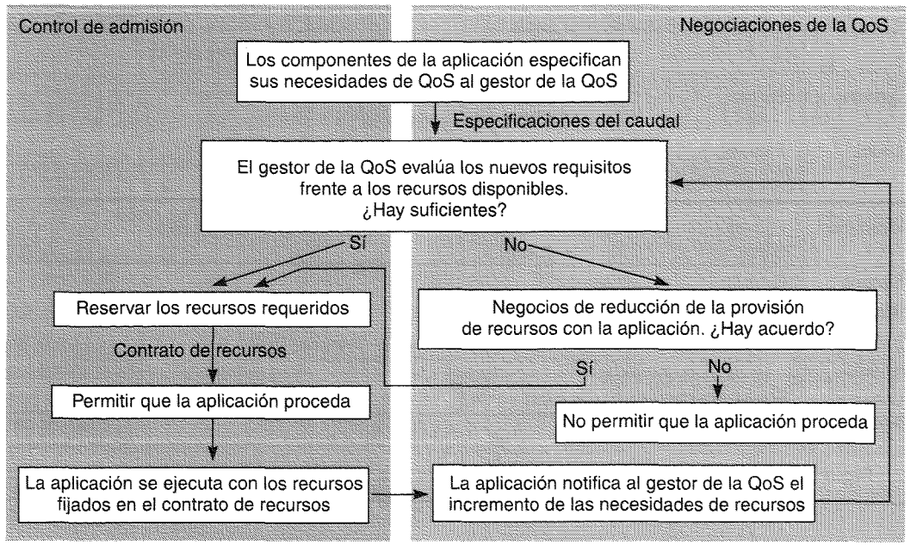


Figura 15.6. Tareas del gestor de la QoS.

Ancho de banda: El ancho de banda de un caudal o componente multimedia es la tasa a la que los datos fluyen a través de él.

Latencia: La latencia es el tiempo requerido por un elemento de datos individual para moverse a través de un caudal desde un origen a un destino. Ésta puede variar dependiendo del volumen de otros datos en el sistema y de otras características de la carga del sistema. Esta variación se denomina fluctuación (*jitter*); formalmente, la fluctuación es la primera derivada de la latencia.

Tasa de pérdidas: Como la entrega tardía de los datos multimedia no tiene sentido, los elementos de datos serán desechados cuando sea imposible entregarlos antes de su tiempo de entrega planeado. En un entorno de QoS perfectamente gestionado, esto nunca debería suceder, pero hasta ahora, existen pocos entornos de ese tipo, por las razones expuestas anteriormente. Por lo tanto, los costos en recursos para garantizar la entrega a tiempo de todos los elementos multimedia es a menudo inaceptable; esto implica probablemente la reserva de unos recursos que excederían las necesidades medias para poder tratar adecuadamente los picos ocasionales. La alternativa adoptada es aceptar cierta tasa de pérdida de datos (marcos de vídeo o muestras de audio desechadas). La proporción aceptable se mantiene normalmente baja; algo más de un 1 % y mucho menor para aplicaciones de calidad crítica.

Se pueden utilizar estos tres parámetros:

1. Para describir las características de un caudal multimedia en un entorno particular. Por ejemplo, un caudal de vídeo puede requerir un ancho de banda medio de 1,5 Mbps, y como está siendo utilizado en una aplicación de conferencia necesita ser transferido con un retardo de 150 ms como mucho para evitar los huecos en la conversación. Los algoritmos de descompresión utilizados en el destino deben producir imágenes aceptables con una tasa de pérdidas de un marco cada 100.
2. Para describir las capacidades de los recursos para transportar un caudal. Por ejemplo, una red puede proporcionar conexiones de 64 kbps de ancho de banda, sus algoritmos de gestión de las colas garantizar retardos menores de 10 ms, y el sistema de transmisión puede garantizar un tasa de pérdidas menores de 1 en 10^6.

Estos parámetros son interdependientes. Por ejemplo:

- La tasa de pérdidas en los sistemas modernos casi nunca depende de los errores en los bits causados por ruido o mal funcionamiento; depende del desbordamiento del búfer y de los datos dependientes del tiempo que llegan tarde. Por lo tanto, cuanto mayor sea el ancho de banda y menor el retardo, más fácil será tener una tasa de errores baja.
- Cuando menor sea el ancho de banda comparado con su carga, más mensajes se acumularán en la entrada y se necesitarán búferes mayores para evitar las pérdidas. Cuanto mayores sean los búferes, más tendrán que esperar los mensajes para ser tratados, por lo que el retardo será mayor.

◇ **Especificación de los parámetros de la QoS para caudales.** Los valores de los parámetros de la QoS se pueden fijar explícitamente (por ejemplo para el caudal de salida de la cámara de la Figura 15.4 podríamos requerir: *ancho de banda*: 50 Mbps, *retardo*: 150 ms, *pérdidas*: < 1 marco en 10^3) o de forma implícita (por ejemplo el ancho de banda del caudal de entrada a la conexión de red K es el resultado de aplicar la compresión MPEG-1 a la salida de la cámara).

Pero el caso más común es que tengamos que especificar un valor y un rango de variación permitida. Vamos a considerar este requisito para cada uno de los parámetros:

Ancho de banda: La mayoría de las técnicas de compresión de vídeo producen un caudal de marcos de diferentes tamaños dependiendo del contenido original del vídeo. Para MPEG, la relación de compresión media está entre 1:50 y 1:100, pero variará dinámicamente dependiendo del contenido; por ejemplo, el mayor ancho de banda requerido se dará cuando el contenido cambie lo más rápidamente posible. Debido a esto, a menudo es útil fijar los parámetros de la QoS como valores mínimo, medio y máximo, dependiendo del tipo de régimen de gestión de la QoS que sea utilizado.

Otro problema que surge en la especificación del ancho de banda es la caracterización de sus ráfagas (*burstiness*). Considérense tres caudales de 1 Mbps. Un caudal transmite un marco de 1 Mbit cada segundo, el segundo es un caudal asíncrono de elementos de animación generados por computador con un ancho de banda medio de 1 Mbps, y el tercero envía una muestra de sonido de 100-bit cada microsegundo. Aunque los tres caudales requieren el mismo ancho de banda, sus patrones de tráfico son muy diferentes.

Un modo de tratar las irregularidades es definir un parámetro de ráfaga además de la tasa y el tamaño de los marcos. El parámetro de ráfaga especifica el número máximo de elementos de datos que pueden llegar temprano, esto es, antes de su instante de llegada establecido. El modelo de *llegada lineal de procesos* (*linear-bounded arrival processes*, LBAP) utilizado en Anderson [1993] define el número máximo de mensajes en un caudal durante cualquier intervalo de tiempo t como $Rt + B$, donde R es la tasa y B es el tamaño máximo de ráfaga. La ventaja de utilizar este modelo es que refleja adecuadamente las características de las fuentes multimedia: los datos multimedia leídos de disco se entregan habitualmente en grandes bloques, y los datos recibidos de las redes llegan, a menudo, bajo la forma de pequeños paquetes. En este caso, el parámetro de ráfaga define la cantidad de espacio necesario en el búfer para evitar pérdidas.

Latencia: Algunos requisitos de tiempo en multimedia son producto del caudal en sí mismo: si los marcos de un caudal no se procesan a la misma tasa a la que llegan, se van a acumular en la entrada y pueden desbordar la capacidad del búfer. Para evitar esto, un marco no debe permanecer en el búfer por término medio más de $1/R$, donde R es la tasa de marcos de un caudal, o se acumulará el trabajo pendiente. Si se acumula el trabajo, la cantidad y el tamaño de la información pendiente de procesar afectará al retardo máximo del caudal entre extremos, a lo que hay que añadir los tiempos de propagación y de procesamiento.

Otros requisitos de latencia surgen del entorno de la aplicación. En aplicaciones de conferencia, la necesidad de interacciones aparentemente instantáneas entre los participantes hacen necesario que los retardos extremo a extremo no sean mayores de 150 ms, para evitar problemas en la percepción humana de la conversación, mientras que en la reproducción de vídeo almacenado, para asegurar que el sistema responda adecuadamente a órdenes como *Pausa* y *Parada*, la latencia máxima debería ser en torno a 500 ms.

Una tercera consideración con respecto al tiempo de entrega de los mensajes multimedia es la fluctuación (variación en el período entre la entrega de dos marcos adyacentes). Mientras la mayoría de los dispositivos multimedia producen datos a su tasa regular sin variación, las herramientas software (por ejemplo un decodificador software para marcos de vídeo) necesitan ser cuidadosas para descartar la fluctuación. La fluctuación se elimina utilizando almacenamiento previo, pero el alcance del la eliminación de la fluctuación es limitado, por que el retardo total extremo a extremo está restringido por las consideraciones mencionadas anteriormente, de modo que la reproducción de secuencias multimedia necesita también que los elementos de información lleguen antes de los tiempos de entrega límite.

Tasa de pérdida: La tasa de pérdida es el parámetro de la QoS más difícil de especificar. Los valores típicos de la pérdida de datos provienen de cálculos de la probabilidad de que se desborden los búferes y de que se retrasen los mensajes. Estos cálculos están basados o en la suposición del peor caso posible o en distribuciones estándar. Ninguno de los cuales es necesariamente una buena aproximación en los casos prácticos. Sin embargo, las especificaciones de las tasas de pérdidas son necesarias para cualificar los parámetros ancho de banda y latencia: dos aplicaciones pueden tener las mismas características de ancho de banda y de latencia y ser totalmente diferentes ya que uno pierde un elemento de información cada cinco y la otra uno cada millón.

Como ocurre con las especificaciones del ancho de banda, donde no sólo es importante el volumen de datos enviados en un intervalo de tiempo, sino también su distribución sobre ese intervalo de tiempo, la especificación de la tasa de pérdidas tiene que determinar el intervalo de tiempo durante en el que espera una cierta pérdida. En concreto, las tasas de pérdidas dadas para intervalos de tiempo infinitos no son válidas, ya que cualquier pérdida en un corto período de tiempo puede superar significativamente a la tasa de pérdidas a largo plazo.

◇ **Moldeado del tráfico.** El moldeado del tráfico es un término utilizado para describir el uso del almacenamiento de salida para suavizar el caudal de elementos de datos. El parámetro de ancho de banda de un caudal multimedia proporciona generalmente una aproximación idealista del patrón del tráfico real que se dará cuando se transmita el caudal. Según más se acerquen a la realidad los patrones de tráfico, mejor será capaz un sistema de gestionar el tráfico, en particular cuando se utilizan métodos de planificación diseñados para solicitudes periódicas.

El modelo de variaciones del ancho de banda LBAP invita a regular las ráfagas de los caudales multimedia. Se puede regular cualquier caudal insertando un búfer en el origen y definiendo un método mediante el cual los elementos dejan el búfer. Una buena ilustración de este método es la imagen de un depósito agujereado (véase la Figura 15.7(a)): el depósito puede ser llenado arbitrariamente con agua hasta que esté lleno; pero por un agujero en la base del depósito discurre un chorro continuo de agua. El algoritmo del depósito agujereado asegura que el caudal nunca discurrirá con una tasa mayor que R. El tamaño del búfer B define el tamaño máximo de ráfaga que se puede dar en un caudal sin que se pierdan elementos. B limita también el tiempo que puede permanecer un elemento en el búfer.

El algoritmo del depósito agujereado elimina completamente las ráfagas. Dicha eliminación no es necesaria siempre ya que el ancho de banda está limitado en cualquier intervalo de tiempo. El algoritmo del depósito de fichas consigue esto mientras que permite la existencia de ráfagas grandes cuando un caudal ha estado inactivo durante un rato (véase la Figura 15.7(b)). Es una variación del algoritmo del depósito agujereado en el cual se generan fichas para ser enviadas a una tasa R

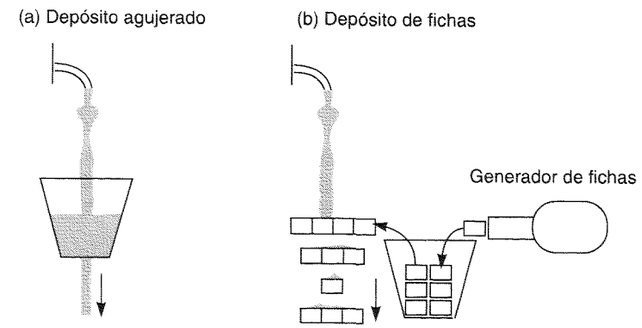


Figura 15.7. Algoritmos de moldeado de tráfico.

fija. Estas fijas se recogen en un depósito de tamaño B. Sólo se puede enviar datos de tamaño S si existen S fichas en el depósito. El proceso emisor elimina entonces esas S fichas. El algoritmo del depósito de fichas asegura que en cualquier intervalo de tiempo t la cantidad de datos enviados no será mayor que $Rt + B$. Es, por lo tanto, una implementación del modelo LBAP.

En un sistema de depósito de fichas sólo se pueden dar picos elevados de tamaño B cuando un caudal ha estado inactivo durante un rato. Para evitar estas ráfagas, se puede colocar un simple depósito agujereado detrás del depósito de fichas. La tasa del caudal F de este depósito debe ser significativamente mayor que R para que este esquema tenga sentido. Su único propósito es romper las ráfagas realmente grandes.

◇ **Especificaciones del caudal.** Una colección de parámetros del caudal es conocida generalmente como una *especificación del caudal*. Existen varios ejemplos de especificaciones de caudal y todas son similares. En RFC 1363 de Internet [Partridge 1992], las especificaciones del caudal se definen como once valores numéricos de 16-bit (Figura 15.8), que reflejan los parámetros de la QoS discutidos arriba del siguiente modo:

	Versión del protocolo
Ancho de banda:	Unidad máxima de transmisión
	Tasa del depósito de fichas
	Tamaño del depósito de fichas
	Tasa máxima de transmisión
Retraso:	Retardo mínimo sensible
	Variación máxima del retardo
Pérdida:	Sensibilidad a la pérdida
	Sensibilidad a las pérdidas en ráfaga
	Intervalo de pérdidas
	Garantía de calidad

Figura 15.8. Especificaciones del flujo del RFC 1363.

- La unidad de transmisión máxima y la tasa de transmisión máxima determinan el ancho de banda máximo requerido por el caudal.
- El tamaño del depósito de fichas y la tasa determinan las ráfagas del sistema.
- Las características del retardo son especificadas por el retardo mínimo al partir del cual puede verse afectada una aplicación (ya que pretendemos evitar la sobreoptimización para retardos cortos) y la máxima fluctuación que puede aceptar.
- Las características de pérdida se definen por el número total de pérdidas aceptable sobre un cierto intervalo y por el número máximo de pérdidas consecutivas.

Existen muchas alternativas para expresar cada grupo de parámetros. En SRP [Anderson y otros 1990a], las características de ráfagas de un caudal se dan mediante un parámetro de trabajo adelantado máximo, que define el número de mensajes de un caudal que pueden adelantarse a su tasa de llegada normal en cualquier instante de tiempo. En Ferrari y Verma [1990], se fija el peor caso del retardo: si el sistema no puede garantizar el transporte de datos dentro de este intervalo de tiempo, el transporte de datos será inútil para la aplicación. En el RFC 1190, la especificación del protocolo ST-II [Topolcic 1990], la pérdida se representa como la probabilidad de que cada paquete sea desechado.

Todos los ejemplos anteriores proveen de un espectro continuo de valores de QoS. Si el conjunto de aplicaciones y de caudales a soportar es limitado, puede ser suficiente con definir un conjunto discreto de clases de QoS: por ejemplo, calidad telefónica, y audio de alta fidelidad, vídeo en directo y reproducido, etc. Los requisitos de todas las clases deben ser conocidos de manera implícita por todos los componentes del sistema; el sistema puede ser configurado incluso para una cierta mezcla de tráfico.

◇ **Procedimientos de negociación.** En las aplicaciones multimedia distribuidas, los componentes de un caudal es probable que estén localizados en diferentes nodos. Existirá un gestor de QoS en cada nodo. Una aproximación directa a la negociación de QoS es seguir el fluir de los datos a lo largo de cada caudal desde el origen al destino. Un componente fuente inició la negociación enviando una especificación del caudal a su gestor de QoS local. El gestor puede comprobar en su base de datos de recursos disponibles si puede satisfacer la QoS solicitada. Si están involucrados otros sistemas en la aplicación, se envían las especificaciones del caudal al siguiente nodo donde se necesitan recursos. La especificación del caudal atraviesa todos los nodos hasta que llega al destino final. Entonces la información sobre si la QoS deseada puede ser satisfecha viaja hasta el origen. Esta simple aproximación a la negociación es satisfactoria para en muchos casos, pero no considera las posibilidades de conflicto entre las negociaciones de la QoS concurrentes que comienzan en nodos distintos. Para resolver este problema se necesitaría un procedimiento de negociación de QoS distribuido transaccional.

Las aplicaciones raramente tienen unos requisitos fijos de QoS. En lugar de devolver un valor booleano dependiendo de si se puede proveer o no una QoS, es más apropiado para el sistema determinar qué tipo de QoS puede dar y dejar a la aplicación que decida si es aceptable. Lo habitual es especificar los valores deseado y peor para cada parámetro de la QoS para evitar una QoS sobreoptimizada o para abortar la negociación cuando es evidente que la calidad deseada no es alcanzable. Una aplicación puede especificar que desea un ancho de banda de 1,5 Mbps pero que también sería capaz de manejar 1 Mbps, o que el retardo debería ser de 200 ms, pero que en el peor caso 300 ms son todavía aceptables. Como sólo se puede optimizar un parámetro a la vez, sistemas como HeiRAT [Vogt y otros 1993], esperan del usuario que defina valores para sólo dos parámetros y deja que el sistema optimice el tercero.

Si el caudal tiene múltiples sumideros, los caminos de negociación se bifurcan de acuerdo con el caudal de datos. Como una extensión directa del esquema anterior, los nodos intermedios pueden agregar mensajes de realimentación acerca de la QoS desde los destinos para conseguir los valores en el peor caso de los parámetros de QoS. Entonces, el ancho de banda disponible se convierte en el menor ancho de banda disponible para todos los destinos, el retardo será el mayor de todos los destinos y la tasa de pérdida la mayor de todos los destinos. Éste es el procedimiento practicado por los protocolos de negociación iniciados por el emisor como SRP, ST-II y RCAP [Banerjea y Mah 1991].

En situaciones con destinos heterogéneos, normalmente no resulta apropiado asignar un valor en el peor caso común a todos los destinos. En su lugar, cada destino debería recibir la mejor QoS posible. Esto sugiere un proceso de negociación iniciado por el receptor, en lugar del orientado al emisor. El protocolo RSVP [Zhang y otros 1993] es un protocolo alternativo de negociación de la QoS en el cual los destinos se conectan a los caudales. Las fuentes comunican la existencia de caudales y sus características inherentes a todos los destinos. Los destinos pueden conectarse entonces al nodo más cercano a través del cual pase el caudal y derivar datos desde ahí. Para conseguir datos con la QoS apropiada, pueden tener que utilizar técnicas como el filtrado (discutido en la Sección 15.5).

15.3.2. CONTROL DE ADMISIÓN

El control de admisión regula el acceso a los recursos para evitar la sobrecarga de los recursos y para proteger los recursos de solicitudes que no se pueden satisfacer. Esto implica rechazar peticiones de servicio en las que los requisitos de recursos de un nuevo caudal multimedia violarían las garantías de QoS actuales.

Un esquema de control de admisión se basa en algún conocimiento sobre la capacidad total del sistema y la carga generada por cada aplicación. La especificación de los requisitos de ancho de banda de una aplicación puede reflejar la cantidad máxima de ancho de banda que una aplicación puede llegar a necesitar, el mínimo ancho de banda necesario para trabajar o un valor promedio entre ambos. En consecuencia, un esquema de control de admisión puede basar sus reservas de recursos en cualquiera de estos valores.

Para aquellos recursos en los que hay un único gestor, el control de admisión es directo. Los recursos que tienen puntos de acceso distribuidos, tales como muchas redes de área local, necesitan o una entidad de control de admisión centralizada o un algoritmo de control de admisión distribuido que evite admisiones concurrentes conflictivas. El arbitraje del bus en las estaciones de trabajo entra dentro de esta categoría; sin embargo, incluso los sistemas multimedia que ejecutan reserva extensiva de ancho de banda no controlan la admisión en el bus, ya que el ancho de banda del bus no se considera en la ventana de escasez.

◇ **Reserva del ancho de banda.** Una forma habitual de asegurar un cierto nivel de QoS para un caudal multimedia es reservar una porción de ancho de banda para uso exclusivo. Si se trata de cumplir los requisitos de un caudal en todo momento, se necesita realizar la reserva para su ancho de banda máximo. Éste es el único modo posible para proporcionar QoS garantizada a una aplicación, siempre que no se den fallos catastróficos en el sistema. Se utiliza en aplicaciones que no se pueden adaptar a distintos niveles de QoS o que se vuelven inservibles cuando se producen descensos en la calidad. Ejemplos de esto son algunas aplicaciones médicas (un síntoma puede aparecer en un vídeo de rayos x justo en el instante en el que los marcos del vídeo fueron desechados) y grabaciones de vídeo (donde los marcos desechados producen un vacío en la grabación que será observable siempre que se reproduzca el vídeo).

La reserva basada en los requisitos máximos es directa: cuando se controla el acceso a una red con un cierto ancho de banda B, se pueden admitir caudales multimedia s de ancho de banda b_s siempre que $\Sigma b_s <= B$. Así, una red token ring de 16 Mb/s puede soportar hasta 10 caudales de vídeo digital de 1,5 Mb/s cada uno.

Desafortunadamente, los cálculos de capacidad no siempre son tan simples como en el caso del ancho de banda de una red. Para reservar ancho de banda de CPU del mismo modo se necesita

conocer el perfil de ejecución de cada proceso de aplicación. Los tiempos de ejecución, sin embargo, dependen del procesador utilizado y a menudo no se pueden determinar con precisión. Aunque existen varias propuestas para el cálculo automático del tiempo de ejecución [Mok 1985, Kopetz y otros 1989], ninguna de ellas ha sido utilizada de forma generalizada. Los tiempos de ejecución son determinados generalmente a partir de medidas con amplios márgenes de error y portabilidad limitada.

Para las codificaciones de medios típicas como MPEG, el ancho de banda consumido realmente por una aplicación es significativamente menor que el ancho de banda máximo. Las reservas basadas en los requisitos máximos pueden conducir, por lo tanto, a desperdiciar ancho de banda: las peticiones de nuevas admisiones son rechazadas aunque podrían ser satisfechas con el ancho de banda reservado, pero no utilizado en la realidad por las aplicaciones existentes.

◇ **Multiplexación estadística.** Dada la potencial infrautilización que se puede producir, es común sobreestimar la reserva de recursos. Las garantías resultantes, a menudo llamadas garantías estadísticas o blandas para distinguirlas de las deterministas o duras presentadas anteriormente, son únicamente válidas con alguna probabilidad (normalmente alta). Las garantías estadísticas tienden a proporcionar una utilización mejor de los recursos ya que no consideran el peor caso. Pero como cuando la reserva de recursos está basada en requisitos mínimos o promedios, picos de carga simultáneos pueden producir caídas en la calidad de servicio; las aplicaciones tienen que ser capaces de manejar estas caídas.

La multiplexación estadística está basada en la hipótesis de que para un gran número de caudales el ancho de banda agregado requerido permanece casi constante, independientemente del ancho de banda de los caudales individuales. Esto supone que cuando un caudal envía una gran cantidad de datos, también existirá otro que envía una pequeña cantidad de datos y que los requisitos generales se equilibran. Esto, sin embargo, es cierto únicamente para caudales no correlacionados.

Como han mostrado los experimentos [Leland y otros 1993], el tráfico multimedia en un entorno típico no obedece a esta hipótesis. Dado un gran número de caudales a ráfagas, el tráfico agregado también tendrá ráfagas. A este fenómeno se le ha aplicado el término *autosimilaridad*, indicando que el tráfico agregado muestra semejanza con los caudales individuales de los que se compone.

15.4. GESTIÓN DE RECURSOS

Para proporcionar un cierto nivel de QoS a una aplicación, un sistema no sólo necesita tener recursos suficientes (prestaciones), sino que necesita poner esos recursos a disposición de la aplicación cuando sean necesarios (planificación o *scheduling*).

15.4.1. PLANIFICACIÓN DE RECURSOS

Los procesos necesitan tener recursos asignados de acuerdo con su prioridad. Un planificador de recursos determina la prioridad de un proceso basándose en ciertos criterios. Los planificadores de CPU tradicionales de los sistemas de tiempo compartido basan sus asignaciones de prioridad en la capacidad de respuesta y en la imparcialidad: las tareas con una entrada/salida intensa obtienen una prioridad alta para garantizar una rápida respuesta a las peticiones de los usuarios, las tareas asociadas a la CPU reciben una prioridad baja y en general, los procesos con igual prioridad son tratados de la misma manera.

Ambos criterios siguen siendo válidos en los sistemas multimedia, pero la existencia de tiempos límite de entrega de los elementos de datos multimedia modifican la naturaleza del problema de la planificación. Los algoritmos de planificación de tiempo real se pueden aplicar a este problema como se trata más adelante. Como los sistemas multimedia tienen que gestionar medios tanto continuos como discretos, se presenta el desafío de proporcionar suficiente servicio a los caudales dependientes del tiempo sin causar la inanición de las aplicaciones que acceden a medios discretos y de otras aplicaciones interactivas.

Los métodos de planificación tienen que ser aplicados a (y coordinarse para) todos los recursos que afectan a las prestaciones de las aplicaciones multimedia. En un escenario típico, un caudal multimedia será recuperado de un disco y enviado a través de la red a otra estación, donde es sincronizado con un caudal de otra fuente y finalmente será mostrado. Los recursos necesarios en el ejemplo comprenden el disco, la red, la CPU, la memoria y el ancho de banda del bus de todos los sistemas involucrados.

◇ **Planificación imparcial.** Si varios caudales compiten por el mismo recurso, se hace necesario considerar un reparto equitativo y prevenir qué caudales con comportamiento anómalo tomen demasiado ancho de banda. Una aproximación directa para asegurar la equidad es aplicar una planificación de ronda a todos los caudales de la misma clase. Mientras que este método fue introducido por Nagle [1987] basándose en paquetes, en Demers y otros [1989] se utiliza basándose en bits, lo que proporciona una mayor equidad con respecto a los tamaños y a los tiempos de llegada variables de los paquetes. Estos métodos se conocen como encolado justo.

Los paquetes no se pueden enviar bit a bit, pero dada una cierta tasa de envío de marcos se puede calcular para cada paquete cuándo debiera haber sido enviado completamente. Si las transmisiones de paquetes se ordenan basándonos en este cálculo, se consigue casi el mismo comportamiento que con la ronda actual basada en bits, excepto que cuando se envía un paquete grande, puede bloquear la transmisión de un paquete más pequeño, que podría haber sido preferido bajo un esquema bit a bit. Sin embargo, los paquetes no se retardan más que el tiempo máximo de transmisión de paquetes.

Todos los esquemas de ronda básicos asignan el mismo ancho de banda a cada caudal. Para tomar en cuenta los anchos de banda de cada uno de los caudales, se puede extender el esquema bit a bit de forma que en algunos casos se puedan enviar un número de bits por ciclo. Este método se llama encolado justo ponderado.

◇ **Planificación de tiempo real.** Se han desarrollado varios algoritmos de planificación de tiempo real para responder a las necesidades de planificación de aplicaciones como procesos de control en la industria aeronáutica. Suponiendo que los recursos de CPU no han sido sobreasignados (lo que es tarea del gestor de la QoS), los algoritmos de planificación asignarán intervalos de tiempo de CPU a un conjunto de procesos de modo que se asegure que completan sus tareas a tiempo.

Los métodos de planificación de tiempo real tradicionales se ajustan muy bien a los modelos de caudales multimedia continuos. El planificador *el más próximo tiempo límite de entrega primero*, (*earliest-deadline-first*, EDF) se ha convertido en sinónimo de estos métodos. Un planificador EDF utiliza un tiempo de entrega límite asociado con cada elemento de trabajo para determinar el siguiente ítem a procesar: el ítem con el tiempo de entrega límite más cercano será el siguiente. En las aplicaciones multimedia, identificamos cada elemento entrante como un elemento de trabajo. El planificador EDF ha demostrado ser óptimo para reservar un recurso único basándose en criterios de tiempo: si existe una planificación que cumpla todos los requisitos temporales, la planificación EDF la encontrará [Dertouzos 1974].

La planificación EDF necesita una sola decisión de planificación por mensaje (o lo que es lo mismo, por elemento multimedia). Podría ser más eficiente basar la planificación en los elementos que existen por un tiempo más largo. La planificación de tasa monótona (*rate-monotonic*), RM, es una técnica prominente de planificación en tiempo real que consigue eso con procesos periódicos. Los caudales reciben prioridades dependiendo de su tasa: cuando mayor la tasa de elementos de trabajo que tenga un caudal, mayor será la prioridad asignada al caudal. La planificación RM se ha

mostrado óptima en situaciones en las que se utiliza menos del 69 por ciento del ancho de banda [Liu y Layland 1973]. Utilizándolo como un esquema de reserva, el ancho de banda remanente podría ser asignado a las aplicaciones que no sean de tiempo real.

Para tratar con las ráfagas de tráfico de tiempo real, los métodos básicos de planificación de tiempo real deberían ajustarse para distinguir entre elementos de trabajo críticos respecto al tiempo y aquellos que no lo son. En Govindan y Anderson [1991] se presenta una planificación *tiempo entrega límite/trabajo adelantado*. Dicha planificación permite que los mensajes de un caudal continuo lleguen antes de tiempo en ráfagas, pero aplica la planificación EDF a los mensajes sólo cuando se cumple su tiempo de llegada normal.

15.5. ADAPTACIÓN DE CAUDALES

Cuando no se puede garantizar una cierta QoS o sólo se puede garantizar con una cierta probabilidad, una aplicación necesita adaptarse a niveles de QoS cambiantes, ajustando sus prestaciones de manera acorde. Para los caudales multimedia continuos, los ajustes se traducen en diferentes niveles de calidad en la presentación de los medios.

La forma más simple de ajuste es desechar elementos de información. Esto es fácilmente realizable en los caudales de audio, donde las muestras son independientes unas de otras, pero pueden ser detectadas inmediatamente por el que escucha. Desechar en un caudal de vídeo codificado en Motion JPEG, donde cada marco es independiente, es más tolerable. Los mecanismos de codificación como MPEG, donde la interpretación de un marco depende de los valores de varios marcos adyacentes, son menos robustos frente a las omisiones: necesitan más tiempo para recuperarse de los errores y el mecanismo de codificación puede amplificar los errores.

Si hay un ancho de banda insuficiente y los datos no son desechados, el retardo de un caudal se incrementará en el tiempo. Para las aplicaciones no interactivas esto puede ser aceptable, aunque pueda conducir eventualmente a desbordamientos del búfer ya que los datos se acumulan entre el origen y el destino. Para conferencias y otras aplicaciones interactivas, los retardos incrementales no son aceptables, o deben existir sólo en un período corto de tiempo. Si un caudal va detrás de su tiempo de reproducción, debería aumentarse su tasa de reproducción hasta alcanzar su planificación prevista: mientras un caudal esté retrasado, sus marcos deberían ser procesados y deberían salir tan pronto estén disponibles.

15.5.1. ESCALADO

Si la adaptación se produce en el destino de un caudal, no se reducirá la carga de cualquier cuello de botella del sistema, y la situación de sobrecarga continuará. Para resolver esta contención, será útil adaptar el caudal al ancho de banda disponible en el sistema antes de que entre en cuello de botella de recurso. Esto se conoce como escalado.

El escalado se aplica mejor cuando se muestrean caudales vivos. Para caudales almacenados, la facilidad para obtener un caudal de menor grado dependerá del método de codificación utilizado. El escalado puede ser demasiado engorroso si se tiene que descomprimir el caudal entero y codificarlo otra vez para dicho propósito. Los algoritmos de escalado son dependientes del medio, aunque la aproximación general del escalado es siempre la misma: obtener una muestra de calidad inferior de la señal dada. Para información de audio, tales muestreos de calidad inferior se pueden conseguir reduciendo las tasas de muestreo. También se pueden conseguir desechando un canal de una transmisión estéreo. Como muestra este ejemplo, diferentes métodos de escalado pueden servir para diferentes niveles de detalle.

Para vídeo, los siguientes métodos de escalado son apropiados:

Escalado temporal: reduce la resolución del caudal de vídeo en el dominio del tiempo decrementando el número de marcos de vídeo enviados dentro de un intervalo de tiempo. El escalado temporal es la mejor elección para los caudales de vídeo en los que los marcos son autocontenidos y se pueden acceder de forma independiente. Las técnicas de compresión delta son más difíciles de manejar ya que no todos los marcos pueden ser desechados fácilmente. Por lo tanto, el escalado temporal es más adecuado para caudales Motion JPEG que para MPEG.

Escalado espacial: reduce el número de píxeles (puntos de la imagen) de cada imagen en el caudal de vídeo. Para el escalado espacial, la disposición jerárquica resulta ideal, porque el vídeo comprimido está disponible inmediatamente en varias resoluciones. Por lo tanto, el vídeo puede ser transferido sobre la red utilizando diferentes resoluciones sin almacenar cada imagen antes de transmitirla finalmente. JPEG y MPEG-2 soportan diferentes resoluciones espaciales de las imágenes y son adecuados para este tipo de escalado.

Escalado en frecuencia: modifican el algoritmo de compresión aplicado a una imagen. Esto produce una cierta pérdida de calidad, pero en una imagen típica, se puede incrementar significativamente la compresión antes de que la reducción de calidad se vuelva visible.

Escalado en amplitud: reduce la profundidad de color para cada píxel de imagen. Este método de escalado se utiliza en las codificaciones H.261 para conseguir que las imágenes lleguen a una tasa constante cuando los contenidos varían.

Escalado en el espacio del color: reduce el número de entradas en el espacio del color. Una forma de realizarlo es pasando una imagen en color a otra en escala de grises.

Se puede utilizar una combinación de estos métodos si fuera necesario.

Un sistema para efectuar escalado está compuesto por un monitor de procesos en el destino y un proceso de escalado en el origen. El monitor lleva cuenta de los tiempos de llegada de los mensajes de un caudal. Los mensajes retrasados son una indicación de la existencia de un cuello de botella en el sistema. El monitor envía un mensaje de disminuir la escala a la fuente y entonces la fuente reduce el ancho de banda del caudal. Después de un período de tiempo, la fuente aumenta la escala otra vez. Si el cuello de botella todavía existiera, el monitor volvería a detectar el retraso y mandaría disminuir la escala [Delgrossi y otros 1993]. Un problema en los sistemas de escalado es evitar las operaciones de aumento de escala innecesarias y prevenir las oscilaciones en el sistema.

15.5.2. FILTRADO

Como el escalado modifica el caudal en el origen, no siempre resulta adecuado para las aplicaciones que involucran varios receptores: cuando el cuello de botella se produce en la ruta a un destino, este destino envía un mensaje al origen para disminuir la escala, y a partir de este momento todos los receptores recibirán un caudal de menor calidad, aunque alguno de ellos no hubiera tenido problemas para manejar el caudal original.

El filtrado es un método que proporciona la QoS mejor posible a cada destino aplicando escalado en cada nodo representativo en el camino desde el origen a cada destino (véase la Figura 15.9). El protocolo RSVP [Zhang y otros 1993] es un ejemplo de protocolo de negociación que soporta filtrado. El filtrado requiere que un caudal sea descompuesto en un conjunto de subcaudales jerárquicos, cada uno añadiendo un nivel superior de calidad. La capacidad de los nodos de un camino determina el número de subcaudales que recibe un destino. Los demás subcaudales serán filtrados en nodos lo más cercanos posible al origen (quizás incluso en el mismo origen) para evitar la transferencia de datos que serán desechados después. Un subcaudal no es filtrado en un nodo intermedio si existe mas allá un camino que puede soportar el caudal entero.

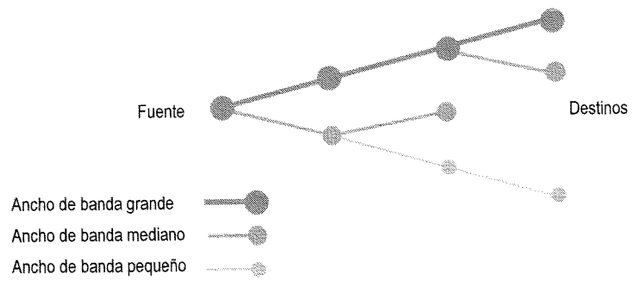


Figura 15.9. Filtrado.

15.6. CASO DE ESTUDIO: EL SERVIDOR DE VÍDEO TIGER

Un sistema de almacenamiento de vídeo que proporcione múltiples caudales de vídeo en tiempo real es un componente importante para las aplicaciones multimedia orientadas al consumidor. Se han desarrollado varios prototipos de este tipo, y algunos han evolucionado hacia productos (véase [Cheng 1998]). Uno de los más avanzados es el servidor de vídeo Tiger desarrollado en los laboratorios de investigación de Microsoft [Bolosky y otros 1996].

◇ **Metas del diseño.** Los principales aspectos del diseño del sistema fueron los siguientes:

Vídeo bajo demanda para un gran número de usuarios: La aplicación típica es un servicio que proporcione películas de pago a los clientes. Las películas son seleccionadas de una gran biblioteca digital de películas. Los clientes deberían recibir los primeros marcos de las películas que han seleccionado a los pocos segundos de emitir una petición, y deberían ser capaces de realizar una pausa, rebobinar o avanzar rápido a voluntad. Aunque la biblioteca de películas sea grande, unas pocas películas pueden ser muy populares y podrán ser objeto de múltiples peticiones no sincronizadas, produciendo varias reproducciones concurrentes de las mismas pero desplazadas en el tiempo.

Calidad de Servicio: Se deben proporcionar caudales de vídeo a una tasa constante con una fluctuación máxima determinada por la (supuestamente pequeña) cantidad de almacenamiento disponible en los clientes y una tasa de pérdidas muy baja.

Escalabilidad y distribución: La idea era diseñar un sistema con una arquitectura extensible (mediante la adición de computadores) para soportar hasta 10.000 clientes de forma simultánea.

Hardware barato: El sistema tenía que ser construido utilizando hardware de bajo costo (como PCs con controladores de disco estándares).

Tolerancia a fallos: El sistema debería continuar funcionando sin una degradación apreciable de la calidad después de un fallo de cualquier servidor o disco.

Considerados conjuntamente, estos requisitos demandan una aproximación radical al almacenamiento y recuperación de datos de vídeo y un algoritmo de planificación efectivo que equilibre la carga entre un gran número de servidores similares. La primera tarea es la transferencia de secuencias de datos de vídeo de ancho de banda alto desde el almacenamiento en disco hasta una red, y ésta es la carga que será compartida entre los servidores.

◇ **Arquitectura.** La Figura 15.10 muestra la arquitectura hardware de Tiger. Todos los componentes son productos corrientes. Los computadores de cada *par* mostrados en la figura son PCs idénticos con el mismo número de discos estándar (normalmente entre 2 y 4). También están equipados con tarjetas de red Ethernet y ATM. El *controlador* es otro PC. No está involucrado en la gestión de datos multimedia y únicamente es responsable de la gestión de las peticiones de los clientes y de la gestión de la planificación del trabajo en los pares.

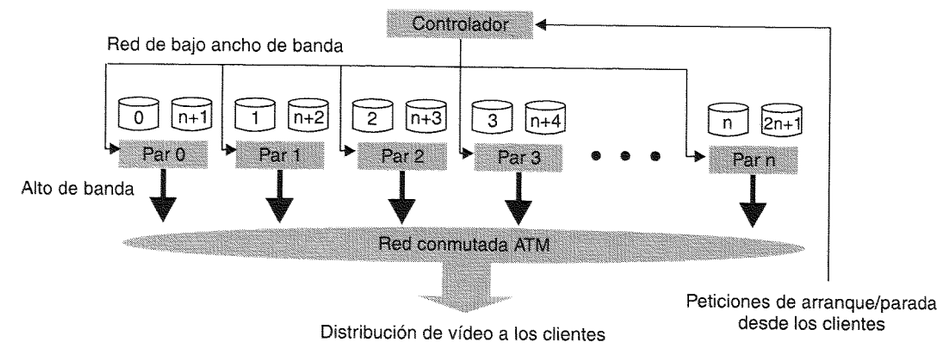


Figura 15.10. Configuración hardware del servidor de vídeo Tiger.

◇ **Organización del almacenamiento.** La clave en el diseño es la distribución de los datos de vídeo entre los discos de los pares para hacer posible que compartan la carga. Cualquier solución basada en el uso de un único disco para almacenar cada película es difícil que consiga el objetivo, ya que la carga puede implicar la aportación de múltiples caudales de la misma película así como caudales de distintas películas. En su lugar, las películas se almacenan en una representación de franjas a lo largo de todos los discos. Esto conduce a un modelo de fallos en el cual la pérdida de un disco o de un par produce un hueco vacío en la secuencia de cada película. Esto se puede solucionar con un esquema de almacenamiento reflejado, que replica los datos, y por un mecanismo de tolerancia a fallos, según se describe más abajo.

Franjas: Una película se divide en *bloques* (trozos de vídeo de igual tiempo de reproducción, generalmente alrededor de 1 segundo, ocupando 0,5 Mbytes), y el conjunto de bloques que componen una película (normalmente unos 7.000 para una película de dos horas) se almacena en discos de distintos pares según la secuencia indicada por los números de los discos de la Figura 15.10. Una película puede comenzar en cualquier disco. Cuando se alcanza el disco de mayor numeración, la película *da la vuelta* de modo que el siguiente bloque está almacenado en el disco 0 y el proceso continúa.

Espejado: El esquema de reflejado divide cada bloque en varias porciones, llamadas *secundarias*. Esto asegura que cuando un par falla, la carga extra de proporcionar los datos de los bloques del par caído se reparte entre varios de los pares restantes, y no sólo en uno de ellos. El número de secundarios por bloque está determinado por un *factor de disociación*, d, con valores típicos en el rango de 4 a 8. Los secundarios de un bloque almacenado en el disco i son almacenados en los discos del $i + 1$ al $i + d$. Hay que destacar que si hay más de d pares, ninguno de esos discos estará en el mismo par que el disco i. Con un factor de disociación de 8, aproximadamente 7/8 de la capacidad de proceso y del ancho de banda de los pares se puede reservar para las tareas libres de fallos. El 1/8 restante de sus recursos deberían ser suficientes para servir los secundarios cuando sea necesario.

◇ **Planificación distribuida.** El corazón del diseño de Tiger es la planificación de la carga de trabajo para los pares. La planificación está organizada según una lista de *ranuras*, donde cada ranura representa el trabajo que se debe realizar para reproducir un bloque de una película; esto es,

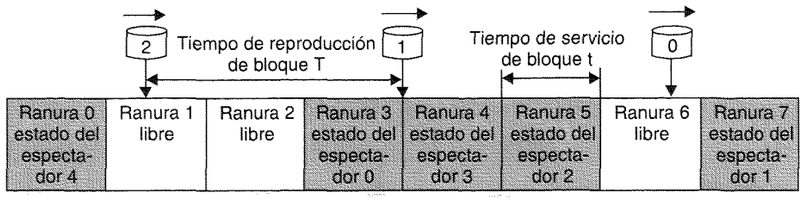


Figura 15.11. Planificación del servidor Tiger.

para leerlo del disco relevante y transferirlo a la red ATM. Hay exactamente una ranura para cada cliente potencial que reciba una película (llamado *espectador*), y cada ranura ocupada representa un espectador recibiendo un caudal de vídeo en tiempo real. El estado de reproducción está representado en el planificador por:

- La dirección del computador cliente.
- La identidad del archivo que se está reproduciendo.
- La posición del espectador en el archivo (el siguiente bloque que hay que entregar al caudal).
- La número de secuencia de reproducción (a partir de la cual se calcula el tiempo de entrega del siguiente bloque).
- Alguna información sobre la reserva.

La planificación se ilustra en la Figura 15.11. El *tiempo de reproducción del bloque T* es el tiempo que será necesario para que un reproductor muestre un bloque en el computador cliente, normalmente es de alrededor de un segundo y se supone que es el mismo para todas las películas almacenadas. El servidor Tiger debe, por lo tanto, mantener un intervalo de tiempo T entre los instantes de entrega de los bloques de cada caudal, con un salto admisible determinado por el almacenamiento disponible en los computadores clientes.

Cada par mantiene un apuntador a la planificación por cada disco que controla. Durante el tiempo de reproducción de cada bloque debe procesar todas las ranuras con números de bloque que caigan en los discos que controla y cuyos tiempos de entrega estén dentro del tiempo de reproducción de bloque actual. Los pasos que siguen los pares para procesar las ranuras en tiempo real son:

1. Leer el siguiente bloque del almacenamiento del par.
2. Empaquetar el bloque con la dirección del computador cliente y entregarlo al controlador de la red ATM del par.
3. Actualizar el estado de reproducción indicando el siguiente número de bloque y número de secuencia de reproducción y pasar la ranura actualizada al siguiente par.

Se supone que estas acciones ocupan un tiempo máximo t, conocido como *tiempo de servicio de bloque*. Como se puede observar en la Figura 15.11, t es substancialmente menor que el tiempo de reproducción de bloque. El valor de t está determinado por el ancho de banda del disco o de la red, dependiendo de cuál sea el más pequeño. (Los recursos de procesamiento en un par son adecuados para ejecutar el trabajo planificado para todos los discos a él conectados a él.) Cuando un par ha completado las tareas planificadas para el tiempo de reproducción de bloque actual, estará disponible para las tareas no planificadas hasta el comienzo del siguiente tiempo de reproducción. En la práctica, los discos no proporcionan bloques con un retardo fijo, y para subsanar esta entrega irregular la lectura en el disco se inicia al menos un tiempo de servicio de bloque antes de que el bloque sea necesitado para empaquetar y entregar.

Un disco puede manejar la carga de servir T/t caudales, y los valores de T y de t generalmente dan un valor > 4 para esta relación. Esto y el número de discos en el sistema entero determinan el número de espectadores que puede servir un sistema Tiger. Por ejemplo, un sistema Tiger con cinco pares y tres discos en cada par entregará aproximadamente 70 caudales de vídeo simultáneos.

◇ **Tolerancia a fallos.** Dado que los archivos de todas las películas se hallan repartidos sobre todos los discos en el sistema Tiger, el fallo de cualquier componente (disco o par) podría producir un interrupción en el servicio a todos los clientes. El diseño de Tiger remedia esto recuperando los datos de las copias secundarias reflejadas cuando un bloque primario no está disponible ya sea por un fallo del disco o del par correspondiente. Hay que recordar que los bloques secundarios son de menor tamaño que los primarios en una proporción dada por el factor de disociación d, y que los secundarios están distribuidos de modo que se almacenan en discos de distintos pares.

Cuando un disco o un par falla, la planificación se modifica por un par adyacente para indicar varios *estados de reproducción reflejados*, representado una carga de trabajo para los d discos que albergan los secundarios para esas películas. Un estado de reproducción reflejado es similar a un estado de reproducción pero con números de bloque y requisitos de sincronización distintos. Debido a que esta carga de trabajo extra es compartida entre d discos y d pares, puede ser soportada sin interrumpir las tareas de las otras ranuras, gracias a que hay una pequeña cantidad de espacio libre en la planificación. El fallo de un par es equivalente al fallo de todos los discos que contiene y se gestiona de un modo similar.

◇ **Soporte de red.** Los bloques de cada película son simplemente pasados a la red ATM por los pares que los contienen, junto con la dirección del cliente. Se confía en la QoS garantizada por los protocolos de red ATM (véase la Sección 3.5.3) para que la entrega de los bloques a los clientes se haga en secuencia y a tiempo. El cliente necesita suficiente espacio de almacenamiento para almacenar dos bloques primarios, uno que se está reproduciendo en la pantalla del cliente y el otro que está llegando por la red. Cuando los bloques primarios están siendo servidos, el cliente necesita comprobar el número de secuencia de cada bloque que llega y pasárselo al gestor de la reproducción. Cuando se sirven secundarios, los d pares responsables de entregar los secundarios del bloque disociado mandan sus secundarios a la red en secuencia, y es responsabilidad del cliente recogerlos y ensamblarlos en su espacio de almacenamiento.

◇ **Otras funciones.** Hemos descrito las actividades críticas respecto al tiempo de un servidor Tiger. Los requisitos de diseño incluían la consideración de las funciones de avance rápido y de rebobinado. Estas funciones implican la entrega de alguna fracción de bloques de la película al cliente de forma que se dé la impresión visual aportada por los reproductores de vídeo clásicos. Esto se realiza por los pares en tiempo fuera de la planificación y según el mejor esfuerzo posible.

Las tareas restantes incluyen la gestión y distribución de la planificación y la gestión de la base de datos de películas, borrado de las antiguas y escritura de las nuevas en los discos, y mantenimiento del índice de películas.

En la implementación inicial del servidor Tiger, la gestión y distribución de la planificación era realizada por el computador controlador. Pero como esto constituye un punto único de fallo y un potencial cuello de botella en las prestaciones, la gestión de la planificación se rediseñó como un algoritmo distribuido [Bolosky y otros 1997]. La gestión de la base de datos de películas se lleva a cabo por los pares en el tiempo no planificado en respuesta a comandos del controlador.

◇ **Prestaciones y escalabilidad.** El prototipo inicial fue desarrollado en 1994 y utilizó cinco PCs Pentium a 133 MHz, cada uno de ellos equipado con 48 Mbytes de RAM y tres discos SCSI de 2 Gbytes y un controlador de red ATM, corriendo sobre Windows NT. Esta configuración fue medida sobre una carga de clientes simulada. Cuando se servía películas a 68 clientes sin fallos en el sistema Tiger, la entrega de datos era perfecta; no se perdieron bloques ni se entregaron tarde a los clientes. Con un par caído (y por lo tanto tres discos) el servicio fue mantenido con una tasa de pérdidas de tan sólo 0,02 por ciento, dentro de los objetivos de diseño.

Otra medida que se tomó fue la latencia de entrega del primer bloque de una película después de recibir una petición de un cliente. El algoritmo utilizado inicialmente colocaba la petición del cliente en la ranura libre más cercana al disco que contenía el bloque 0 de la película solicitada. Esto producía unos valores medidos para la latencia de arranque entre 2 y 12 segundos. Trabajos recientes han producido un algoritmo de reserva de ranuras que reduce el agrupamiento de ranuras ocupadas en la planificación y mejora la latencia de arranque media [Douceur y Bolosky 1999].

Aunque los experimentos iniciales fueron hechos con una configuración pequeña, se realizaron medidas posteriores con una configuración de catorce pares y 56 discos, y el algoritmo de planificación distribuido descrito por Bolosky y otros [1997]. La carga que podría ser servida por este sistema aumentó hasta entregar 602 caudales simultáneos de 2 Mbps con una tasa de pérdidas de un bloque entre 180.000 cuando todos los pares funcionaban. Con un par caído, se perdían menos de 1 cada 40.000 bloques. Estos resultados son impresionantes y parecen confirmar la impresión de que podría configurarse un sistema Tiger con hasta 1.000 pares sirviendo hasta 30.000 ó 40.000 reproducciones simultáneas. El sistema Tiger ha evolucionado hacia un producto software llamado Microsoft NetShow Theater Sever [www.microsoft.com].

15.7. RESUMEN

Las aplicaciones multimedia necesitan nuevos mecanismos que permitan gestionar grandes volúmenes de datos dependientes del tiempo. Los más importantes de estos mecanismos están centrados en la gestión de la calidad de servicio. Deben reservar ancho de banda y otros recursos de modo que aseguren que se pueden satisfacer los requisitos de una aplicación, y deben planificar el uso de los recursos de forma que se puedan cumplir los muy frecuentes tiempos límite de entrega de las aplicaciones multimedia.

Las gestión de la calidad de servicio maneja las peticiones de QoS de las aplicaciones que especifican el ancho de banda, la latencia y la tasa de pérdidas aceptables para los caudales multimedia, y efectúan los mecanismos de control de admisión, determinado cuándo están disponibles los recursos suficientes para satisfacer cada nueva petición y negociando con la aplicación si fuera necesario. Una vez que una petición de QoS es aceptada, se reservan los recursos y se envía una garantía a la aplicación.

La capacidad del procesador y el ancho de banda reservados a una aplicación debe ser planificada para satisfacer las necesidades de la aplicación. Se necesita un algoritmo de planificación de tiempo real como el *tiempo de entrega límite más cercano* o como el de *tasa monótona* para asegurar que cada elemento del caudal es procesado a tiempo.

El moldeado del tráfico es el nombre dado a los algoritmos que almacenan los datos en tiempo real para suavizar las irregularidades temporales que inevitablemente se darán. Los caudales se pueden adaptar para utilizar menos recursos reduciendo el ancho de banda de la fuente (escalado) o en puntos a lo largo del camino (filtrado).

El servidor de vídeo Tiger es un excelente ejemplo de sistema escalable que proporciona entrega de caudales a una escala potencialmente grande con una fuerte garantía de calidad de servicio. Su planificación de los recursos está altamente especializada, y ofrece un excelente ejemplo del cambio en el diseño que estos sistemas necesitan.

EJERCICIOS

15.1. Diseñe a grandes rasgos un sistema que soporte una aplicación distribuida de reproducción de música. Sugiera los requisitos adecuados de QoS y la configuración hardware y software que podría utilizarse.

15.2. Internet actualmente no ofrece ninguna posibilidad de reserva de recursos ni de gestión de la calidad del servicio. ¿Cómo consiguen una calidad aceptable las aplicaciones existentes de audio y vídeo basadas en Internet? ¿Qué limitaciones imponen a las aplicaciones multimedia las soluciones que adoptan?

15.3. Explique las diferencias entre las tres formas de sincronización (estado distribuido síncrono, sincronización de medios, sincronización externa) que pueden necesitarse en las aplicaciones multimedia distribuidas. Sugiera mecanismos por los que cualquiera de ellas pueda ser conseguida, por ejemplo en una aplicación de videoconferencia.

15.4. Esboce el diseño de un gestor de QoS que posibilite que computadores conectadas por una red ATM soporten varias aplicaciones multimedia concurrentes. Defina un API para su gestor de QoS, dando las operaciones principales con sus argumentos.

15.5. Para especificar las necesidades de componentes software que procesen datos multimedia, necesitamos estimar sus cargas de procesamiento. ¿Cómo se puede obtener esta información sin un esfuerzo excesivo?

15.6. ¿Cómo hace el sistema Tiger para soportar un gran número de clientes todos solicitando la misma película en instantes de tiempo aleatorios?

15.7. La planificación del servidor Tiger es potencialmente una estructura de datos que cambia frecuentemente, pero cada par necesita una representación actualizada de las porciones que está gestionando actualmente. Sugiera un mecanismo para la distribución de la planificación entre los pares.

15.8. Cuando el sistema Tiger trabaja con un disco caído o con un par caído, los bloques de datos secundarios se utilizan para sustituir a los primarios perdidos. Los bloques secundarios son n veces más pequeños que los primarios (donde n es el factor de disociación), ¿cómo hace el sistema para acomodarse a la variabilidad en el tamaño del bloque?

16

MEMORIA COMPARTIDA DISTRIBUIDA

16.1. Introducción
16.2. Cuestiones de diseño e implementación
16.3. Consistencia secuencial e Ivy
16.4. Liberación de consistencia y Munin
16.5. Otros modelos de consistencia
16.6. Resumen

En este capítulo se describe la memoria compartida distribuida (*Distributed Shared Memory*, DSM), una abstracción utilizada para la compartición de datos entre procesos en computadores que no comparten su memoria física. La motivación de DSM consiste en poder utilizar un modelo de programación de memoria compartida, que tiene algunas ventajas sobre los modelos basados en mensajes. Por ejemplo, en DSM los programadores no tienen que empaquetar los datos para su envío.

El principal problema en las implementaciones DSM es cómo conseguir mantener unas buenas prestaciones a medida que crece el número de computadores del sistema. Los accesos a DSM implican la utilización potencial de la red de comunicaciones. Los procesos que compiten para acceder a los mismos datos o a datos cercanos pueden generar un gran número de operaciones de comunicación. La cantidad de operaciones de comunicación está muy relacionada con el modelo de consistencia en un sistema DSM; éste es el modelo que determina cuál de los múltiples datos escritos deberá devolverse cuando el proceso lee desde una cierta posición DSM.

En este capítulo se discuten cuestiones de diseño de DSM, como el modelo de consistencia, y cuestiones de implementación, como cuál de las copias de un mismo dato son invalidadas o actualizadas cuando el dato es modificado. Los protocolos de invalidación se estudian con mayor nivel de detalle. Finalmente se describe la consistencia relajada (*release*), un modelo de consistencia relativamente débil, adecuado para múltiples propósitos y poco costoso en su implementación.

16.1. INTRODUCCIÓN

La memoria compartida distribuida (DSM) es una abstracción utilizada para compartir datos entre computadores que no comparten memoria física. Los procesos acceden a DSM para leer y actualizar, dentro de sus espacios de direcciones, sobre lo que aparenta ser la memoria interna normal asignada a un proceso. Sin embargo, existe un sistema subyacente en tiempo de ejecución que asegura de forma transparente que procesos diferentes ejecutándose en computadores diferentes observen las actualizaciones realizadas entre ellos. Es como si los procesos accedieran a una única memoria compartida, pero de hecho la memoria física está distribuida (véase la Figura 16.1).

La principal característica de DSM es que ahorra al programador todo lo concerniente al paso de mensajes al escribir sus aplicaciones, cuestión que en otro sistema debería tenerse muy presente. DSM es fundamentalmente una herramienta para aplicaciones paralelas o para aplicaciones o grupos de aplicaciones distribuidas en las que se puede acceder directamente a datos individuales que ellas comparten. En general, DSM es menos apropiado para sistemas cliente-servidor, ya que los clientes ven al servidor como un gestor de recursos en forma de datos abstractos que se acceden a través de peticiones (por razones de modularidad y protección). Sin embargo, los servidores pueden proporcionar DSM compartido entre los clientes. Por ejemplo, los archivos plasmados en memoria (*memory mapped*) que son compartidos y sobre los que se gestiona un cierto grado de consistencia son una forma de DSM. (Los archivos reflejados en memoria se introdujeron en el sistema operativo MULTICS [Organick 1972].)

El paso de mensajes no puede ser eliminado completamente en un sistema distribuido: en ausencia de memoria compartida físicamente, el soporte en tiempo de ejecución de DSM envía las actualizaciones mediante mensajes entre computadores. Los sistemas DSM gestionan datos replicados: cada computador tiene una copia local de aquellos datos almacenados en DSM que han sido usados recientemente, con el fin de acelerar sus accesos. Los problemas asociados a la implementación de DSM tienen relación con los discutidos en el Capítulo 14 y con los relativos a los archivos compartidos mediante cachés, discutidos en el Capítulo 8.

Uno de los primeros y más notables ejemplos de una implementación DSM fue el sistema de archivos de Apollo Domain [Leach y otros 1983], en el que los procesos ejecutándose en las diferentes estaciones de trabajo comparten archivos mediante su correspondencia (*mapping*) simultánea sobre sus espacios de direcciones. Este ejemplo muestra que la memoria compartida distribuida puede ser persistente. Es decir, puede sobrepasar la ejecución de cualquier proceso o grupo de ellos que accede a ella y ser compartida por diferentes grupos de procesos a lo largo del tiempo.

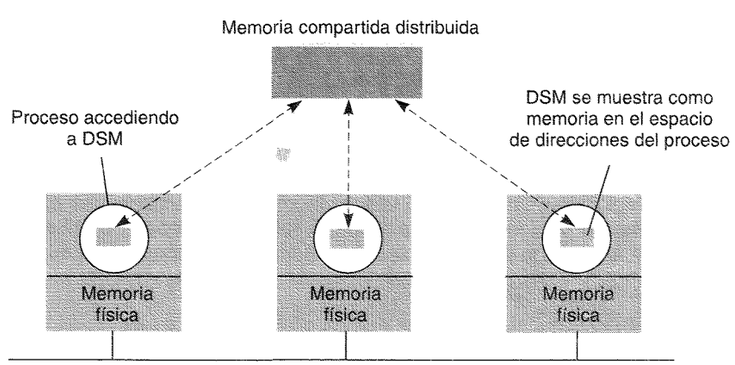


Figura 16.1. La abstracción de la memoria compartida distribuida.

La importancia de DSM ha crecido junto con el desarrollo de los multiprocesadores de memoria compartida (véase la Sección 6.3). Una gran parte de la investigación se ha orientado al desarrollo de algoritmos adaptados para la computación paralela en ese tipo de multiprocesadores. En el nivel de arquitectura del hardware, los desarrollos incluyen estrategias de memoria caché e interconexiones rápidas procesador-memoria con el objetivo de maximizar el número de procesadores que pueden ser conectados manteniéndose la productividad y minimizando las latencias de accesos a memoria [Dubois y otros 1988]. Si los procesadores se conectan a los módulos de memoria mediante bus común, el límite práctico oscila entre diez y veinte procesadores antes de que las prestaciones se degraden de forma drástica debido a la contención en el bus. Los procesadores que comparten memoria se agrupan normalmente de cuatro en cuatro, compartiendo cada grupo un módulo de memoria sobre un bus en un único circuito impreso. Se construyen de esta forma multiprocesadores con hasta 64 procesadores en arquitectura de *Memoria de Acceso No Uniforme* (*Non-Uniform Memory Access*, NUMA). Se trata de una arquitectura jerárquica en la que las placas de cuatro procesadores se conectan entre sí utilizando un switch de altas prestaciones o un bus de alto nivel. En una arquitectura NUMA, los procesadores ven un único espacio de direcciones que contiene toda la memora de todos los circuitos. Sin embargo, la latencia de acceso para la memoria dentro del mismo circuito es menor que la de acceso a un módulo de memoria de una tarjeta diferente, de ahí el nombre de la arquitectura.

En los *multiprocesadores de memoria distribuida* los procesadores no comparten memoria pero se conectan mediante una red de muy altas prestaciones. Estos multiprocesadores, al igual que en los sistemas distribuidos de propósito general, pueden escalar hasta un número mucho mayor de procesadores o computadores que los 64 de los multiprocesadores de memoria compartida. Una cuestión central perseguida por la comunidad investigadora de multiprocesadores y DSM, es si la inversión en conocimiento sobre algoritmos de memoria compartida y el software asociado puede ser directamente transferida a una arquitectura de memoria distribuida más escalable.

16.1.1. DSM FRENTE A PASO DE MENSAJES

Como mecanismo de comunicación, DSM es comparable con los sistemas de paso de mensajes en lugar de con la comunicación basada en peticiones y respuestas, ya que su aplicación concreta al procesamiento paralelo impone el uso de comunicación asíncrona. Los sistemas de programación DSM y de paso de mensajes pueden compararse de la siguiente forma:

Modelo de programación: en el modelo de paso de mensajes, las variables deben empaquetarse desde un cierto proceso, transmitirse y desempaquetarse sobre variables en el proceso receptor. Por el contrario, con memoria compartida, los procesos involucrados comparten directamente las variables, de forma que el empaquetamiento no es necesario (incluso cuando se trata de punteros a variables compartidas) y por lo tanto no son necesarias operaciones de comunicación separadas. La mayor parte de las implementaciones permiten nombrar y acceder a las variables almacenadas en DSM de forma similar a las variables no compartidas. Un aspecto favorable a los sistemas de paso de mensajes es que permiten que los procesos se comuniquen mientras se protegen entre ellos al disponer de espacios de direcciones privados, mientras que los procesos compartiendo DSM pueden, por ejemplo, provocar fallos entre ellos mediante la alteración errónea de los datos. Es más, el empaquetamiento en los sistemas de paso de mensajes entre computadores heterogéneos, resuelve las diferencias en la representación de los datos; pero, ¿cómo puede compartirse memoria entre computadores que, por ejemplo, representan los enteros de forma diferente?

La sincronización entre procesos se consigue, en el modelo de mensajes, mediante las propias primitivas de paso de mensajes, utilizando técnicas como la implementación del servidor

de bloqueo discutida en el Capítulo 11. Para DSM, la sincronización se realiza a través de construcciones normales de programación de memoria compartida, como los bloqueos y los semáforos (a pesar de que su implementación es diferente en un entorno de memoria distribuida). En el Capítulo 6 se discutieron brevemente estos objetos de sincronización en el contexto de la programación con hilos.

Finalmente, al ser DSM persistente, los procesos que se comunican mediante DSM pueden ejecutarse en instantes temporales no solapados. Un proceso puede dejar datos en una posición de memoria acordada para que otro proceso la examine cuando se ejecute. Por el contrario, los procesos que se comunican mediante paso de mensajes deben ejecutarse al mismo tiempo.

Eficiencia: los experimentos muestran que ciertos programas paralelos desarrollados para DSM pueden tener prestaciones similares a aquellos otros programas que se ejecutan sobre la misma plataforma y son funcionalmente equivalentes pero que han sido construidos usando un modelo de paso de mensajes [Carter y otros 1991] (por lo menos cuando el número de computadores es relativamente bajo, alrededor de diez). Sin embargo, este resultado no puede generalizarse. Las prestaciones de un programa basado en DSM dependen de muchos factores, como discutiremos más adelante; por ejemplo un factor importante es el patrón de compartición de datos (como si un dato es actualizado o no por varios procesos).

Hay una diferencia en la visibilidad de los costes asociados con los dos tipos de modelos de programación. En el paso de mensajes, todos los accesos remotos son explícitos y por lo tanto el programador siempre sabe si una cierta operación es local al proceso o bien supone un gasto en comunicación. Sin embargo, en DSM cualquier lectura o actualización puede suponer o no comunicación sobre el sistema de soporte en tiempo real. Si existe o no comunicación depende de factores del tipo de si los datos han sido o no accedidos antes y del patrón de compartición entre procesos en diferentes computadores.

No existe una respuesta definitiva sobre si DSM es preferible al paso de mensajes para una cierta aplicación. DSM es una herramienta prometedora cuya evolución dependerá de la eficiencia con la que pueda implementarse.

16.1.2. APROXIMACIONES A LA IMPLEMENTACIÓN DE DSM

La memoria compartida distribuida se implementa utilizando uno de los siguientes métodos o bien una combinación de ellos, hardware especializado, memoria virtual paginada convencional o middleware:

Hardware: las arquitecturas multiprocesador de memoria compartida basadas en una arquitectura NUMA (por ejemplo, Dash [Lenoski y otros 1992] y PLUS [Bisiani y Ravishankar 1990] se basan en hardware especializado para proporcionar a los procesadores una visión consistente de la memoria compartida. Gestionan las instrucciones de acceso a memoria LOAD y STORE de forma que se comuniquen con la memoria remota y los módulos de caché según sea necesario para almacenar y obtener datos. Esta comunicación se realiza sobre sistemas de interconexión de alta velocidad similares a una red. El prototipo del multiprocesador Dash tiene 64 nodos conectados mediante una arquitectura NUMA.

Memoria virtual paginada: muchos sistemas, incluyendo Ivy [Li y Hudak 1989], Munin [Carter y otros 1991], Mirage [Fleisch y Popek 1989], Clouds [Dasgupta y otros 1991] (véase www.cdk3.net/oss), Choices [Sane y otros 1990], COOL [Lea y otros 1993] y Mether [Minnich y Farber 1989], implementan DSM como una región de memoria virtual que ocupa el mismo rango de direcciones en el espacio de direcciones de cada proceso participante. Este tipo de implementación normalmente sólo es factible sobre una colección de computadores homogéneos con formatos de datos y de paginación comunes.

Middleware: algunos lenguajes del tipo de Orca [Bal y otros 1990] y middleware como Linda [Carriero y Gelernter 1989] junto con sus derivados JavaSpaces [java.sun.com VI] y TSpaces [Wyckoff y otros 1998] proporcionan DSM sin necesidad de soporte hardware o de paginación, de una forma independiente de la plataforma. En este tipo de implementación, la compartición se implementa mediante la comunicación entre instancias del nivel de soporte de usuario en los clientes y los servidores. Los procesos realizan llamadas a este nivel cuando acceden a datos en DSM. Las instancias de este nivel en las diferentes computadores acceden a los datos locales y se intercambian información siempre que sea necesario para el mantenimiento de la consistencia.

Este capítulo se ha enfocado a la utilización del software para implementar DSM sobre computadores estándar. Incluso con soporte hardware, se pueden utilizar técnicas software de alto nivel para minimizar la cantidad de comunicación entre componentes de una implementación DSM.

La aproximación basada en páginas tiene la ventaja de que no impone una estructura particular en la DSM, que se muestra como una secuencia de bytes. Inicialmente permite que los programas diseñados para multiprocesadores de memoria compartida se ejecuten en computadores sin memoria compartida, con muy poca o ninguna adaptación. Los micronúcleos como Mach y Chorus proporcionan soporte nativo para DSM (y otras abstracciones de memoria; los recursos de memoria virtual en Mach serán descritos en el Capítulo 18). Actualmente, los sistemas DSM basados en páginas son implementados en su mayor parte en el nivel de usuario debido a su mayor flexibilidad. La implementación utiliza el soporte del núcleo para los manejadores de fallos de las páginas de nivel de usuario. UNIX y algunas variantes de Windows proporcionan esta posibilidad. Los microprocesadores con espacios de direcciones de 64-bits amplían el ámbito de los sistemas DSM basados en páginas mediante la relajación de las restricciones en la gestión del espacio de direcciones [Bartoli y otros 1993].

En el ejemplo de la Figura 16.2 aparecen dos programas en C, *Lector* y *Escritor*, que se comu-

```
#include "world.h"
struct compartida { int a, b; } ;
Programa Escritor:
    main()
    {
        struct compartida *p;
        methersetup();                  /* Inicialización del sistema Mether */
        p = (struct compartida *)METHERBASE;
                                        /* estructura de overlay en el segmento METHER */
        p->a=p->b=0;                    /* inicialización a cero de los campos de la esctructura */
        while (TRUE){                   /* actualización continua de los campos de la estructura */
            p->a=p->a+1;
            p->b=p->b-1;
        }
    }
Programa Lector:
    main()
    {
        struct compartida *p;
        methersetup();
        p=(struct compartida *)METHERBASE;
        while(TRUE){                    /* lectura de los campos una vez por segundo */
            printf("a=%d, b=%d\n", p->a, p->b);
            sleep(1);
        }
    }
```

Figura 16.2. Programa en el sistema Mether.

nican a través de la DSM basada en páginas proporcionadas por el sistema Mether [Minnich y Farber 1989]. *Escritor* actualiza dos campos de una estructura asociada al principio de un segmento DSM en Mether (comenzando en la dirección METHERBASE) y *Lector*, imprime periódicamente los valores que lee de esos dos campos.

No existen operaciones especiales en los dos programas anteriores; se compilan para generar instrucciones máquina que acceden al rango común de direcciones de memoria virtual (comenzando a partir de METHERBASE). Mether se ejecutó en estaciones de trabajo convencionales Sun interconectadas por una red.

La aproximación del middleware es muy diferente a la utilización de hardware especializado y paginación ya que no está pensado para utilizar código existente de memoria compartida. Su importancia estriba en que nos permite el desarrollo de abstracciones de mayor nivel sobre objetos compartidos, en lugar de hacerlo sobre posiciones de memoria compartidas.

16.2. CUESTIONES DE DISEÑO E IMPLEMENTACIÓN

En esta sección se discuten las opciones de diseño e implementación relativas a las principales características de un sistema DSM. Es decir, se discute la estructura de los datos gestionados en DSM; el modelo de sincronización utilizado para acceder a DSM de forma consistente desde el nivel de aplicación; el modelo de consistencia de DSM, encargado de gobernar la consistencia de datos accedidos desde diferentes computadores; las opciones de actualización para la comunicación de valores escritos en diferentes computadores; la granularidad de la compartición en una implementación DSM; y el problema del *thrashing* (fustigamiento).

16.2.1. ESTRUCTURA

En el Capítulo 14 estudiamos sistemas que replicaban una colección de objetos como diarios y archivos. Estos sistemas permiten a los programas cliente realizar operaciones sobre los objetos como si existiera únicamente una copia de cada objeto, aunque en realidad están accediendo a réplicas físicas diferentes. Estos sistemas ofrecen garantías acerca de hasta qué punto se permite que se diferencien los objetos.

Un sistema DSM es equivalente a estos sistemas de replicación. Cada proceso de aplicación dispone de una abstracción de una colección de objetos, pero en este caso la *colección* tiene un aspecto muy parecido al que ofrece la memoria. Es decir, los objetos pueden ser direccionados de una u otra forma. Las diferentes aproximaciones de DSM difieren entre sí en qué es lo que consideran un *objeto* y en cómo se direccionan los objetos. Consideramos tres aproximaciones, que tienen una visión de DSM como formada por, respectivamente, una secuencia contigua de bytes, objetos de nivel de lenguaje o datos inmutables.

◇ **Orientada a byte.** Este tipo de DSM se utiliza como la memoria virtual ordinaria, es decir, como una cadena de bytes contiguos. Es la visión del sistema Mether explicada previamente. También es la imagen proporcionada por muchos otros sistemas DSM, incluyendo Ivy, que será descrito en la Sección 16.3. Permite que las aplicaciones (y las implementaciones de los lenguajes) almacenen cualquier tipo de estructura de datos sobre la memoria compartida. Los objetos compartidos son posiciones de memoria direccionables directamente (en la práctica, las posiciones de memoria pueden ser palabras multi-byte en lugar de bytes individuales). Las únicas operaciones permitidas

sobre esos objetos son *lee* (o LOAD) y *escribe* (o STORE). Si x e y son dos posiciones de memoria, las dos operaciones tienen la siguiente notación:

$L(x)a$ – una operación de *lectura* del valor a desde la posición x.

$E(x)b$ – una operación de *escritura* que almacena el valor b en la posición x.

Un ejemplo de ejecución es $E(x)1$, $L(x)2$. Este proceso escribe el valor 1 en la posición x y lee el valor 2 desde la misma posición. Otro proceso debe haber sido el encargado de escribir el valor 2 en la posición compartida.

◇ **Orientado a objetos.** La memoria compartida se estructura como una colección de objetos de nivel de lenguaje, como pilas o diccionarios, con una semántica de mayor nivel que la simple de *lectura/escritura* de variables. Los contenidos de la memoria compartida se modifican únicamente mediante invocaciones sobre dichos objetos y nunca a través del acceso directo a sus variables miembro. Una ventaja de esta visión de la memoria es que la semántica de objetos puede utilizarse para forzar la consistencia. La visión de DSM proporcionada por Orca es la de una colección de objetos compartidos en los que automáticamente se serializan las operaciones simultáneas realizadas sobre cualquiera de ellos.

◇ **Datos inmutables.** En este caso DSM se muestra como una colección de datos inmutables que los procesos pueden leer, sumar y eliminar. Algunos ejemplos son Agora [Bisiani y Forin 1988] y, más importante, Linda y sus derivados, TSpaces y JavaSpaces.

Los sistemas de tipo Linda proporcionan al programador una colección de tuplas llamado *espacio de tuplas*. Una tupla está formada por una secuencia de uno o más campos de datos con tipo, como <«tino», 1958>, <«teo», 1964> y <4, 9.8, «Sí»>. En el mismo espacio de tuplas puede existir cualquier combinación de tipos de tuplas. Los procesos comparten los datos mediante el acceso al mismo espacio de tuplas: sitúan las tuplas en el espacio utilizando la operación *escribe* y las leen o extraen utilizando las operaciones *lee* o *toma*. La operación *escribe* añade una tupla sin afectar al resto de tuplas existentes en el espacio. La operación *lee* devuelve el valor de una tupla sin afectar al contenido del espacio de tuplas. La operación *toma* también devuelve una tupla, pero la elimina del espacio.

Cuando se lee o se toma una tupla desde su espacio, un proceso proporciona una especificación de tuplas y el espacio devuelve cualquiera que cumpla dicha especificación (se trata de una variante del direccionamiento asociativo). Para permitir que los procesos sincronicen sus actividades, las operaciones *lee* y *toma* se bloquean hasta que se encuentre la tupla dentro del espacio. Una especificación de tupla está formada por el número de campos y los valores o tipos de los campos solicitados. Por ejemplo, *toma*(<String, integer>) podría extraer tanto <«tino», 1958> como <«teo», 1964>; *toma*(<String, 1958>) extraería únicamente <«tino», 1958>.

En Linda no se permite el acceso directo a las tuplas en un espacio de tuplas y los procesos las deben reemplazar en lugar de modificarlas. Sea, por ejemplo, un conjunto de procesos que accede a un contador compartido en un espacio de tuplas. El valor actual del contador (64) está en la tupla <«contador», 64>. Para incrementar el contador en un espacio de tuplas *misTups* se debe ejecutar el siguiente fragmento de código:

```
<s, contador> := misTups.take(<«contador», integer>);
misTups.write(<«contador», contador+1>);
```

El lector deberá comprobar que las condiciones de carrera no pueden ocurrir, ya que *toma* extrae la tupla contador del espacio de tuplas.

16.2.2. MODELO DE SINCRONIZACIÓN

Muchas aplicaciones aplican restricciones sobre los valores almacenados en la memoria compartida. Esto ocurre tanto en aplicaciones basadas en DSM como en las mismas aplicaciones escritas para multiprocesadores de memoria compartida (o para cualquier programa concurrente que comparte datos, como los núcleos de los sistemas operativos y los servidores multi-hilo). Por ejemplo, si a y b son dos variables almacenadas en DSM, una restricción puede ser que siempre se cumpla $a = b$. Si dos o más procesos ejecutan el siguiente fragmento de código:

$a := a + 1;$
$b := b + 1;$

entonces se puede llegar a una situación inconsistente. Supóngase que a y b se han inicializado a cero y que el proceso 1 incrementa a. Antes de que pueda incrementar b, el proceso 2 pone un 2 en a y un 1 en b. La restricción ha dejado de cumplirse. La solución consiste en convertir este fragmento de código en una sección crítica: sincronizar los procesos para asegurar que sólo uno de ellos puede estar a la vez ejecutándola.

Para poder utilizar DSM se debe construir un servicio de sincronización distribuida que incluya construcciones familiares como bloqueos y semáforos. Incluso cuando DSM se estructura como un conjunto de objetos, la implementación de estos objetos depende de la sincronización. Las construcciones de sincronización se implementan utilizando paso de mensajes (véase el Capítulo 11 para una descripción de un servidor de bloqueos distribuido). Las instrucciones máquina especiales del tipo de *testAndSet* (comprueba y escribe), utilizadas para la sincronización en multiprocesadores de memoria compartida, se pueden aplicar a sistemas DSM basados en páginas, pero su operación en el caso distribuido puede ser muy ineficiente. Las implementaciones de DSM utilizan la sincronización a nivel de aplicación para reducir la cantidad de transmisión de actualización. DSM incluye la sincronización como un componente integrado.

16.2.3. MODELO DE CONSISTENCIA

Tal y como fue descrito en el Capítulo 14, la cuestión de la consistencia adquiere importancia en los sistemas DSM que replican el contenido de la memoria compartida mediante su almacenamiento en las cachés de computadores separados. En la terminología utilizada en el Capítulo 14, cada proceso tiene un gestor de réplicas local, el cual está encargado de mantener copias en caché para los objetos. En la mayor parte de las implementaciones, los datos se leen desde las réplicas locales por cuestiones de eficiencia, pero las actualizaciones deben propagarse al resto de gestores de réplica.

El gestor de réplica local se implementa mediante una combinación del middleware (el nivel DSM en tiempo de ejecución en cada proceso) y del núcleo. Es normal que el middleware realice la mayor parte del procesamiento DSM. Incluso en las implementaciones de DSM basadas en páginas, el núcleo normalmente proporciona únicamente una correspondencia de páginas básica, el manejo de fallos de página y los mecanismos de comunicación, mientras que el middleware es responsable de implementar las políticas de compartición de páginas. Si los segmentos DSM son persistentes, entonces uno o más servidores de almacenamiento (por ejemplo, servidores de archivos) actuarán también como gestores de réplicas.

Además de la gestión de la caché, una implementación DSM puede almacenar las actualizaciones y reducir los costes de comunicación mediante la propagación de múltiples actualizaciones a la vez. Una aproximación similar se estudió sobre la arquitectura de cotilleo (*gossip*) en el Capítulo 14.

Un modelo de *consistencia de memoria* [Mosberger 1993] especifica las garantías de consistencia que un sistema DSM realiza sobre los valores que los procesos leen desde los objetos, dado que en realidad accden sobre una réplica de cada objeto y que múltiples procesos pueden actualizar los objetos. Téngase en cuenta que esto es diferente de la noción de consistencia de alto nivel y dependiente de aplicación, ya discutida en el apartado de sincronización de aplicaciones.

Cheriton [1985] describe cómo diferentes formas de DSM pueden tener previsto que un determinado nivel de inconsistencia pueda ser aceptable. Por ejemplo, DSM puede utilizarse para almacenar la carga de diferentes computadores en una cierta red para que los clientes puedan seleccionar, para ejecutar sus aplicaciones, el computador más descargado. Puesto que esta información es, por su propia naturaleza, muy inexacta en escalas de tiempo relativamente pequeñas, sería un desperdicio gastar recursos en el mantenimiento continuo de la consistencia para todos los computadores del sistema.

Sin embargo, la mayor parte de las aplicaciones tienen requisitos de consistencia muy estrictos. Es preciso proporcionar a los programadores un modelo que se ajuste razonablemente al comportamiento que la memoria debería tener. Antes de describir los requisitos de consistencia de memoria en mayor detalle será útil estudiar un ejemplo.

Sea una aplicación en la que dos procesos acceden a dos variables, a y b (véase la Figura 16.3), cuyo valor inicial es cero. El proceso 2 incrementa a y b, en ese orden. El proceso 1 lee los valores de b y a depositándolos sobre las variables locales br y ar, en ese orden. Fíjese en que no existe sincronización a nivel de aplicación. De forma intuitiva el proceso 1 espera encontrar una de las siguientes combinaciones de valores, en función del momento en el que las operaciones de lectura hayan sido aplicadas sobre a y b (implícito en las sentencias $br := b$ y $ar := a$) relativo a la ejecución del proceso 2: $ar = 0$, $br = 0$; $ar = 1$, $br = 0$; $ar = 1$, $br = 1$. En otras palabras, la condición $ar \geqslant br$ se debería cumplir siempre y el proceso 1 debería imprimir *OK*. Sin embargo, una cierta implementación DSM puede enviar las actualizaciones de a y b fuera de orden al gestor de réplicas del proceso 1, en cuyo caso la combinación $ar = 0$, $br = 1$ podría ocurrir.

La reacción inmediata del lector respecto a este ejemplo será, probablemente, considerar que la implementación DSM, que cambia el orden de las dos actualizaciones, es incorrecta. Si el proceso 1 y el proceso 2 se ejecutan juntos en un computador con un único procesador, podríamos suponer que el subsistema de memoria está averiado. Sin embargo, para el caso distribuido se puede tratar de una implementación correcta de un modelo de consistencia más débil del que muchos de nosotros suponemos de forma intuitiva, pero que en cualquier caso puede ser útil y es relativamente eficiente.

Mosberger [1993] realiza un bosquejo de un conjunto de modelos que han sido pensados para multiprocesadores de memoria compartida y sistemas DSM software. Los principales modelos de consistencia que se pueden implementar en la práctica en sistemas DSM son la consistencia secuencial y los modelos basados en consistencia débil.

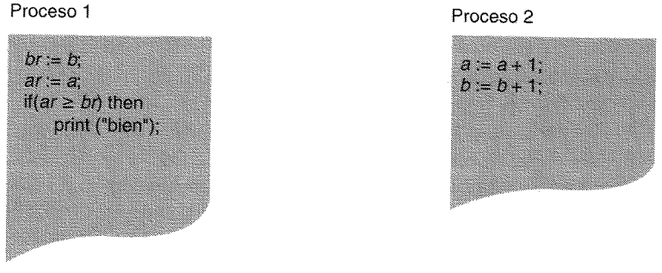


Figura 16.3. Dos procesos accediendo a variables compartidas.

La principal cuestión que se plantea al caracterizar un modelo particular de consistencia de memoria es: cuando se realiza un acceso de lectura sobre una posición de memoria, ¿qué accesos de escritura a esa posición son candidatos para que sus valores sean proporcionados a la lectura? En el extremo más débil, la respuesta es: cualquier escritura que haya sido iniciada antes que la lectura. Este modelo se podría obtener si se permitiera a los gestores de réplica retrasar la propagación de las actualizaciones a sus parejas de forma indefinida. Es demasiado débil para ser útil.

En el extremo más fuerte, todos los valores que se escriban aparecen disponibles de forma instantánea a todos los procesos: una lectura devuelve el valor escrito más reciente, respecto al instante en el que se realiza la lectura. Se trata de una definición problemática en dos sentidos. En primer lugar, ni las escrituras ni las lecturas se realizan en un instante temporal único, de forma que el significado de *más reciente* no siempre es claro. Cada tipo de acceso tiene un punto de inicio bien definido, pero se completa en un cierto instante posterior (por ejemplo, después de que se haya enviado un mensaje). En segundo lugar, en el Capítulo 10 se mostró que existen límites acerca de con qué proximidad se pueden sincronizar de relojes en un sistema distribuido. De forma que no es siempre posible determinar de forma exacta si un evento ocurrió antes que otro.

En cualquier caso, este modelo ha sido especificado y estudiado. El lector puede haberlo reconocido: es lo que llamamos secuenciación en el Capítulo 14. La secuenciación es más conocida como *consistencia atómica* en la literatura DSM. Volveremos a plantear la definición de secuenciación realizada en el Capítulo 14.

Se dice que un servicio de objetos compartidos replicados es secuenciable si *para cualquier ejecución* existe un entrelazado de las series de operaciones iniciadas por todos los clientes que satisfacen los dos criterios siguientes:

L1: La secuencia de operaciones entrelazada cumple la especificación de una (única) copia correcta de los objetos.

L2: El orden de las operaciones en el entrelazado es consistente con los instantes reales en los que las operaciones ocurrieron en la ejecución actual.

Se trata de una definición genérica que se aplica a cualquier sistema que contenga objetos replicados compartidos. Al estar tratando con memoria compartida, podemos ahora ser más explícitos. Sea el caso más simple en el que la memoria compartida se estructura como un conjunto de variables que pueden ser leídas o escritas. Todas las operaciones son de lectura y de escritura, y para ellas introdujimos una notación en la Sección 16.2.1: la lectura del valor a desde la variable x se indica como $L(x)a$; la escritura del valor b sobre la variable x se indica como $E(x)b$. Podemos ahora expresar el primer criterio, L1, en términos de variables (objetos compartidos):

L1′: La secuencia entrelazada de operaciones es tal que si $L(x)a$ ocurre en la secuencia, entonces o bien la última operación de escritura que ocurre antes que ella en la secuencia entrelazada es $E(x)a$, o bien no ocurre ninguna operación de escritura antes que ella y a es el valor inicial de x.

Este criterio se ajusta a nuestra intuición de que una variable puede ser cambiada únicamente por una operación de escritura. El segundo criterio de serialización, L2, no cambia.

◇ **Consistencia secuencial.** La secuenciación es demasiado estricta en la mayor parte de las ocasiones. El modelo de memoria más fuerte para DSM que se usa en la práctica es el de *consistencia secuencial* [Lamport 1979], explicado en el Capítulo 14. Adaptaremos a continuación la definición realizada en el Capítulo 14 para el caso particular de variables compartidas.

Un sistema DSM se dice que es secuencialmente consistente si *para cualquier ejecución* existe algún entrelazado de las series de operaciones realizadas por todos los procesos que satisface los dos siguientes criterios:

SC1: La secuencia entrelazada de operaciones es tal que si $L(x)a$ ocurre en la secuencia, entonces o bien la última operación de escritura que ocurrió antes en la secuencia entrelazada fue $E(x)a$, o bien no ha ocurrido ninguna operación de escritura antes que ella y a es el valor inicial de x.

SC2: El orden de las operaciones en el entrelazado es consistente con el orden de programa en el que dichas operaciones fueron ejecutadas por cada cliente individual.

El criterio SC1 es el mismo que el L1′. El criterio SC2 se refiere al orden de programa en lugar de referirse al orden temporal, que es el que hace posible implementar la consistencia secuencial.

La condición puede ser retomada de la siguiente forma: existe un entrelazado virtual para todas las operaciones de *lectura* y *escritura* de los procesos sobre una única imagen virtual de la memoria; en este entrelazado se mantiene el orden de programa de cada proceso individual y cada proceso siempre lee el último valor escrito dentro del entrelazado.

En una ejecución real, las operaciones de memoria pueden ser solapadas y algunas actualizaciones pueden ser ordenadas de forma diferente en diferentes procesos, siempre que las limitaciones de la definición no dejen de cumplirse. Fíjese que, para satisfacer las condiciones de la consistencia secuencial, se deben tener en cuenta las operaciones de memoria sobre el sistema DSM completo y no únicamente las operaciones desde cada posición individual.

La combinación $ar = 0$, $br = 1$ en el ejemplo anterior puede no ocurrir bajo la consistencia secuencial, ya que el proceso 1 estaría leyendo valores que entrarían en conflicto con el orden de programa del proceso 2. En la Figura 16.4 se muestra un ejemplo de entrelazado de los accesos a memoria de un proceso en una ejecución con consistencia secuencial. Una vez más, mientras que aquí se muestra un entrelazado real de las operaciones de lectura y escritura, la definición únicamente estipula que la ejecución debe producirse *como si* se realizara dicho entrelazado estricto.

Los sistemas DSM con consistencia secuencial se pueden implementar utilizando un único servidor para gestionar todos los datos compartidos y haciendo que todos los procesos envíen las solicitudes de lectura y escritura al servidor, que las ordena de forma global. Esta arquitectura es demasiado ineficiente para una implementación DSM por lo que a continuación se describen algunos medios prácticos para conseguir la consistencia secuencial. En cualquier caso, se trata de un modelo cuya implementación es muy costosa.

◇ **Coherencia.** Una reacción al coste de la consistencia secuencial consiste en buscar un modelo más débil pero con sus propiedades bien definidas. La *coherencia* es un ejemplo de una forma más débil de consistencia. Bajo la coherencia, cada proceso llega a acuerdos sobre el orden de las operaciones de *escritura* sobre la misma posición, pero no acuerdan necesariamente el orden de las operaciones de *escritura* sobre posiciones diferentes. Se puede pensar en la coherencia como una forma de consistencia secuencial realizada posición a posición. Los sistemas DSM coherentes pueden implementarse mediante un protocolo para implementar la consistencia secuencial aplicado de forma separada a cada unidad de datos replicados (por ejemplo, a cada página). El ahorro se

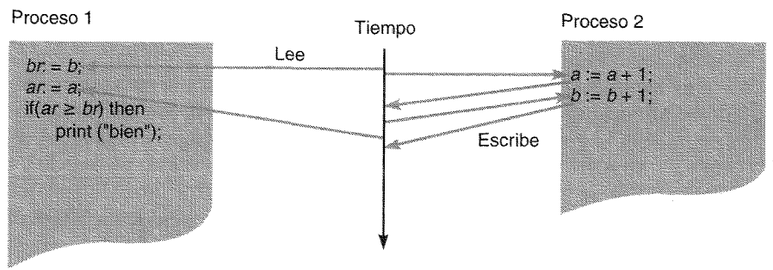


Figura 16.4. Entrelazado bajo la consistencia secuencial.

produce del hecho de que los accesos a dos páginas diferentes se hacen de forma independiente y no existe retardo entre ellos al aplicarse el protocolo de forma separada a ambos.

◇ **Consistencia débil.** Dubois y otros [1988] desarrollaron un modelo de consistencia débil en un intento de evitar los costes de la consistencia secuencial en los multiprocesadores, pero conservando el *efecto* de esta consistencia secuencial. Este modelo aprovecha el conocimiento de las operaciones de sincronización para relajar la consistencia de memoria, mientras se muestra al programador para implementar una consistencia secuencial (al menos, bajo ciertas condiciones que superan los objetivos de este libro). Por ejemplo, si el programador utiliza un bloqueo para implementar una sección crítica, entonces un sistema DSM puede asumir que ningún otro proceso puede acceder a los datos accedidos bajo la exclusión mutua. Será por lo tanto redundante que un sistema DSM propague las actualizaciones de dichos datos antes que el proceso deje la sección crítica. A pesar de que se mantienen valores *inconsistentes* durante algún tiempo para los datos, durante ese tiempo dichos datos no son utilizados; la ejecución aparenta ser secuencialmente consistente. Adve y Hill [1990] describen una generalización de esta idea llamada ordenamiento débil: «(Un sistema DSM) está ordenado débilmente respecto a un modelo de sincronización si y sólo si aparenta ser secuencialmente consistente para el software que aplica el modelo de sincronización.» En la Sección 16.4 se describe la consistencia relajada (*release*), la cual es un desarrollo de la consistencia débil.

16.2.4. OPCIONES DE ACTUALIZACIÓN

Para la propagación de las actualizaciones realizadas desde un cierto proceso sobre varios otros se han desarrollado dos posibles opciones: escritura actualizante y escritura invalidante. Ambos se pueden aplicar a varios modelos de consistencia DSM, incluyendo la consistencia secuencial. En detalle las opciones se describen así:

Escritura actualizante: las actualizaciones de un proceso se realizan de forma local y se envían por multidifusión a todos los gestores de réplica que posean una copia del dato, los cuales modifican inmediatamente el dato leído por los procesos locales (véase la Figura 16.5). Los procesos leen las copias locales de los datos, sin necesidad de comunicación. Además de permitir múltiples lectores, varios procesos pueden escribir el mismo dato de forma simultánea; esto se conoce como *compartición de múltiples lectores/múltiples escritores*. El modelo de consistencia de memoria implementado en la escritura actualizante depende de varios factores, siendo el principal la propiedad de ordenamiento de la multidifusión. Se puede conseguir la consistencia secuencial mediante la utilización de multidifusiones totalmente ordenadas (véase el Capítulo 11 para una definición de los multidifusiones totalmente ordenadas), que no terminan hasta que el mensaje de actualización haya sido entregado localmente. Se consigue así que todos los procesos acuerden el orden de las actualizaciones. El conjunto de lecturas que se realizan entre dos actualizaciones consecutivas está bien definido y su ordenamiento es independiente de la consistencia secuencial.

Las lecturas son baratas en los sistemas de escritura actualizante. Sin embargo, en el Capítulo 11 se mostró que los protocolos de multidifusión ordenada son relativamente caros de implementar en software. Orca utiliza escritura actualizante a través del protocolo de multidifusión de Amoeba [Kaashoek y Tanenbaum 1991] (véase www.cdk3.net/coordination), que utiliza el soporte hardware para implementar la multidifusión. Munin soporta la escritura actualizante como una opción. En la arquitectura multiprocesador PLUS se utiliza un protocolo de escritura actualizante soportado mediante hardware especializado.

Escritura invalidante: está implementada normalmente en la forma de compartición de múltiples lectores/un único escritor. En cualquier instante, una cierta posición puede ser accedida

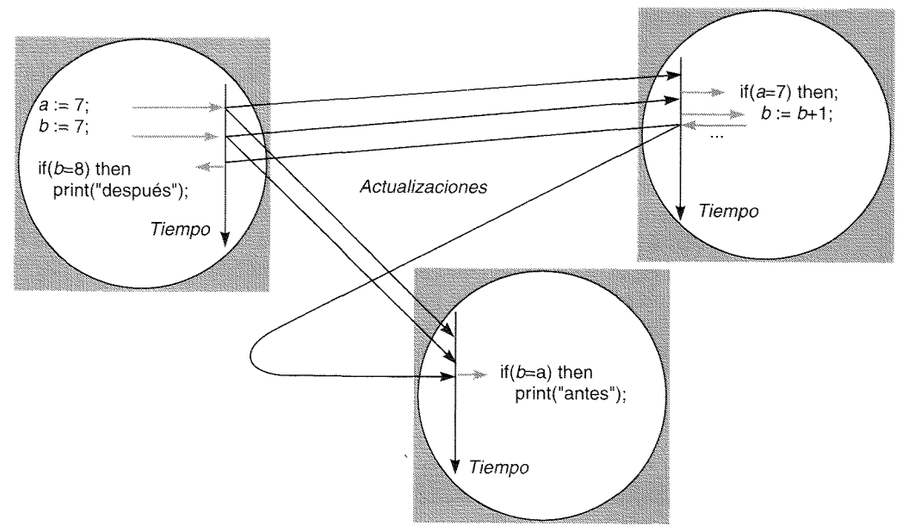


Figura 16.5. DSM utilizando escritura actualizante.

para lectura por uno o más procesos, o bien puede ser leída y escrita por un único proceso. Una posición que está siendo utilizada en modo de sólo lectura puede ser copiada a cualquier otro proceso. Cuando un proceso intenta escribir sobre ella, en primer lugar se envía un mensaje multidifusión a todas sus copias para invalidarlas y se espera el reconocimiento de esta operación antes de que la escritura tenga lugar; el resto de los procesos están por lo tanto prevenidos para no leer datos viejos (es decir, datos que no están actualizados). Cualquier proceso que esté intentando acceder al dato se bloquea hasta que la escritura termine. Finalmente, el control se transfiere desde el proceso escritor, y se puedan realizar otros accesos una vez que la actualización haya sido enviada. El resultado es que se procesan todos los accesos al dato mediante una política del tipo primero en llegar, primero en ser servido. Lamport [1979] demostró que, con este esquema, se consigue consistencia secuencial. Veremos en la Sección 16.4 que en la consistencia relajada, las invalidaciones pueden ser retrasadas.

En el esquema de invalidación, las actualizaciones únicamente se propagan cuando los datos son leídos y además se pueden realizar varias actualizaciones consecutivas sin necesidad de realizar ninguna comunicación. Como contrapartida, se deben invalidar todas las copias de sólo lectura antes de que se pueda realizar una escritura. Esto puede ser potencialmente caro en el esquema de múltiples lectores/un único escritor. Sin embargo, si la relación lectura/escritura es suficientemente alta, el paralelismo que se obtiene permitiendo múltiples lectores simultáneos amortiza su coste. Cuando la relación lecturas/escrituras es relativamente baja, un esquema de tipo único lector/único escritor puede ser más apropiado: es decir, como máximo un proceso en cada instante puede obtener acceso de sólo lectura.

16.2.5. GRANULARIDAD

Una cuestión relacionada con la estructura de DSM es la granularidad de la compartición. Teóricamente, todos los procesos comparten el contenido completo del DSM. Sin embargo, según se van

ejecutando los programas que comparten DSM, únicamente ciertas partes de los datos son en realidad compartidas y únicamente durante ciertos períodos de la ejecución. Para una implementación DSM sería un derroche transmitir el contenido completo de DSM cuando los procesos acceden y actualizan los datos compartidos. ¿Cuál debería ser la unidad de compartición en una implementación DSM? Es decir, cuando un proceso ha escrito sobre DSM, ¿qué datos debe enviar el sistema DSM para conseguir que en todos los nodos los valores sean consistentes?

En este punto nos centraremos en las implementaciones basadas en páginas, a pesar de que la cuestión de la granularidad aparece también en otro tipo de implementaciones (véase el Ejercicio 16.11). En un sistema DSM basado en páginas, el hardware soporta de forma eficiente las alteraciones sobre un espacio de direcciones siempre que la unidad básica sea la página; esto se consigue mediante la inserción de un nuevo puntero de marco de página en la tabla de páginas (véase, por ejemplo, Bacon [1998] para una descripción de la paginación). Los tamaños de las páginas pueden llegar a ser de 8 kilobytes, cantidad apreciable de datos que debe ser transmitida sobre una red para mantener consistentes las copias remotas cuando se genera una actualización. Por defecto, el precio de la transferencia completa de la página debe ser pagado tanto si se ha modificado la página de forma completa como si únicamente se ha variado un bit.

La utilización de un tamaño de página menor (entre 512 bytes y 1 kilobyte) no implica necesariamente una mejora global de las prestaciones. Primero, en aquellos casos en los que los procesos actualizan grandes cantidades de datos contiguos, es mejor enviar una página grande que varias páginas pequeñas a través de actualizaciones separadas, ya que las sobrecargas fijas debidos al software se generan por paquete de red. Segundo, la utilización de una página pequeña como unidad de distribución supone un mayor número de unidades que deben ser administradas separadamente por la implementación DSM.

Para complicar aún más el problema, los procesos tienden a competir más por las páginas cuando el tamaño de éstas es mayor, debido a que la probabilidad de que los datos utilizados estén en la misma página se incrementa con su tamaño. Considere, por ejemplo, dos procesos, uno que accede únicamente a la posición *A* mientras que el otro accede sólo a la posición *B*, asociada a la misma página (véase la Figura 16.6). Supongamos, para ser más concretos, que un proceso lee *A* y el otro actualiza *B*. A nivel de aplicación no existe conflicto en el acceso a los datos. Sin embargo, la página debe ser transmitida completamente entre los procesos, ya que el sistema DSM no sabe por defecto, en tiempo de ejecución, qué posiciones han sido alteradas en la página. A esta situación se la conoce como *compartición falsa*: dos o más procesos comparten zonas de una página, pero de hecho sólo uno de ellos accede a cada zona. En los protocolos de escritura invalidante, la compartición falsa puede derivar en invalidaciones innecesarias. En los protocolos de escritura actualizante, cuando varios escritores comparten datos de forma falsa puede ocurrir que las páginas sean sobreescritas con versiones antiguas.

En la práctica, la elección de la unidad de compartición debe realizarse en función de los tamaños de página física disponibles, a pesar de que, si el tamaño de la página es pequeño, se pueda utilizar como unidad un cierto número de páginas contiguas. La disposición de los datos respecto a las fronteras entre páginas es un factor importante para determinar el número de transferencias de páginas realizadas durante la ejecución de un cierto programa.

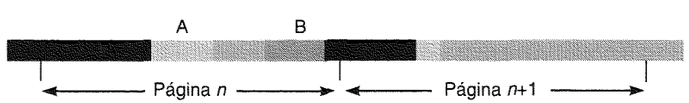


Figura 16.6. Disposición de los datos sobre las páginas.

16.2.6. *THRASHING* (FUSTIGAMIENTO)

Un problema potencial de los protocolos de escritura invalidante es el thrashing. Se dice que un sistema DSM está en thrashing cuando realiza un gasto desmesurado de tiempo en la invalidación y transferencia de datos compartidos en comparación con el tiempo empleado por los procesos de aplicación en la realización de trabajo útil. Ocurre cuando varios procesos compiten por el mismo dato o bien por datos que están bajo compartición falsa. Si, por ejemplo, un proceso lee de forma repetida el mismo dato que otro proceso está actualizando regularmente, entonces el dato será continuamente transferido desde el escritor e invalidado en el lector. Éste es un ejemplo de patrón de compartición para el que la escritura invalidante es menos apropiada que la escritura actualizante. En la siguiente sección se describe la aproximación Mirage al thrashing, en la que los computadores *poseen* las páginas durante un período de tiempo mínimo; en la Sección 16.4 se describió cómo Munin permite al programador declarar patrones de acceso al sistema DSM de forma que se pueden elegir opciones de actualización que se adapten al patrón de compartición de cada dato y se evite así el thrashing.

16.3. CONSISTENCIA SECUENCIAL E IVY

En esta sección se describen métodos para implementar sistemas DSM basados en páginas con consistencia secuencial. La descripción se basa en Ivy [Li y Hudak 1989] como caso de estudio.

16.3.1. EL MODELO DE SISTEMA

El modelo básico a considerar es aquel en el que una colección de procesos comparten un segmento de DSM (véase la Figura 16.7). El segmento se hace corresponder sobre el mismo rango de direcciones en cada proceso, de forma que sobre él se pueden almacenar valores de punteros significativos. Los procesos se ejecutan en computadores equipados con unidades de gestión de memoria paginada. Asumiremos que únicamente hay un proceso que accede al segmento DSM en cada computador. En realidad puede haber varios procesos de ese tipo en cada computador. Sin embargo, esos procesos podrían entonces compartir las páginas directamente (el mismo marco de página puede aparecer en las tablas de páginas de diferentes procesos). La única complicación sería la de coordinar el envío y la propagación de actualizaciones a una página cuando dos o más procesos locales acceden a ella. En esta descripción se ignoran este tipo de detalles.

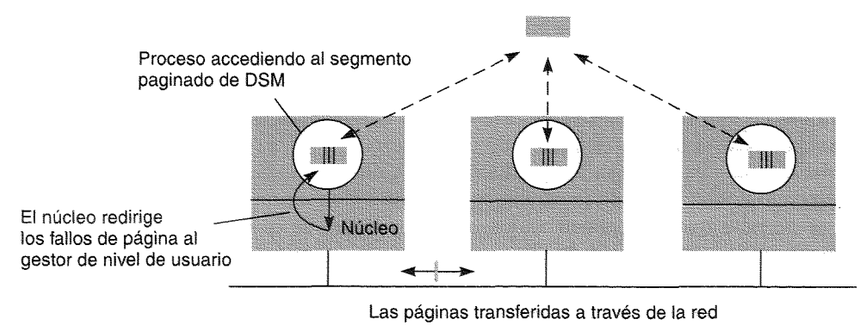


Figura 16.7. Modelo del sistema de un DSM basado en páginas.

La paginación es transparente a los componentes de la aplicación dentro de los procesos; ellos pueden leer y escribir de forma lógica cualquier posición sobre DSM. Sin embargo, para mantener la consistencia secuencial, el soporte en tiempo de ejecución de DSM restringe los permisos de acceso a páginas cuando se procesan lecturas y escrituras. La unidad de gestión de memoria paginada permite configurar los permisos de acceso a los datos de una página a *ninguno*, *sólo lectura* o *lectura escritura*. Si un proceso trata de saltarse los permisos de acceso actuales, se dispara un fallo de página en lectura o escritura, en función del tipo de acceso. El núcleo redirige el fallo de página a un manejador especificado por el nivel de soporte en tiempo de ejecución de DSM para cada proceso. El manejador de fallos de página (que se ejecuta de forma transparente a la aplicación) realiza un procesamiento especial del mencionado fallo de página, según se describe a continuación, antes de devolver el control a la aplicación. En sistemas DSM primitivos, como Ivy, el propio núcleo realiza gran parte del procesamiento. Explicaremos cómo los propios procesos realizan la gestión de los fallos de página y la comunicación consiguiente. En la actualidad, estas funciones son realizadas por una combinación del nivel en tiempo de ejecución de DSM en el proceso y del núcleo. Normalmente, el módulo de DSM insertado en el proceso contiene la mayor parte de la funcionalidad de forma que pueda ser modificada y ajustada sin tener que afrontar los problemas asociados con la alteración del núcleo.

En esta descripción no se tendrá en cuenta el procesamiento de fallo de página que debe realizarse como parte de una implementación típica de memoria virtual. Aparte del hecho de que los segmentos DSM compiten con otros segmentos por los marcos de páginas, las implementaciones son independientes.

◇ **El problema de la escritura actualizante.** En la sección anterior se perfilaron las principales alternativas de implementación de la escritura actualizante y de la escritura invalidante. En la práctica, si el sistema DSM está basado en páginas, la escritura actualizante sólo se usa si las escrituras pueden ser almacenadas en búferes. Esto se debe a que la gestión estándar de fallos de página no se adapta bien al procesamiento aislado de cada escritura a una página.

Para entender esto, supongamos que cada actualización deba ser enviada vía multidifusión a todas las copias y supongamos también que una página ha sido protegida contra escritura. Cuando un proceso intenta escribir en la página, se genera un fallo de página y se invoca a una rutina de gestión. Este gestor debería, inicialmente, examinar la instrucción que provocó el fallo para determinar el valor y la dirección que estaba siendo escrita y a continuación realizar una multidifusión de la actualización antes de restablecer el acceso de la escritura y devolver el control a la instrucción que generó el fallo.

Sin embargo, ahora que el acceso de escritura ha sido restablecido, las siguientes actualizaciones no generarán un fallo de página. Para provocar que cada acceso de escritura genere un fallo de página, será necesario que el manejador de fallos de página configure el proceso en modo TRAZA, de forma que en el procesador se activará una excepción de éste tipo después de cada instrucción. El manejador de la excepción TRAZA debería desactivar los permisos de escritura de la página y desactivar el modo TRAZA. Todos los pasos anteriores deberán repetirse cuando ocurra el siguiente fallo en escritura. Claramente este método tiende a ser muy caro ya que podrían ocurrir muchas excepciones durante la ejecución de un proceso.

En la práctica, la escritura actualizante se utiliza en las implementaciones basadas en páginas, pero sólo donde se deja a la página con permisos de escritura después de un fallo de páginas inicial y se permite que se realicen varias escrituras antes de que la página actualizada sea propagada. Munin utiliza esta técnica de almacenamiento de escrituras. Para mejorar la eficiencia, Munin intenta evitar la propagación de la página completa, propagando únicamente la parte que haya sido modificada. Cuando un proceso realiza un primer intento de escritura sobre una página, Munin gestiona el fallo de páginas realizando una copia de la página y guardando la copia antes de habilitar el acceso de escritura. Posteriormente, cuando Munin está dispuesto para propagar la página, compara la página actualizada con la copia original y codifica las actualizaciones en forma de conjunto de diferencias entre las dos páginas. Estas diferencias a menudo ocupan mucho menos espacio que la página completa. Los procesos receptores regeneran la página actualizada partiendo de la copia antes de actualizar y del conjunto de diferencias.

16.3.2. INVALIDACIÓN DE ESCRITURA

Los algoritmos basados en invalidaciones utilizan la protección de páginas para forzar la consistencia en la compartición de datos. Cuando un proceso está actualizando una página, tiene localmente los permisos de lectura y escritura sobre dicha página; el resto de procesos no tienen permisos de acceso sobre la página. Cuando uno o más procesos están leyendo una página, sólo tienen permiso de lectura; el resto de procesos no tienen permiso de acceso (a pesar de que puedan adquirir los permisos de lectura). No son posibles otras combinaciones. Un proceso con la versión actualizada de una página p es marcado como su *propietario* y esto se indica mediante la notación *propietario*(p). Se trata del único escritor o bien de uno de los lectores. Al conjunto de procesos que tienen una copia de una cierta página p se le denomina *conjunto de copia* y se utiliza la notación *conjuntocopia*(p) para referirse a ellos.

Las transiciones de estado posibles se muestran en la Figura 16.8. Cuando un proceso P_w intenta escribir sobre una página p sobre la que o no tiene acceso o sólo tiene acceso de lectura, se genera un fallo de página. El manejador de fallos de página realiza los siguientes pasos:

- Si el procesador P_w no tienen una copia actualizada de la página, se le transfiere.
- Se invalidan el resto de copias: los permisos de acceso a las páginas se modifican para impedir el acceso de los miembros de *conjuntocopia*(p).
- *conjuntocopia*(p) $:= \{P_w\}$.
- *propietario*(p) $:= P_w$.
- El soporte en tiempo de ejecución de P_w sitúa la página con permisos de lectura y escritura en la posición correspondiente en su espacio de direcciones y reinicia la instrucción que generó el fallo.

Nótese que dos o más procesos con copias de sólo lectura pueden generar fallos de escritura de forma más o menos simultánea. Una copia de sólo lectura de una página puede quedar caducada cuando la propiedad sea finalmente cedida. Para detectar si la copia actual de sólo lectura de una página está o no caducada, se asocia un número de secuencia a cada página, el cual se incrementa cada vez que la propiedad es transferida. Un proceso que solicita acceso de escritura inserta en la solicitud el número de secuencia de su copia de sólo lectura, si posee uno. El propietario actual

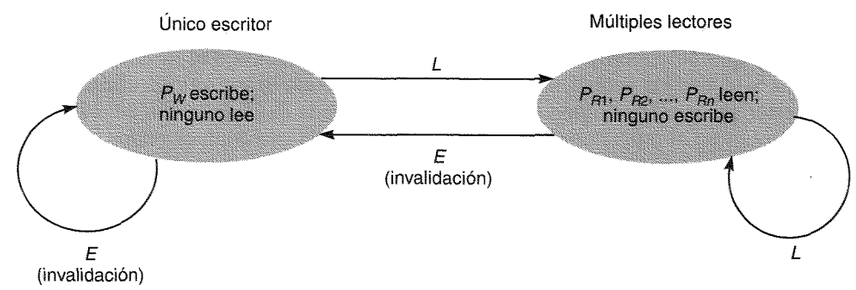


Figura 16.8. Transiciones de estado bajo invalidación de escritura.

utiliza este número para decidir si la página ha sido modificada y por lo tanto es necesario enviarla. Este esquema es descrito por Kessler y Livny [1989] bajo el nombre de *algoritmo astuto*.

Cuando un proceso P_R intenta leer una página p sobre la que no tiene permisos de acceso, se genera un fallo de página en lectura. El manejador de fallos realiza los siguientes pasos:

- La página es copiada desde *propietario*(p) hasta P_R.
- Si el propietario actual es un único escritor, entonces éste mantiene la propiedad de p y se colocan sus permisos de acceso en sólo lectura. La retención de los derechos de acceso en lectura es deseable en el caso que el proceso intente leer la página posteriormente, se habrá retenido una versión actualizada de la página. Sin embargo, al ser el propietario deberá gestionar las solicitudes que se reciban sobre la página, incluso si no se vuelve a utilizar de nuevo. Por lo tanto se puede considerar que en lugar de reducir los derechos de acceso sería más apropiado simplemente prohibirlos y transferir la propiedad a P_R.
- *conjuntocopia*(p) := *conjuntocopia*(p) $\cup \{P_R\}$.
- El nivel de soporte de sistema DSM en P_R sitúa la página con permisos de sólo lectura en la posición adecuada en su espacio de direcciones y reinicia la instrucción que generó el fallo.

Es posible que se genere un segundo fallo de página durante los algoritmos de transición que se acaban de describir. Para que las transiciones se realicen de forma consistente, cualquier nueva solicitud sobre la página no será procesada hasta que la transición actual se haya completado.

La descripción que se acaba de proporcionar se ha centrado en la explicación de *qué* debe hacerse. A continuación se estudia el problema de *cómo* implementar de forma eficiente el gestor de fallos de página.

16.3.3. PROTOCOLOS DE INVALIDACIÓN

Quedan por resolver dos problemas importantes en un protocolo que implemente el esquema de invalidación:

1. Cómo localizar el *propietario*(p) *para una cierta página p*.
2. Dónde almacenar *conjuntocopia*(p).

Li y Hudak [1989] describen varias arquitecturas y protocolos para Ivy, que realizan diferentes aproximaciones sobre los problemas mencionados. La más simple que describiremos es el algoritmo mejorado de gestión centralizada. En esta solución, se utiliza un único servidor, llamado gestor, para almacenar la dirección (dirección de nivel de transporte) del *propietario*(p) para cada página p. El gestor podría ser uno de los procesos que ejecutan la aplicación o bien cualquier otro proceso. En este algoritmo el conjunto *conjuntocopia*(p) se almacena en *propietario*(p). Es decir, se almacenan los identificadores y las direcciones de transporte de los miembros de *conjuntocopia*(p).

Tal y como se puede observar en la Figura 16.9, cuando ocurre un fallo de página el proceso local (al que nos referiremos como *cliente*) envía un mensaje al gestor conteniendo el número de página y el tipo de acceso solicitado (lectura o lectura-escritura). El cliente espera una respuesta. El gestor procesa la solicitud realizando una búsqueda de la dirección de *propietario*(p) y reenviando la solicitud al propietario. Si se trata de un fallo de escritura, el gestor configura al cliente como nuevo propietario. Las siguientes solicitudes se insertan en una cola en el cliente a la espera de que éste complete la transferencia de la propiedad sobre él mismo.

El anterior propietario envía la página al cliente. En el caso de un fallo en escritura, también envía el conjunto de copia de la página. El cliente realiza la invalidación cuando recibe el conjunto de copia. Envía una solicitud multidifusión a los miembros del conjunto de copia, y espera el reconocimiento desde todos los procesos para considerar que la invalidación se ha completado. La multidifusión no necesita ser ordenada. El anterior propietario no necesita ser incluido en la lista

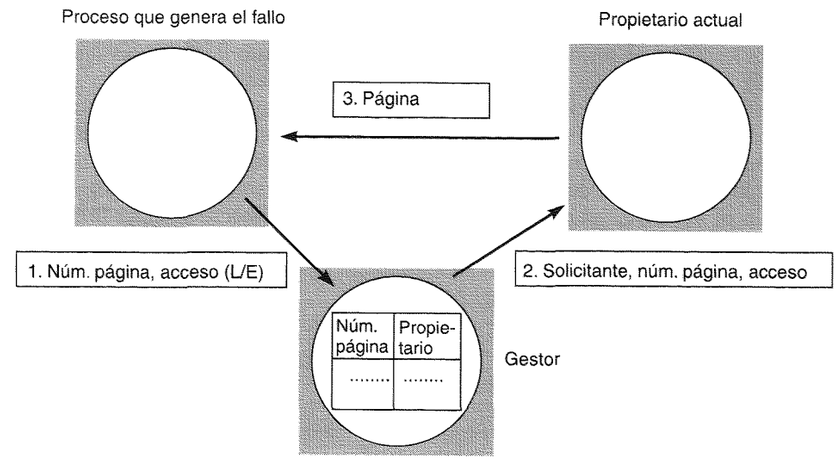


Figura 16.9. Gestor central y mensajes asociados.

de destinatarios, al haberse invalidado a sí mismo. Los detalles de la gestión del conjunto de copia se dejan al lector, que deberá consultar los algoritmos de invalidación generales ya explicados.

El gestor es un cuello de botella para las prestaciones y un punto de fallo crítico. Li y Hudak sugirieron tres alternativas para que la carga de la gestión de páginas fuera dividida entre computadores: gestión de páginas distribuida fija, gestión distribuida basada en multidifusión y gestión distribuida dinámica. En la primera alternativa, se utilizan múltiples gestores, cada uno con funcionalidad equivalente al gestor centralizado ya descrito, pero dividiendo de forma estática las páginas entre ellos. Por ejemplo, cada gestor podría manejar sólo aquellas páginas cuyos números de página estén dentro de un cierto rango de valores. Los clientes calculan el número hash de la página que se necesita y utilizan una tabla de configuración predeterminada para buscar la dirección del gestor correspondiente.

Este esquema debería mejorar en general el problema de la carga, pero tiene la desventaja de que una correspondencia fija de las páginas sobre los gestores puede no ser factible. Si los procesos no acceden a las páginas de forma uniforme, algunos gestores estarán más cargados que otros. Describimos a continuación la gestión basada en multidifusión y la gestión distribuida dinámica.

◇ **Uso de la multidifusión para localizar el propietario.** La multidifusión puede utilizarse para eliminar completamente al gestor. Cuando se genera un fallo en un proceso, éste envía una multidifusión con su solicitud de página al resto de los procesos. Sólo contesta el proceso que posea la página. Se debe tener cuidado para garantizar el comportamiento correcto si los clientes solicitan la misma página de forma simultánea: cada cliente debe terminar obteniendo la página, incluso si su multidifusión se ha realizado durante la transferencia de la propiedad.

Considérense dos clientes C_1 y C_2 que utilizan multidifusión para localizar una página cuyo propietario es O. Supongamos que O recibe la solicitud de C_1 en primer lugar y le transfiere la propiedad. Antes de que la página llegue, la solicitud de C_2 llega a O y a C_1. O descartará la solicitud de C_2 ya que ha dejado de poseer la página. Li y Hudak consideraron que C_1 debería retrasar el procesamiento de la solicitud de C_2 hasta que haya obtenido la página ya que en otro caso debería descartar dicha solicitud al no ser el propietario y la solicitud de C_2 se perdería. Sin embargo, queda todavía un problema sin resolver. La solicitud de C_1 ha sido almacenada mientras tanto en una cola en C_2. Después de que C_1 haya finalmente proporcionado a C_2 la página, C_2 recibirá y procesará la solicitud de C_1, ¡la cual ahora es obsoleta!

Una solución consiste en utilizar multidifusión totalmente ordenada, de forma que los clientes puedan descartar de forma segura aquellas solicitudes que llegan antes que la suya (además de al resto de los procesos las solicitudes son enviadas a ellos mismos). Otra solución, basada en una multidifusión sin orden más barata, pero que utiliza mayor ancho de banda, consiste en asociar a cada página un vector de marcas temporales, con una entrada por cada proceso (véase el Capítulo 10 para una descripción de los vectores de marcas temporales). La marca de tiempo se transfiere junto con la propiedad de una página. Cuando un proceso obtiene la propiedad, incrementa su entrada en el vector de marcas temporales. Cuando un proceso solicita la propiedad, inserta en la solicitud la última marca temporal que mantiene para esa página. En nuestro ejemplo, C_2 debería descartar la solicitud de C_1 ya que la entrada de C_1 en la marca temporal de la solicitud es menor que la marca que llegó con la página.

Independientemente de que se utilice una multidifusión ordenada o desordenada, este esquema tiene las desventajas típicas de los sistema basados en multidifusión: los procesos que no poseen una cierta página son interrumpidos con mensajes irrelevantes, desperdiciando tiempo de procesamiento.

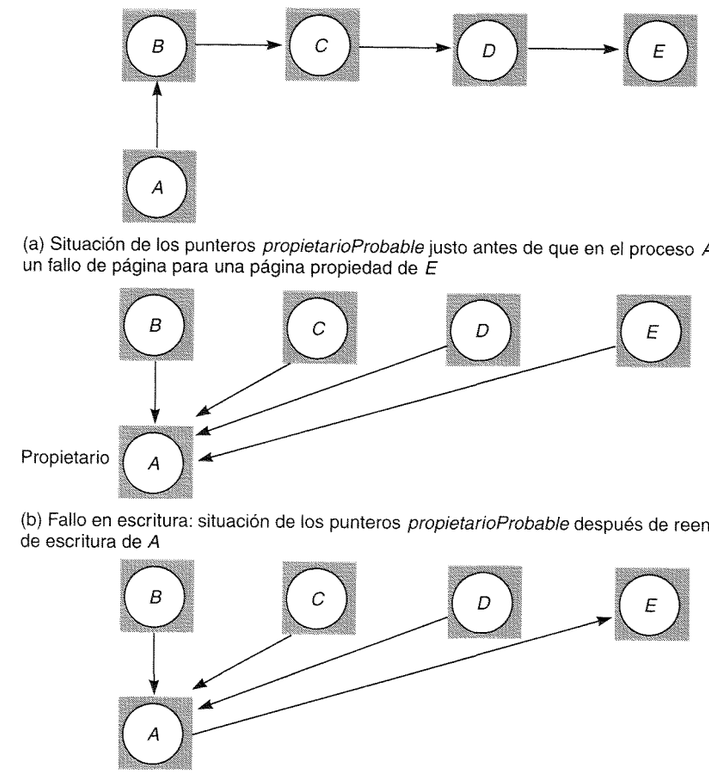
16.3.4. UN ALGORITMO DE GESTIÓN DISTRIBUIDA DINÁMICO

Li y Hudak propusieron el algoritmo de gestión distribuida dinámico, que permite que la propiedad de una página se transfiera entre procesos pero utiliza un método alternativo a la multidifusión para localizar al propietario de una página. La idea se basa en la división de las sobrecargas de localización de páginas entre aquellos computadores que acceden a ellas. Para cada página p, cada proceso mantiene una marca del propietario actual de la página, que en realidad es el propietario probable de p o *propietarioProbable*(p). Inicialmente a cada proceso se le proporcionan las localizaciones exactas para las páginas. Sin embargo, estos valores son meros *indicios*, ya que las páginas se transfieren entre los nodos en cualquier momento. Como ocurre en los algoritmos anteriores, la propiedad se transfiere sólo cuando ocurre un fallo en escritura.

El propietario de una página es localizado siguiendo una cadena de marcas que son actualizadas al nuevo propietario cuando la página es transferida entre computadores. La longitud de la cadena, es decir, el número de mensajes de reenvío que son necesarios para localizar al propietario, amenaza con crecer de forma indefinida. El algoritmo resuelve este problema mediante la actualización inmediata de las marcas según vayan estando disponibles los valores más actualizados. El procedimiento para actualizar las marcas y reenviar las solicitudes es el siguiente:

- Cuando un proceso transfiere la propiedad de una página p a otro proceso, marca como nuevo *propietarioProbable*(p) al receptor de la página.
- Cuando un proceso gestiona una solicitud de invalidación para una página p, marca como nuevo *propietarioProbable*(p) al solicitante.
- Cuando un proceso que ha solicitado acceso de lectura a una página p la recibe, marca como nuevo *propietarioProbable*(p) al nodo que se la proporcionó.
- Cuando un proceso recibe una solicitud para una página p de la que no es propietario, reenvía la solicitud al *propietarioProbable*(p) y marca como nuevo *propietarioProbable*(p) al solicitante.

Las primeras tres actualizaciones son una consecuencia del protocolo en lo que se refiere a la transferencia de la propiedad de la página y la obtención de copias de sólo lectura. La actualización cuando se reenvía una solicitud se justifica en que, para las solicitudes de escritura, el solicitante será pronto el propietario, incluso si no lo es actualmente. De hecho, en el algoritmo de Li y Hudak, la actualización de *propietarioProbable* se realiza tanto si la solicitud es para acceso de lectura como si lo es para escritura. Volveremos sobre este punto pronto.



(a) Situación de los punteros *propietarioProbable* justo antes de que en el proceso *A* se produzca un fallo de página para una página propiedad de *E*

(b) Fallo en escritura: situación de los punteros *propietarioProbable* después de reenviar la solicitud de escritura de *A*

(c) Fallo en lectura: situación de los punteros *propietarioProbable* después de reenviar la solicitud de lectura de *A*

Figura 16.10. Actualizaciones de los punteros *propietarioProbable*.

En la Figura 16.10 ((a) y (b)) se ilustran los punteros de *propietarioProbable* antes y después de que en el proceso *A* se produzca un fallo de página en escritura. El puntero *propietarioProbable* de *A* para la página apunta inicialmente a *B*. Los procesos *B*, *C* y *D* reenvían la solicitud a *E* siguiendo sus propios punteros *propietarioProbable*; después de eso, todos los punteros apuntan a *A* como resultado de las reglas de actualización descritas. El resultado una vez gestionado el fallo es claramente mejor que antes de hacerlo: la cadena de punteros ha desaparecido.

Sin embargo, si en *A* se produce un fallo de lectura, entonces el proceso *B* se comporta mejor (dos pasos en lugar de tres para apuntar a *E*), la situación de *C* no cambia (dos pasos) pero para *D* la situación empeora ya que necesita dos pasos en lugar de uno (véase la Figura 16.10(c)). Para investigar el efecto en las prestaciones de esta táctica es necesario realizar simulaciones.

El tamaño medio de las cadenas de punteros puede ser controlado además difundiendo periódicamente a todos los procesos la localización del propietario actual. Esto provoca que todas las cadenas pasen a tener tamaño 1.

Li y Hudak describen los resultados de la simulaciones que realizaron para investigar la eficacia de sus actualizaciones de punteros. Eligieron de forma aleatoria los procesos que generaban fallos de entre 1.024 procesadores y encontraron que el número medio de mensajes necesarios para

conseguir el propietario de una página fue de 2,34 si las difusiones de anuncio de la posición del propietario se realizaban cada 256 fallos y de 3,64 si las difusiones se realizaban cada 1.024 fallos. Estos valores se muestran aquí únicamente a modo de ilustración: en Li y Hudak [1989] se muestra un conjunto completo de resultados. Nótese que un sistema DSM que utiliza un gestor central necesita dos mensajes para conseguir el propietario de una página.

Finalmente, Li y Hudak describen una optimización que potencialmente consigue realizar la invalidación de forma más eficiente y reduce el número de mensajes necesarios para manejar un fallo de página en lectura. En lugar de tener que obtener una copia de la página desde su nodo propietario, un cliente puede obtener una copia de cualquier proceso con una copia válida. Existe la posibilidad de que un cliente que está intentando localizar el propietario en la cadena de punteros encuentre antes un nodo con una copia válida.

Esto se realiza bajo la condición de que los procesos mantengan un registro de clientes a los que han cedido una copia de una página de su propiedad. El conjunto de procesos que poseen copias de sólo lectura de una página forma un árbol cuya raíz es el propietario, con cada nodo apuntando a los nodos hijo, los cuales obtuvieron copias de él. La invalidación de una página comienza en el propietario y avanza hacia abajo a través del árbol. Cuando un nodo recibe un mensaje de invalidación, lo reenvía a sus hijos además de invalidar su propia copia. El efecto en su conjunto es el de que algunas invalidaciones se realizan en paralelo. Esto puede reducir el tiempo global necesario para invalidar una página, especialmente en un entorno sin soporte hardware para la multidifusión.

16.3.5. THRASHING

Se puede argumentar que es responsabilidad del programador evitar el thrashing. El programador debería anotar las posiciones de sus datos para ayudar al soporte DSM a minimizar tanto el número de veces que se copian las páginas como el número de transferencias de propiedad. Esta última aproximación es discutida en la siguiente sección en el contexto del sistema DSM Munin.

Mirage [Fleisch y Popek 1989] adopta una aproximación al thrashing transparente a los programadores. Mirage asocia cada página con un pequeño intervalo de tiempo. Una vez que un proceso ha accedido a una página, se le permite retener el acceso para el intervalo dado, que puede ser considerado como un intervalo de tiempo. Cualquier otra solicitud realizada sobre la página durante dicho intervalo de tiempo es rechazada. Una desventaja obvia de este esquema es que resulta muy difícil elegir la longitud del intervalo de tiempo. Si el sistema utiliza una longitud de tiempo elegida de forma estática, ese valor puede ser inapropiado en muchos casos. Por ejemplo un proceso podría escribir una página una única vez y no volver a acceder a ella; el resto de procesos tendrían entre tanto el acceso prohibido a dicha página. Análogamente, el sistema podría ceder a otro proceso el acceso a la página antes de acabar de usarla.

Un sistema DSM podría elegir el tamaño del intervalo de tiempo de forma dinámica. Un posible punto de partida para su elección podría ser la observación de los accesos sobre la página (utilizando los bits de *referencia* de la unidad de gestión de memoria). Otro factor que podría considerarse es el tamaño de la cola de procesos que esperan por la página.

16.4. LIBERACIÓN DE CONSISTENCIA Y MUNIN

Los algoritmos de la sección anterior fueron diseñados para conseguir la consistencia secuencial en sistemas DSM. La ventaja de la consistencia secuencial es que el sistema DSM se comporta de la forma que sus programadores esperan que lo haga una memoria compartida. Su desventaja es el

coste de implementación. Los sistemas DSM a menudo necesitan utilizar multidifusión en sus implementaciones, tanto si se basan en escritura actualizante como en invalidación de escritura (aunque para la invalidación sea suficiente una multidifusión desordenada). La búsqueda del propietario de una página tiende a ser cara: un gestor central que conoce la ubicación del propietario de cada página se convierte en un cuello de botella; las secuencias de punteros implican, de media, la necesidad de más mensajes. Además, los algoritmos basados en invalidaciones pueden sufrir de thrashing.

La consistencia relajada fue utilizada por primera vez en el multiprocesador Dash, el cual realiza una implementación hardware de DSM basada en un protocolo de invalidación de escritura [Lenoski y otros 1992]. Munin y Treadmarks [Keleher y otros 1992] utilizan una implementación software. La consistencia relajada es más débil que la consistencia secuencial y más barata en su implementación, pero conserva una semántica razonable que resulta tratable para los programadores.

La consistencia relajada se basa en la reducción de las sobrecargas de DSM explotando el hecho de que los programadores utilizan objetos de sincronización, como semáforos, bloqueos y barreras. Una implementación DSM puede utilizar el conocimiento del acceso a estos objetos para permitir que la memoria sea inconsistente en determinados puntos, pero asegurando la consistencia a nivel de aplicación mediante la utilización de estos objetos de sincronización.

16.4.1. ACCESOS A MEMORIA

Para comprender la consistencia relajada (o cualquier otro modelo de memoria que utilice la sincronización) comenzamos realizando una clasificación de los accesos a memoria en función de su papel, si existe, en la sincronización. Además estudiaremos cómo los accesos a memoria se pueden realizar de forma asíncrona para mejorar las prestaciones y proporcionaremos un modelo operacional sencillo de cómo se producen los accesos a memoria.

Tal y como hemos mencionado previamente, las implementaciones DSM para sistemas distribuidos de propósito general pueden utilizar, por razones de eficiencia, paso de mensajes en lugar de variables compartidas para implementar la sincronización. Sin embargo, resultará útil mantener en mente la sincronización basada en variables compartidas para la siguiente discusión. El pseudocódigo que aparece a continuación implementa bloqueos utilizando la operación *testAndSet* sobre variables. La función *testAndSet* pone la variable lock a 1 y devuelve 0 si valía previamente cero; en otro caso devuelve 1. Esto lo hace de forma atómica.

```
adquiereBloqueo(var int bloqueo):      // bloqueo se pasa por referencia
        while (testAndSet(bloqueo) = 1)
                skip;
liberaBloqueo(var int bloqueo):        // bloqueo se pasa por referencia
        lock := 0;
```

◇ **Tipos de accesos a memoria.** La principal distinción se realiza entre accesos *competitivos* y accesos *no competitivos* (ordinarios). Dos accesos son competitivos si:

- Pueden ocurrir de forma concurrente (no se fuerza el mantenimiento de un orden entre ellos).
- Al menos uno de ellos es una *escritura*.

Por lo tanto dos operaciones de *lectura* no pueden nunca ser competitivas; una *lectura* y una *escritura* sobre la misma posición realizadas por dos procesos que se sincronizan entre las operaciones (y por lo tanto las ordenan) no son competitivas.

Además dividimos los accesos competitivos entre accesos de *sincronización* y de *no sincronización*:

- Los accesos de sincronización son operaciones de *lectura* o *escritura* que contribuyen a la sincronización.
- Los accesos de no sincronización son operaciones de *lectura* o de *escritura* concurrentes pero que no contribuyen a la sincronización.

La operación de *escritura* implícita en *bloqueo* := 0 dentro de *liberaBloqueo* (arriba) es un acceso de sincronización. También lo es la operación de *lectura* implícita en *testAndSet*.

Los accesos de sincronización son competitivos, ya que los procesos que potencialmente necesitan sincronizarse acceden a variables de sincronización de forma concurrente y deben actualizarlas: las operaciones de *lectura* por sí solas no permiten conseguir la sincronización. Pero no todos los accesos competitivos son accesos de sincronización ya que existen algunos tipos de algoritmos paralelos en los que los procesos realizan accesos competitivos a variables compartidas simplemente para actualizar y leer resultados de otro y no para sincronizar.

Los accesos de sincronización también se dividen en accesos de *adquisición* y accesos de *liberación*, en función de su papel en el bloqueo potencial del proceso que realiza el acceso o en el desbloqueo de otros procesos.

◇ **Realización de operaciones asíncronas.** En la descripción de la implementación DSM con consistencia secuencial, se explicó que las operaciones de memoria pueden sufrir retardos significativos. Existen varios tipos de operaciones asíncronas que permiten incrementar la velocidad a la que se ejecutan los procesos, a pesar de los mencionados retardos. En primer lugar, las operaciones de *escritura* pueden implementarse de forma asíncrona. Un valor escrito es almacenado en un búfer antes de ser propagado y los efectos de la *escritura* se observan con posterioridad desde otros procesos. En segundo lugar, las implementaciones de DSM pueden realizar de forma anticipada la búsqueda de valores con anticipación a su lectura, evitándose así la parada de un proceso cuando necesita dichos valores. En tercer lugar, los procesadores pueden ejecutar las instrucciones fuera de orden. Mientras se espera la finalización del acceso a memoria actual, se puede iniciar la siguiente instrucción, siempre que ésta no dependa de la actual.

A la vista de las operaciones asíncronas que se acaban de presentar, distinguiremos entre el punto en el que una operación de *lectura* o *escritura* es iniciada (cuando el proceso comienza la ejecución de la operación) y el punto en el que la instrucción es *efectuada* o completada.

Supondremos que nuestro DSM es, al menos, coherente. Como se explicó en la Sección 16.2.3, esto significa que cada proceso está de acuerdo en el orden de las operaciones de *escritura* sobre una misma posición. Dada esta suposición, podemos hablar de forma no ambigua del orden de las operaciones de *escritura* sobre una cierta posición.

En un sistema de memoria compartida distribuida, podemos dibujar un diagrama temporal para cualquier operación de memoria o que ejecute el proceso P (véase la Figura 16.11).

Decimos que una operación de *escritura* $E(x)v$ se ha realizado con respecto al proceso P si desde ese punto las operaciones de *lectura* de P devuelven el valor v escrito por la operación de *escritura*, o bien el valor escrito por alguna operación posterior sobre x (nótese que otra operación puede escribir el mismo valor v).

De forma similar, decimos que una operación de *lectura* $L(x)v$ se ha realizado con respecto al proceso P cuando ninguna *escritura* posterior iniciada sobre la misma posición pueda proporcionar

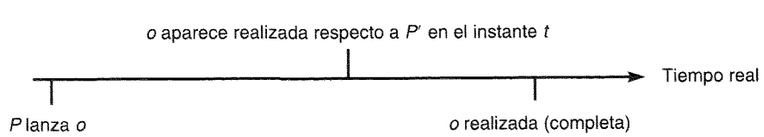


Figura 16.11. Diagrama temporal de una operación de *lectura* o *escritura* sobre DSM.

el valor v que P lee. Por ejemplo, P puede haber obtenido mediante prebúsqueda el valor que necesita *leer*.

Finalmente, se dice que la operación o se ha *realizado* si lo ha hecho respecto a todos los procesos.

16.4.2. CONSISTENCIA RELAJADA

Los requisitos que pretendemos cumplir son:

- Preservar la semántica de sincronización de objetos del tipo de bloqueos y barreras.
- Mejorar las prestaciones, para lo cual se permite un cierto grado de asincronía en las operaciones de memoria.
- Limitar el solapamiento entre los accesos a memoria para garantizar ejecuciones cuyos resultados sean equivalentes a los obtenidos con consistencia secuencial.

La memoria con consistencia relajada se ha diseñado para satisfacer estos requisitos. Gharachorloo y otros [1990] definen la consistencia relajada de la siguiente forma:

CR1: Antes de permitir que una operación ordinaria de *lectura* o *escritura* se realice respecto a cualquier otro proceso, deben haberse realizado todos los accesos de *adquisición* previos.

CR2: Antes de permitir que una operación de *liberación* sea realizada con respecto a cualquier otro proceso, deben haberse realizado todas las operaciones ordinarias de *lectura* y *escritura* previas.

CR3: Las operaciones de *adquisición* y *liberación* son secuencialmente consistentes entre ellas.

CR1 y CR2 garantizan que, cuando se haya realizado una liberación, ningún otro proceso que haya adquirido un bloqueo pueda leer versiones caducadas de los datos modificados por el proceso que realiza la liberación. Esto coincide con las expectativas de los programadores ya que cuando, por ejemplo, se libera un bloqueo, eso implica que un proceso ha terminado de modificar los datos dentro de la sección crítica.

El soporte en tiempo de ejecución de DSM sólo podrá forzar la consistencia relajada si es consciente de los accesos de sincronización. Por ejemplo, en Munin, el programador se ve forzado a utilizar las primitivas propias de Munin *acquireLock* (adquiere bloqueo), *releaseLock* (libera bloqueo) y *waitAtBarrier* (espera en la barrera). (Una barrera es un objeto de sincronización que bloquea cada uno de los procesos de un conjunto hasta que todos ellos hayan llegado a ella; entonces todos los procesos continúan.) Un programa debe utilizar la sincronización para asegurar que las actualizaciones se hacen visibles al resto de procesos. Dos procesos que comparten DSM pero no utilizan nunca objetos de sincronización pueden que nunca vean las actualizaciones que realizan entre ellos si la implementación garantiza estrictamente las condiciones expuestas previamente.

Nótese que el modelo de consistencia relajada permite que una implementación realice algunas operaciones asíncronas. Por ejemplo, un proceso necesita no ser bloqueado cuando realiza actualizaciones dentro de una sección crítica. Sus actualizaciones tampoco necesitan propagarse hasta que deja la sección crítica mediante la liberación del bloqueo. Además las actualizaciones pueden recolectarse y ser enviadas en un único mensaje. Únicamente la actualización final a cada dato debe ser enviada.

Considérense los procesos de la Figura 16.12, los cuales adquieren y liberan un bloqueo para acceder a una pareja de variables a y b (a y b son inicializadas a cero). El proceso 1 actualiza a y b bajo condiciones de exclusión mutua, de forma que el proceso 2 no puede leer a y b al mismo tiempo y por lo tanto para él se cumplirá que $a = b = 0$ o bien $a = b = 1$. Las secciones críticas

```
Proceso 1:
        adquiereBloqueo();              // entra a la sección crítica
        a := a + 1;
        b := b + 1;
        liberaBloqueo()                 // sale de la sección crítica
Proceso 2:
        adquiereBloqueo();              // entra a la sección crítica
        print ("Los valores de a y b son: ", a, b);
        liberaBloqueo();                // sale de la sección crítica
```

Figura 16.12. Procesos ejecutándose en un DSM con consistencia relajada.

fuerzan la consistencia (igualdad de *a* y *b*) al *nivel de aplicación*. Es redundante propagar las actualizaciones a las variables afectadas durante la sección crítica. Si el proceso 2 hubiera, por ejemplo, intentado acceder a la variable *a* fuera de la sección crítica, entonces hubiera encontrado un valor caducado. Ése es un problema del programador de aplicaciones.

Supongamos que el proceso 1 adquiere primero el bloqueo. El proceso 2 se bloqueará y no generará ninguna actividad sobre DSM hasta que haya adquirido el bloqueo e intente acceder a *a* y *b*. Si los dos procesos se ejecutaran en una memoria con consistencia secuencial, entonces el proceso 1 se hubiera bloqueado durante la actualización de *a* y *b*. Bajo un protocolo de escritura actualizante, se hubiera bloqueado durante la actualización de todas las versiones de los datos; bajo un protocolo de invalidación de escritura, se hubiera bloqueado hasta que se hubieran invalidado todas las copias.

Bajo la consistencia relajada, el proceso 1 no se bloqueará cuando acceda a los datos *a* y *b*. El soporte en tiempo de ejecución de DSM simplemente anota qué datos han sido actualizados pero no realiza otras tareas en ese instante. Será únicamente cuando el proceso 1 libere el bloqueo cuando se necesite la comunicación. Bajo un protocolo de escritura actualizante, se propagarán las actualizaciones sobre *a* y *b*; en un protocolo de invalidación de escritura, se enviarán las invalidaciones.

El programador (o un compilador) es el responsable de etiquetar las operaciones de lectura y escritura como accesos de tipo *liberación*, *adquisición* o *no-sincronización*; el resto de instrucciones se considerarán ordinarias. Etiquetar el programa supone la dirección del sistema DSM para forzar las condiciones de la consistencia relajada.

Gharachorloo y otros [1990] describen el concepto de programa *etiquetado correctamente*. Ellos probaron que este tipo de programa no puede distinguir entre una ejecución DSM con consistencia relajada y otra ejecución con consistencia secuencial.

16.4.3. MUNIN

El diseño del sistema DSM Munin [Carter y otros 1991] intenta mejorar la eficiencia de DSM mediante la implementación del modelo de consistencia relajada. Además, Munin permite a los programadores anotar sus datos según el tipo de compartición, de forma que se pueden realizar optimizaciones en las opciones de actualización seleccionadas para la gestión de la consistencia. Ha sido implementado sobre el núcleo V [Cheriton y Zwaenepoel 1985], que fue uno de los primeros núcleos que permitían a los hilos de nivel de usuario manejar los fallos de página y manipular las tablas de páginas.

Los siguientes puntos se refieren a la implementación de la consistencia relajada en Munin:

- Munin envía la información de actualización o invalidación tan pronto como se libere el bloqueo.

- El programador puede realizar anotaciones que asocien un bloqueo con ciertos conjuntos de datos. En este caso, el soporte en tiempo de ejecución de DSM puede propagar las actualizaciones relevantes en el mismo mensaje utilizado para transferir el bloqueo al proceso en espera, asegurando que el receptor del bloqueo tiene copias de los datos que necesita antes de utilizarlos.

Keleher y otros [1992] describen una alternativa a la aproximación *impaciente* de Munin, es decir, al envío de información de actualización o invalidación el mismo instante de la liberación. En su lugar, esta implementación *débil* realiza dicho envío únicamente cuando dicho bloqueo es adquirido a continuación. Además, envía esta información sólo a los procesos que adquieran el bloqueo y la inserta en el mensaje que sirve para conceder la adquisición de bloqueo. Es innecesario hacer visible las actualizaciones a los otros procesos hasta que ellos adquieran, por turno, el bloqueo.

◇ **Anotaciones de compartición.** Munin implementa diferentes protocolos de consistencia aplicados con una granularidad a nivel de datos individuales. Los protocolos están parametrizados según las siguientes opciones:

- Protocolo de escritura actualizante o de invalidación de escritura.
- Que puedan o no existir de forma simultánea varias copias de un dato modificable.
- Que se retarden o no las actualizaciones o las invalidaciones (por ejemplo, bajo la consistencia relajada).
- Que los datos tengan o no un propietario fijo, al que se deben enviar todas las actualizaciones.
- Que el mismo dato pueda o no ser modificado concurrentemente por varios escritores.
- Que los datos puedan o no ser compartidos por un conjunto fijo de procesos;
- Que los datos puedan o no ser modificados.

Estas opciones son elegidas en función de la naturaleza de los datos y de sus patrones de compartición entre procesos. El programador puede elegir de forma explícita qué opciones usar para cada dato. Sin embargo, Munin proporciona un conjunto pequeño, estándar de anotaciones válidas para una gran variedad de aplicaciones y datos para que el programador las aplique sobre los datos; cada una de estas anotaciones supone una elección adecuada de los parámetros. Son las siguientes:

Sólo lectura: después de la inicialización no se realizan actualizaciones y el dato puede ser copiado libremente.

Migratorio: los procesos realizan por turno varios accesos al dato, siendo al menos uno de ellos una actualización. Por ejemplo, el dato puede ser accedido dentro de una sección crítica. Munin siempre proporciona los accesos de lectura y escritura juntos en dicho objeto, incluso cuando un proceso genera un fallo de lectura. Esto ahorra el procesamiento de los fallos de escritura posteriores.

Escritura compartida: varios procesos actualizan simultáneamente la misma variable (por ejemplo, un array) de forma que esta anotación es una declaración por parte del programador de que los procesos no actualizan las mismas partes de la variable. Esto significa que Munin puede evitar la compartición falsa pero debe propagar sólo aquellas palabras del dato que cada proceso está actualizando actualmente. Para conseguirlo, Munin realiza una copia de la página (dentro de un gestor de fallos de escritura) justo antes de que sea actualizada localmente. Sólo las diferencias entre las dos versiones de la página son enviadas en una actualización.

Productor-consumidor: el dato es compartido por un conjunto fijo de procesos, siendo actualizado sólo por uno de ellos. Como ya se explicó anteriormente en la discusión del thrashing, en este caso es más apropiado un protocolo de escritura actualizante. Además, las actualizaciones pueden ser retrasadas utilizando el modelo de consistencia relajada, suponiendo que el proceso utiliza bloqueos para sincronizar su acceso.

Reducción: el dato siempre es modificado siguiendo la secuencia de bloqueo, lectura, actualización y desbloqueo. Por ejemplo, un mínimo global en una computación paralela debe ser localizado y modificado atómicamente si es mayor que el mínimo local. Estas variables son almacenadas en un propietario fijo. Las actualizaciones son enviadas al propietario, que se encarga de su propagación.

Resultado: varios procesos actualizan diferentes palabras dentro de elemento de datos mientras que un único proceso lee el elemento completo. Por ejemplo, diferentes procesos *trabajadores* escriben en diferentes elementos de un array, que es procesado posteriormente por un proceso *maestro*. La optimización consiste en propagar las actualizaciones sólo al maestro y no a los trabajadores (como ocurriría bajo la anotación de *escritura compartida* ya descrita).

Convencional: los datos se manejan mediante un protocolo de invalidación similar al descrito en la sección previa. Por lo tanto ningún proceso podrá leer versiones caducadas de un dato.

Carter y otros [1991] detallaron las opciones de parámetros utilizadas en cada una de las anotaciones descritas. Este conjunto de anotaciones no es fijo. Pueden crearse otras anotaciones de la misma forma que pueden encontrarse otros patrones de compartición que necesiten diferentes opciones de parámetros.

16.5. OTROS MODELOS DE CONSISTENCIA

Los modelos de consistencia de memoria se pueden dividir entre *modelos uniformes*, caracterizados por no distinguir entre diferentes tipos de accesos a memoria y *modelos híbridos*, los cuales distinguen entre accesos ordinarios y de sincronización (al igual que otros tipos de accesos).

Existen varios modelos uniformes más débiles que el modelo de consistencia secuencial. En la Sección 16.2.3 se presentó la coherencia como la consistencia secuencial sobre cada posición de memoria. Los procesos acuerdan el orden de todas las escrituras sobre una cierta posición, pero el orden de las escrituras desde diferentes procesadores sobre diferentes posiciones puede variar [Goodman 1989, Gharachorloo y otros 1990].

Otros modelos de consistencia uniforme son:

Consistencia causal: las lecturas y escrituras se asocian mediante la relación ocurrió-antes (véase el Capítulo 10). Esta relación define el orden entre dos operaciones de memoria cuando o bien (a) se han realizado desde el mismo proceso; o bien (b) un proceso lee un valor escrito por otro proceso; o bien (c) existe una secuencia de los dos tipos de operaciones anteriores que enlazan las dos operaciones de memoria. El modelo fuerza a que el valor devuelto por una lectura deba ser consistente con la relación ocurrió-antes. Esto es descrito por Hutto y Ahamad [1990].

Consistencia de procesador: la memoria cumple dos condiciones, es coherente y cumple el modelo de RAM encauzada (véase a continuación). La forma más sencilla de pensar en la consistencia de procesador es que la memoria es coherente y todos los procesos acuerdan el orden de dos accesos de escritura cualquiera realizados por el mismo proceso, es decir, acuerdan el orden de programa. Fue inicialmente descrito informalmente por Goodman [1989] y posteriormente definido formalmente por Gharachorloo y otros [1990] y Ahamad y otros [1992].

RAM encauzada: todos los procesadores acuerdan el orden de las escrituras iniciadas por un cierto procesador [Lipton y Sandberg 1988].

Además de la consistencia relajada, otros modelos híbridos son los siguientes:

Consistencia con admisión: la consistencia con admisión (*entry*) fue propuesta inicialmente en el sistema DSM Midway [Bershad y otros 1993]. En este modelo, cada variable compartida se enlaza a un objeto de sincronización, como por ejemplo un bloqueo, que gobierna el acceso a dicha variable. El primer proceso que adquiera el bloqueo tiene garantizada la lectura del valor más reciente de la variable. Un proceso que quiera escribir en la variable debe obtener primero el correspondiente bloqueo en modo *exclusivo*, de forma que dicho proceso sea el único capaz de acceder a la variable. Varios procesos podrán leer de forma concurrente la variable manteniendo el bloqueo en modo no exclusivo. Midway evita la tendencia a la compartición falsa de la consistencia relajada, pero a costa de incrementar la complejidad de la programación.

Consistencia de ámbito: este modelo de memoria [Iftode y otros 1996] trata de simplificar el modelo de programación de la consistencia *entry*. En la consistencia de ámbito, las variables se asocian con objetos de sincronización de forma automática en su mayor parte, en lugar de depender del programador para la asociación explícita de bloqueos a variables. Por ejemplo, el sistema puede monitorizar qué variables son actualizadas en una sección crítica.

Consistencia débil: la consistencia débil [Dubois y otros 1988] no distingue entre los accesos de sincronización *adquiere* y *libera*. Asegura que todos los accesos ordinarios previos se completan antes que lo haga *cualquiera* de los dos tipos de accesos de sincronización.

◇ **Discusión.** La consistencia relajada y algunos de los modelos de consistencia más débiles que la secuencial parecen ser los más prometedores para DSM. No parece muy significativa la desventaja del modelo de consistencia relajada consistente en que las operaciones de sincronización necesiten ser conocidas por el soporte en tiempo de ejecución de DSM, siempre y cuando las operaciones de sincronización proporcionadas por el sistema sean suficientemente potentes para cubrir las necesidades de los programadores.

Es importante darse cuenta de que, bajo los modelos híbridos, la mayor parte de los programadores no están forzados a tener en cuenta la semántica del modelo de consistencia de memoria utilizado, siempre que sincronicen los accesos a sus datos de forma adecuada. Pero existe el peligro general en los diseños DSM de pedir al programador que realice múltiples anotaciones a su programa para conseguir una ejecución eficiente. Esto incluye tanto anotaciones para asociar datos con objetos de sincronización como las anotaciones de compartición usadas en Munin. Se supone que una de las ventajas de la programación de memoria compartida sobre el paso de mensajes debería ser su relativa comodidad.

16.6. RESUMEN

En este capítulo se ha descrito y justificado el concepto de memoria compartida distribuida como una abstracción de la memoria compartida que sirve de alternativa a la comunicación basada en mensajes en un sistema distribuido. El objetivo principal de DSM es el procesamiento paralelo y la compartición de los datos. Se ha demostrado que sus prestaciones son comparables a las del paso de mensajes para ciertas aplicaciones paralelas, pero es difícil conseguir una implementación eficiente y sus prestaciones dependen en gran medida de las aplicaciones.

Este capítulo se ha centrado en las implementaciones software de DSM (en particular aquellas basadas en el subsistema de memoria virtual) con un cierto soporte del hardware.

Las principales cuestiones de diseño e implementación son la estructura de DSM, la forma de sincronizar las aplicaciones, el modelo de consistencia de memoria, la utilización de protocolos de escritura actualizante o de invalidación de escritura, la granularidad de la compartición y el thrashing.

DSM puede estructurarse como una serie de bytes, como una colección de objetos compartidos o como una colección de datos inmutables como las tuplas.

Las aplicaciones en DSM necesitan la sincronización para cumplir los requisitos de consistencia específicos de la aplicación. Para este propósito utilizan objetos como los bloqueos, implementados utilizando paso de mensajes por razones de eficiencia.

El modelo de consistencia más estricto implementado en los sistemas DSM es la consistencia secuencial. Debido a su coste, se han desarrollado otros modelos de consistencia más débiles, como la coherencia y la consistencia relajada. La consistencia relajada permite a la implementación utilizar los objetos de sincronización para conseguir mayor eficiencia sin romper las restricciones de consistencia del nivel de aplicación. Se han mencionado brevemente otros modelos de consistencia, incluyendo la consistencia de entrada, la de ámbito y la débil, todas ellas basadas en la sincronización.

Los protocolos de escritura actualizante son aquellos en los que las actualizaciones de los datos son propagadas a todas sus copias. Normalmente son implementadas en hardware, a pesar de que también existen implementaciones software que utilizan multidifusiones totalmente ordenadas. Los protocolos de invalidación de escritura evitan la lectura de datos no válidos mediante la invalidación de todas las copias cuando los datos son actualizados. Estos protocolos se adaptan mejor a los sistemas DSM basados en páginas, para los que la escritura actualizante puede ser una opción costosa.

La granularidad de DSM modifica la probabilidad de contención entre procesos con compartición falsa de datos ya que dichos datos están contenidos en la misma unidad de compartición (una página, por ejemplo). También afecta al coste por byte de la transferencia de actualizaciones entre computadores.

El thrashing puede ocurrir cuando se utiliza invalidación de escritura. Consiste en la transferencia repetida de datos entre procesos competidores a costa del progreso de la aplicación. Este efecto puede ser reducido mediante la sincronización a nivel de aplicación, permitiendo a los computadores retener una página durante una mínima cantidad de tiempo, o mediante el etiquetado de los datos de forma que las lecturas y las escrituras sean concedidas conjuntamente.

En este capítulo se han descrito los tres principales protocolos de invalidación de escritura de Ivy para DSM basada en páginas, que tratan el problema de la gestión del conjunto de copias de una página y de la localización de su propietario. Se trata del protocolo de gestión central, en el que un único proceso almacena la dirección del propietario actual de cada página; el protocolo basado en la multidifusión para localizar el propietario actual de una página; y el protocolo de gestión distribuida dinámica, que envía cadenas de punteros para localizar al propietario actual de una página.

Munin es un ejemplo de implementación de la consistencia relajada. Implementa una consistencia relajada impaciente caracterizada porque la propagación de los mensajes de actualización o invalidación se realiza tan pronto como el bloqueo es liberado. Existen otras implementaciones más débiles que propagan sólo aquellos mensajes que son solicitados. Munin permite a los programadores anotar sus datos para seleccionar aquellas opciones de protocolo que mejor se adapten, dada la forma en la que se comparten.

EJERCICIOS

16.1. Explique las diferentes razones que justifiquen si lo sistemas DSM son viables o no en sistemas cliente-servidor.

16.2. Discuta qué es más apropiado para aplicaciones tolerantes a fallos, si el paso de mensaje o DSM.

16.3. ¿Cómo resolvería el problema de diferentes representaciones de datos para una implementación DSM basada en middleware sobre computadores heterogéneos? ¿Cómo se abordaría el problema en una implementación basada en páginas? ¿Su solución puede extenderse a punteros?

16.4. ¿Por qué razón se podría querer implementar un sistema DSM basado en páginas a nivel de usuario y qué se necesita para conseguirlo?

16.5. ¿Cómo implementaría un semáforo utilizando un espacio de tuplas?

16.6. La memoria subyacente a la ejecución de estos dos procesos, ¿tiene consistencia secuencial? (asumiendo que todas las variables son inicializadas a cero).

P_1: $L(x)1$; $L(x)2$; $E(y)1$
P_2: $E(x)1$; $L(y)1$; $E(x)2$

16.7. Utilizando la notación $L()$, $E()$, proporcione un ejemplo de una ejecución en una memoria que es coherente pero no tiene consistencia secuencial. ¿Puede una memoria tener consistencia secuencial pero no ser coherente?

16.8. Sobre escritura actualizante, muestre que la consistencia secuencial puede dejar de cumplirse si cada actualización se realiza de forma local antes de enviarla vía multidifusión asíncrona a otros gestores de réplica, incluso si la multidifusión es totalmente ordenada. Discuta si una multidifusión asíncrona puede o no utilizarse para conseguir consistencia secuencial. (Pista: considere si se deben o no bloquear las operaciones subsiguientes.)

16.9. La memoria con consistencia secuencial puede implementarse mediante un protocolo de escritura actualizante utilizando una multidifusión síncrona y totalmente ordenada. Discuta qué requisitos de ordenamiento de multidifusión serían necesarios para implementar memoria coherente.

16.10. Explique por qué, en un protocolo de escritura actualizante, es preciso tener cuidado para propagar únicamente aquellas palabras que hayan sido actualizadas localmente dentro de una variable compleja.

Desarrolle un algoritmo para representar las diferencias entre una página y su versión actualizada. Discuta las prestaciones de este algoritmo.

16.11. Explique por qué la granularidad es un aspecto importante en los sistemas DSM. Compare la cuestión de la granularidad entre los sistemas DSM orientados a objetos y los orientados a bytes, teniendo en mente sus implementaciones.

¿Por qué la granularidad es relevante en espacios de tuplas, caracterizados por contener datos inmutables?

¿Qué es la compartición falsa? ¿ Puede provocar ejecuciones incorrectas?

16.12. ¿Cuáles son las implicaciones de DSM en las políticas de reemplazo de páginas (es decir, la elección de qué página eliminar desde la memoria principal para hacer espacio a una página nueva)?

16.13. Pruebe que el protocolo de invalidación de escritura de Ivy garantiza la consistencia secuencial.

16.14. En el algoritmo de gestión distribuida dinámica de Ivy, ¿qué pasos deben tomarse para minimizar el número de saltos necesarios para encontrar una página?

16.15. ¿Por qué el thrashing es una cuestión importante en sistemas DSM y qué métodos están disponibles para su resolución?

16.16. Discuta cómo podría relajarse la condición CR2 de la consistencia relajada. Por lo tanto distinga entre consistencia relajada impaciente y débil.

16.17. Un proceso sensor escribe la temperatura actual sobre una variable t almacenada en un sistema DSM con consistencia relajada. De forma periódica, un proceso monitor lee t. Explique por qué se necesita la sincronización para propagar las actualizaciones sobre t, incluso aunque ninguna de ellas sea necesaria en el nivel de aplicación. ¿Qué proceso de entre los mencionados debe realizar las operaciones de sincronización?

16.18. Muestre que la siguiente historia no tiene consistencia causal:

P_1: $E(a)0$; $E(a)1$
P_2: $L(a)1$; $E(b)2$
P_3: $L(b)2$; $L(a)0$

16.19. ¿Qué ventaja puede obtener una implementación DSM si conoce la asociación entre datos y objetos de sincronización? ¿Cuál es la desventaja de realizar dicha asociación de forma explícita?

EL CASO DE ESTUDIO CORBA

17.1. Introducción
17.2. CORBA RMI
17.3. Servicios de CORBA
17.4. Resumen

CORBA es un diseño de middleware que permite que los programas de aplicación se comuniquen unos con otros con independencia de sus lenguajes de programación, sus plataformas hardware y software, las redes sobre las que se comunican y sus implementadores.

Las aplicaciones se construyen mediante objetos CORBA, que implementan interfaces definidas en el lenguaje de definición de interfaces de CORBA, IDL. Los clientes acceden a los métodos de las interfaces de los objetos de CORBA mediante RMI. El componente middleware que da soporte a RMI se denomina Object Request Broker (Intermediario de Petición de Objetos) u ORB.

La especificación de CORBA ha sido promovida por los miembros del Object Management Group (OMG). Se han implementado versiones muy diferentes de ORB, que soportan variedad de lenguajes de programación.

Los servicios CORBA proporcionan un equipamiento genérico que puede utilizarse en una gran variedad de aplicaciones. Estos servicios incluyen el Servicio de Nombres, los Servicios de Eventos y Notificación, el Servicio de Seguridad, los Servicios de Transacción y Concurrencia y el Servicio de Comercio.

17.1. INTRODUCCIÓN

El OMG (Object Management Group) fue formado en 1989 con la perspectiva de promocionar la adopción de sistemas de objetos distribuidos para conseguir los beneficios de la programación orientada al objeto para el desarrollo del software y la utilización sobre los sistemas distribuidos, que comenzaban entonces a difundirse. Para lograr sus objetivos, el OMG abogaba por el uso de sistemas abiertos basados en interfaces estándar orientadas al objeto. Estos sistemas podrían construirse partiendo de una base heterogénea de hardware, redes de computadores, sistemas operativos y lenguajes de programación.

Una motivación importante fue el permitir que los objetos distribuidos fueran implementados en cualquier lenguaje de programación fueran capaces de comunicarse uno con otro. Así, diseñaron un lenguaje de interfaz independiente de cualquier lenguaje específico de implementación.

Se introdujo una metáfora, el *intermediario de petición de objetos* (ORB, *object request broker*) cuyo papel es el de ayudar a un cliente a invocar un método de un objeto. Esta función conlleva localizar el objeto, activar el objeto si fuera necesario y después comunicar la petición del cliente hacia el objeto, que la realiza y responde al cliente.

En 1991, un grupo de compañías se pusieron de acuerdo en una especificación de una arquitectura de intermediario de petición de objetos conocida como CORBA (Common Object Request Broker Arquitecture o Arquitectura Común de Intermediario de Petición de Objetos). A ésta le siguió, en 1996, la especificación CORBA 2.0 [OMG 1998a], que definió estándares para permitir la comunicación entre implementaciones realizadas por desarrolladores diferentes. Estos estándares se denominan General Inter-ORB Protocol (Protocolo General Inter-ORB) o GIOP. Se pretendía que este GIOP pudiera ser implementado sobre cualquier capa de transporte con conexiones. La implementación de GIOP para Internet utiliza el protocolo TCP/IP y se denomina Internet Inter-ORB Protocol (IIOP, Protocolo Inter-ORB para Internet).

Los componentes principales del esquema RMI independiente del lenguaje de CORBA son los siguientes:

- Un lenguaje de definición de interfaces conocido como IDL, que se ilustra al principio de la Sección 17.2 y se describe en más profundidad en la Sección 17.2.3.
- Una arquitectura, que se discute en la Sección 17.2.2.
- El GIOP define una representación de datos externa, denominada CDR, que se describe en la Sección 4.3. También define formatos específicos para los mensajes de un protocolo petición-respuesta. Además de los mensajes de petición y de respuesta, especifica los mensajes de interrogación sobre la ubicación de objetos, para cancelar peticiones y para informar de errores.
- EL IIOP define una forma estándar para las referencias a objetos remotos, que se describe en la Sección 17.2.4.

La arquitectura de CORBA también permite servicios CORBA: un conjunto de servicios genéricos que pueden ser de utilidad para las aplicaciones distribuidas. Éstas se presentan en la Sección 17.3 donde se incluye una discusión más detallada del Servicio de Nombres, el Servicio de Eventos, el Servicio de Notificación y el Servicio de Seguridad. Para consultar una interesante colección de artículos sobre CORBA, véase el número especial de *CACM* [Seetharamanan 1998].

Previo a la discusión de los componentes de CORBA anteriores, presentaremos CORBA RMI desde el punto de vista del programador.

17.2. CORBA RMI

La programación en un sistema RMI multilenguaje como CORBA RMI requiere más del programador que la programación sobre un sistema RMI de un solo lenguaje, como Java RMI. Es necesario aprender los siguientes nuevos conceptos:

- El modelo de objeto que ofrece CORBA.
- El lenguaje de definición de interfaz y su correspondencia sobre el lenguaje de implementación.

Los otros aspectos de la programación sobre CORBA son similares a los que se discutieron en el Capítulo 5. En particular, el programador define interfaces remotas para los objetos remotos y después usa un compilador de interfaces para producir cada proxy y cada esqueleto correspondiente. Aunque en CORBA, el proxy se genera en el lenguaje del cliente y el esqueleto en el lenguaje del servidor. Utilizando el ejemplo de un tablero simple, presentado en la Sección 5.5, mostraremos cómo escribir una especificación IDL y cómo construir los programas cliente y servidor.

◇ **El modelo de objetos de CORBA.** El modelo de objetos de CORBA es similar al que se describió en la Sección 5.2, aunque los clientes no son objetos necesariamente; un cliente podrá ser cualquier programa que envíe mensajes de petición a los objetos remotos y reciba las respuestas. El término *objeto CORBA* se emplea para referirse a los objetos remotos. Así, un objeto CORBA implementa una interfaz de un IDL, tiene una referencia a un objeto remoto y es capaz de responder a las invocaciones de los métodos en su interfaz del IDL. Se puede implementar un objeto CORBA en un lenguaje que no sea orientado al objeto, por ejemplo sin el concepto de clase. Dado que los lenguajes de implementación pueden tener diferentes nociones de clase, o incluso ninguna en absoluto, el concepto de clase no existe en CORBA. Es así que no se pueden definir clases en CORBA IDL, lo que quiere decir que no podemos pasar instancias de clases como argumento. Sin embargo, sí se pueden pasar como argumentos estructuras de datos de varios tipos y de complejidad arbitraria.

◇ **CORBA IDL.** Una interfaz CORBA IDL especifica un nombre y un conjunto de métodos que podrán utilizar los clientes. La Figura 17.1 muestra dos interfaces de nombre *Forma* (en la línea 3) y *ListaForma* (en la línea 5), que son las versiones IDL de las interfaces definidas en la Figura 5.11. Éstas vienen precedidas de las definiciones de dos *estructuras* (*struct*), que que se emplean como tipos de los parámetros en la definición de los métodos. Adviértase que *ObjetoGrafico* se define como un *struct*, mientras que en el ejemplo en Java RMI era una clase. Un componente cuyo tipo sea un *struct* posee un conjunto de campos que contienen valores de diversos tipos, como las variables de instancia de un objeto, pero no posee métodos. Habrá más sobre IDL en la Sección 17.2.3.

Parámetros y resultados en CORBA IDL: Cada parámetro se marca como de entrada o de salida, o ambos, mediante las palabras clave *in*, *out* o *inout*. La Figura 5.2 muestra un ejemplo simple del uso de estas palabras clave. En la Figura 17.1, en la línea 7, el parámetro de *nuevaForma* es un parámetro *in* que indica que el argumento debería pasarse desde el cliente al servidor en el mensaje de petición. El valor de retorno proporciona un parámetro *out* adicional; si no hubiera parámetro *out* se indicaría como *void*.

Los parámetros podrán ser de uno cualquiera de los tipos primitivos, como *long* y *boolean*, o de uno de los tipos construidos como un *struct* o un *array*. Los tipos primitivos y estructurados se describen en mayor detalle en la Sección 17.2.3. Nuestro ejemplo muestra las definiciones de dos *struct* en las líneas 1 y 2. Las secuencias y las cadenas se definen mediante *typedef*, como se ve en la línea 4, que muestra una secuencia de elementos de tipo *Forma* de longitud 100. La semántica de paso de parámetros es como sigue:

Paso de objetos CORBA: cualquier parámetro cuyo tipo esté especificado mediante el nombre de una interfaz IDL, tal como el valor de vuelta *Forma* en la línea 7, es una referencia a un objeto CORBA y se pasa el valor de una referencia a un objeto remoto.

Paso de tipos primitivos y construidos de CORBA: los argumentos de tipo primitivo y construido se copian y se pasan por valor. En el destino, se crea un nuevo valor en el proceso

```
struct Rectangulo{                                                          1
        long anchura;
        long altura;
        long x;
        long y;
};
struct ObjetoGrafico {                                                      2
        string tipo;
        Rectangle enmarcado;
        boolean estaRelleno;
};
interface Forma {                                                           3
        long dameVersion() ;
        ObjetoGrafico dameTodoEstado();     // devuelve el estado del ObjetoGrafico
};
typedef sequence <Forma, 100> Todo;                                         4
interface ListaForma {                                                      5
    exception ExcepcionLlena{ };                                            6
    Forma nuevaForma(in ObjetoGrafico g) raises (ExcepcionLlena);           7
    Todo TodasFormas( );          // devuelve una secuencia de referencias a objetos remotos   8
    long dameVersion();
};
```

Figura 17.1. Interfaces IDL *Forma* y *ListaForma*.

receptor. Por ejemplo, la estructura *ObjetoGrafico* pasada como argumento (en la línea 7) produce una copia nueva de esta estructura en el servidor.

En el método *todasFormas* se combinan estas dos formas de paso de parámetros (véase la línea 8), cuyo valor devuelto es una cadena de tipo *Forma*: esto es, una cadena de referencias a objetos remotos. El valor devuelto es una copia de la cadena en la que en cada uno de sus elementos hay una referencia a un objeto remoto.

Tipo *Object*: *Object* es el nombre de un tipo cuyos valores son referencias a objetos remotos. Su efecto es el de ser un supertipo común de todos los tipos de interfaz IDL tales como *Forma* y *ListaForma*.

Excepciones en CORBA IDL: CORBA IDL permite que se definan excepciones en las interfaces y sean lanzadas por sus métodos. Para ejemplificar este punto, hemos definido nuestra lista de formas en el servidor como una secuencia de longitud fija (véase la línea 4) y se hemos definido *ExcepcionLlena* (véase la línea 6), que es lanzada por el método *nuevaForma* (véase la línea 7) si el cliente intenta añadir una forma cuando la secuencia está llena.

Semántica de invocación: la invocación remota en CORBA define, por defecto, la semántica *como máximo una vez*. Sin embargo, IDL puede especificar que la invocación de un método concreto tenga semántica *puede ser* mediante la palabra clave *oneway*. El cliente no se bloquea en las peticiones *oneway*, que solamente puede emplearse para los métodos sin resultado. Como ejemplo de una petición *oneway*, véase el ejemplo sobre devoluciones de llamada al final de la Sección 17.2.1.

◇ **El Servicio de Nombres de CORBA.** El Servicio de Nombres de CORBA (*Naming Service*) se discute en la Sección 17.3.1. Es un enlazador que proporciona operaciones como *rebind* para que los servidores registren referencias a objetos remotos de objetos CORBA mediante su nombre y *resolve* para que los clientes las busquen mediante este nombre. Los nombres se estructuran al modo jerárquico, y cada nombre de una ruta se encuentra en una estructura denominada *NameComponent*. Esto hace que el acceso en un ejemplo sencillo parezca algo bastante complejo.

◇ **Pseudo objetos CORBA.** Las implementaciones de CORBA proporcionan algunas interfaces sobre las funciones del ORB que los programadores necesitan conocer. Éstas se denominan pseudo-objetos dado que no pueden utilizarse como si fueran objetos CORBA; por ejemplo, no pueden pasarse como argumentos en una RMI. Tienen interfaces IDL y vienen implementadas en bibliotecas. Para nuestro ejemplo sencillo, la relevante es:

- ORB es el nombre de una interfaz que representa la funcionalidad del ORB a la que necesitan acceder los programadores. Ésta incluye:
  - — El método *init*, que deberá llamarse para inicializar el ORB.
  - — El método *connect*, que se emplea para registrar objetos CORBA con el ORB.
  - — Otros métodos, que permiten conversiones entre referencias a objetos remotos y cadenas de texto.

17.2.1. EJEMPLO DE CLIENTE Y SERVIDOR CORBA

Esta sección apunta las etapas necesarias al producir programas cliente y servidor que emplean las interfaces IDL *Forma* y *ListaForma* que se muestran en la Figura 17.1. Se sigue con una discusión sobre devoluciones de llamada en CORBA. Se emplea Java como lenguaje para el cliente y el servidor, pero la aproximación es similar para los otros lenguajes. El compilador de interfaz *idltojava* se aplica a las interfaces CORBA para generar los siguientes elementos:

- Cada interfaz Java equivalente. Por ejemplo, en la Figura 17.2 se muestra la interfaz Java para *ListaForma*.
- Un esqueleto de servicio para cada interfaz IDL. Los nombres de las clases esqueleto terminan en *ImplBase*, por ejemplo *_ListaFormaImplBase*.
- Una clase proxy o un resguardo de cliente, una por cada interfaz IDL. Los nombres de estas clases terminan en *Stub*, por ejemplo *_ListaFormaStub*.
- Una clase Java para cada construcción *struct* definida con las interfaces IDL. En nuestro ejemplo, se generan las clases *Rectangulo* y *ObjetoGrafico*. Cada una de estas clases contiene una declaración de una variable de instancia para cada campo del correspondiente *struct* y un par de constructores, aunque ningún método más.
- Clases denominadas ayudantes y propietarios, para cada uno de los tipos definidos en la interfaz IDL. Una clase ayudante contiene el método *narrow*, que se emplea para convertir a partir de una referencia dada a un objeto hacia la clase a la que pertenece, que estará en un punto más bajo de la jerarquía de clases. Por ejemplo, el método *narrow* en *FormaHelper* convierte hacia la clase *Forma*. La clase propietario trata con los argumentos de carácter *out* y *inout*, los cuales no se corresponden directamente con Java. Véase el Ejercicio 17.9 para un ejemplo del uso de propietarios.

◇ **Programa servidor.** El programa servidor deberá contener implementaciones de una o más interfaces IDL. Para un servidor escrito en un lenguaje orientado al objeto como Java o C++, estas implementaciones vienen dadas en las clases sirvientes. Los objetos CORBA son instancias de las clases sirvientes.

```
public interface Listaforma extends org.omg.CORBA.Object {
   Forma nuevaForma(ObjetoGrafico g) throws ListaFormaPackage.ExcepcionLlena;
   Forma [] todasFormas();
   int dameVersion();
}
```

Figura 17.2. Interfaz Java *ListaForma* generado por *idltojava* de la interfaz CORBA *ListaForma*.

```
import org.omb.CORBA.*;
class SirvienteListaForma extends _ListaFormaImplBase {
    ORB elORB;
    private Forma laLista[];
    private int version;
    private static int n=0;
    private SirvienteListaForma(ORB orb){
        elOrb=orb;
        // inicialización de las otras variables de instancia
    }
    public Forma nuevaForma(ObjetoGrafico g) throws ListaFormaPackage.ExcepcionLlena{      1
        version++;
        Forma f=new SirvienteForma(g, version);
        if(n> =100) throw new ListaFormaPackage.ExcepcionLlena();
        laLista[n++]=f;
        elOrb.connect(f);                                                                   2
        return s;
    }
    public Forma[] todasFormas(){...}
    public int dameVersion() {...}
}
```

Figura 17.3. Clase *SirvienteListaForma* del programa servidor para la interfaz CORBA *ListaForma*.

Cuando un servidor crea una instancia de una clase sirviente, deberá registrarla frente al ORB, que convierte la instancia en un objeto CORBA y le da una referencia a un objeto remoto. A menos que se haya hecho esto, no podrá recibir invocaciones remotas. Los lectores que hayan estudiado el Capítulo 5 cuidadosamente se darán cuenta de que al registrar el objeto se registra en el equivalente CORBA de la tabla de objetos remotos.

En nuestro ejemplo, el servidor contiene las implementaciones de las interfaces *Forma* y *ListaForma* en forma de dos clases sirvientes, junto con una clase servidora que contiene una sección de *inicialización* en su método *main*.

Las clases sirvientes: cada clase sirviente extiende la clase esqueleto correspondiente e implementa los métodos de una interfaz IDL empleando las signaturas de los métodos que se definen en la interfaz Java equivalente. La clase sirviente que implementa la interfaz *ListaForma* se denomina *SirvienteListaForma*, aunque podría haberse elegido cualquier otro nombre. Su diseño se muestra en la Figura 17.3. Considere el método *nuevaForma* en la línea 1, que es un método factoría puesto que crea objetos *Forma*. Para hacer de un objeto *Forma* un objeto CORBA, se registra frente al ORB por medio de su método *connect*, como se muestra en la línea 2. Las versiones completas de la interfaz IDL y de las clases del cliente y del servidor se encuentran disponibles en cdk3.net/corba.

El servidor: el método *main* de la clase del servidor *ServidorListaForma* se muestra en la Figura 17.4. Primero crea e inicializa el ORB (véase la línea 1). Posteriormente crea una instancia de *SirvienteListaForma*, que no es más que un objeto Java (véase la línea 2). Lo convierte en un objeto CORBA al registrarlo frente al ORB (véase la línea 3). Tras esto, obtiene una referencia al Servicio de Nombres (véase la línea 4) e invoca la operación *rebind* para registrar al servidor frente al Servicio de Nombres (véase la línea 7). Es entonces cuando espera las peticiones de los clientes.

Los servidores al hacer uso del Servicio de Nombres primero obtienen un contexto de nominación raíz (véase la línea 4), después crean un *NameComponent* (véase la línea 5), definen una ruta


```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingcontextPackage.*;
import org.omg.CORBA.*;
public class ServidorListaForma {
    public static void main(String args[]) {
        try{
            ORB orb=ORB.init(args, null);                                                   1
            SirvienteListaForma formaRef=new SirvienteListaForma(orb)                       2
            orb.connect(formaRef);                                                          3
            org.omg.CORBA.Object objRef=
                orb.resolve_initial_references("NameService");                              4
            NamingContext ncRef=NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ListaForma", "");                         5
            NameComponent ruta[]={nc};                                                      6
            ncRef.rebind(ruta, formaRef);                                                   7
            java.lang.Object sincr=new java.lang.Object();
            synchronized(sincr) { sincr.wait();}
        } catch (Exception e) {...}
    }
}
```

Figura 17.4. Clase Java *ServidorListaForma*.

(véase la línea 6) y finalmente emplean el método *rebind* (véase la línea 7) para registrar el nombre y la referencia al objeto remoto. Los clientes llevan a cabo los mismos pasos pero emplean el método *resolve* según se muestra en la Figura 17.5, en la línea 2.

◇ **El programa cliente.** En la Figura 17.5 se muestra un ejemplo de un programa cliente. Crea e inicializa un ORB (véase la línea 1), entonces contacta con el Servicio de Nombres para obtener una referencia al objeto remoto *ListaForma* mediante su método *resolve* (véase la línea 2). Tras ello, invoca su método *todasFormas* (véase la línea 3) para conseguir una secuencia de referencias a objetos remotos de todas las *Formas* que posee el servidor en la actualidad. Después invoca el método *dameTodoEstado* (véase la línea 4), dándole como argumento la referencia al primer objeto remoto de la secuencia obtenida; el resultado se proporciona como una instancia de la clase *ObjetoGrafico*.

El método *dameTodoEstado* parece contradecir nuestra afirmación anterior de que no es posible pasar objetos por valor en CORBA, dado que tanto el cliente como el servidor trabajan con instancias de la clase *ObjetoGrafico*. Sin embargo, no hay tal contradicción: el objeto CORBA devuelve un elemento *struct*, y los clientes que emplearan otro lenguaje diferente también lo verían de forma diferente. O por ejemplo, en el lenguaje C++ el cliente lo vería también como un *struct*. Incluso en Java, la clase generada *ObjetoGrafico* es más parecida a un *struct* dado que no posee métodos.

Los programas cliente deberían atrapar siempre las excepciones CORBA *SystemExceptions*, que informan de los errores debidos a la distribución (véase la línea 5). Los programas cliente también deberían atrapar las excepciones definidas en la interfaz IDL, tales como *ExcepcionLlena* que lanza el método *nuevaForma*.

Este ejemplo ejemplifica el uso de la operación *narrow*: la operación *resolve* del Servicio de nombres devuelve un valor del tipo *Object*; este tipo se restringe para adecuarse al tipo concreto requerido: *ListaForma*.

◇ **Devoluciones de llamada.** Las devoluciones de llamada (*retrollamadas*) pueden implementarse en CORBA de modo similar a la descrita para Java RMI en la Sección 5.5.1. Por ejemplo la interfaz *RetrollamadaTablero* podría definirse como sigue:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingcontextPackage.*;
import org.omg.CORBA.*;
public class ClienteListaForma{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                  1
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ListaForma", "");
            NameComponent ruta [] = { nc };
            ListaForma listaFormaRef =
                ListaFormaHelper.narrow(ncRef.resolve(ruta));                2
            Forma[] listaF = listaFormaRef.todasFormas();                    3
            ObjetoGrafico g = listaF[0].dameTodoEstado();                    4
        } catch (org.omg.CORBA.SystemException e) {...}
}
```

Figura 17.5. Programa cliente Java para las interfaces CORBA *Forma* y *ListaForma*.

```
interface retrollamadaTablero {
    oneway void retrollamada(in int version);
};
```

Esta interfaz se implementa como un objeto CORBA por parte del cliente, permitiendo que el servidor envíe al cliente un número de versión cuando se añadan objetos nuevos. Pero antes de que el servidor pueda hacer esto, el cliente necesita informar al servidor de la referencia a este objeto remoto. Para posibilidad esto, la interfaz *ListaForma* requiere métodos adicionales tales como *registra* y *desregistra*, como sigue:

```
int registra(in RetrollamadaTablero retrollamada);
void desregistra(in int idRetrollamada);
```

Después de que el cliente haya obtenido una referencia al objeto *ListaForma* y creado una instancia de *RetrollamadaTablero*, emplea el método *registra* de *ListaForma* para informar al servidor de que está interesado en recibir retrollamadas. El objeto *ListaForma* del servidor es el responsable de mantener una lista de clientes interesados y de notificar a todos ellos cada vez que se incrementa el número de versión al añadir un objeto nuevo. El método *retrollamada* se declara como *oneway* de modo que el servidor pueda utilizar llamadas asíncronas para evitar retrasos cuando notifica a cada cliente.

17.2.2. LA ARQUITECTURA DE CORBA

La arquitectura está diseñada para dar soporte al papel de un intermediario de peticiones de objetos que permita que los clientes invoquen métodos en objetos remotos, donde tanto los clientes como los servidores puedan implementarse en variedad de lenguajes de programación. Los componentes principales de la arquitectura CORBA se muestran en la Figura 17.6.

Al comparar esta figura con la Figura 5.6 advertimos que la arquitectura CORBA contiene tres componentes mayores: el adaptador de objeto, el repositorio de implementación y el repositorio de interfaz.

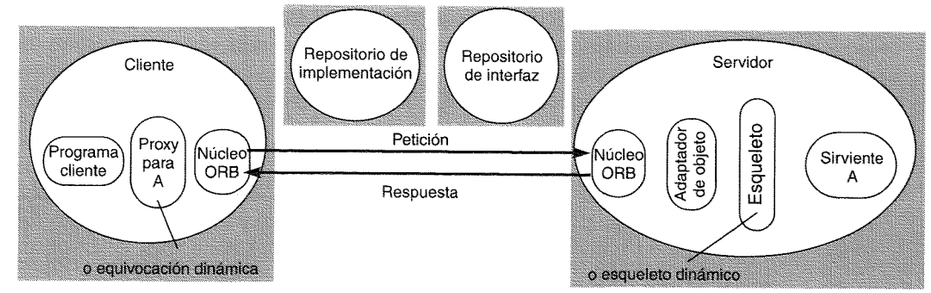


Figura 17.6. Componentes principales de la arquitectura CORBA.

CORBA distingue entre invocaciones estáticas y dinámicas. Las invocaciones estáticas se emplean cuando se conoce en tiempo de compilación la interfaz remota del objeto CORBA, lo que permite emplear un resguardo de cliente y un esqueleto de servidor. Si la interfaz remota no es conocida en tiempo de compilación, debe emplearse una invocación dinámica. La mayoría de los programadores prefieren emplear la invocación estática porque proporciona un modelo de programación más natural.

A continuación, discutiremos los componentes de la arquitectura, dejando para el final aquellos relacionados con la invocación dinámica.

◇ **El núcleo de ORB.** El papel del núcleo de ORB es similar al del módulo de comunicación de la Figura 5.6. Además, un núcleo de ORB proporciona una interfaz que incluye lo siguiente:

- Operaciones que permiten su arranque y parada.
- Operaciones para la conversión entre referencias a objetos remotos y cadenas de texto.
- Operaciones para obtener listas de argumentos para las llamadas que emplean invocación dinámica.

◇ **Adaptador de objeto.** El papel de un *adaptador de objeto* es servir de puente sobre el hueco que media entre los objetos con interfaces IDL y las interfaces del lenguaje de programación de las correspondientes clases sirviente. Este papel también cubre la existente entre la referencia remota y los módulos de despacho en la Figura 5.6. Un adaptador de objeto tiene los siguientes cometidos:

- Crea referencias a objetos remotos para los objetos CORBA.
- Despacha cada RMI vía un esqueleto hacia el sirviente apropiado.
- Activa objetos.

Un adaptador de objeto da a cada objeto CORBA un único *nombre de objeto*, que forma parte de su referencia a objeto remoto. Cada vez que se activa un objeto se empleará este mismo nombre. El nombre de objeto puede venir especificado por el programa de aplicación o ser generado por el adaptador de objeto. Cada objeto CORBA se registra frente a su adaptador de objeto, que puede mantener una tabla a objetos remotos que relaciona los nombres de objetos CORBA con sus sirvientes.

Cada adaptador de objeto tiene su propio nombre, que también forma parte de las referencias a objeto remoto de todos los objetos CORBA que gestiona. También este nombre puede venir especificado por el programa de aplicación o generarse automáticamente. El estándar CORBA 2.2 para los adaptadores de objetos se denomina Adaptador de Objetos Portable (*Portable Object Adapter*, POA). Se dice que es portable porque permite lanzar aplicaciones y sirvientes sobre cualquier ORB producido por desarrolladores diferentes [Vinoski 1998].

◇ **Esqueletos.** Las clases esqueleto se generan en el lenguaje del servidor mediante un compilador de IDL. Igual que antes, las invocaciones de métodos remotos se despachan mediante el esqueleto apropiado a un sirviente concreto, y el esqueleto desempaqueta los argumentos desde los mensajes de petición y empaqueta las excepciones y los resultados en mensajes de respuesta.

◇ **Resguardo/proxy del cliente.** Se encuentran en el lenguaje del cliente. La clase de un proxy (para los lenguajes orientados al objeto) o un conjunto de procedimientos de resguardo (para los lenguajes procedimentales) se genera desde una interfaz IDL mediante un compilador de IDL para el lenguaje del cliente. Igual que antes, cada resguardo/proxy empaqueta los argumentos de los mensajes de invocación y desempaqueta las excepciones y los resultados de las respuestas.

◇ **Repositorio de implementación.** Cada repositorio de implementación es responsable de activar los servidores registrados bajo demanda y localizar los servidores que están en ejecución en cada momento. Para hacer referencia a los servidores, cuando se registran y se activan se emplea el nombre del adaptador del objeto.

Un repositorio de implementación almacena la correspondencia de los nombres de ciertos adaptadores de objetos con las rutas de los archivos que contienen los objetos de la implementación. Los objetos de la implementación y los nombres del adaptador de objetos se registran usualmente frente al repositorio de implementación cuando se instalan los programas servidores. Cuando se activan las implementaciones de los objetos en los servidores, se añade a la relación el nombre del computador y el número de puerto donde se ubica el servidor.

Entrada del repositorio de implementación:

Nombre de adaptador de objeto	Ruta de la implementación de objeto	Nombres host y número de puerto del servidor

No todos los objetos CORBA necesitan ser activados bajo demanda. Algunos objetos, por ejemplo los objetos de retrollamada creados por los clientes, se lanzan una vez y dejan de existir cuando ya no se necesitan. Éstos no utilizan el repositorio de implementación.

Un repositorio de implementación permite almacenar generalmente cierta información extra sobre cada servidor, por ejemplo, información de control de acceso sobre quién está capacitado para activarlo o para invocar sus operaciones. Es posible replicar información en los repositorios de implementación para proporcionar disponibilidad o tolerancia a fallos.

◇ **Repositorio de interfaz.** El papel del repositorio de interfaz es proporcionar información sobre las interfaces IDL registradas a los clientes y a los servidores que lo requieran. Para una interfaz de un cierto tipo puede aportar los nombres de los métodos y para cada método los nombres y los tipos de los argumentos y las excepciones. Así, el repositorio de interfaz añade posibilidades de reflexión a CORBA. Suponga que un programa cliente recibe una referencia remota a un nuevo objeto CORBA. Suponga también que el cliente no tiene un proxy para él; entonces se podrá consultar al repositorio de interfaz sobre los métodos del objeto y los tipos de los parámetros que requiere.

El compilador de IDL asigna un identificador de tipo único a cada tipo IDL de cada interfaz que compila. En las referencias a objetos remotos se incluye un identificador de tipo para los objetos CORBA de ese tipo. Este identificador se denomina ID del repositorio porque puede usarse como clave para las interfaces IDL registradas en el repositorio de interfaz.

Aquellas aplicaciones que utilicen la invocación estática (ordinaria) con resguardos de cliente y esqueletos del IDL no necesitan un repositorio de interfaz. No todos los ORB proporcionan un repositorio de interfaz.

◇ **Interfaz de invocación dinámica.** En algunas aplicaciones, puede que un cliente sin la clase proxy apropiada necesite invocar un método de un objeto remoto. Por ejemplo, un visualizador puede necesitar mostrar información sobre todos los objetos CORBA disponibles en los diferentes servidores de un sistema distribuido. No es razonable que tal programa tenga que enlazar un proxy para cada uno de estos objetos, y más concretamente si se tiene en cuenta que se pueden añadir más objetos nuevos según transcurre el tiempo en el sistema. CORBA no permite cargar clases de proxy en tiempo de ejecución como en Java RMI. La alternativa de CORBA es la interfaz de invocación dinámica.

Ésta permite que los clientes realicen invocaciones dinámicas sobre objetos remotos CORBA. Se emplea cuando no es práctico emplear proxies. El cliente puede obtener la información necesaria sobre los métodos disponibles para un objeto CORBA dado desde el repositorio de interfaz. El cliente puede utilizar esta información para construir una invocación con los argumentos apropiados y enviarla al servidor.

◇ **Interfaz de esqueleto dinámica.** Ésta permite a un objeto CORBA aceptar invocaciones sobre una interfaz para la que no hay un esqueleto, posiblemente porque no se conocía el tipo de su interfaz en tiempo de compilación. Cuando un esqueleto dinámico recibe una invocación, inspecciona los contenidos de la petición para descubrir el objeto destino, el método que se invoca y los argumentos. A continuación invoca sobre el objetivo.

◇ **Código de legado.** El término *código de legado* se refiere al código existente que no fue diseñado previendo los objetos distribuidos. Se puede convertir un fragmento de código de legado en un objeto CORBA definiendo una interfaz IDL para él y proporcionando la implementación de un adaptador de objeto y los esqueletos.

17.2.3. LENGUAJE DE DEFINICIÓN DE INTERFAZ DE CORBA

El Lenguaje de Definición de Interfaz (*Interface Definition Language*) de CORBA, IDL, ofrece mecanismos para definir módulos, interfaces, tipos, atributos y signaturas de métodos. Hemos mostrado ejemplos de todo lo anterior, excepto módulos, en las Figuras 5.2 y 17.1. IDL tiene las mismas reglas léxicas que C++ pero tiene palabras adicionales para dar soporte a la distribución, por ejemplo *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly* y *raises*. También proporciona las posibilidades de preprocesamiento estándar de C++. Véase, por ejemplo, el *typedef* para *Todo* en la Figura 17.7. La gramática de IDL es un subconjunto de ANSI C++ con constructores adicionales para soportar signaturas de métodos. Aquí mostramos sólo un breve panorama de IDL. En Baker [1997] y Henning y Vinoski [1999] se da una interesante visión de conjunto y muchos ejemplos. La especificación completa se encuentra disponible en [OMG 1997d].

◇ **Módulos IDL.** La construcción módulo permite agrupar en unidades lógicas las interfaces y otras definiciones de tipo IDL. *module* define un alcance léxico, que previene la colisión de los nombres definidos en el interior de un módulo con los nombres definidos fuera de él. Por ejemplo, las definiciones de las interfaces *Forma* y *ListaForma* podrían pertenecer a un módulo denominado *Tablero*, como se indica en la Figura 17.7.

◇ **Interfaces IDL.** Como ya se vio, una interfaz IDL describe los métodos disponibles en los objetos CORBA que implementan esa interfaz. Podrían diseñarse clientes de un objeto CORBA partiendo del único conocimiento de esta interfaz IDL. Estudiando nuestros ejemplos, los lectores verán que una interfaz define un conjunto de operaciones y atributos y depende generalmente de un conjunto de tipos definidos con ésta. Por ejemplo, la interfaz *ListaPersona* de la Figura 5.2 define un atributo y tres métodos y depende del tipo *Persona*.

```
módulo Tablero {
    struct Rectangulo{
    ...} ;
    struct ObjetoGrafico {
    ...};
    interface Forma {
    ...};
    typedef secuencia <Forma, 100> Todo;
    interface ListaForma {
    ...};
};
```

Figura 17.7. Módulo IDL de *Tablero*.

◇ **Métodos IDL.** La forma general de la signatura de un método es:

[oneway] <tipo_devuelto> <nombre_método> (parámetro1, ..., parámetroL)
[raises (excepc1, ..., excepcN)] [contexto (nombre1, ..., nombreM)]

donde las expresiones entre corchetes son opcionales. Como ejemplo de un método que contenga sólo las partes necesarias, considere:

void damePersona(in string nombre, out Persona p);

Según se explicó en la Sección 17.2, los parámetros se etiquetan como *in*, *out* o *inout*, donde el valor de un parámetro *in* se pasa desde el cliente al objeto CORBA invocado y el valor de un parámetro *out* se devuelve desde el objeto CORBA invocado hacia el cliente. Los parámetros etiquetados como *inout* se emplean raramente, e indican que su valor puede pasarse en ambas direcciones. El tipo de valor retornado puede especificarse como *void* si no se devuelve ningún valor.

La expresión *oneway* indica que el cliente que invoca el método no se bloqueará mientras el objeto destino lleva a cabo el método. Además, las invocaciones *oneway* se realizan con la semántica de llamadas *pudiera ser*. En la sección 17.2.1 vimos el siguiente ejemplo:

oneway void retrollamada(in int version);

En este ejemplo, donde el servidor llama al cliente cada vez que se añade una nueva forma, la pérdida ocasional de una petición no es un problema para el cliente, dado que la llamada indica exáctamente que es improbable que se pierdan el número de la última versión y las llamadas subsiguientes.

La expresión opcional *raises* indica excepciones definidas por el usuario que pueden lanzarse para terminar la ejecución del método. Por ejemplo, considere el siguiente ejemplo de la Figura 17.1:

exception ExcepcionLlena{ };
Forma nuevaForma(in ObjetoGrafico g) raises (ExcepcionLlena);

El método *nuevaForma* especifica mediante la expresión *raises* que puede lanzar una excepción llamada *ExcepcionLlena*, definida dentro de la interfaz *ListaForma*. En nuestro ejemplo, la excepción no contiene variables, Sin embargo, pueden definirse excepciones que contengan variables, por ejemplo:

exception ExcepcionLlena{ ObjetoGrafico g };

Tipo	*Ejemplos*	*Uso*
sequence	*typedef sequence <Forma, 100> Todo;* *typedef sequence <Forma> Todo;* secuencias de *Formas* con límite y sin límite	Define un tipo para una secuencia de longitud variable de elementos de un tipo IDL especificado. Puede especificarse una talla superior límite.
string	*string nombre;* *typedef string <8> CadenaCorta;* secuencias de caracteres con límite y sin límite	Define secuencias de caracteres, terminadas en el carácter nulo. Puede especificarse una talla superior límite.
array	*typedef octet idUnico[12];* *typedef ObjetoGrafico OG[10][8];*	Define un tipo para una secuencia multidimensional de talla fija de elementos de un tipo IDL especificado.
record	*struct ObjetoGrafico {* *string tipo;* *Rectangulo enmarcado;* *boolean estaRelleno;* *};*	Define un tipo para un registro que contiene un grupo de entidades relacionadas. Cada *struct* se pasa por valor en los argumentos y en los resultados.
enumerated	*enum Arbitr* *(Exp, Numero, Nombre);*	El tipo enumerado en IDL hace corresponder un nombre de tipo sobre un pequeño conjunto de valores enteros.
union	*union Exp switch (Arbitr)* *case Exp: string voto;* *case Numero: long n;* *case Nombre: string s;* *};*	La unión IDL discriminada permite pasar como argumento uno de un conjunto de tipos dado. La cabecera está parametrizada por un *enum*, que especifica qué miembro está en uso.

Figura 17.8. Tipos IDL construidos.

Cuando se lanza una excepción que contiene variables, el servidor puede utilizar las variables para devolver información al cliente acerca del contexto de la excepción.

CORBA puede producir también excepciones de sistema sobre los problemas con los servidores, tales como estar demasiado ocupados o la imposibilidad de ser activados, problemas con la comunicación y problemas del lado del cliente. Los programas cliente deberían gestionar las excepciones de usuario y de sistema. La expresión opcional *context* se utiliza para aportar relaciones entre cadenas de nombres y cadenas de texto de valores. Vea Baker [1997] para una explicación de *context*.

◇ **Tipos IDL.** IDL soporta quince tipos primitivos, que incluyen *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits), y *any* (que podrá representar cualquier tipo primitivo o construido). Mediante la palabra clave *const*, es posible declarar constantes de la mayoría de los tipos primitivos y constantes de cadenas de texto. IDL proporciona un tipo especial llamado *Object*, cuyos valores son referencias a objetos remotos. Si un parámetro o el resultado es de tipo *Object*, entonces el argumento correspondiente podría referirse a cualquier objeto CORBA.

Los tipos IDL construidos se describen en la Figura 17.8, siendo que todos ellos se pasan por valor como argumentos y como resultados. Todas las cadenas o secuencias que se empleen como argumentos deben definirse en un *typedef*. Ninguno de los tipos de datos primitivos o construidos podrán contener referencias.

◇ **Atributos.** Las interfaces IDL pueden tener tanto atributos como métodos. Los atributos son como los campos de clase públicos en Java. Los atributos se pueden definir como *readonly* (sólo

lectura) allá donde se necesite. Los atributos son privados a los objetos CORBA, pero se genera un par de métodos de acceso, para cada atributo declarado, de modo automático por el compilador de IDL. Para los atributos *readonly*, sólo se proporciona el método de lectura. Por ejemplo, la interfaz *ListaPersona* definida en la Figura 5.1 incluye la siguiente definición de atributo:

readonly attribute string listanombre;

◇ **Herencia.** Las interfaces IDL se pueden extender. Por ejemplo, si la interfaz *B* extiende la interfaz *A*, quiere decir que pudiera haber añadido tipos, constantes, excepciones, métodos y atributos nuevos a los de *A*. Una interfaz extendida puede redefinir tipos, constantes y excepciones, pero no podrá redefinir métodos. Un valor de un tipo extendido es válido como valor de parámetro o resultado del tipo padre. Por ejemplo, el tipo *B* es válido como valor de un parámetro o resultado del tipo *A*.

interface A { };
interface B: A{ };
interface C {};
interface Z: B, C {};

Además, una interfaz IDL podrá extender más de una interfaz. Por ejemplo, la interfaz *Z*, extiende *B* y *C*. Esto indica que *Z* tiene todos los componentes de ambos, *B* y *C* (aparte de aquellos que redefine) así como aquellos que define como extensión.

Cuando una interfaz, como *Z*, extiende más de una interfaz, existe la posibilidad de que pueda heredar un tipo, constante o excepción con el mismo nombre desde dos interfaces diferentes. Por ejemplo, suponga que ambos, *B* y *C*, definen un tipo denominado *Q*; entonces el uso de *Q* en la interfaz *Z* será ambiguo a menos que se proporcione un nombre cualificado como *B*::*Q* o *C*::*Q*. IDL no permite que una interfaz herede métodos o atributos con nombres comunes desde dos interfaces diferentes.

Todas la interfaces IDL heredan del tipo *Object*, lo que implica que todas la interfaces IDL son compatibles con el tipo *Object*. Esto posibilita el definir operaciones IDL que puedan tomar como argumento o devolver como resultado una referencia a un objeto remoto de cualquier tipo. Las operaciones *bind* y *resolve* del Servicio de Nombres son ejemplo de ello.

◇ **Nombres de tipo IDL.** En la Sección 17.2.2 se mencionó que para cada tipo de una interfaz IDL se generan identificadores de tipo únicos, mediante el compilador IDL. Por ejemplo, el tipo IDL para el tipo *Forma* de la interfaz (véase la Figura 17.7) podría ser:

IDL:Tablero/Forma:1.0

Este ejemplo ilustra las tres partes del nombre de un tipo IDL: el prefijo IDL, un nombre de tipo y un número de versión. Los desarrolladores pueden usar el prefijo IDL *pragma* (véase Henning y Vinoski [1999]) para prefijar una cadena de texto adicional al nombre del tipo con el objeto de distinguir sus propios tipos de los del resto de desarrolladores. Los nombres de tipo IDL de las interfaces se incluyen en las referencias a objetos remotos.

◇ **Extensiones a CORBA.** Recientemente, se han añadido algunas características nuevas a la especificación de CORBA. Éstas incluyen la posibilidad de pasar objetos no-CORBA por valor y una variante asíncrona de RMI. Ambas se discuten en el artículo de la *CACM* de Vinoski [1998].

Objetos que pueden pasarse por valor: Como hemos visto antes, los argumentos y los resultados IDL de tipos primitivos o construidos se pasan por valor, mientras que aquellos que se refieren a objetos CORBA se pasan por referencia. Recientemente, se ha añadido a CORBA el soporte para pasar por valor objetos no-CORBA. Un *valuetype* es un *struct* con signaturas de métodos adicionales (como las de una interfaz). Los argumentos y resultados *valuetype* se pasan por valor: esto es, se pasa el estado al lugar remoto y se emplea para producir un nuevo objeto en el destino. Los métodos de este nuevo objeto pueden ser invocados localmente, provocando que su estado pueda diferir del estado del objeto original. El pasar la implementación de los métodos no es algo sencillo, dado que el cliente y el servidor pueden usar lenguajes diferentes. Sin embargo, si el cliente y el servidor están implementados en Java, el código podrá descargarse. Para ser implementado en C++, el código requerido deberá estar ya presente tanto en el cliente como en el servidor.

RMI asíncrono: El RMI síncrono de CORBA se adapta bien para su uso en entornos donde los clientes pueden quedar desconectados temporalmente (como por ejemplo un cliente que use un computador portátil en un tren). La especificación de mensajería de CORBA [OMG 1998d] propone una forma alternativa de RMI para abastecer a aquellos clientes susceptibles de desconectarse temporalmente. Por ejemplo, permite que las peticiones viajen mediante un agente intermediario, que se asegura que se lleva a cabo la petición y si es necesario almacena la respuesta. Además, la semántica de invocación de RMI tiene dos variantes nuevas: *callback*, en la que un cliente pasa una referencia a una devolución de llamada con la invocación, de modo que el servidor pueda devolver la llamada con los resultados; y *polling*, en la que el servidor devuelve un objeto *valuetype* que puede utilizarse para sondear o esperar por la respuesta.

17.2.4. REFERENCIAS A OBJETOS REMOTOS EN CORBA

CORBA 2.0 especifica un formato para las referencias a objetos remotos que es adecuada para su uso tanto si el objeto es activable remotamente como si no. Las referencias que utilizan este formato se denominan referencias a objetos remotos interoperables (IOR, *interoperable object reference*). La siguiente figura se basa en Henning [1998] y contiene el detalle del IOR:

Formato de IOR

Nombre de tipo de interfaz IDL	Protocolo y dirección detallada			Clave de objeto	
Identificador de repositorio de interfaz	IIOP	Nombre de dominio del host	Número de puerto	Nombre de adaptador	Nombre de objeto

- El primer campo de un IOR especifica el nombre de tipo de la interfaz IDL del objeto CORBA. Observe que si el ORB tiene un repositorio de interfaz, este nombre de tipo es también el identificador de la interfaz IDL en el repositorio de interfaz.
- El segundo campo especifica el protocolo de transporte y los detalles necesarios para que ese protocolo de transporte identifique al servidor. En concreto, el protocolo Internet Inter-ORB (IIOP) emplea TCP/IP, en el que la dirección del servidor consta de un nombre de dominio de la máquina y un número de puerto.
- El tercer campo es empleado por el ORB para identificar un objeto CORBA. Consta del nombre de un adaptador de objeto en el servidor y el nombre de objeto de un objeto CORBA especificado por el adaptador de objeto.

El *IOR transitorio* de un objeto CORBA dura sólo el tiempo en que dura el proceso que lo aloja, mientras que un *IOR persistente* perdura entre las activaciones de los objetos CORBA. Un IOR transitorio contiene los detalles de las direcciones del servidor que contiene el objeto CORBA,

mientras que un IOR persistente contendrá los detalles de las direcciones del repositorio de implementación. En ambos casos, el ORB cliente envía el mensaje de petición al servidor cuya dirección pormenorizada proporciona el IOR. A continuación describiremos cómo se utiliza el IOR para encontrar el sirviente que representa el objeto CORBA en ambos casos.

IOR transitorio: el núcleo del ORB del servidor recibe el mensaje de petición que contiene el nombre del adaptador de objeto y el nombre del destino. Emplea el nombre del adaptador del objeto para localizar el adaptador de objeto, el cual utiliza el nombre del objeto para localizar el sirviente.

IOR persistente: un repositorio de implementación recibe la petición. Extrae el nombre del adaptador de objeto del IOR de la petición. Dado que el nombre del adaptador de objeto está en su tabla, intenta, si es necesario, activar el objeto CORBA de la dirección del host especificado. Una vez activado el objeto CORBA, el repositorio de implementación devuelve su dirección detallada al ORB del cliente, que lo emplea como destino de los mensajes de petición RMI, lo que incluye el nombre del adaptador de objeto y el nombre del objeto. Esto permite al núcleo del ORB del servidor encontrar el adaptador de objeto, el cual emplea el nombre del objeto para localizar al sirviente, como antes.

El segundo campo, en un IOR que especifique el nombre de dominio del host y el número de puerto, puede aparecer varias veces con el fin de permitir que un objeto o un repositorio de implementación esté replicado en varias ubicaciones diferentes.

El mensaje de respuesta en el protocolo petición-respuesta incluye un encabezamiento con información que permite llevar a cabo el mecanismo de IOR persistente. En particular, incluye una entrada de estatus que puede indicar si la petición debiera dirigirse a un servidor diferente, en cuyo caso el cuerpo de la respuesta incluye un IOR que contiene la dirección del servidor del objeto recientemente activado.

17.2.5. CORRESPONDENCIA CON EL LENGUAJE EN CORBA

Hemos visto en nuestros ejemplos que la correlación de los tipos en el IDL con los tipos en Java es bastante directa. Los tipos primitivos en IDL se corresponden con tipos primitivos en Java. Los *struct*, *enum* y *union* se escriben como clases Java; las secuencias y las cadenas en IDL se corresponden con las cadenas (*array*) en Java. Una excepción de IDL se corresponde con una clase Java que proporciona variables de instancia a los campos de la excepción y los constructores. La relación con C++ es igualmente directa.

Sin embargo, hemos visto que aparecen algunas dificultades con la correspondencia en la semántica de paso de parámetros de IDL con Java. En particular, IDL permite que los métodos devuelvan varios valores separados vía parámetros de salida, mientras que Java sólo puede tener un solo resultado. Para soslayar esta diferencia se proporcionan clases *Holder* (propietario), que aun así requiere que el programador haga un uso explícito de ellas, lo cual no es directo en la misma medida que lo anterior. Por ejemplo, el método *damePersona* de la Figura 5.2 se define en IDL como sigue:

void damePersona(in string nombre, out Persona p);

y el método equivalente en la interfaz Java quedaría definido como:

void damePersona(String nombre, HolderPersona p);

y el cliente debe proporcionar una instancia de *HolderPersona* como argumento de su invocación. La clase propietaria tiene una variable de instancia que sustenta el valor del argumento para que el cliente acceda a éste cuando retorna la invocación. También tiene métodos para transmitir el argumento entre el servidor y el cliente.

A pesar de que las implementaciones en C++ de CORBA pueden gestionar parámetros *out* e *inout* de una forma bastante natural, los programadores de C++ padecen un diferente conjunto de problemas con los parámetros, relacionado con la gestión del almacenamiento. Estas penalidades aparecen cuando se pasan como argumentos referencias a objetos y entidades de longitud variable como las cadenas de texto o las secuencias.

Por ejemplo, en Orbix [Baker 1997] el ORB mantiene contadores de referencias a objetos remotos y proxy y los libera cuando no se necesitan más. Proporciona a los programadores métodos para liberarlos o duplicarlos. Cuando un método del servidor ha acabado su ejecución, los argumentos *out* y los resultados se liberan y el programador deberá duplicarlos si aún los necesitara. Por ejemplo, un sirviente C++ que implementara la interfaz *ListaForma* necesitará duplicar las referencias devueltas por el método *todasFormas*. Las referencias a objeto pasadas a los clientes deberán liberarse cuando no se vayan a necesitar. Las mismas reglas se aplican a los parámetros de longitud variable.

En general, los programadores que utilicen IDL no sólo tendrán que aprender la notación IDL en sí, sino que también tendrán que comprender cómo se correlacionan sus parámetros con los parámetros del lenguaje de implementación.

17.3. SERVICIOS DE CORBA

CORBA incluye especificaciones para los servicios que pudieran necesitarse en los objetos distribuidos. En concreto, el Servicio de Nombres es una parte esencial para cualquier ORB, como ya se vio en nuestro ejemplo de programación en esta sección. En el lugar web de OMG en [www.omg.org] se puede encontrar un índice de la documentación de estos servicios. Los servicios de CORBA se describen en Orfali y otros [1996] e incluyen los siguientes:

Servicio de Nombres: el Servicio de Nombres de CORBA se detalla en la Sección 17.3.1.

Servicio de Eventos y *Servicio de Notificación*: el Servicio de Eventos de CORBA (*Event Service*) se discute en la Sección 17.3.2 y el Servicio de Notificación (*Notification Service*) en la Sección 17.3.3.

Servicio de Seguridad: el Servicio de Seguridad de CORBA (*Security Service*) se discute en la Sección 17.3.4.

Servicio de Comercio: en contraste con el Servicio de Nombres que permite buscar los objetos CORBA por nombre, el Servicio de Comercio (*Trading Service*) permite que sean localizados por sus atributos (es un servicio de directorio). Su base de datos contiene una relación de tipos de servicio y sus atributos asociados con referencias a objetos remotos de objetos CORBA. Este tipo del servicio es un nombre, y cada atributo es un par nombre-valor. Los clientes hacen consultas especificando restricciones sobre los valores de los atributos, y sus preferencias del orden en que quieren recibir las ofertas concordantes. Los servidores de comercio pueden formar federaciones en las que no sólo emplean sus propias bases de datos sino que también realizan consultas en representación del cliente de alguna otra. Para una descripción detallada del Servicio de Comercio, véase Henning y Vinoski [1999].

Servicio de Transacciones y Servicio de Control de Concurrencia: el Servicio de Transacción de Objetos (*Object Transaction Service*) permite que los objetos distribuidos CORBA participen en transacciones tanto sencillas como anidadas. El cliente especifica una transacción como una secuencia de llamadas RMI, que comienzan por *begin* (comenzar) y terminan por *commit* (consumar) o *rollback* (abortar, retractar). El ORB adhiere un identificador de transacción a

cada invocación remota y trata con las peticiones *begin*, *commit* y *rollback* (*abort*). Los clientes también pueden suspender y retomar las transacciones. El servicio de transacciones lleva a cabo un protocolo de consumación en dos fases. El Servicio de Control de Concurrencia (*Concurrency Control Service*) emplea bloqueos para aplicar el control de concurrencia al acceso a objetos CORBA. Puede utilizarse desde el interior de las transacciones o aisladamente.

Servicio de Objetos Persistentes: la Sección 5.2.5 explicó que los objetos persistentes pueden almacenarse en una forma pasiva en un almacén de objetos persistentes, mientras no están siendo usados, y pueden activarse cuando se los necesite. A pesar de que un ORB activa objetos CORBA con referencias a objetos persistentes, obteniendo sus implementaciones desde el repositorio de implementación, no se hacen responsables de almacenar y restaurar el estado de los objetos CORBA. En su lugar, cada objeto CORBA es responsable de guardar y recuperar su propio estado, por ejemplo mediante archivos. El Servicio de Objetos Persistentes (*Persistent Object Service*, POS) de CORBA está dispuesto para servir como almacén de objetos persistentes de objetos CORBA. La especificación define interfaces para sus componentes. El POS puede implementarse de variedad de formas, por ejemplo, utilizando archivos o mediante un sistema de base de datos relacional u orientada al objeto.

La arquitectura de POS permite que se disponga de un conjunto de almacenes de datos; cada objeto persistente tiene un identificador persistente que incluye la identidad de su almacén de datos y un número de objeto dentro de ese almacén. Tanto el cliente como el objeto CORBA pueden decidir cuándo almacenar su estado, enviando una petición al POS, que la dirige al almacén de datos apropiado. Para permitir que el cliente controle su persistencia, un objeto CORBA hereda una interfaz que incluye operaciones para almacenarlo, restaurarlo o eliminarlo.

La implementación de un objeto CORBA persistente debe elegir un protocolo de comunicación con un almacén de datos. Por ejemplo, podría transferir sus datos vía un stream. La especificación POS ofrece una selección de protocolos alternativos para esta tarea, proporcionando un abanico de funcionalidades. Por ejemplo, el protocolo de acceso directo permite que los objetos CORBA accedan a los atributos de los objetos persistentes del almacén de datos. En este caso, el programador del objeto CORBA utiliza un lenguaje de definición de datos, al estilo del IDL, para definir los atributos en el estado persistente.

17.3.1. EL SERVICIO DE NOMBRES DE CORBA

El Servicio de Nombres de CORBA es un sofisticado ejemplo del enlazador que se describió en el Capítulo 5. Permite enlazar nombres con referencias a objetos remotos de objetos CORBA con contextos de nombres.

Como ya se explicó en la Sección 9.2, un contexto de nominación es el alcance dentro del que se aplican un conjunto de nombres; cada uno de estos nombres en un contexto será único. Se puede

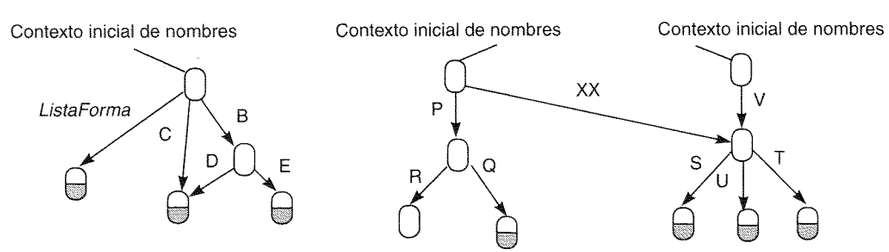


Figura 17.9. Grafo de nombres en el Servicio de Nombres de CORBA.


```
struct NameComponent { string id; string clase; };
typedef sequence <NameComponent> Name;
interface NamingContext {
  void bind(in Name n, in Object obj);
    enlaza el nombre dado y la referencia al objeto remoto en mi contexto.
  void unbind(in Name n);
    elimina un enlace existente con el nombre dado.
  void bind_new_context(in Name n);
    crea un nuevo contexto de nominación y lo enlaza con un nombre dado en mi contexto.
  Object resolve(in Name n);
    busca el nombre en mi contexto y devuelve su referencia a objeto remoto.
  void list(in unsigned long cuantos, out BindingList le, out BindingIterator ie);
    devuelve los nombres de los enlaces en mi contexto.
};
```

Figura 17.10. Parte de la interfaz IDL *NamingContext* del Servicio de Nombres de CORBA.

asociar un nombre tanto con una referencia a un objeto para un objeto CORBA en una aplicación como con otro contexto en el servicio de nombres. Los contextos pueden estar anidados para proporcionar un espacio de nombres jerárquico, como se muestra en la Figura 17.9, en la que se muestran los objetos CORBA al modo habitual, y los contextos de nominación como óvalos sencillos. El grafo de la izquierda muestra una entrada para el objeto *ListaForma* que se describió en el ejemplo de programación de la Sección 17.2.1.

Un *contexto inicial de nominación* proporciona una raíz para un conjunto de enlaces. Observe que más de un contexto inicial de nominación podrá apuntar sobre el mismo grafo. En la práctica, cada instancia de un ORB tiene un único contexto inicial de nominación, pero los servidores de nombres asociados con diferentes ORB pueden formar federaciones, como se describirá más tarde en esta sección. Los programas cliente y servidor solicitan el contexto inicial de nominación desde el ORB, como se muestra en la Figura 17.5, invocando su método *resolve_initial_references*, dándole «*NameService*» como argumento. El ORB devuelve una referencia a un objeto del tipo *NamingContext* (véase la Figura 17.10). Éste se refiere al contexto inicial del servidor de nombres para ese ORB. Dado que puede haber varios contextos iniciales, los objetos no tienen nombres absolutos: los nombres siempre se interpretan con respecto a un contexto inicial de nominación.

Se puede resolver cualquier nombre con uno o más componentes, comenzando en cualquier contexto de nominación. Para resolver un nombre con varios componentes, el servicio de nombres busca en el contexto de arranque por un enlace que concuerde con el primer componente. Si tal enlace existe, será bien una referencia a un objeto remoto o una referencia a otro contexto de nominación. Si el resultado es un contexto de nominación, se resuelve el segundo nombre en ese contexto. El procedimiento se repite hasta que se hayan resuelto todos los componentes del nombre y se obtenga una referencia a un objeto remoto, a menos que falle la concordancia por el camino.

Los nombres que emplea el Servicio de Nombres de CORBA son del tipo *NameComponent*, y constan de dos cadenas, una para el *nombre* y otra para la *clase* del objeto. El campo *clase* proporciona un solo atributo cuyo uso se previó para las aplicaciones y puede contener cualquier información descriptiva de utilidad; no es interpretado por el Servicio de Nombres.

A pesar de que a los objetos CORBA se les dan nombres jerárquicos por parte del Servicio de Nombres, estos nombres no pueden expresarse como rutas similares a las de los archivos en UNIX. Así, en la Figura 17.9, no podemos hacer referencia al objeto del extremo de la derecha como /V/T. Esto es así porque los nombres pueden contener cualquier carácter, lo que elimina la posibilidad de contar con un delimitador.

La Figura 17.10 da cuenta de las operaciones principales que proporciona la clase *NamingContext* del Servicio de Nombres de CORBA, definida en CORBA IDL. La especificación completa puede obtenerse de OMG [1997b]. Por simplicidad, nuestra figura no incluye las excepciones lanzadas por los métodos. Por ejemplo, el método *resolve* puede lanzar una excepción *NotFound*, y *bind* puede lanzar una excepción *AlreadyBound*.

Los clientes emplean el método *resolve* para buscar referencias a objetos por su nombre. El tipo del valor de vuelta es *Object*, de este modo puede devolver referencias a cualquier tipo de objeto perteneciente a cada aplicación. El tipo del resultado deberá ser estrechado (*narrow*) antes de poder ser utilizado para invocar un método en un objeto remoto de la aplicación, según se indica en la Figura 17.5. El argumento de *resolve* es de tipo *Name*, que está definido como una secuencia de componentes de nombre. Esto viene a decir que el cliente debe construir una secuencia de componentes de nombre antes de realizar la llamada. La Figura 17.5 mostraba un cliente construyendo una cadena denominado *ruta* que consta de un solo nombre componente de nombre, que se utiliza como argumento para *resolve*. Esto no parece una alternativa demasiado conveniente al uso de una ruta de nombres normal.

Los servidores de los objetos remotos emplean la operación *bind* para registrar los nombres de sus objetos y *unbind* para eliminarlos. La operación *bind* enlaza cierto nombre con una referencia a un objeto remoto y se invoca en el contexto en el que se añade el enlace. En la Figura 17.4, el nombre *ListaForma* se enlaza en el contexto inicial de nombres. En ese ejemplo, se emplea el método *rebind* dado que la operación *bind* lanzará una excepción si se le llama con un nombre que ya tiene un enlace, mientras que *rebind* permite remplazar enlaces.

La operación *bind_new_context* se usa para crear un contexto nuevo y para enlazarlo con el nombre dado en el contexto sobre el que fue invocada. Otro método llamado *bind_context* enlaza un nombre de contexto dado a un nombre dado en el contexto sobre el que fue invocado. El método *unbind* puede usarse para tanto contextos como nombres.

La operación *list* está preparada para ojear la información disponible en un contexto en el Servicio de Nombres. Devuelve una lista de enlaces desde el *NameContext* objetivo. Cada enlace consta de un nombre y un tipo; un objeto o un contexto. Algunas veces, un nombre de contexto puede contener un número muy grande de enlaces, en cuyo caso sería deseable que pudiera proporcionarlos como resultado de una sola invocación. Por esta razón, el método *list* devuelve cierto número máximo de enlaces como resultado de la llamada a *list*, si es el caso de que faltan enlaces por enviar, se puede arreglar para que envíe los resultados en tandas. Esto es posible dado que retorna un iterador como resultado secundario. El cliente usa el iterador para recuperar el resto de los resultados, uno cada vez.

El método *lista* se muestra en la Figura 17.10, pero se han omitido las definiciones de los tipos de sus argumentos en virtud de la simplicidad. El tipo *BindingList* es una secuencia de enlaces, cada uno de los cuales contiene un nombre y su tipo, que es bien un contexto o bien una referencia a un objeto remoto. El tipo *BindingIterator* proporciona un método *next_n* para acceder al próximo conjunto de enlaces; su primer argumento especifica cuántos enlaces se desean y en el segundo recibimos una secuencia de enlaces. El cliente llama al método *lista* dándole como primer argumento el número máximo de enlaces que se recibirán inmediatamente a través del segundo argumento. El tercer argumento es un iterador, que podrá ser usado para obtener el resto de los enlaces, si los hubiera.

El espacio de nombres de CORBA permite la federación de Servicios de Nombres, usando un esquema en el que cada servidor proporciona un subconjunto del grafo de nombres. Por ejemplo, en la Figura 17.9, el contexto inicial de nombres en el grafo del medio y en el de la derecha se maneja desde servidores diferentes. El grafo del medio tiene un enlace etiquetado *XX* hacia un contexto en el grafo de la derecha por el que los clientes podrían acceder a los objetos con nombre del grafo remoto. Para completar esta federación, el grafo de la derecha necesitaría añadir un enlace a un nodo del grafo del medio. Una organización puede proporcionar acceso a algunos o a todos los contextos en su espacio de nombres si proveen servidores de nombres remotos con referencias remotas a ellos.

La implementación Java del Servicio de Nombres de CORBA es muy simple y se la denomina transitoria porque almacena todos sus enlaces en memoria volátil. Cualquier implementación seria debería, al menos, conservar en archivos copias de sus grafos de nominación. Tal y como hemos visto en el estudio de DNS, se puede proporcionar replicación para proporcionar una mejor disponibilidad.

17.3.2. SERVICIO DE EVENTOS DE CORBA

La especificación del Servicio de Eventos de CORBA define interfaces que permiten a los objetos de interés, denominados *proveedores* (*supplier*), comunicar notificaciones a sus subscriptores, llamados *consumidores* (*consumer*). Las notificaciones se comunican como argumentos o resultados de invocaciones síncronas ordinarias de métodos remotos CORBA. Las notificaciones pueden propagarse tanto *impulsadas* (*push*) por el proveedor hacia el consumidor como *extraídas* (*pull*) por parte del consumidor desde el proveedor. En el primer caso, el consumidor implementa la interfaz *PushConsumer* que incluye un método *push* que toma cualquier tipo de dato CORBA como argumento. Los consumidores registran sus referencias a objetos remotos con los proveedores. Los proveedores invocan el método *push*, pasando una notificación como argumento. En el segundo caso, el proveedor implementa la interfaz *PullSupplier*, que incluye un método *pull* que recibe cualquier tipo de dato CORBA como valor de retorno. Los proveedores registran sus referencias a objetos remotos frente a los consumidores. Los consumidores invocan el método *pull* y reciben como resultado una notificación.

La notificación en sí misma se transmite como un argumento o resultado cuyo tipo es *any*, que indica que los objetos que intercambian notificaciones deben haberse puesto de acuerdo sobre los contenidos de las notificaciones. Los programadores de aplicaciones, sin embargo, podrán definir sus propias interfaces IDL con notificaciones de cualquier tipo deseado.

Los *canales de eventos* son objetos CORBA que pueden emplearse para permitir que múltiples proveedores se comuniquen con múltiples consumidores de modo asíncrono. Un canal de eventos actúa como un búfer entre los proveedores y los consumidores. También puede multidifundir las notificaciones a los consumidores. La comunicación vía canal de eventos podrá ser mediante impulsión o mediante extracción. Es posible mezclar ambos estilos; por ejemplo, los proveedores podrán impulsar notificaciones hacia el canal y los consumidores podrán extraer notificaciones de él.

Cuando una aplicación distribuida necesita emplear notificaciones asíncronas, crea un canal de eventos, que es un objeto CORBA cuya referencia a objeto remoto podrá ser aportada a los componentes de la aplicación a través del Servicio de Nombres o mediante un RMI. Los procesos proveedores en la aplicación ofrecen ellos mismos las suscripciones obteniendo *los proxy de consumidor* desde el canal de eventos y conectando los proveedores a ellos pasándoles sus referencias a objetos remotos. Los procesos consumidores de la aplicación se suscriben a las notificaciones obteniendo *los proxy de proveedor* del canal de notificación y conectando los consumidores a ellos. Los proxy de proveedor y consumidor se encuentran disponibles tanto en el estilo de impulsión como en el de extracción. Cuando un proveedor genera una notificación usando el estilo de interacción por impulsión, llama al método *push* del proxy de un consumidor por impulsión. La notificación pasa a través del canal y se entrega a los proxy de los consumidores, que las pasa a los consumidores, como se indica en la Figura 17.11. Si los consumidores emplean el estilo de interacción por extracción, llamarán al método *pull* del proxy de un proveedor por extracción.

La presencia de proxy de proveedores y proxy de consumidores hace posible construir cadenas de canales de eventos en las que cada canal suministra notificaciones que serán consumidas por el

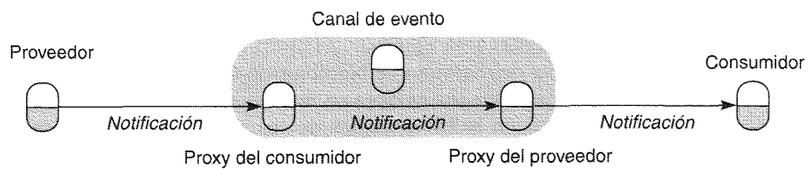


Figura 17.11. Canales de eventos en CORBA.

siguiente canal. Los canales de eventos en el modelo CORBA son similares a los observadores definidos en la Figura 5.10. Éstos pueden programarse para llevar a cabo algunos de los papeles de los observadores que se discutieron en la Sección 5.4. Sin embargo, las notificaciones no conllevan forma alguna de identificadores, y así los patrones de reconocimiento o de filtrado de notificaciones deberán basarse en la información de tipo puesta en las notificaciones por la aplicación.

En Farley [1998] se proporciona una explicación más detallada del Servicio de Eventos de CORBA y un bosquejo de sus interfaces principales. La especificación completa del Servicio de Eventos de CORBA está en [OMG 1997c]. A pesar de ello, la especificación no muestra cómo crear un canal de eventos ni cómo solicitar la fiabilidad que se requiere de él.

17.3.3. SERVICIO DE NOTIFICACIÓN DE CORBA

El Servicio de Notificación de CORBA [OMG 1998c] extiende el Servicio de Eventos de CORBA, conservando todas sus características, incluyendo canales de eventos, consumidores de eventos y proveedores de eventos. El servicio de eventos no proporciona ningún soporte para el filtrado de eventos o para especificar requisitos de reparto. Sin el uso de filtros, todos los consumidores conectados a un canal tendrán que recibir las mismas notificaciones que cualquier otro. Y sin la posibilidad de especificar requisitos de reparto, todas las notificaciones que se envíen a través de este canal tendrán las garantías de reparto incluidas en la implementación.

El servicio de notificación añade las siguientes nuevas funciones:

- Las notificaciones se pueden definir como estructuras de datos. Esto es una mejora de la utilidad limitada que proporcionaban las notificaciones en el servicio de eventos, cuyo tipo sólo podía ser *any* o un tipo especificado por el programador de la aplicación.
- Los consumidores de eventos pueden usar filtros que especifican exactamente en qué eventos están interesados. Los filtros pueden adherirse a cada proxy de un canal. Cada proxy dirigirá las notificaciones a los consumidores de eventos según las restricciones especificadas en los filtros, en términos de los contenidos de cada notificación.
- Los proveedores de eventos poseen mecanismos para descubrir en qué eventos están interesados los consumidores. Esto les permite generar sólo aquellos eventos que requieran los consumidores.
- Los consumidores de eventos pueden descubrir los tipos de eventos que ofrecen los proveedores en un canal, lo que les permite suscribirse a nuevos eventos según se hagan disponibles.
- Es posible configurar las propiedades de un canal, un proxy o un evento particular. Estas propiedades incluyen la confiabilidad del reparto de eventos, la prioridad de los eventos, el ordenamiento requerido (por ejemplo, FIFO o por prioridad) y la política para descartar eventos almacenados.
- Como extra adicional se da un repositorio de tipos de eventos. Proporciona acceso a la estructura de eventos, lo que le hace conveniente para definir restricciones de filtrado.

El tipo Evento Estructura presentado por el servicio de notificación proporciona una estructura de datos en la que encajan una gran variedad de tipos de notificación diferentes. Se pueden definir filtros en términos de las componentes de un tipo de Evento Estructurado. Un evento estructurado consta de una cabecera de evento y un cuerpo de evento. La cabecera contiene:

Tipo de dominio	*Tipo de evento*	*Nombre de evento*	*Requisito*
«casa»	«alarma antirrobo»	«21 marzo, 2 pm»	«prioridad», 1.000

El tipo de dominio se refiere al dominio de definición (por ejemplo, «*finanzas*», «*hotel*» u «*hogar*»). El tipo de evento categoriza el tipo del evento de modo único dentro del dominio (por ejemplo, «*cotización stock*», «*hora desayuno*», «*alarma antirrobo*»). El nombre de evento identifica de modo único la instancia específica del evento que está siendo transmitido. El resto de la cabecera contiene una lista de pares <nombre, valor>, proyectadas para ser usadas al especificar la fiabilidad y otros requisitos sobre el reparto de eventos.

El cuerpo de un evento estructurado contiene la siguiente información:

Parte filtrable			
Nombre, valor	*Nombre, valor*	*Nombre, valor*	*Resto*
«campana», «sonando»	«puerta», «abierta»	«gato», «fuera»	

La primera parte del cuerpo del evento contiene una secuencia de pares <nombre, valor> proyectadas para su uso por los filtros. Se espera que diferentes áreas de la industria definan estándares para los pares <nombre, valor> que se usan en la parte filtrable del cuerpo del evento; al definir los filtros se emplearán los mismos nombres y valores. Quizás cuando la alarma antirrobo se apague, el evento pueda incluir el estado de la campana de alarma, si la puerta principal está abierta y la situación del gato. El resto del cuerpo del evento está propuesto para transmitir datos referentes a un evento particular; por ejemplo, cuando la alarma antirrobo se apague, pudiera contener una fotografía digital del interior de las estancias.

Los objetos filtro se emplean desde los proxy para realizar decisiones sobre si encaminar o no cada notificación. Un filtro se diseña como una colección de restricciones, cada una de las cuales es una estructura con dos componentes:

- Una lista de estructuras de datos, cada una de las cuales indica un tipo de evento en términos de su nombre de dominio y tipo de evento, por ejemplo «*hogar*», «*alarma antirrobo*». La lista incluye todos los tipos de eventos a los que se pueda aplicar la restricción.
- Una cadena de texto que contiene una expresión booleana que incluye los tipos de eventos listados anteriormente. Por ejemplo:

  («domain type» = = «hogar» && «event type» = = «alarma antirrobo») &&
  («campana» ! = «sonando» !! «puerta» = = «abierta»)

Nuestro ejemplo emplea una sintaxis informal. La especificación del servicio de notificación incluye la definición de un lenguaje de restriccioens, que es una extensión del lenguaje de restricciones empleado por el Servicio de Comercio.

17.3.4. SERVICIO DE SEGURIDAD DE CORBA

El servicio de Seguridad de CORBA [Blakly 1999, Baker 1997, OMG 1998b] incluye lo siguiente:

- Autenticación de principales (usuarios y servidores); generando credenciales para los principales (esto es, certificados confirmando sus derechos); la delegación de credenciales está soportada en los términos de la Sección 7.2.5.
- Se puede aplicar control de acceso a los objetos CORBA cuando reciben invocaciones remotas. Los derechos de acceso se pueden especificar, por ejemplo, mediante listas de control de acceso (ACL).
- Auditoría de las invocaciones a métodos remotos por parte de los servidores.
- Medios para el no repudio. Cuando un objeto lleva a cabo una invocación remota en nombre de un principal, el servidor crea y almacena las credenciales que prueban que se hizo la invocación al servidor en representación del principal peticionario.

Para garantizar la aplicación correcta de la seguridad a las invocaciones de métodos remotos, el servicio de seguridad requiere cooperación en nombre del ORB. Para realizar una invocación segura a un método remoto, se envían las credenciales del cliente en el mensaje de petición. Cuando el servidor recibe un mensaje de petición, se validan las credenciales del cliente para ver, por ejemplo, si son actuales y están firmadas por una autoridad aceptable. Si las credenciales son válidas, se utilizan para decidir si el principal tiene derecho a acceder al objeto remoto utilizando el método del mensaje de petición. Esta decisión se hace consultando un objeto que contiene información sobre qué principales tienen derecho a acceder a cada método del objeto destino (posiblemente en forma de un ACL). Si el cliente tiene suficientes derechos, se lleva a cabo la invocación y se devuelve el resultado al cliente, junto con las credenciales del servidor, si fuera necesario. El objeto destino también podría almacenar los detalles de la invocación en un histórico o guardar las credenciales de no repudio.

CORBA permite especificar una variedad de políticas de seguridad según los requisitos. Una política de protección del mensaje declara: si deben autenticarse el cliente o el servidor (o ambos), y si debe protegerse la privacidad y/o la integridad de los mensajes. También se pueden especificar políticas con respecto a la auditoría y al no repudio; por ejemplo, una política podría declarar a qué métodos y argumentos deberían aplicarse.

El control de acceso tiene en cuenta que muchas aplicaciones tienen un gran número de usuarios y mayor número incluso de objetos, cada uno de ellos con su propio conjunto de métodos. Se proporciona a los usuarios un tipo especial de credencial, denominada privilegio, a la medida de sus funciones. Los objetos se agrupan en *dominios*. Cada dominio tiene una única política de control de acceso que especifica los derechos de acceso a los objetos del dominio de conjuntos de usuarios con privilegios concretos. Para dar cabida a una variedad impredecible de métodos, cada método se clasifica en términos de uno de cuatro métodos genéricos (*get*, *set*, *use* y *manage*). Los métodos *get* sólo devuelven partes del estado de un objeto, los métodos *set* alteran el estado del objeto, los métodos *use* hacen que el objeto realice algún trabajo, y los métodos *manage* realizan funciones especiales no proyectadas para el uso general. Dado que los objetos CORBA tienen una variedad de interfaces diferentes, hay que especificar los derechos de acceso de cada nueva interfaz en términos de los métodos genéricos anteriores. Esto implica que los diseñadores de la aplicación estarán involucrados en la aplicación del control de acceso, la determinación de los atributos de privilegio apropiados (por ejemplo, grupos o funciones) y en ayudar al usuario en la obtención de los privilegios apropiados para su tarea.

En su forma más simple, se puede aplicar seguridad de forma que sea transparente a las aplicaciones. Esto incluye la aplicación de la política de protección requerida para las invocaciones a métodos remotos, junto con la auditoría. El servicio de seguridad permite que los usuarios obtengan sus credenciales y privilegios individuales en respuesta a la aportación de datos de autenticación tales como claves de acceso.

17.4. RESUMEN

El principal componente de CORBA es el Intermediario de Peticiones de Objetos (*Object Request Broker*) u ORB, que permite que clientes escritos en un lenguaje invoquen operaciones de objetos remotos (denominados objetos CORBA) escritos en otro lenguaje. CORBA trata otros aspectos de la heterogeneidad como sigue:

- El protocolo General Inter-ORB (GIOP) de CORBA incluye una representación externa de datos llamada CDR, que hace posible que los clientes y los servidores se comuniquen independientemente de su hardware. También especifica una forma estándar para las referencias a objetos remotos.
- GIOP también incluye una especificación para las operaciones de un protocolo petición-respuesta que pueda ser usado independientemente del sistema operativo subyacente.
- El protocolo Internet Inter-ORB (IIOP) implementa el protocolo petición-respuesta sobre TCP/IP. Las referencia a objetos remotos IIOP incluyen el nombre de dominio y el número de puerto de un servidor.

Un objeto CORBA implementa las operaciones de una interfaz IDL. Todo lo que los clientes necesitan saber para acceder a un objeto CORBA es el conjunto de operaciones disponible en su interfaz. El programa cliente accede a los objetos CORBA vía proxy o resguardo, que se generan automáticamente desde sus interfaces IDL hacia el lenguaje del cliente. Los esqueletos del servidor para los objetos CORBA se generan automáticamente desde sus interfaces IDL hacia el lenguaje del servidor. El adaptador de objeto es un componente importante de los servidores CORBA. Su papel es crear referencias a objetos remotos y relacionar referencias a objetos remotos de los mensajes de petición con las implementaciones de los objetos CORBA.

La arquitectura de CORBA permite activar los objetos CORBA bajo demanda. Esto se logra mediante un componente denominado repositorio de implementación, que aloja una base de datos de las implementaciones indexadas por sus nombres de adaptadores de objeto. Cuando un cliente invoca un objeto CORBA, puede activarse si fuera necesario para llevar a cabo la invocación.
Un repositorio de interfaz es una base de datos de definiciones de interfaces IDL indexada por un ID de repositorio, que está incluida en un IOR. Se puede utilizar un repositorio de interfaz para obtener información sobre los métodos de la interfaz de un objeto CORBA para permitir la invocación dinámica de métodos.

Los servicios CORBA proporcionan funcionalidades sobre RMI, que pudieran necesitarse en las aplicaciones distribuidas, permitiéndolas utilizar servicios adicionales como servicios de nombres y de directorio, notificaciones de eventos, transacciones o seguridad según se desee.

EJERCICIOS

17.1. La Bolsa de Tareas es un objeto que almacena pares de clave, valor. Una clave es una cadena de texto y un valor es una secuencia de bytes. Su interfaz proporciona los siguientes métodos remotos:

parSalida: con dos parámetros con los que el cliente especifica que se almacene una *clave* y un *valor*.

parEntrada: cuyo primer parámetro permite que el cliente especifique una *clave* de un par que se eliminará de la Bolsa de Tareas. El *valor* del par se devuelve al cliente vía un segúndo parámetro. Si no existe el par correspondiente, se lanza una excepción.

leePar: es lo mismo que *parEntrada* excepto que el par permanece en la Bolsa de Tareas.

Utilice CORBA IDL para definir la interfaz de la Bolsa de Tareas. Defina una excepción que pueda lanzarse cuando no pueda llevarse a cabo cualquiera de las operaciones. Su excepción debería devolver un entero que indique el número del problema y una cadena de texto que lo describa. La interfaz de la bolsa de Tareas debería definir sólo un atributo que proporciona el número de tareas en la bolsa.

17.2. Defina una signatura alternativa para los métodos *parEntrada* y *leePar*, cuyo valor de retorno indique cuándo no hay un par que concuerde. El valor de retorno debería definirse como un tipo enumerado cuyos valores pueden ser *ok* y *espera*. Discuta los méritos relativos de las dos aproximaciones alternativas. ¿Qué aproximación utilizaría usted para indicar un error tal como que una clave contiene caracteres ilegales?

17.3. ¿Cuál de los métodos de la interfaz de la Bolsa de Tareas podría definirse como una operación *oneway*? De una regla general con respecto a los parámetros y excepciones de los métodos *oneway*. ¿En qué forma difiere el significado de la palabra clave *oneway* del resto del IDL?

17.4. El tipo *union* del IDL puede usarse para un parámetro que necesite pasar uno de un pequeño número de tipos. Utilícelo para definir el tipo de un parámetro que a veces está vacío y a veces es del tipo *Valor*.

17.5. En la Figura 17.1 el tipo de *Todo* se definió como una secuencia de longitud fija. Redefina éste como una cadena de la misma longitud. Dé algunas recomendaciones para elegir entre cadenas y secuencias en una interfaz IDL.

17.6. La Bolsa de Tareas está pensada para utilizarse por clientes cooperantes, algunos de los cuales añaden pares (que describen tareas) o otros los eliminan (y realizan las tareas descritas). Cuando se informa a un cliente de que no hay ningún par de tareas disponible, no podrá continuar con su trabajo hasta que haya alguno disponible. Defina una interfaz de devolución de llamada apropiada para su uso en esta situación.

17.7. Describa las modificaciones necesarias a la interfaz de la Bolsa de Tareas para permitir el uso de devoluciones de llamadas.

17.8. ¿Cuáles de los parámetros de los métodos de la interfaz de la Bolsa de Tareas se pasan por valor y cuáles se pasan por referencia?

17.9. Utilice el compilador de IDL de Java para procesar la interfaz que definió en el Ejercicio 17.1. Inspeccione la definición de las signaturas de los métodos *parEntrada* y *leePar* en el código Java equivalente generado por la interfaz IDL. Observe también la definición generada para el método propietario para el argumento *valor* en los métodos *parEntrada* y *leePar*. Ahora dé un ejemplo mostrando cómo invocaría el cliente el método *parEntrada*, explicando cómo adquirirá el valor devuelto mediante el segundo argumento.

17.10. Dé un ejemplo que muestre cómo un cliente Java accederá al atributo que da el número de tareas en el objeto Bolsa de Tareas. ¿En qué difiere un atributo de una variable de instancia de un objeto?

17.11. Explique por qué las interfaces a los objetos remotos en general y a los objetos CORBA en particular no proporcionan constructores. Explique cómo se pueden crear los objetos CORBA en ausencia de constructores.

17.12. Redefina la interfaz de la Bolsa de Tareas del Ejercicio 17.1 en IDL de modo que haga uso de *struct* para representar un *Par*, que consiste en una *Clave* y un *Valor*. Observe que no hay necesidad de usar *typedef* para definir un *struct*.

17.13. Discuta las funciones del repositorio de implementación desde el punto de vista de la escalabilidad y la tolerancia a fallos.

17.14. ¿Hasta qué punto se pueden migrar objetos CORBA de un servidor a otro?

17.15. Discuta los beneficios y desventajas de los nombres bipartitos o *NameComponent* en el Servicio de Nombres de CORBA.

17.16. Dé un algoritmo que describa cómo se resuelve un nombre multiparte en el Servicio de Nombres de CORBA. Un programa cliente necesita resolver un nombre multiparte con componentes «A», «B» y «C», relativos a un contexto inicial de nominación. ¿Cómo se especificarían los argumentos para la operación *resolve* en el servicio de nombres?

17.17. Una empresa virtual consta de un conjunto de compañías que cooperan una con otra para sacar adelante un proyecto concreto. Cada compañía desea proporcionar a las otras compañías acceso sólo a los objetos CORBA relevantes al proyecto. Describa una forma apropiada de agrupar o federar sus Servicios de Nombres de CORBA.

17.18. Discuta cómo utilizar proveedores y consumidores conectados directamente a un Servicio de Eventos de CORBA en el contexto de una aplicación de tablero compartido. Las interfaces *PushConsumer* y *PushSupplier* están definidas en IDL como sigue:

```
interface PushConsumer {
    void push(in any datos) raises (Disconnected);
    void disconnect_push_consumer();
}
interface PushSupplier {
    void disconnect_push_supplier();
}
```

Tanto el proveedor como el consumidor pueden decidir terminar la comunicación de eventos llamando a *disconnect_push_supplier()* o a *disconnect_push_consummer()* respectivamente.

17.19. Describa cómo interponer un Canal de Eventos entre el proveedor y los consumidores en su solución al Ejercicio 17.18. Un canal de eventos tiene la siguiente interfaz IDL:

```
interface EventChannel {
    consumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
};
```

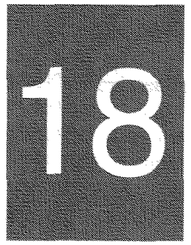
donde las interfaces *SupplierAdmin* y *ConsumerAdmin*, que permiten que el proveedor y el consumidor obtengan cada proxy se definen en IDL como sigue:

```
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ---
};
interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ---
};
```

La interfaz para el proxy del consumidor y el proxy del proveedor se definen en IDL como sigue:

```
interface ProxyPushConsumer : PushConsumer{
    void connect_push_supplier(in PushSupplier proveedor)
        raises (AlreadyConnected);
};
interface ProxyPushSupplier : PushSupplier{
    void connect_push_consummer(in PushConsumer consumidor)
    raises (AlreadyConnected);
};
```

¿Cuál es la ventaja de usar el canal de eventos?



ESTUDIO DEL CASO: MACH

18.1. Introducción
18.2. Puertos, nombres y protección
18.3. Tareas e hilos
18.4. Modelo de comunicación
18.5. Implementación de la comunicación
18.6. Gestión de memoria
18.7. Resumen

Este capítulo describe el diseño del micronúcleo de Mach. Mach puede ejecutarse tanto en multiprocesadores como en computadores monoprocesador conectadas por redes. Fue diseñado para permitir la evolución de los nuevos sistemas distribuidos pero manteniendo la compatibilidad con UNIX.

Mach es un diseño seminal. Incluye mecanismos sofisticados de comunicación entre procesos y de gestión de memoria virtual. Describimos la arquitectura de Mach y discutimos sus principales abstracciones: hilos y tareas (procesos); puertos (puntos terminales de la comunicación), y memoria virtual, incluyendo su papel en la comunicación entre tareas y el soporte para la compartición de objetos entre computadores mediante la paginación.

18.1. INTRODUCCIÓN

El proyecto Mach [Acetta y otros 1986, Loepere 1991, Boykin y otros 1993] estaba asentado en la Universidad Carnegie-Mellon en USA hasta 1994. Allí continuó su desarrollo hacia un núcleo en tiempo real [Lee y otros 1996], y su desarrollo fue continuado por grupos de la Universidad de Utah y la Fundación para el Software Abierto (*Open Software Foundation*, OSF). El proyecto Mach fue el sucesor de otros dos proyectos, RIG [Rashid 1986] y Accent [Rashid y Robertson 1981, Rashid 1985, Fitzgerald y Rashid 1986]. RIG fue desarrollado en la Universidad de Rochester en la década de los años setenta, y Accent fue desarrollado en Carnegie-Mellon durante la primera mitad de la década de los años ochenta. A diferencia de sus predecesores, RIG y Accent, el proyecto Mach nunca tuvo como objetivo el desarrollo de un sistema operativo distribuido completo. En vez de esto, el núcleo Mach se desarrolló para proporcionar una compatibilidad directa con UNIX BSD. Fue diseñado para proporcionar recursos avanzados que complementaran los de UNIX y que permitiera a una implementación UNIX proliferar sobre una red de multiprocesadores y computadores monoprocesador. Desde el principio, la intención de los diseñadores fue la de implementar la mayor parte de UNIX mediante procesos de nivel de usuario.

A pesar de estas intenciones, la versión 2.5 de Mach, la primera de las dos principales versiones, incluía todo el código de compatibilidad con UNIX dentro del propio núcleo. Se ejecutaba, entre otros computadores, en SUN-3, IBM RT PC, sistemas uniprocesadores y multiprocesadores VAX y en los multiprocesadores Encore Multimax y Sequent. Desde 1989, Mach 2.5 fue incorporado como la tecnología base para OSF/1, el rival propuesto por la Fundación para el Software Abierto para competir contra System V Release 4 como versión estándar de UNIX para la industria. Una versión más antigua de Mach se utilizó como base para el sistema operativo de la estación de trabajo NeXT.

A partir de la versión 3.0 del núcleo Mach se eliminó el código UNIX, con todo, ésta es la versión que estudiaremos. Más recientemente, Mach 3.0 es la base de la implementación de MkLinux, una variante del sistema operativo Linux para computadores Power Macintosh [Morin 1997]. La versión 3.0 del núcleo Mach también se ejecuta en PCs basados en Intel x86. Funcionaba en las series de computadores DECstation 3100 y 5000, algunos computadores basados en el Motorola 88000 y SUN SPARCStations; se emprendieron versiones para IBM RS6000, la Arquitectura de Precisión (*Precission Arquitecture*, PA) de Hewlett-Packard y Alpha de Digital Equipment Corporation.

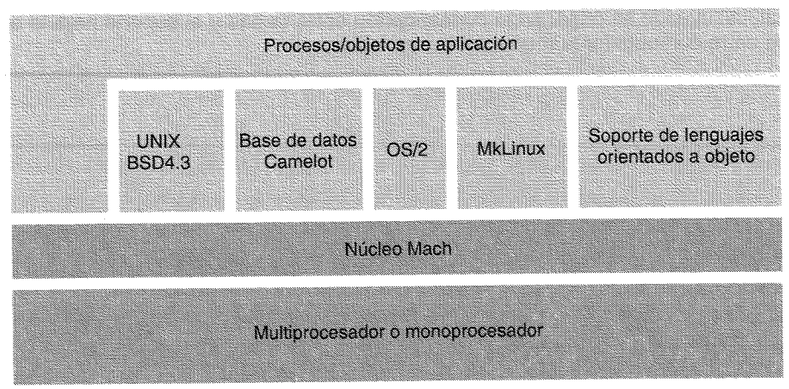


Figura 18.1. Mach proporciona soporte a sistemas operativos, bases de datos y otros subsistemas.

La versión 3.0 de Mach se utiliza como soporte para la construcción de emulaciones de sistemas operativos, sistemas de base de datos, sistemas de soporte de lenguajes en tiempo de ejecución y otros tipos de software de sistema que llamaremos subsistemas (véase la Figura 18.1).

La emulación de sistemas operativos convencionales permite ejecutar binarios ya existentes desarrollados para ellos. Además se pueden desarrollar nuevas aplicaciones para dichos sistemas operativos convencionales. Al mismo tiempo, se puede desarrollar middleware y aplicaciones que aprovechen las ventajas de la distribución; y se pueden crear versiones distribuidas de las implementaciones de los sistemas operativos convencionales. Surgen dos cuestiones importantes en la emulación de los sistemas operativos. La primera es que las emulaciones distribuidas no pueden ser completamente exactas debido a los nuevos modos de fallo que aparecen con la distribución. La segunda, es la cuestión todavía sin resolver de si se pueden conseguir niveles de prestaciones aceptables para que su utilización sea generalizada.

18.1.1. OBJETIVOS Y PRINCIPALES CARACTERÍSTICAS DEL DISEÑO

Los principales objetivos y características de diseño de Mach son los siguientes:

Operación multiprocesador: Mach fue diseñado para ejecutarse en un multiprocesador de memoria compartida de forma que tanto los hilos del núcleo como los hilos en modo usuario pueden ejecutarse en cualquier procesador. Mach proporciona un modelo multi-hilo para procesos de usuario, con entornos de ejecución llamados *tareas*. Para permitir la ejecución paralela en un multiprocesador de memoria compartida, la planificación de los hilos es prioritaria, o apropiativa, tanto si pertenecen a las mismas como a diferentes tareas.

Extensión transparente para operar en red: para permitir que los programas distribuidos se extiendan de forma transparente sobre mono y multiprocesadores en una red, Mach ha adoptado un modelo de comunicación independiente de la ubicación usando puertos como destino de la comunicación. Sin embargo, el núcleo de Mach ha sido diseñado para ser 100 % independiente de redes concretas. El diseño de Mach confía en procesos servidores de red de nivel de usuario para el envío de mensajes de forma transparente sobre la red (véase la Figura 18.2). Se trata de una decisión de diseño polémica dados los costes del cambio de contexto, que examinamos en el Capítulo 6. Sin embargo, permite una absoluta flexibilidad en el control de la política de comunicación de la red.

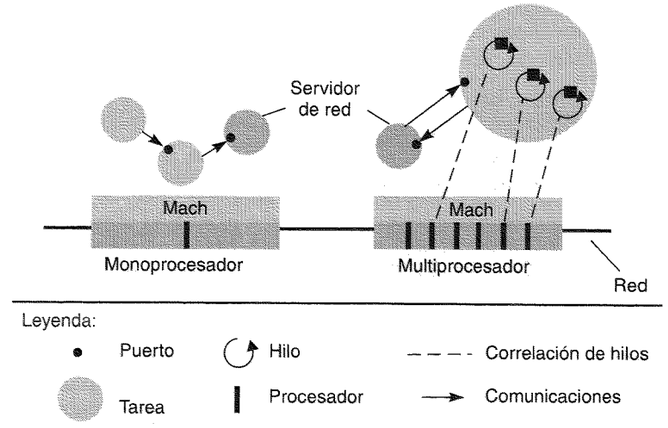


Figura 18.2. Tareas, hilos y comunicación en Mach.

Servidores de nivel de usuario: Mach implementa un modelo basado en objetos en el que los recursos se gestionan o bien por el núcleo, o bien mediante servidores cargados dinámicamente. Inicialmente sólo se permitían servidores de nivel de usuario, pero posteriormente se adaptó Mach para acomodar servidores en el espacio de direcciones del núcleo. Tal y como ya hemos mencionado, un objetivo primordial fue el de implementar la mayor parte de los recursos de UNIX desde el nivel de usuario, a la vez que se proporcionaba compatibilidad binaria con sistemas UNIX existentes. Con la excepción de algunos recursos gestionados por el núcleo, los recursos son accedidos de forma uniforme mediante paso de mensajes, independientemente de cómo sean gestionados. A cada recurso le corresponde un puerto gestionado por un servidor. El Generador de Interfaz de Mach (*Mach Interface Generator*, MiG) se desarrolló para generar enlaces RPC utilizados para ocultar los accesos basados en mensajes del nivel de lenguaje [Draves y otros 1989].

Emulación de sistema operativo: para dar soporte a la emulación de nivel de usuario de UNIX y otros sistemas operativos, Mach permite la redirección transparente de las llamadas al sistema operativo sobre llamadas a una biblioteca de emulación y de ahí hacia los servidores del sistema operativo del nivel de usuario; a esta técnica se le conoce como *trampolining*. También se incluye la posibilidad de permitir que las excepciones del tipo de violaciones del espacio de direcciones que ocurren en las tareas de aplicación sean gestionadas por servidores. En www.cdk3.net/oss se pueden encontrar los casos de estudio de emulaciones UNIX bajo Mach y Chorus.

Implementación de memoria virtual flexible: se realizó un gran esfuerzo en la mejora de la memoria virtual para dotar a Mach de la posibilidad de emular sistemas UNIX y dar soporte a otros subsistemas. Para ello se realizó una aproximación flexible a la distribución del espacio de direcciones de un proceso. Mach soporta un espacio de direcciones grande y disperso, con capacidad potencial para contener múltiples regiones. Por ejemplo, tanto los mensajes como los archivos abiertos pueden aparecer como regiones de memoria virtual. Las regiones pueden ser privadas a una tarea, compartidas entre tareas o copiadas desde regiones sobre otras tareas. El diseño incluye el uso de técnicas de correlación de memoria, principalmente la *copia en escritura*, para evitar la copia de datos cuando, por ejemplo, se pasan mensajes entre tareas. Finalmente, Mach se diseñó para permitir que fueran los servidores, en lugar del propio núcleo, los que implementaran el almacenamiento secundario para las páginas de memoria virtual. Las regiones pueden correlacionarse con datos gestionados por los servidores llamados *paginadores externos*. Estos datos correlacionados (*mapped*) pueden residir en cualquier abstracción de un recurso de memoria, desde memoria compartida distribuida hasta archivos.

Portabilidad: Mach fue diseñado para ser portable sobre varias plataformas hardware. Por esta razón, se trató de aislar, tanto como fue posible, el código dependiente de la máquina. Por ejemplo, el código de memoria virtual fue dividido en una parte dependiente de la máquina y otra independiente de la máquina [Rashid y otros 1988].

18.1.2. REPASO DE LAS PRINCIPALES ABSTRACCIONES DE MACH

Se pueden resumir las diferentes abstracciones proporcionadas por el núcleo de Mach de la siguiente forma (todas ellas serán descritas en detalle posteriormente en este capítulo):

Tareas: una tarea en Mach es un entorno de ejecución. Básicamente está compuesto por un espacio de direcciones protegido y una colección de habilitaciones *gestionadas por el núcleo* que se utilizan para el acceso a los puertos.

Hilos: las tareas pueden contener múltiples hilos. Los hilos que pertenecen a una misma tarea se pueden ejecutar en paralelo sobre diferentes procesadores en un multiprocesador de memoria compartida.

Puertos: un puerto en Mach es un canal de comunicaciones *uno a uno* y unidireccional con una cola de mensajes asociada. Los puertos no son accesibles directamente por el programador de Mach y tampoco forman parte de una tarea. En su lugar, el programador dispone de apuntadores a *derechos de puerto*. Se trata de habilitaciones para el envío de mensajes a un puerto o para la recepción de mensajes desde un puerto.

Conjuntos de puertos: un conjunto de puertos es una colección de derechos de recepción de puertos, locales a una cierta tarea. Se utilizan para recibir mensajes desde cualquier puerto de entre una colección de ellos. Los conjuntos de puertos no deben confundirse con los *grupos* de puertos; estos últimos son destinos de multidifusión y en Mach no están implementados.

Mensajes: un mensaje en Mach puede contener derechos de puerto además de datos. El núcleo utiliza técnicas de gestión de memoria para transferir de forma eficiente los datos de los mensajes entre las tareas.

Dispositivos: ciertos servidores como los servidores de archivos que se ejecutan a nivel de usuario deben poder acceder a los dispositivos. Para ello el núcleo exporta una interfaz de bajo nivel sobre los dispositivos.

Objeto de memoria: cada región del espacio de direcciones virtuales de una tarea Mach se corresponde con un objeto de memoria. Se trata de un objeto que normalmente está implementado fuera del propio núcleo pero que es accedido por éste cuando realiza operaciones de paginación en la gestión de la memoria virtual. Un objeto de memoria es una instancia de un tipo abstracto de datos que incluye operaciones de búsqueda y almacenamiento de datos que se emplean cuando los hilos generan faltas de página en el intento de hacer referencia a direcciones en la región correspondiente.

Objeto de memoria caché: para cada objeto correlacionado en memoria existe un objeto gestionado por el núcleo que contiene una caché de páginas residentes en la memoria principal para la correspondiente región. A esto se le llama objeto de memoria caché. Da soporte a las operaciones necesarias para el paginador externo encargado de implementar el objeto de memoria.

Estudiaremos a continuación las principales abstracciones. La abstracción de dispositivo se omite en interés de la brevedad.

18.2. PUERTOS, NOMBRES Y PROTECCIÓN

Mach asocia recursos individuales con puertos. Para acceder a un recurso se envía un mensaje sobre el puerto correspondiente. En Mach, se presupone que los servidores manejarán, por lo general, múltiples puertos: uno por cada recurso. Un único servidor UNIX utiliza alrededor de 2.000 puertos [Draves 1990]. Por lo tanto la creación y gestión de estos puertos debe ser barata.

El problema de la protección de un recurso frente a accesos ilegales se equipara al de la protección del correspondiente puerto frente a envíos ilegales. En Mach esto se consigue mediante el control que realiza el núcleo sobre la adquisición de habilitaciones para el puerto y también por el control de los mensajes realizado por el servidor de red.

La habilitación de un puerto tiene un campo que especifica los derechos de acceso sobre el puerto pertenecientes a la tarea que lo posee. Existen tres tipos diferentes de derechos de puerto. Los *derechos de envío* permiten a los hilos dentro de la tarea que los posee el envío de mensajes al

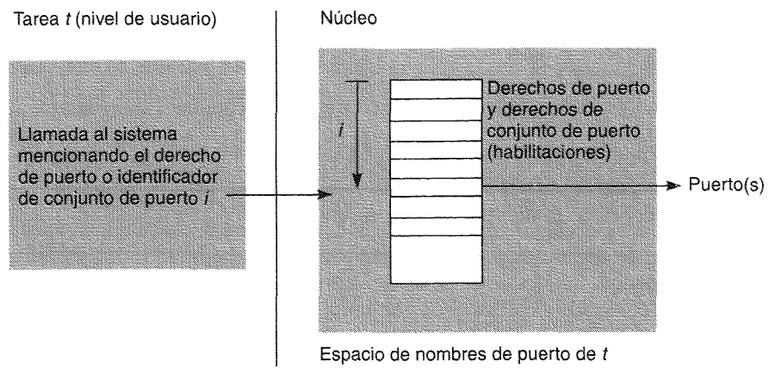


Figura 18.3. Un espacio de nombres de puerto de una tarea.

puerto correspondiente. Una forma más restringida son los *derechos de un solo envío*, que permiten el envío de tan sólo un mensaje, después de lo cual los derechos son destruidos automáticamente por el núcleo. Esta forma restringida permite, por ejemplo, que un cliente obtenga una respuesta desde un servidor sabiendo que el servidor no le enviará más mensajes (y por lo tanto protegiéndole de servidores erróneos); además, se libera al servidor de la tarea de eliminar los derechos de envío recibidos desde los clientes. Finalmente los *derechos de recepción* permiten a los hilos de una tarea recibir mensajes desde la cola de mensajes del puerto. En cada momento sólo una tarea puede poseer los derechos de recepción, mientras que un número variable de tareas pueden poseer los derechos de envío o de *un solo envío*. Mach soporta sólo la comunicación *muchos a uno*: la multidifusión no aparece implementada directamente en el núcleo.

En el momento de su creación a cada tarea se la proporciona un *derecho de puerto de inicio*, que es un derecho de envío que sirve para obtener servicios de otras tareas. Una vez creada, los hilos pertenecientes a la tarea adquieren más derechos de puerto ya sea mediante su creación o mediante su recepción a través de mensajes.

Los derechos de puerto de Mach se almacenan en el núcleo y son protegidos por él (véase la Figura 18.3). Las tareas nombran los derechos de puerto mediante identificadores locales válidos únicamente en el *espacio de nombres de puerto* local a la tarea. Esto permite al equipo de desarrollo del núcleo elegir representaciones eficientes para estas habilitaciones (tales como punteros a colas de mensajes) y elegir nombres locales de tipo entero a conveniencia del núcleo para permitir búsquedas fácilmente la habilitación dado el nombre. De hecho, análogamente a los descriptores de archivo de UNIX, los identificadores locales son enteros utilizados para indexar una tabla del núcleo que contiene las habilitaciones de la tarea.

El esquema de nombres y protección de Mach permite un acceso rápido a las colas de mensajes locales dado un identificador de usuario. Como contrapartida a esta ventaja está el gasto realizado por el núcleo al procesar todos los derechos transmitidos en los mensajes entre tareas. Como mínimo, los derechos de envío deben tener asociado un nombre local en el espacio de nombres de la tarea receptora, junto con espacio en sus tablas del núcleo. Hay que resaltar que, en un entorno seguro, la transmisión de derechos de puerto a través de los servidores de red requiere su encriptación como media de protección frente a ataques de seguridad como las escuchas no autorizadas [Sansom y otros 1986].

18.3. TAREAS E HILOS

Una tarea es un entorno de ejecución: las propias tareas no pueden realizar ninguna acción; únicamente lo pueden realizar sus hilos. Sin embargo, por comodidad, a veces nos referiremos a tareas realizando acciones cuando en realidad nos referimos a los hilos dentro de la tarea. Los principales recursos asociados directamente con una tarea son su espacio de direcciones, sus hilos, sus derechos de puerto, sus conjuntos de puertos y el espacio de nombres local en el que se buscan los derechos de puerto y los conjuntos de puertos. Examinaremos el mecanismo de creación de una tarea nueva y todo lo relacionado con la gestión de tareas y la ejecución de sus hilos.

◇ **Creación de una nueva tarea.** La operación *fork* de UNIX crea un nuevo proceso copiando uno ya existente. El modelo de creación de procesos de Mach es una generalización del de UNIX. Las tareas son creadas con referencia a lo que llamaremos *tarea tipo* (no necesariamente la creadora). La nueva tarea reside en el mismo computador que la tarea tipo. Al no proporcionar Mach la posibilidad de migración de tareas, la única forma de establecer una tarea sobre un computador remoto es a través de otra tarea que ya resida allí. El nuevo derecho de puerto de inicio de la tarea se hereda de la tarea tipo y su espacio de direcciones puede estar vacío o bien ser heredado de la misma forma (la herencia del espacio de direcciones se discute posteriormente en la subsección de memoria virtual de Mach). Una tarea recién creada no tiene hilos. En su lugar, el creador de la tarea solicita la creación de un hilo dentro de la tarea hijo. Posteriormente podrán crearse otros hilos a partir de los hilos existentes dentro de la tarea. En la Figura 18.4 aparecen algunas de las llamadas de Mach relacionadas con la creación de tareas e hilos.

◇ **Invocación de operaciones del núcleo.** Cuando se crea una tarea o hilo en Mach se le asigna un denominado *puerto del núcleo*. Las llamadas al sistema de Mach se dividen entre las implementadas directamente como trampas (*traps*) y las implementadas mediante paso de mensajes a los puertos del núcleo. Este último método tiene la ventaja de permitir operaciones con transparencia de red sobre tareas e hilos que pueden ser remotos o locales. Un servicio del núcleo gestiona recursos del núcleo de la misma forma en que un servidor de nivel de usuario maneja otros recursos. Cada tarea tiene derechos de envío sobre su puerto del núcleo, lo que la permite invocar operaciones por sí misma (como la creación de un nuevo hilo). Cada uno de los servicios del núcleo accedido mediante paso de mensajes tiene una definición de interfaz. Las tareas acceden a esos servicios mediante procedimientos de resguardo, los cuales se generan de sus definiciones de interfaz mediante el Generador de Interfaz de Mach.

---

task_create(tarea_padre, hereda_memoria, hijo_tarea)
tarea_padre es la tarea utilizada como tarea tipo en la creación, *hereda_memoria* especifica si el hijo debe heredar el espacio de direcciones de su padre o bien se le debe asignar un espacio de direcciones vacío, *tarea_hijo* es el identificador de la nueva tarea.

thread_create(padre_tarea, hijo_hilo)
tarea_padre es la tarea en la que se quiere crear el nuevo hilo, *hilo_hijo* es el identificador del nuevo hilo. El hilo creado no tiene estado de ejecución y está suspendido.

thread_set_state(hilo, aroma, nuevo_estado, contador)
hilo es el identificador del hilo al que se quiere proporcionar un estado de ejecución, *aroma* especifica la arquitectura de la máquina, *nuevo_estado* especifica el estado (el contador de programa y el puntero de pila, por ejemplo), *contador* es la talla del estado.

thread_resume(hilo)
Se utiliza para reanudar el hilo suspendido identificado por *hilo*.

Figura 18.4. Creación de tareas e hilos.

◇ **Gestión de excepciones.** Las tareas y (a veces) los hilos poseen, además del puerto del núcleo, un *puerto de excepciones*. Cuando ocurren ciertos tipos de excepciones el núcleo reacciona intentando enviar un mensaje que describe la excepción a un puerto de excepciones asociado. Si el hilo carece de dicho puerto, busca el de la tarea. El hilo que recibe el mensaje puede intentar reparar el problema (puede, por ejemplo, incrementar el tamaño de la pila del hilo como respuesta a una violación del espacio de direcciones) y devuelve, a continuación, un valor de estado en un mensaje de respuesta. Si el núcleo encuentra un puerto de excepciones y recibe una respuesta indicando éxito, entonces reinicia el hilo que generó la excepción. En otro caso el núcleo elimina dicho hilo.

Por ejemplo, cuando una tarea realiza una violación de acceso sobre el espacio de direcciones o bien una división por cero, el núcleo envía un mensaje a un puerto de excepciones. El propietario de este puerto podría ser una tarea de depuración de errores, que podría estar ejecutándose en cualquier punto de la red gracias al sistema de comunicación independiente de ubicación de Mach. Las faltas de página son manejados por paginadores externos. En la Sección 18.4 se describe cómo Mach gestiona las llamadas al sistema dirigidas a un sistema operativo emulado.

◇ **Gestión de tareas e hilos.** Alrededor de cuarenta procedimientos en la interfaz del núcleo están relacionados con la creación y gestión de tareas e hilos. El primer argumento de cada procedimiento es un derecho de envío al puerto del núcleo asociado, y se usan llamadas al sistema por paso de mensajes para solicitar la operación al núcleo destino. Algunas de esas llamadas de gestión de tareas e hilos se muestran en la Figura 18.4. En resumen, las prioridades de planificación de los hilos se pueden configurar de forma individual; los hilos y las tareas pueden ser suspendidos, reanudados y terminados; y el estado de ejecución de los hilos puede ser creado de forma externa, leído y modificado. La última posibilidad es importante tanto para la depuración como para el manejo de interrupciones software. Otras invocaciones a la interfaz del núcleo sirven para asignar hilos sobre ciertos *conjuntos de procesadores*. Un conjunto de procesadores es un subconjunto de procesadores en un multiprocesador. Mediante la asignación de hilos a conjuntos de procesadores, los recursos computacionales disponibles pueden asignarse toscamente a tipos de actividades diferentes. El lector puede consultar Loepere [1991] para profundizar en los detalles de soporte del núcleo para la gestión de tareas e hilos y para la asignación de procesadores. Tokuda y otros [1990] y Lee y otros [1996] describen extensiones de Mach para la planificación y sincronización de hilos en tiempo real.

18.4. MODELO DE COMUNICACIÓN

Mach proporciona una única llamada al sistema para el paso de mensajes: *mach_msg*. Antes de describirla, debemos profundizar algo más sobre los mensajes y puertos en Mach.

18.4.1. MENSAJES

Un mensaje está formado por una cabecera de tamaño fijo seguida por una lista de tamaño variable de campos de datos (véase la Figura 18.5).

La cabecera de tamaño fijo contiene:

El puerto de destino: por simplicidad forma parte del mensaje en lugar de especificarse como un parámetro separado de la llamada al sistema *mach_msg*. Se indica mediante el identificador local de los correspondientes derechos de envío.

Un puerto de respuesta: si se necesita una respuesta, entonces se insertan los derechos de envío a un puerto local (es decir, un puerto para el que el hilo que envía tenga derechos de recepción) en el mensaje.

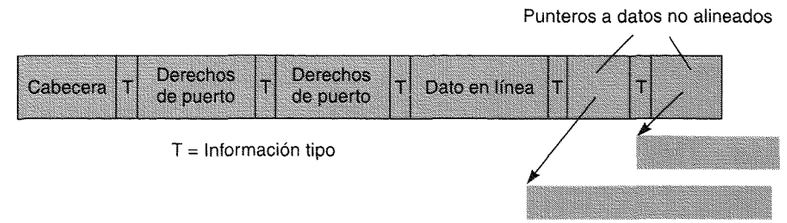


Figura 18.5. Un mensaje en Mach conteniendo derechos sobre puerto y datos no alineados.

Un identificador de operación: identifica una operación (procedimiento) en la interfaz de servicio que sólo puede ser interpretada por las aplicaciones.

Tamaño de los datos extra: a continuación de la cabecera aparece una lista de tamaño variable de datos tipados. No existe límite para su tamaño, excepto el número de bits del campo y el tamaño total del espacio de direcciones.

Cada ítem en la lista consecutiva a la cabecera del mensaje puede ser uno de los siguientes (pueden aparecer en cualquier orden dentro del mensaje):

Datos de mensaje tipado: individuales, datos en línea etiquetados con su tipo;

Derechos de puerto: referidos a sus identificadores locales;

Punteros a datos no alineados: datos que residen en un bloque de memoria separado y no contiguo.

Los mensajes en Mach están formados por una cabecera de tamaño fijo y múltiples bloques de datos de tamaño variable, algunos de los cuales pueden estar *no alineados* (es decir en bloques separados). Sin embargo, cuando se envían mensajes con datos no alineados, el núcleo (no la tarea receptora) escoge la ubicación de los datos recibidos en el espacio de direcciones de la tarea receptora. Se trata de un efecto colateral de la técnica de la *copia en escritura* utilizada para transferir estos datos. Las regiones de memoria virtual extra recibidas en un mensaje que no se vayan a usar más deben ser liberadas de forma explícita por la tarea receptora. Al ser los costes de las operaciones de memoria virtual superiores a los coste de la copia de datos para pequeñas cantidades de datos, se supone que sólo se envían de modo no alineado grandes cantidades de datos.

La ventaja de permitir múltiples componentes de datos en los mensajes consiste en que posibilita al programador la asignación de memoria de forma separada para los datos y para los metadatos. Por ejemplo, un servidor de archivos puede asignar bajo petición un bloque de disco desde su caché. En lugar de copiar el bloque en el búfer del mensaje, de forma contigua a la cabecera, los datos pueden ser tomados directamente desde donde residen mediante la inserción de un puntero en el mensaje de respuesta. Se trata de una variante de la *E/S de acopio disperso* (*scatter-gather I/O*), en la que los datos son leídos o escritos desde múltiples ubicaciones del espacio de direcciones del invocador mediante una llamada al sistema. Las llamadas al sistema *readv* y *writev* de UNIX actúan de esta forma [Leffler y otros 1989].

El tipo de cada elemento de datos en un mensaje Mach es especificado por el emisor (mediante, por ejemplo, ASN.1). Esto permite a los servidores de red de nivel de usuario empaquetar los datos con formato estándar, cuando se transmiten sobre la red. Sin embargo, este esquema de empaquetamiento tiene peores prestaciones en comparación con el empaquetamiento y desempaquetamiento realizado por los procedimientos de resguardo generados desde las definiciones de la interfaz. Los procedimientos de resguardo conocen los tipos de datos implicados, no necesitan in-

cluirlos en los mensajes y pueden empaquetarlos directamente en el mensaje (véase la Sección 4.2). Un servidor de debería copiar los datos con tipo del emisor dentro de otro mensaje al empaquetarlos.

18.4.2. PUERTOS

Un puerto en Mach tiene una cola de mensajes cuyo tamaño puede ser cambiado dinámicamente por la tarea con los derechos de recepción. Esta posibilidad permite a los receptores implementar cierta forma de control de flujo. Cuando se utiliza un derecho de envío normal, un hilo que intente enviar un mensaje a un puerto cuya cola de mensajes esté llena se bloqueará hasta que haya espacio disponible. Cuando un hilo utiliza un derecho de *un solo envío*, el receptor siempre inserta en la cola el mensaje, aunque la cola de mensajes esté llena. Al utilizarse el derecho de *un solo envío*, se sabe que no habrá más mensajes desde dicho emisor. Los hilos del servidor pueden evitar la espera enviando derechos de *un solo envío* en sus respuestas a los clientes.

◇ **Envío de derechos de puerto.** Cuando los derechos de envío de un puerto se insertan en un mensaje, el receptor los adquiere sobre el mismo puerto. Sin embargo, cuando lo que se transmite son los derechos de recepción son desasignados automáticamente de la tarea que los emite. La razón es que los derechos de recepción no pueden asignarse a más de una tarea al tiempo. Todos los mensajes en la cola del puerto y todos los mensajes transmitidos posteriormente podrán ser recibidos por el nuevo propietario de los derechos de recepción, y esto se consigue de forma transparente a las tareas que envían mensajes al puerto. La transferencia transparente de derechos de recepción es relativamente fácil de conseguir cuando estos derechos se transfieren dentro de un único computador. La habilitación adquirida es simplemente un puntero a una cola local de mensajes. Para el caso de transferencia entre distintos computadores se plantean problemas más complejos de diseño que serán discutidos posteriormente.

◇ **Monitorización de la conectividad.** El núcleo está diseñado para informar a los emisores y receptores de la aparición de ciertas condiciones bajo las que la emisión o recepción de mensajes es inútil. Para conseguirlo gestiona información sobre el número de derechos de envío y recepción sobre un puerto dado. Si no existe ninguna tarea con los derechos de recepción sobre cierto puerto (por ejemplo, al fallar la tarea que disponía de dichos derechos), entonces todos los derechos de envío en los espacios de nombres de puerto de las tareas locales se convierten en *nombres muertos*. Cuando un emisor intenta utilizar un nombre referido a un puerto para el que los derechos de recepción ya no existen, el núcleo convierte el nombre en un nombre muerto y devuelve una indicación del error. Análogamente las tareas pueden solicitar al núcleo la notificación asíncrona de la condición de que ya no existan derechos de envío para un cierto puerto. El núcleo realiza esta notificación mediante el envío de un mensaje, utilizando los derechos de envío proporcionados para ello por el hilo. La condición de no existencia de derechos de envío puede ser detectada mediante un contador de referencias que se incrementa cuando se crea un derecho de envío y que se decrementa cuando se destruye.

Debe resaltarse que las condiciones de ausencia de emisores/ausencia de receptor pueden ser abordadas dentro del dominio de un único núcleo con un coste relativamente bajo. El chequeo de estas condiciones en un sistema distribuido es, por el contrario, una operación compleja y cara. Dado que los derechos pueden ser enviados en los mensajes, los derechos de envío o recepción para un cierto puerto podrían residir en cualquier tarea, o estar en un mensaje dentro de la cola de un puerto, o en tránsito entre computadores.

◇ **Conjuntos de puertos.** Los conjuntos de puertos son colecciones de puertos creadas dentro de una única tarea y gestionadas localmente. Cuando un hilo realiza una operación de recepción sobre un conjunto de puertos, el núcleo devuelve un mensaje que fue entregado a algún miembro del conjunto. También devuelve el identificador de los derechos de recepción del puerto para que el hilo pueda realizar el correspondiente tratamiento del mensaje.

Los conjuntos de puertos son útiles ya que normalmente un servidor debe proporcionar servicio a mensajes de clientes que llegan por todos los puertos y en cualquier instante. El intento de recibir un mensaje desde un puerto cuya cola de mensajes está vacía provoca la detención del hilo y esta espera se mantiene aunque desde otro puerto se reciba posteriormente un mensaje. La asignación de un hilo a cada puerto resuelve este problema pero no es factible para servidores con un número grande de puertos ya que los hilos son recursos más caros que los puertos. Mediante la asociación de puertos en conjuntos de puertos, puede utilizarse un único hilo para dar servicio a los mensajes entrantes sin temor a perder ninguno de ellos. Además este hilo se inmovilizará si no hay mensajes disponibles desde ningún puerto del conjunto, evitando esperas activas en las que el hilo sondea los puertos hasta que llega de un mensaje por alguno de ellos.

18.4.3. MACH_MSG

La llamada al sistema *Mach_msg* es extremadamente complicada ya que proporciona tanto servicios de paso de mensajes asíncrono como interacciones de tipo petición-respuesta. Únicamente proporcionaremos una breve descripción de su semántica. La llamada completa es como sigue:

---

mach_msg(cabecera_msg, opción, tam_envío, tam_recep, nombre_recep, timeout, notifica)

cabecera_msg es un puntero a una cabecera de mensajes común para los mensajes de envío y recepción, *opcion* indica si es un envío, una recepción o ambos, *tam_envío* y *tam_recep* son los tamaños de los búferes de mensajes de envío y recepción, *nombre_recep* especifica los derechos de recepción del puerto o conjunto de puertos (si se recibe un mensaje), *timeout* pone un límite al tiempo total para el envío y/o recepción de un mensaje, *notifica* proporciona derechos de puerto que utiliza el núcleo para enviar mensajes de notificación bajo circunstancias excepcionales.

Mach_msg tanto envía un mensaje, como lo recibe o ambas cosas. Es una única llamada al sistema que utilizan los clientes para enviar un mensaje de petición y recibir una respuesta, y los servidores para responder al último cliente y recibir el siguiente mensaje de petición. Otro beneficio de la utilización de llamadas de envío/recepción combinadas es que para el caso de un cliente y servidor ejecutándose en el mismo computador la implementación puede utilizar una optimización llamada *planificación con traspaso*. Consiste en que cuando una tarea va a inmovilizarse una vez que ha enviado un mensaje a otra tarea, *dona* el resto de su intervalo de tiempo de ejecución (*timeslice*) al hilo de la otra tarea. Esto es más barato que ir a la cola de hilos PREPARADOS para seleccionar el siguiente a ejecutar.

Los mensajes enviados por el mismo hilo se entregan en el orden de envío, siendo esta entrega fiable. Por lo menos, esto se garantiza cuando los mensajes se envían entre tareas bajo un mismo núcleo, incluso cuando no hay espacio en el búfer. Cuando los mensajes se transmiten a través de la red a un computador independiente, se utiliza la semántica de entrega *como máximo una vez*.

El timeout es útil en situaciones en las que no es deseable que un hilo se inmovilice indefinidamente, por ejemplo en espera de un mensaje que puede no llegar nunca, o esperando por espacio en una cola sobre un puerto de servidor erróneo.

18.5. IMPLEMENTACIÓN DE LA COMUNICACIÓN

Uno de los aspectos más interesantes de la implementación del sistema de comunicación en Mach es la utilización de servidores de red de nivel de usuario. Los servidores de red (*netmsgservers* en la literatura de Mach), uno por computador, son colectivamente responsables de extender la semántica de comunicación local sobre toda la red. Esto incluye preservar, en la medida de lo posible, las garantías de reparto y hacer que la comunicación sobre la red sea transparente. También incluye la realización y monitorización de las transferencias de derechos de puertos. En concreto, los servidores de red son los responsables de la protección de los puertos frente a accesos ilegales y del mantenimiento de la privacidad de los mensajes enviados por la red. Los detalles completos de cuestiones del tratamiento de Mach de la protección están disponibles en Sansom y otros [1986].

18.5.1. GESTIÓN TRANSPARENTE DE MENSAJES

Debido a que los puertos son siempre locales al núcleo de Mach, es necesario añadir una abstracción impuesta de forma externa, el *puerto de red*, utilizada para enviar los mensajes de red. Un puerto de red es un identificador de canal globalmente único gestionado exclusivamente por los servidores de red y que éstos asocian a cada único puerto Mach en cada momento. Los servidores de red poseen los derechos de envío y recepción sobre los puertos de red, de la misma forma que las tareas poseen los derechos de envío y recepción sobre los puertos Mach.

En la Figura 18.6 aparece una descripción de la transmisión de un mensaje entre tareas localizadas en diferentes computadores. Los derechos de la tarea emisora se refieren a los puertos locales, sobre los cuales el servidor de red local dispone de los derechos de recepción. En la figura, el identificador local del servidor de red para los derechos de recepción es 8. El servidor de red en el computador emisor utiliza este identificador para buscar una entrada en una tabla de puertos de red para los que tiene derechos de envío. Esta tabla proporciona un puerto y sugiere una dirección de red. A continuación envía el mensaje, al que se añade el puerto de red, al servidor de red de la dirección indicada en la tabla. Allí este servidor de red local extrae el puerto de red y lo utiliza para buscar en una tabla de puerto de red para los que tiene derechos de recepción. Si encuentra

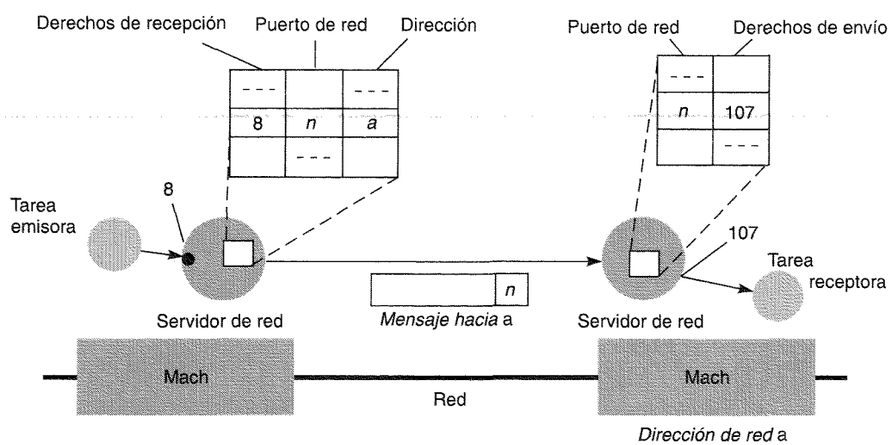


Figura 18.6. Comunicación de red en Mach.

una entrada válida (el puerto de red podría haber sido reasignado a otro núcleo), entonces esa entrada contiene el identificador de los derechos de envío a un puerto local en Mach. El servidor de red reenvía el mensaje utilizando estos derechos, consiguiendo así enviar el mensaje al puerto apropiado. El proceso completo realizado por los servidores de red es transparente tanto para el emisor como para el receptor.

¿Cómo se establecen las tablas mostradas en la Figura 18.6? El servidor de red de un computador recién inicializado se enrola en un protocolo de inicialización, por el cual se obtienen derechos de envío sobre los servicios de toda red. Considérese que ocurre después de esto, cuando se transfiere entre servidores de red un mensaje que contiene derechos de puerto. Estos derechos están tipados de modo que los servidores de red pueden seguirles la pista. Si una tarea emite los derechos de envío de un puerto local, el servidor de red local crea un identificador de puerto de red y una entrada en la tabla para el puerto local (si no existía ninguna) y enlaza el identificador al mensaje que envía. El servidor de red receptor también crea una entrada en la tabla si no existe ninguna.

La situación se vuelve más complicada cuando se transmiten los derechos de recepción. Este caso es un ejemplo en el que se necesita transparencia en la migración: los clientes deben ser capaces de enviar mensajes al puerto mientras éste está migrando. En primer lugar, el servidor de red local en el emisor adquiere los derechos de recepción. Todos los mensajes destinados al puerto, locales y remotos, comienzan llegando a este servidor. A continuación emplea un protocolo por el que se transfieren consistentemente los derechos de recepción al servidor de red destino.

La principal cuestión relativa a esta transferencia de derechos es cómo conseguir que los mensajes enviados al puerto ahora lleguen al computador a la que se han transferido los derechos de recepción. Una posibilidad sería que Mach siguiera la pista de todos los servidores de red que dispusieran de los derechos de envío sobre un cierto puerto de red, y notificar directamente a dichos servidores en el momento en que los derechos de recepción hubieran sido transferidos. Este esquema se rechazó por su coste de gestión. Una alternativa más barata sería utilizar un medio de difusión (*broadcast*) hardware que difunda el cambio de ubicación a todos los servidores de red. Sin embargo, este servicio de broadcast no es fiable y no está disponible en todas las redes. En su lugar, la responsabilidad de localizar un puerto de red se asignó a los servidores de red que disponen de los derechos de envío sobre dicho puerto.

Recuerde que un servidor de red utiliza una indicación de ubicación cuando reenvía un mensaje a otro servidor de red. Las posibles respuestas son:

el puerto está aquí: el destino dispone de los derechos de recepción;
el puerto está muerto: el puerto ha sido eliminado;
el puerto no está aquí, está transferido: los derechos de recepción fueron transferidos a una cierta dirección;
el puerto no está aquí, lo desconozco: no existe información sobre el puerto de red;
no hay respuesta: el computador destino no responde.

Si se devuelve una dirección de reenvío, el servidor de red emisor envía el mensaje a dicha dirección; sin embargo se trata sólo de una sugerencia que puede ser inexacta. Si en algún punto el emisor se queda sin direcciones de reenvío, entonces recurre a la difusión. En este tipo de algoritmos de localización, y en concreto cuando se utilizan sobre una WAN, las dos principales cuestiones de diseño son cómo manejar las cadenas de direcciones de reenvío y qué hacer cuando un computador que contiene una dirección de reenvío falla. El empleo de direcciones de reenvío sobre una WAN se describe en Black y Artsy [1990].

Una segunda cuestión para conseguir la transparencia en la migración es cómo sincronizar la entrega de los mensajes. Mach garantiza que dos mensajes enviados por el mismo hilo son entregados en el mismo orden. ¿Cómo puede garantizarse esto mientras los derechos de recepción están siendo transmitidos? Si esto no se resuelve correctamente un mensaje podría ser entregado por el

nuevo servidor de red antes de que se haya redirigido un mensaje anterior almacenado en la cola del computador original. El servidor de red puede conseguir esto reteniendo la entrega de todos los mensajes en el computador origen hasta que todos los mensajes almacenados en la cola hayan sido transferidos al computador destino. Así, el reparto de mensajes puede redirigirse de forma segura, y devolver a los emisores la dirección de reenvío.

18.5.2. EXTENSIBILIDAD: PROTOCOLOS Y GESTORES DE DISPOSITIVOS

◇ **Protocolos de transporte.** A pesar de que inicialmente la intención era conseguir que pudiera usarse una amplia variedad de protocolos de transporte mediante la ejecución de los servidores de red en el espacio de usuario, en el momento en que se escribe este libro los servidores de red de Mach de usados ampliamente usan únicamente TCP/IP como protocolo de transporte. Esto fue impulsado, en parte por la compatibilidad con UNIX, y en parte al ser seleccionado por la Universidad Carnegie-Mellon dada la complejidad de su red, que contiene más de 1.700 computadores, alrededor de 500 de ellos ejecutando Mach. Se ajustó TCP/IP para conseguir robustez de forma bastante eficiente sobre dicha red. Sin embargo, por motivos de prestaciones, esto puede no ser apropiado en una LAN donde predominan las interacciones petición-respuesta. Discutiremos las prestaciones de las comunicaciones en Mach en la Sección 18.6.

◇ **Gestores de dispositivos de red del nivel de usuario.** Algunos servidores de red proporcionan sus propios gestores de dispositivos (*drivers*) de red a nivel de usuario. Su objetivo es acelerar los accesos a la red. El situar los drivers en el nivel de usuario es, además de un método para mejorar la flexibilidad, una forma de compensar la degradación de prestaciones debida a la utilización de servidores de red al nivel de usuario. El núcleo exporta una abstracción para cada dispositivo, que incluye una operación para proyectar los registros de control del dispositivo sobre el espacio de usuario. Para el caso de una Ethernet, tanto los registros como los búferes de paquetes utilizados por el controlador pueden correlacionarse dentro el espacio de direcciones del servidor de red. Además, el núcleo ejecuta un código especial para despertar al hilo de nivel de usuario (que pertenece, en este caso, al servidor de red) cuando ocurre una interrupción. Este hilo es así capaz de gestionar la interrupción transfiriendo datos hacia o desde los búferes utilizados por el controlador y reiniciando el controlador para la siguiente operación.

18.6. GESTIÓN DE MEMORIA

Mach no sólo destaca por emplear espacios de direcciones grandes y dispersos, sino también por sus técnicas de memoria virtual entre tareas que permiten la compartición de memoria entre las tareas. No sólo puede compartirse la memoria física entre tareas en ejecución sobre el mismo núcleo Mach, sino que además el soporte de paginadores externos de Mach (llamados *gestores de memoria* en la literatura de Mach) permite compartir los contenidos de la memoria virtual entre tareas, aunque residan en computadores diferentes. Finalmente, es relevante la implementación de memoria virtual de Mach que se estratifica en niveles dependientes e independientes de máquina, lo que facilita la portabilidad del núcleo [Rashid y otros 1988].

18.6.1. ESTRUCTURA DEL ESPACIO DE DIRECCIONES

En el Capítulo 6 se presentó un modelo genérico de un espacio de direcciones formado por regiones. Cada región es un rango de direcciones lógicas contiguas con ciertas propiedades comunes.

Estas propiedades incluyen los permisos de accesos (lectura/escritura/ejecución), y también la elongabilidad. Por ejemplo se permite que las regiones de pila crezcan hacia direcciones bajas; y los montones (*heaps*) puedan crecer hacia las altas. El modelo utilizado por Mach es similar.

A pesar de ello, la visión del espacio de direcciones en Mach es la de una colección de grupos de páginas contiguas que se nombran mediante sus direcciones, en lugar de regiones identificables separadamente. Consecuentemente, la protección en Mach se aplica a las páginas en lugar de a las regiones. Las llamadas al sistema de Mach se refieren a direcciones y tamaños en lugar de usar identificadores de región. Por ejemplo, Mach no *hace crecer la región de pila*. En su lugar se asignan algunas páginas justo por debajo de las que están siendo utilizadas para la pila. Sin embargo, en la mayoría de los casos estas diferencias no son demasiado importantes.

Llamaremos región a una colección contigua de páginas con propiedades comunes a la región. Como hemos visto, Mach soporta un gran número de regiones, que pueden usarse para múltiples fines como almacenar datos o proyectar archivos en memoria.

Las regiones pueden crearse mediante cualquiera de las siguientes formas:

- Pueden ser asignadas de forma explícita mediante una llamada a *vm_allocate*. Las regiones creadas con *vm_allocate* están puestas a cero por defecto.
- Puede crearse una región en asociación con un *objeto de memoria*, utilizando *vm_map*.
- Pueden asignarse regiones a tareas recién creadas a través de la declaración (mejor dicho, la declaración y la de sus contenidos) como herencia desde una tarea tipo, utilizando *vm_inherit* aplicado a la región de la tarea tipo.
- Pueden asignarse automáticamente regiones en el espacio de direcciones de una tarea como efecto colateral del paso de mensajes.

Todas las regiones pueden establecer permisos de lectura/escritura/ejecución, mediante *vm_protect*. Pueden copiarse regiones dentro de las tareas mediante *vm_copy*, y otras tareas podrán leer o modificar los contenidos mediante *vm_read* y *vm_write*. Cualquier región previamente asignada puede ser liberada usando *vm_deallocate*.

Describiremos ahora la aproximación de Mach para la implementación de la operación UNIX *fork* y los aspectos relativos a la memoria virtual del paso de mensajes, incorporados en Mach para permitir la compartición de memoria cuando resulte conveniente.

18.6.2. COMPARTICIÓN DE MEMORIA: HERENCIA Y PASO DE MENSAJES

Mach permite una generalización de la semántica de UNIX *fork* mediante el mecanismo de *herencia de memoria*. Hemos visto que cada tarea se crea desde otra tarea, que actúa como plantilla. Una región heredada desde una tarea tipo contiene el mismo rango de direcciones y su memoria puede ser:

Compartida: secundada por la misma memoria.

Copiada: almacenada en una zona de memoria copiada de la memoria de la tarea tipo en el mismo instante en que se creó la región.

También es posible, rechazar la herencia de cierta región que no sea necesaria en el hijo.

Para el caso de *fork* en UNIX, el texto del programa de la tarea tipo se hereda y se comparte con la tarea hija. Lo mismo ocurre para una región que contiene código de biblioteca compartida. Sin embargo, la pila y el montón del se heredan mediante copias de las regiones de la tarea tipo. Además, si se solicita a la tarea tipo que comparta con su hijo cierta región de datos (operación permitida en UNIX System V), podrá configurarse esta región como heredada para compartir.

Los datos de los mensajes no alineados se transfieren entre las tareas con un método similar a la herencia con copia. Mach crea una región en el espacio de direcciones del receptor, y sus conte-

nidos iniciales son una copia de la región que contiene los datos no alineados del emisor. Al contrario de la herencia, la región del receptor no suele ocupar el mismo rango de direcciones que la región emisora. El rango de direcciones de la región de envío puede haber sido previamente utilizada por una región existente en el receptor.

Mach utiliza *copia en escritura* tanto para la herencia con copia como para el paso de mensajes. Para ello Mach realiza una suposición optimista: parte o toda la memoria que es heredada con copia o pasada como datos no alineados de mensajes no será escrita por tarea alguna, incluso cuando existan permisos de escritura.

Para justificar esta suposición, considérese de nuevo la llamada al sistema de UNIX *fork*. Normalmente la utilización de *fork* está seguida de cerca por una llamada a *exec*, que sobreescribe los contenidos del espacio de direcciones, incluyendo el montón y la pila. Si la memoria ha sido copiada físicamente al efectuar *fork*, entonces la mayor parte del copiado se desperdiciará: se modifican pocas páginas entre esas dos llamadas.

Un segundo ejemplo, también importante, considere un mensaje muy largo enviado al servidor de red local para su transmisión. Este mensaje pudiera ser muy largo. Si la tarea emisora no modifica el mensaje, o modifica sólo partes, en el momento de transmitirlo, se podrá ahorrar gran parte de la copia de memoria. El servidor de red no tiene razones para modificar el mensaje. La optimización de *copia en escritura* ayuda a compensar los costes del cambio de contexto que se generan en la transmisión a través de un servidor de red.

En contraposición a la *copia en escritura*, Chorus y DASH [Tzou y Anderson 1991] soportan la comunicación *moviendo* (no copiando) páginas entre espacios de direcciones. El diseño Fbufs aprovecha las manipulaciones de la memoria virtual tanto para realizar la copia como el movimiento de páginas [Druschel y Peterson 1993].

18.6.3. EVALUACIÓN DE LA *COPIA-EN-ESCRITURA*

A pesar de que la *copia en escritura* ayuda a pasar datos de mensajes entre las tareas y el servidor de red, no puede usarse para facilitar la transmisión sobre la red. Esto es porque los computadores involucrados no comparten la memoria física.

La *copia en escritura* es eficiente siempre que haya una cantidad suficiente de datos para enviar. La ventaja de evitar la copia física debe superar los costes de las manipulaciones de la tabla de páginas (y la gestión de la caché si se trata de una caché virtual; véase la Sección 6.3). En Abrossimov y otros [1989] aparecen valores de prestaciones de una implementación Mach en una Sun 3/60; reproducimos algunos de ellos en la Figura 18.7. Estos valores se muestran sólo a efectos ilustrativos y no representan necesariamente las prestaciones de la implementación Mach más actual.

Se dan los tiempos para regiones de dos tamaños, 8 kilobytes (1 página) y 256 kilobytes (32 páginas). Las primeras dos columnas sirven sólo de referencia. La columna *Copia simple* muestra el tiempo necesario para copiar todos los datos entre dos regiones ya existentes (es decir, sin utilizar la *copia en escritura*). La columna *Crear región* indica el tiempo necesario para crear una región inicializada a cero (pero a la que no se accede). El resto de columnas proporcionan los tiempos medidos en los experimentos en los que una región existente fue copiada en otra utilizando la *copia en escritura*. Estos valores incluyen el tiempo necesario para la creación de la región copia, la copia de datos en modificación y la destrucción de la región copia. Para cada tamaño de región, se proporcionan valores asociados a diferentes cantidades de datos modificados en la región fuente.

Tamaño de la región	*Copia simple*	*Crear región*	*Cantidad de datos copiados (en escritura)*		
			0 kilobytes (0 páginas)	*8 kilobytes (1 página)*	*256 kilobytes (32 páginas)*
8 kilobytes	1,4	1,57	2,7	4,82	—
256 kilobytes	44,8	1,81	2,9	5,12	66,4

Nota: todos los tiempos están en milisegundos.

Figura 18.7. Sobrecargas de la *copia en escritura*.

Si comparamos los tiempos para una región con tamaño de una página, existe una sobrecarga de 4,82 − 2,7 = 2,12 milisegundos debidos a la página que se está escribiendo, de los cuales 1,4 milisegundos pueden asignarse a la copia de la página; los restantes 0,72 milisegundos se emplean en la gestión del fallo en escritura y las modificaciones de las estructuras internas de datos de gestión de la memoria virtual. Para el envío de un mensaje de tamaño 8 kilobytes entre dos tareas locales, el envío de datos no alineados (es decir, la utilización de la *copia en escritura*) no parece tener una clara ventaja sobre el envío alineado, en el que simplemente se copia. Las transferencias alineadas implican la realización de dos copias (usuario-núcleo y núcleo-usuario) por lo que se necesitará algo más de 2,8 milisegundos para las dos. Sin embargo, en el peor caso no alineado, el coste es significativamente mayor. Por otra parte, la transmisión de un mensaje no alineado de tamaño 256 kilobytes es mucho menos cara que la transmisión alineada si se mantiene la suposición ya mencionada; e incluso en el peor caso, 66,4 milisegundos es menos que 2 × 44,8 milisegundos que son los necesarios para copiar 256 kilobytes de datos alineados hacia y desde el núcleo.

Como apunte final sobre las técnicas de memoria virtual para la copia y la compartición de datos, nótese que la especificación de los datos implicados debe realizarse cuidadosamente. A pesar de no haberlo mencionado hasta este momento, un usuario puede especificar regiones mediante rangos de direcciones que no comiencen ni terminen en fronteras de páginas. Sin embargo, en Mach es obligado compartir memoria a nivel de página. Todos los datos dentro de las páginas involucradas pero no especificados por los usuarios serán en cualquier caso copiados entre las tareas. Éste es un ejemplo de lo que se conoce como *compartición falsa* (véase también la discusión sobre la granularidad de los datos compartidos en el Capítulo 16).

18.6.4. PAGINADORES EXTERNOS

En la línea de la filosofía de micronúcleo, el núcleo de Mach no soporta directamente archivos o cualquier otra abstracción de almacenamiento externo. En su lugar asume que dichos recursos son implementados por paginadores (*pagers*) externos. Al igual que Multics [Organick 1972], Mach ha elegido el modelo de acceso correlacionado para objetos de memoria. En lugar de acceder a los datos utilizando explícitamente operaciones *lee* y *escribe*, el programador sólo necesita acceder directamente a las correspondientes posiciones de memoria virtual. Una ventaja del acceso correlacionado es su uniformidad: se presenta al programador un único modelo de acceso a los datos, en lugar de dos. Sin embargo, la cuestión de si el acceso correlacionado es preferible a la utilización de operaciones explícitas es compleja en cuanto a sus ramificaciones, especialmente las asociadas a las prestaciones, por lo que no intentaremos aquí tratar con ella. Nos concentraremos ahora en los aspectos distribuidos de la implementación de memoria virtual de Mach. Ésta consta primordialmente del protocolo entre el núcleo y un paginador externo necesario para gestionar la correspondencia de los datos almacenados por este último.

Mach permite, mediante la llamada a *vm_map*, asociar una región con datos contiguos que arrancan de cierta posición de un objeto de almacenamiento. Esta asociación implica que los accesos de lectura a direcciones de la región se satisfacen con datos almacenados en el objeto de memoria, y los datos de la región modificados por accesos de escritura se propagan al objeto de almacenamiento. En general, el objeto de memoria se manejado desde un paginador externo, a pesar de que existe un paginador por defecto, implementado por el propio núcleo. El objeto de memoria

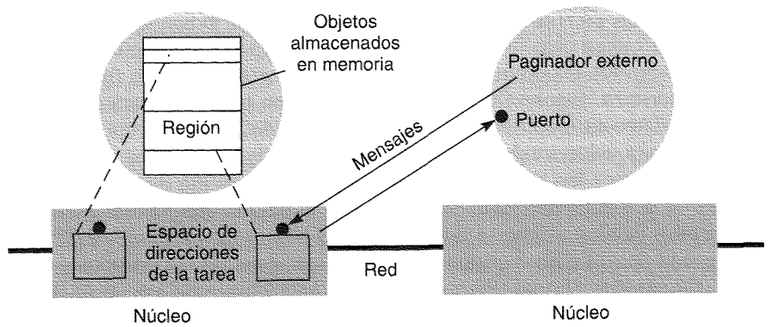


Figura 18.8. Paginador externo.

está representado por los derechos de envío a un puerto utilizado por el paginador externo, el cual satisface las peticiones concernientes al objeto de memoria provenientes del núcleo.

Para cada objeto de memoria correlacionado, el núcleo gestiona un recurso local llamado *objeto cacheado en memoria* (véase la Figura 18.8). En esencia se trata de una lista de páginas que contienen datos respaldados por el correspondiente objeto de memoria.

Las funciones de un paginador externo son (a) almacenar los datos desalojados por el núcleo desde su caché de páginas, (b) proporcionar los datos de las páginas según los vaya solicitando el núcleo, y (c) imponer las limitaciones de consistencia asociadas a la abstracción del recurso de memoria subyacente para el caso en que el recurso de memoria sea compartido y varios núcleos puedan mantener de forma simultánea varios objetos almacenados en memoria para el mismo objeto de memoria.

Los principales componentes del protocolo de paso de mensajes entre el núcleo (N) y el paginador externo (PE) se resumen en la Figura 18.9. Cuando se invoca a *vm_map*, el núcleo local contacta con el paginador externo utilizando el derecho de envío del puerto asociado al objeto de memoria proporcionado en la llamada *vm_map*, enviándole un mensaje de tipo *memory_object_init*. El núcleo inserta en este mensaje los derechos de envío, los cuales son utilizados por el paginador externo para controlar el objeto almacenado en memoria. También declara el tamaño y la posición de los datos solicitados al objeto de memoria, y el tipo de acceso (lectura/escritura). El paginador externo responde con un mensaje de tipo *memory_object_set_attributes*, que indica al núcleo que el paginador está preparado para gestionar solicitudes de datos y proporciona información adicional sobre los requisitos del paginador en relación con el objeto de memoria. Cuando un paginador externo recibe un mensaje *memory_object_init*, es capaz de determinar si necesita o no implementar un protocolo de consistencia, ya que todos los núcleos que quieren acceder al correspondiente objeto de memoria deben enviar este mensaje.

18.6.5. GESTIÓN DE ACCESOS A UN OBJETO DE MEMORIA

Comenzaremos analizando el caso en el que un objeto de memoria no sea compartido (sólo está correlacionado en un computador). Para ser más concretos, utilizaremos un archivo como objeto de memoria. Suponiendo que no se realiza prebúsqueda de los datos del archivo desde el paginador externo, todas las páginas en la región correlacionada con este archivo se protegen inicialmente por el hardware frente a todos los accesos, al no existir ningún dato del archivo. Cuando un hilo intenta leer una de las páginas de la región, se genera un fallo de página. El núcleo busca el derecho de envío del puerto asociado al objeto de memoria correspondiente a la región correlacionada y envía

Evento	*Emisor*	*Mensaje*
vm_map invocado por la tarea	N → PE PE → N PE → N	*memory_object_init* *memory_object_set_atributes*, o *memory_object_data_error*
Faltas de página de la tarea al no existir un marco de página	N → PE PE → N PE → N	*memory_object_data_request* *memory_object_data_provided*, o *memory_object_data_unavailable*
El núcleo escribe una página modificada sobre almacenamiento persistente	N → PE	*memory_object_data_write*
El paginador externo ordena al núcleo escribir una página/poner permisos de acceso	PE → N N → PE	*memory_object_lock_request* *memory_object_lock_completed*
Faltas de página de la tarea al no existir suficientes datos	N → PE PE → N	*memory_object_data_unlock* *memory_object_lock_request*
El objeto de memoria deja de estar mapeado	N → PE	*memory_object_terminate*
Paginador externo retira un objeto de memoria	PE → N N → PE	*memory_object_destroy* *memory_object_terminate*

Figura 18.9. Mensajes del paginador externo.

un mensaje *memory_object_data_request* al paginador externo (que es, en nuestro ejemplo, un servidor de archivos). Si todo se realiza correctamente, el paginador externo responde con los datos de la página, utilizando un mensaje *memory_object_data_provided*.

Cuando el archivo de datos es modificado por el computador que lo ha correlacionado, a veces el núcleo necesitará escribir la página desde su objeto almacenado en memoria. Para ello envía un mensaje de tipo *memory_object_data_write* al paginador externo en el que se inserta la página de datos. Las páginas modificadas se transmiten al paginador externo como en efecto colateral del reemplazo de páginas (cuando el núcleo necesita espacio para otra página). Además, el núcleo puede decidir la escritura de la página en el almacenamiento de respaldo (aunque la deja en la caché de memoria) para garantizar su persistencia. Por ejemplo, las implementaciones de UNIX escriben en disco los datos modificados normalmente cada 30 segundos, en caso de un fallo del sistema. Algunos sistemas operativos permiten a los programas controlar la seguridad de sus datos mediante una operación *flush* sobre un archivo abierto, el cual provoca que todas las páginas modificadas sean escritas en el archivo para el momento en que haya vuelto la llamada.

Los diferentes tipos de recursos de memoria pueden tener diferentes garantías de persistencia. El propio paginador externo puede solicitar al núcleo, mediante un mensaje de tipo *memory_object_lock_request*, que los datos modificados en cierto rango sean enviados al paginador para su consumación sobre el almacén permanente de acuerdo con estas garantías. Cuando el núcleo haya completado las acciones solicitadas, envía al paginador externo un mensaje *memory_object_lock_completed*. (El paginador externo necesita este mensaje porque no puede saber qué páginas han sido modificadas y requieren serle enviadas.)

Observe que todos los mensajes que estamos describiendo son enviados asíncronamente, incluso si se generan en una combinación petición-respuesta. Esto ocurre, en primer lugar, para que los hilos no sean suspendidos y puedan realizar otras tareas después del envío de las solicitudes. Además, un hilo no queda colgado cuando ha enviado una solicitud a un paginador externo o núcleo

que resulta que ha fallado (o cuando el núcleo envía una solicitud a un paginador externo erróneo que no responde). Finalmente, un paginador externo puede utilizar el protocolo de mensajes asíncrono basado en mensajes para implementar una política de prebúsqueda de páginas. Puede enviar datos de páginas en mensajes *memory_object_data_provided* a los objetos cacheados en memoria anticipándose a su utilización, en lugar de esperar que se genere un fallo de páginas y se soliciten los datos.

◇ **Gestión de accesos compartidos a un objeto de memoria.** Supongamos, según nuestro ejemplo, que varias tareas residentes en diferentes computadores correlacionan un mismo archivo. Si la correlación se realiza únicamente para lectura en todas las regiones que lo usan, no existe problema de consistencia y las solicitudes a las páginas de los archivos pueden satisfacerse inmediatamente. Sin embargo, si al menos una tarea correlaciona el archivo para su escritura, entonces el paginador externo (es decir, el servidor de archivos) debe implementar un protocolo para asegurar que las tareas no leen versiones inconsistentes de la misma página.

Se invita al lector a traducir el protocolo de escritura invalidante para obtener consistencia secuencial, descrito en el Capítulo 16, sobre mensajes enviados entre el núcleo de Mach y los paginadores externos.

18.7. RESUMEN

El núcleo de Mach se ejecuta tanto en computadores multiprocesador como monoprocesador conectados por redes. Fue diseñado para facilitar la evolución de los nuevos sistemas distribuidos manteniendo la compatibilidad con UNIX. Muy recientemente, se ha utilizado el micronúcleo Mach 3.0 como base de la la implementación MkLinux del sistema operativo Linux.

El núcleo de Mach es complejo en sí mismo y esto se debe en parte a lo sofisticado de algunas de las posibilidades que es capaz de emular. La interfaz del núcleo está formada por varios cientos de llamadas, siendo muchas de ellas simplemente resguardos que únicamente invocan a llamadas al sistema del tipo *mach_msg*. Un sistema operativo como UNIX no puede ser emulado utilizando únicamente paso de mensajes. Se necesita una gestión de memoria virtual sofisticada, proporcionada por Mach. El modelo de tareas e hilos de Mach y la integración de la gestión de memoria virtual con la comunicación representan una mejora considerable sobre las posibilidades básicas de UNIX, incorporando lecciones aprendidas de los intentos de implementación de servidores UNIX. Su modelo de comunicación entre tareas es funcionalmente rico y extremadamente complejo en su semántica. Sin embargo, debe tenerse presente que sólo unos pocos programadores de sistema necesitarán utilizarlos en su forma más cruda: por ejemplo, sobre este modelo de comunicación se proporcionan canalizaciones de UNIX y llamadas a procedimientos remotos.

A pesar de que el núcleo de Mach se considera a menudo un micronúcleo, su tamaño ronda los 500 kilobytes de código y datos (incluyendo un porcentaje sustancial de código de gestores de dispositivos). Los micronúcleos posteriores, llamados de segunda generación, proporcionan una gestión de la memoria y de la comunicación entre procesos más sencilla.

Los micronúcleos de segunda generación con comunicación entre procesos optimizada superan las prestaciones de Mach en la emulación de UNIX. Por ejemplo, los diseñadores del micronúcleo L4 [Hartig y otros 1997] informan que su implementación de nivel de usuario de Linux es sólo entre un 5 % y un 10 % más lenta que la implementación nativa de Linux sobre la misma máquina. Como contrapunto, indican que la emulación de nivel de usuario de MkLinux para Linux en Mach 3.0 proporciona de media un 50 % de degradación de prestaciones respecto a una implementación de Linux nativa en dicha máquina.

La filosofía de Mach de proporcionar toda la funcionalidad extendida en el nivel de usuario ha sido finalmente abandonada y se permitió que los servidores residieran en el núcleo [Condict y otros 1994]. Sin embargo, Hartig y otros indican que incluso la emulación de MkLinux a nivel de núcleo es alrededor de un 30 % más lenta que el Linux nativo.

A pesar de estas limitaciones de prestaciones en la emulación de UNIX, Mach, junto con Chorus, es un diseño innovador que sigue siendo relevante. Ambos continúan utilizándose y son una referencia inestimable para los desarrollos de arquitecturas de núcleo. El lector puede encontrar información adicional acerca del diseño del núcleo en www.cdk3.net/oss: Amoeba, Chorus y la emulación de UNIX sobre Mach y Chorus.

EJERCICIOS

18.1. ¿En qué se diferencia un núcleo diseñado para operar sobre multiprocesadores de otro pensado sólo para computadores monoprocesador?

18.2. Defina (a nivel binario) la emulación de un sistema operativo. ¿Por qué se considera deseable y, en caso de que lo sea, por qué no se realiza de forma rutinaria?

18.3. Explique por qué se asocian tipos a los contenidos de los mensajes en Mach.

18.4. Discuta si la habilidad del núcleo de Mach para monitorizar el número de derechos de envío sobre un cierto puerto debería extenderse a la red.

18.5. ¿Por qué se proporcionan en Mach conjuntos de puertos cuando también existen los hilos?

18.6. ¿Por qué existe en Mach una única llamada al sistema (*mach_msg*) para las comunicaciones? ¿Cómo se utiliza desde los clientes y los servidores?

18.7. ¿Qué diferencia existe entre un puerto de red y un puerto (local)?

18.8. Un servidor Mach gestiona múltiples recursos *thingumajig*.

(i) Discuta las ventajas e inconvenientes de las siguientes asociaciones:

a) Un único puerto con todos los thingumajigs.
b) Un único puerto para cada thingumajig.
c) Un puerto por cada cliente.

(ii) Un cliente proporciona un identificador thingumajig al servidor, el cual responde con un derecho de puerto. ¿Qué tipo de derecho de puerto debería enviar el servidor al cliente? Explique por qué el identificador del servidor para el derecho de puerto y el del cliente pueden ser diferentes.

(iii) Un cliente thingumajig reside en un computador distinto a la del servidor. Explique detalladamente cómo consigue el cliente llegar a poseer el derecho de puerto que le habilita para comunicarse con el servidor, aunque el núcleo de Mach sólo pueda transmitir derechos de puerto entre tareas locales.

(iv) Explique la secuencia de eventos de comunicación que tienen lugar en Mach cuando el cliente envía un mensaje solicitando una operación sobre un thingumajig, asumiendo de nuevo que el cliente y el servidor residen en computadores diferentes.

18.9. Una tarea Mach en una máquina *A* envía un mensaje a otra tarea en una máquina diferente *B*. ¿Cuántas transiciones de dominio ocurren y cuántas veces se copian los contenidos del mensaje si éste está alineado en la página?

18.10. Diseñe un protocolo para conseguir transparencia en la migración de puertos.

18.11. ¿Cómo puede un gestor de dispositivo, como un gestor de red, operar a nivel de usuario?

18.12. Explique dos tipos de compartición de regiones utilizados por Mach en la emulación de la llamada al sistema UNIX *fork*, suponiendo que el hijo se ejecuta en el mismo computador. Un proceso hijo puede invocar de nuevo a *fork*. Explique cómo esto origina un problema de implementación y sugiera una solución.

18.13. (i) ¿Es necesario que el núcleo elija el rango de direcciones de un mensaje recibido cuando se utiliza *copia en escritura*?

(ii) ¿Utiliza Mach la *copia en escritura* para el envío de mensajes a destinos remotos?

(iii) Una tarea envía un mensaje de 16 kilobytes de forma asíncrona a una tarea local en una máquina de 10 MIPS, 32 bits y tamaño de página de 8 kilobytes. Compare los costes de (1) copia simple de los datos del mensaje (sin usar *copia en escritura*) (2) el mejor caso usando *copia en escritura* y (3) el peor caso usando *copia en escritura*. Realice las siguientes suposiciones:

- La creación de una región vacía de tamaño 16 kilobytes cuesta 1.000 instrucciones.
- La gestión de un fallo de páginas y la asignación de una nueva página en la región necesita 100 instrucciones.

18.14. Resuma las razones para la existencia de los paginadores externos.

18.15. Un archivo es abierto y correlacionado al mismo tiempo por dos tareas que residen en máquinas sin memoria física compartida. Discuta el problema de consistencia que surge de dicha situación. Diseñe un protocolo utilizando los mensajes del paginador externo de Mach que asegure la consistencia secuencial para los contenidos del archivo (véase el Capítulo 16).

REFERENCIAS

REFERENCIAS EN LÍNEA

La lista de referencias en línea en www.cdk3.net/refs proporciona enlaces a los siguientes tipos de referencia:

- Enlaces a versiones en línea de los artículos publicados. La entrada en la lista impresa de referencias se marca con el símbolo www para indicar que el material está disponible en el Web a través de nuestra lista de referencias en línea.
- Enlaces a documentos que existen únicamente en el Web. Éstos están identificados por una etiqueta subrayada, por ejemplo, www.omg.org o Linux AFS y también están marcados con el símbolo ωωω en la lista de referencias impresa que se presenta a continuación.

Los enlaces conducen al documento en sí o a una página de índice que contiene un enlace a documento. Las referencias a los RFC hacen alusión a las series de estándares y especificaciones de Internet llamadas «request for comments» que se encuentran disponibles en el *Internet Engineering Task Force*, en www.ietf.org/frc/ y en otros sitios bien conocidos.

El material en línea escrito o editado por los autores para completar este libro se referencia en el libro por www.cdk3.net, pero no se incluye en la lista de referencias. Por ejemplo, www.cdk3.net/ip se refiere a material adicional sobre comunicación entre procesos que se encuentra en nuestras páginas web.

Abadi y Gordon 1999 — Abadi, M. y Gordon, A. D. (1999). A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, vol. 148, n.º. 1, pp. 1-70, enero. ωωω

Abadi y otros 1998 — Abadi, M., Birrell, A. D., Stata, R. y Wobber, E. P (1998). Secure Web tunneling. En *Proceedings 7th International World Wide Web Conference*, pp. 531-9. Elsevier, en *Computer Networks and ISDN Systems*, volumen 30, n.os 1-7. ωωω

Abrossimov y otros 1989 — Abrossimov, V., Rozier, M. y Shapiro, M. (1989). Generic virtual memory management for operating system kernels. *Proceedings of 12th ACM Symposium on Operating System Principles*, pp. 123-36, diciembre. ωωω

Accetta y otros 1986 — Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. y Young, M. (1986). Mach: A new kernel foundation for UNIX development. En *Proceedings Summer* 1986 *USENIX Conference* pp. 93-112.

Adjie-Winoto y otros 1999 — Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. y Lilley, J. (1999). The design and implementation of an intentional naming system. En *Proceedings 17th ACM Symposium on Operating System Principles*, publicado como *Operating Systems Review*, vol. 34, n.° 5, pp. 186-201.

Adve y Hill 1990 — Adve, S. y Hill, M. (1990). Weak ordering – a new definition. En *Proceedings 17th. Annual Symposium on Computer Architecture*, IEEE, pp. 2-14.

Agrawal y otros 1987 — Agrawal, D., Bernstein, A., Gupta, P. y Sengupta, S. (1987). Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, vol. 2: pp. 45-59. Springer-Verlag.

Ahamad y otros 1992 — Ahamad, M., Bazzi, R., John, R., Kohli, P. y Neiger, G. (1992). *The Power of Processor Consistency*. Technical report GIT-CC-92/34, Georgia Institute of Technology, Atlanta. ωωω

Anderson 1993 — Anderson, D. P. (1993). Meta-scheduling for distributed continuous media. *ACM Transactions on Computer Systems*, vol. 11, n.° 3.

Anderson y otros 1990a — Anderson, D. P., Herrtwich, R. G. y Schaefer, C. (1990). *SRP – A Resource Reservation Protocol for Guaranteed- Performance Communication in the Internet*. Technical report 90-006, International Computer Science Institute, Berkeley, Calif.

Anderson y otros 1990b — Anderson, D. P., Tzou, S., Wahbe, R., Govindan, R. y Andrews, M. (1990). Support for continuous media in the DASH System. *Tenth International Conference on Distributed Computing Systems*, Paris.

Anderson y otros 1991 — Anderson, T., Bershad, B., Lazowska, E. y Levy, H. (1991). Scheduler activations: efficient kernel support for the user- level management of parallelism. En *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 95-109.

Anderson y otros 1995 — Anderson, T., Culler, D., Patterson, D. y the NOW team. (1995). A case for NOW (Networks Of Workstations), *IEEE Micro*, vol. 15, n.° 1.

Anderson y otros 1996 — Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S. y Wang, R. Y. (1996). Serverless Network File Systems. *ACM Trans. on Computer Systems* 14,1. pp. 41-79, febrero. ωωω

ANSA 1989 — ANSA (1989). *The Advanced Network Systems Architecture (ANSA) Reference Manual*. Castle Hill, Cambridge, England: Architecture Project Management.

ANSI 1985 — American National Standards Institute (1985). *American National Standard for Financial Institution Key Management*, Standard X9.17 (revisado).

Arnold y otros 1999 — Arnold, K., O'Sullivan, B., Scheifler, R. W., Waldo, J. y Wollrath, A. (1999). *The Jini Specification*, Reading, Mass: Addison-Wesley. ωωω

Attiya y Welch 1998 — Attiya, H., y Welch, J. (1998). *Distributed Computing – Fundamentals, Simulations and Advanced Topics*, McGraw-Hill.

Babaoglu y otros 1998 — Babaoglu, O., Davoli, R., Montresor, A. y Segala, R. (1998). System support for partition-aware network applications. En *Proceedings 18th International Conference on Distributed Computing Systems* (ICDCS '98), pp. 184-191.

Bacon 1998 — Bacon, J. (1998). *Concurrent Systems*, segunda edición. Wokingham, England: Addison-Wesley.

Baker 1997 — Baker, S. (1997). *CORBA Distributed Objects Using Orbix*, Harlow, England: Addison-Wesley.

Bal y otros 1990 — Bal, H. E., Kaashoek, M. F. y Tanenbaum, A. S. (1990). Experience with distributed programming in Orca. En *Proceedings International Conference on Computer Languages '90*, IEEE, pp. 79-89.

Balakrishnan y otros 1995 — Balakrishnan, H., Seshan, S. y Katz, R. H. (1995). Improving reliable transport and hand-off performance in cellular wireless networks. En *Proceedings ACM Mobile Computing and Networking Conference*, ACM, pp. 2-11.

Balakrishnan y otros 1996 — Balakrishnan, H., Padmanabhan, V., Seshan, S. y Katz, R. (1996). A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *Proceedings of the ACM SIGCOMM '96 Conference*, pp. 256-69.

Banerjea y Mah 1991 — Banerjea, A. y Mah, B.A. (1991). The real-time channel administration protocol. *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg.

Baran 1964 — Baran, P. (1964). *On Distributed Communications*. Research Memorandum RM-3420-PR, Rand Corporation. ωωω

Barborak y otros 1993 — Barborak, M., Malek, M. y Dahbura, A. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, vol. 25, n.° 2, pp. 171-220.

Barghouti y Kaiser 1991 — Barghouti, N. S. y Kaiser G. E. (1991). Concurrency control in advanced database applications. *Computing Surveys*, vol. 23, n.° 3, pp. 269-318.

Bartoli y otros 1993 — Bartoli, A., Mullender, S. J. y van der Valk, M. (1993). Wide-address spaces – exploring the design space. *ACM Operating Systems Review*, vol. 27, n.° 1, pp. 11-17.

Bates y otros 1996 — Bates, J., Bacon, J., Moody, K. y Spiteri, M. (1996). Using events for the scalable federation of heterogeneous components, *European SIGOPS Workshop*.

Bell y LaPadula 1975 — Bell, D. E. y LaPadula, L. J. (1975). *Computer Security Model: Unified Exposition and Multics Interpretation*, Mitre Corporation, 1975.

Bellman 1957 — Bellman, R. E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bellovin y Merritt 1990 — Bellovin, S. M. y Merritt, M. (1990). Limitations of the Kerberos authentication system. *ACM Computer Communications Review*, vol. 20, n.°. 5, pp. 119-32.

Berners-Lee 1991 — Berners-Lee, T. (1991). World Wide Web Seminar. ωωω

Berners-Lee 1999 — Berners-Lee, T. (1999). *Weaving The Web*. HarperCollins.

Bernstein y otros 1980 — Bernstein, P. A., Shipman, D. W. y Rothnie, J. B. (1980). Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions Database Systems*, vol. 5, n.° 1, pp. 18-51.

Bernstein y otros 1987 — Bernstein, P., Hadzilacos, V. y Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison-Wesley.

Bershad y otros 1990 — Bershad, B., Anderson, T., Lazowska, E. y Levy, H. (1990). Lightweight remote procedure call. *ACM Transactions Computer Systems*, vol. 8, n.° 1, pp. 37-55.

Bershad y otros 1991 — Bershad, B., Anderson, T., Lazowska, E. y Levy, H. (1991). User-level interprocess communication for shared memory multiprocessors. *ACM Transactions Computer Systems*, vol. 9, n.° 2, pp. 175-198.

Bershad y otros 1993 — Bershad, B., Zekauskas, M. y Sawdon, W. (1993). The Midway distributed shared memory system. En *Proceedings IEEE COMPCON Conference*, IEEE, pp. 528-37.

Bershad y otros 1995 — Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. y Eggers, S. (1995). Safety and performance in the SPIN operating system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267-84.

Bhatti y Friedrich 1999 — Bhatti, N. y Friedrich, R. (1999). *Web Server Support for Tiered Services*. Hewlett-Packard Corporation Technical Report HPL-1999-160.

Birman 1993 — Birman, K. P. (1993). The process group approach to reliable distributed computing. *Comms. ACM*, vol. 36, n.° 12, pp. 36-53.

Birman 1996 — Birman, K. P. (1996). *Building Secure and Reliable Network Applications*. Greenwich, CT: Manning.

Birman y Joseph 1987a — Birman, K. P. y Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Transactions Computer Systems*, vol. 5, n.° 1, pp. 47-76.

Birman y Joseph 1987b — Birman, K., y Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. En *Proceedings 11th ACM Symposium on Operating Systems Principles*, pp. 123-38.

Birman y otros 1991 — Birman, K. P., Schiper, A. y Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions Computer Systems*, vol. 9, n.° 3, pp. 272-314.

Birrell y otros 1995 — Birrell, A., Nelson, G. y Owicki, S. (1993). Network objects. En *Proceedings 14th ACM Symposium on Operating Systems Principles*, pp. 217-30.

Birrell y Needham 1980 — Birrell, A. D. y Needham, R. M. (1980). A universal file server. *IEEE Transactions Software Engineering*, vol. SE-6, n.° 5, pp. 450-3.

Birrell y Nelson 1984 — Birrell, A. D. y Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions Computer Systems*, vol. 2, pp. 39-59.

Birrell y otros 1982 — Birrell, A. D., Levin, R., Needham, R. M. y Schroeder, M. D. (1982). Grapevine: an exercise in distributed computing. *Comms. ACM*, vol. 25, n.° 4, pp. 260-73.

Bisiani y Forin 1988 — Bisiani, R. y Forin, A. (1988). Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions Computers*, vol. 37, n.° 8, pp. 930-45.

Bisiani y Ravishankar 1990 — Bisiani, R. y Ravishankar, M. (1990). Plus: a distributed shared memory system. En *Proceedings 17th International Symposium on Computer Architecture*, pp. 115-24.

Black 1990 — Black, D. (1990). Scheduling support for concurrency and parallelism in the Mach operating system, *IEEE Computer*, vol. 23, n.° 5, pp. 35-43.

Black y Artsy 1990 — Black, A. y Artsy, Y. (1990). Implementing location independent invocation, *IEEE Transactions Parallel and Distributed Systems*, vol. 1, n.° 1.

Blair y Stefani 1997 — Blair, G. S. y Stefani, J.-B. (1997). *Open Distributed Processing and Multimedia*. Harlow, England: Addison-Wesley.

Blakley 1999 — Blakley, R. (1999). *CORBA Security – An Introduction to Safe Computing with Objects*. Reading, Mass.: Addison-Wesley.

Bolosky y otros 1996 — Bolosky, W., Barrera, J., Draves, R., Fitzgerald, R., Gibson, G., Jones, M., Levi, S., Myhrvold, N. y Rashid, R. (1996). The Tiger video fileserver, *6th NOSSDAV Conference*, Zushi, Japón, abril. ωωω

Bolosky y otros 1997 — Bolosky, W., Fitzgerald, R. y Douceur, J. (1997). Distributed schedule management in the Tiger video fileserver, *16th ACM Symposium on Operating System Principles*, pp. 212-23, St. Malo, Francia, octubre. ωωω

Bonnaire y otros 1995 — Bonnaire, X., Baggio, A. y Prun, D. (1995). Intrusion free monitoring: an observation engine for message server based applications. En *Proceedings of the 10th International Symposium on Computer and Information Sciences* (ISCIS X), pp. 541-48.

Borenstein y Freed 1993 — Borenstein, N. y Freed, N., (1993). *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, septiembre. Internet RFC 1521. ωωω

Bowman y otros 1990 — Bowman, M., Peterson, L. y Yeatts, A. (1990). Univers: an attribute-based name server. *Software-Practice y Experience*, vol. 20, n.° 4, pp. 403-24.

Box 1998 — Box, D. (1998). *Essential COM*. Reading, Mass: Addison-Wesley.

Boykin y otros 1993 — Boykin, J., Kirschen, D., Langerman, A. y LoVerso, S. (1993). *Programming under Mach*. Reading, Mass.: Addison-Wesley.

Buford 1994 — Buford, J. K. (1994). *Multimedia Systems*. Addison-Wesley.

Burns y Wellings 1998 — Burns, A. y Wellings, A. (1998). *Concurrency in Ada*, Cambridge University Press.

Burrows y otros 1989 — Burrows, M., Abadi, M. y Needham, R. (1989). *A logic of authentication*. Technical Report 39, Palo Alto, Calif.: Digital Equipment Corporation Systems Research Center.

Burrows y otros 1990 — Burrows, M., Abadi, M. y Needham, R. (1990). A logic of authentication. *ACM Transactions Computer Systems*, vol. 8, pp. 18-36.

Bush 1945 — Bush, V. (1945). As we may think. *The Atlantic Monthly*, julio. ωωω

Callaghan 1996a — Callaghan, B. (1996). *WebNFS Client Specification*, Internet RFC 2054, octubre. ωωω

Callaghan 1996b — Callaghan, B. (1996). *WebNFS Server Specification*, Internet RFC 2055, octubre. ωωω

Callaghan 1999 — Callaghan, B. (1999). *NFS Illustrated*, Reading, Mass.: Addison-Wesley.

Callaghan y otros 1995 — Callaghan, B., Pawlowski, B. y Staubach, P. (1995). *NFS Version 3 Protocol Specification*, Internet RFC 1813, Sun Microsystems, junio. ωωω

Campbell 1997 — Campbell, R. (1997). *Managing AFS: The Andrew File System*, Prentice-Hall.

Campbell y otros 1993 — Campbell, R., Islam, N., Raila, D. y Madany, P. (1993). Designing and implementing Choices: an object-oriented system in C++. *Comms. ACM*, vol. 36, n.° 9, pp. 117-26.

Canetti y Rabin 1993 — Canetti, R. y Rabin, T. (1993). Fast asynchronous byzantine agreement with optimal resilience. En *Proceedings 25th ACM Symposium on Theory of Computing*, pp. 42-51.

Carriero y Gelernter 1989 — Carriero, N. y Gelernter, D. (1989). Linda in context. *Comms. ACM*, vol. 32, n.° 4, pp. 444-58.

Carter y otros 1991 — Carter, J. B., Bennett, J. K. y Zwaenepoel, W. (1991). Implementation and performance of Munin. En *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 152-64.

Carter y otros 1998 — Carter, J., Ranganathan, A. y Susarla, S. (1998). Khazana, An Infrastructure for Building Distributed Services, En *Proceedings of ICDCS '98*, Amsterdam, The Netherlands. ωωω

CCITT 1988a — CCITT (1988). *Recommendation X.500 : The Directory – Overview of Concepts, Models and Service*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

CCITT 1988b — CCITT (1988). *Recommendation X.509: The Directory – Authentication Framework*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

CCITT 1990 — CCITT (1990). *Recommendation I.150 : B-ISDN ATM Functional Characteristics*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.

Ceri y Owicki 1982 — Ceri, S. y Owicki, S. (1982). On the use of optimistic methods for concurrency control in distributed databases. En *Proceedings 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, Calif. pp. 117-30.

Ceri y Pelagatti 1985 — Ceri, S. y Pelagatti, G. (1985). *Distributed Databases – Principles and Systems*. McGraw-Hill.

Chandra y Toueg 1996 — Chandra, T. y Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, abril, pp. 374-82.

Chandy y Lamport 1985 — Chandy, K. y Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, vol. 3, n.° 1, pp. 63-75.

Chang y Maxemchuk 1984 — Chang, J. y Maxemchuk, N. (1984). Reliable Broadcast Protocols, *ACM Transactions on Computer Systems*, vol. 2. n.° 3. pp. 251-75.

Chang y Roberts 1979 — Chang, E.G. y Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Comms. ACM*, vol. 22, n.° 5, pp. 281-3.

Charron-Bost 1991 — Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, vol. 39, julio, pp. 11-16.

Chen y otros 1994 — Chen, P., Lee, E., Gibson, G., Katz, R. y Patterson, D. (1994). RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, vol. 26, n.° 2, junio, pp. 145-188.

Cheng 1998 — Cheng, C. K. (1998). *A survey of media servers*. Hong Kong University CSIS, noviembre, ωωω

Cheriton 1984 — Cheriton, D. R. (1984). The V kernel: a software base for distributed systems. *IEEE Software*, vol. 1 n.° 2, pp. 19-42.

Cheriton 1985 — Cheriton, D. R. (1985). Preliminary thoughts on problem- oriented shared memory: a decentralized approach to distributed systems. *ACM Operating Systems Review*, vol. 19, n.° 4, pp. 26-33.

Cheriton 1986 — Cheriton, D. R. (1986). VMTP: A protocol for the next generation of communication systems. En *Proceedings SIGCOMM '86 Symposium on Communication Architectures and Protocols*, ACM, pp. 406-15.

Cheriton y Mann 1989 — Cheriton, D. y Mann, T. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions Computer Systems*, vol. 7, n.° 2, pp. 147-83.

Cheriton y Skeen 1993 — Cheriton, D. y Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. En *Proceedings 14th ACM Symposium on Operating System Principles*, diciembre, pp. 44-57.

Cheriton y Zwaenepoel 1985 — Cheriton, D. R. y Zwaenepoel, W. (1985). Distributed process groups in the V kernel. *ACM Transactions Computer Systems*, vol. 3, n.° 2, pp. 77-107.

Cheswick y Bellovin 1994 — Cheswick, E. R. y Bellovin, S. M. (1994). *Firewalls and Internet Security*. Reading, Mass.: Addison-Wesley.

Choudhary y otros 1989 — Choudhary, A., Kohler, W., Stankovic, J. y Towsley, D. (1989). A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions Software Engineering*, vol. 15, n.° 1.

Clark 1982 — Clark, D. D. (1982). *Window and Acknowledgement Strategy in TCP*, Internet RFC 813. ωωω

Comer 1991 — Comer, D. E. (1991). *Internetworking with TCP/IP, Volume* 1: *Principles, Protocols and Architecture*. Segunda edición. Englewood Cliffs, NJ: Prentice-Hall.

Comer 1995 — Comer, D. E. (1995). *The Internet Book*. Englewood Cliffs, NJ. Prentice-Hall.

Condict y otros 1994 — Condict, M., Bolinger, D., McManus, E., Mitchell, D. y Lewontin, S. (1994). *Microkernel modularity with integrated kernel performance*. Technical report, OSF Research Institute, Cambridge, Mass, abril.

Coulouris y otros 1998 — Coulouris, G. F., Dollimore, J. B. y Roberts, M. (1998). Role and task-based access control in the PerDiS groupware platform. *Third ACM Workshop on Role-Based Access Control*, George Mason University, Washington DC, octubre 22-23. ωωω

Cristian 1989 — Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, vol. 3, pp. 146-58.

Cristian 1991a — Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Comms. ACM*, vol. 34, n.° 2.

Cristian 1991b — Cristian, F. (1991). Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, Springer Verlag, vol. 4, pp 175-87.

Cristian y Fetzer 1994 — Cristian, F., y Fetzer, C. (1994). Probabilistic Internal Clock Synchronization. En *Proceedings 13th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, octubre 25-27, pp. 22-31.

Crow y otros 1997 — Crow, B., Widjaja, I., Kim, J. y Sakai, P. (1997). IEEE 802.11 Wireless local area networks. *IEEE Communications Magazine*, septiembre, 1997, pp. 116-26.

cryptography.org — *North American Cryptography Archives*. ωωω

Curtin y Dolske 1998 — Kurtin, M. y Dolski, J. (1998). A brute force search of DES Keyspace. *;login: – the Newsletter of the USENIX Association*, mayo. ωωω

Custer 1998 — Custer, H. (1998). *Inside Windows NT*, segunda edición. Microsoft Press.

Czerwinski y otros 1999 — Czerwinski, S., Zhao, B., Hodes, T., Joseph, A. y Katz, R. (1999). An architecture for a secure discovery service. En *Proceedings Fifth Annual International Conference on Mobile Computing and Networks*.

Dasgupta y otros 1991 — Dasgupta, P., LeBlanc Jr., R. J. Ahamad, M. y Ramachandran, U. (1991). The Clouds distributed operating system. *IEEE Computer*, vol. 24, n.° 11, pp. 34-44.

Davidson 1984 — Davidson, S. B. (1984). Optimism and consistency in partitioned database systems. *ACM Transactions Database Systems*, vol. 9, n.° 3, pp. 456-81.

Davidson y otros 1985 — Davidson, S. B., García-Molina, H. y Skeen, D. (1985). Consistency in partitioned networks. *Computing Surveys*, vol. 17, n.° 3, pp. 341-70.

DEC 1990 — Digital Equipment Corporation. (1990). *In Memoriam: J. C. R. Licklider* 1915-1990, Technical Report 61, DEC Systems Research Center. ωωω

Delgrossi y otros 1993 — Delgrossi, L., Halstrick, C., Hehmann, D., Herrtwich, R. G., Krone, O., Sandvoss, J. y Vogt, C. (1993). Media scaling for audiovisual communication with the Heidelberg transport system. *ACM Multimedia '93*, Anaheim, Calif.

Demers y otros 1989 — Demers, A., Keshav, S., Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM* '89.

Denning y Denning 1977 — Denning, D. y Denning, P. (1977). Certification of programs for secure information flow. *Comms. ACM*, vol. 20, n.° 7, pp. 504-13.

Dertouzos 1974 — Dertouzos, M. L. (1974). Control robotics – the procedural control of physical processes. *IFIP Congress*.

Dierks y Allen 1999 — Dierks,T. y Allen, C. (1999). *The TLS Protocol Version* 1.0, Internet RFC 2246. ωωω

Diffie 1988 — Diffie, W. (1988). The first ten years of public-key cryptography. *Proceedings of the IEEE*, vol. 76, n.° 5, mayo 1988, pp. 560-77.

Diffie y Hellman 1976 — Diffie, W. y Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions Information Theory*, vol. IT-22, pp. 644-54.

Diffie y Landau 1998 — Kahn, D. y Landau, S. (1998). *Privacy on the Line*. Cambridge, Mass: MIT Press.

Dijkstra 1959 — Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematic*, vol. 1, pp. 269-71.

Dolev y Malki 1996 — Dolev, D. y Malki, D. (1996). The Transis approach to high availability cluster communication. *Comms. ACM*, vol. 39, n.° 4, pp. 64-70.

Dolev y Strong 1983 — Dolev, D. y Strong, H. (1983). Authenticated algorithms for byzantine agreement. *SIAM Journal of Computing*, vol. 12, n.° 4, pp. 656-66.

Dolev y otros 1986 — Dolev, D., Halpern, J., y Strong, H. (1986). On the possibility and impossibility of achieving clock synchronization. *Journal of Computing Systems Science* 32, 2 (abril), pp. 230-50.

Dorcey 1995 — Dorcey, T. (1995) CU-SeeMe Desktop Video Conferencing Software, *Connexions*, vol. 9, no. 3 (marzo).

Douceur y Bolosky 1999 — Douceur, J.R. y Bolosky, W. (1999). Improving responsiveness of a stripe-scheduled media server. SPIE Proceedings, vol. 3.654. *Multimedia Computing and Networking*. pp. 192- 203. ωωω

Douglis y Ousterhout 1991 — Douglis, F. y Ousterhout, J. (1991). Transparent process migration: design alternatives and the Sprite implementation, *Software – Practice and Experience*, vol. 21, n.° 8, pp. 757-89.

Draves 1990 — Draves, R. (1990). A revised IPC interface, In *Proceedings USENIX Mach Workshop*, pp. 101-21, octubre.

Draves y otros 1989 — Draves, R. P., Jones, M. B. y Thompson, M. R. (1989). *MIG – the Mach Interface Generator*. Technical Report, Dept. of Computer Science, Carnegie-Mellon University.

Druschel y Peterson 1993 — Druschel, P. y Peterson, L. (1993). Fbufs: a high-bandwidth cross-domain transfer facility. En *Proceedings 14th ACM Symposium on Operating System Principles*, pp. 189-202.

Dubois y otros 1988 — Dubois, M., Scheurich, C. y Briggs, F. A. (1988). Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, vol. 21, n.° 2, pp. 9-21.

Dwork y otros 1988 — Dwork, C., Lynch, N. y Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, vol. 35, n.° 2, pp. 288-323.

Eager y otros 1986 — Eager, D., Lazowska, E. y Zahorjan, J. (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, vol. SE-12, n.° 5, pp. 662-675.

EFF 1998 — Electronic Frontier Foundation (1998). *Cracking DES, Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastapol Calif.: O'Reilly & Associates.

Eisler y otros 1997 — Eisler, M., Chiu, A, y L. Ling, L. (1997). *RPCSEC—GSS Protocol Specification*. Sun Microsystems. Internet RFC 2203, septiembre. ωωω

El Abbadi y otros 1985 — El Abbadi, A., Skeen, D. y Cristian, C. (1985). An efficient fault-tolerant protocol for replicated data management. En *4th Annual ACM SIGACT/SIGMOD Symposium on Principles of Data Base Systems*, Portland, Ore.

Ellis y otros 1991 — Ellis, C., Gibbs, S. y Rein, G. (1991). Groupware – some issues and experiences. *Comms. ACM*, vol. 34, n.° 1, pp. 38-58.

Ellison 1996 — Ellison, C. (1996). Establishing identity without certification authorities. En *6th USENIX Security Symposium*, San José, julio 22-25. ωωω

Ellison y otros 1999 — Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. y Ylonen, T. (1999). *SPKI Certificate Theory*. Internet RFC 2693, septiembre. ωωω

Farley 1998 — Farley, J. (1998). *Java Distributed Computing*. Cambridge, Mass: O'Reilly.

Farrow 2000 — Farrow, R. (2000). Distributed denial of service attacks – how Amazon, Yahoo, eBay and others were brought down. *Network Magazine*, abril. ωωω

Ferrari y Verma 1990 — Ferrari, D. y Verma, D. (1990). A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, vol. 8, n.° 4.

Ferreira y otros 2000 — Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., García, J., Kloostermann, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D. y Krakowiak, S. (2000). PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. En *LNCS* 1752: *Advances in Distributed Systems*. Springer-Verlag Berlin, Heidelberg, New York. pp. 427-53. ωωω

Fidge 1991 — Fidge, C. (1991). Logical Time in Distributed Computing Systems. *IEEE Computer*, vol. 24, n.° 8, pp. 28-33.

Fielding y otros 1999 — Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P. y Berners-Lee T. (1999). *Hypertext Transfer Protocol – HTTP*/1.1. Internet RFC 2616. ωωω

Fischer 1983 — Fischer, M. (1983). The Consensus Problem in Unreliable Distributed Systems (a Brief Survey). En M. Karpinsky, ed., *Foundations of Computation Theory*, vol. 158 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 127-140. Yale University Technical Report YALEU/DCS/TR- 273.

Fischer y Lynch 1982 — Fischer, M. y Lynch, N. (1982). A lower bound for the time to assure interactive consistency. *Inf. Process. Letters*, vol. 14, n.° 4, junio, pp. 183-6.

Fischer y Michael 1982 — Fischer, M. J. y Michael, A. (1982). Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. En *Proceedings Symposium on Principles of Database Systems*, ACM, pp. 70-5.

Fischer y otros 1985 — Fischer, M., Lynch, N. y Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, vol. 32, n.° 2, abril, pp. 374-82.

Fitzgerald y Rashid 1986 — Fitzgerald, R. y Rashid, R. F. (1986). The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions Computer Systems*, vol. 4, n.° 2, pp. 147-77.

Flanagan 1997 — Flanagan, D. (1997). *Java in a Nutshell*. Cambridge, England: O'Reilly.

Fleisch y Popek 1989 — Fleisch, B. y Popek, G. (1989). Mirage: a coherent distributed shared memory design. En *Proceedings 12th ACM Symposium on Operating System Principles*, diciembre, pp. 211-23.

Floyd 1986 — Floyd, R. (1986). *Short term file reference patterns in a UNIX environment*. Technical Rep. TR 177, Rochester, NY: Dept. of Computer Science, University of Rochester.

Floyd y Johnson 1993 — Floyd, S. y Jacobson, V. (1993). The Synchronization of Periodic Routing Messages. *ACM Sigcomm '93 Symposium*.

Floyd y otros 1997 — Floyd, S., Jacobson, V., Liu, C., McCanne, S. y Zhang, L. (1997). A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, vol. 5, n.° 6, pp. 784-803.

Ford y Fulkerson 1962 — Ford, L. R. y Fulkerson, D. R. (1962). *Flows in Networks*. Princeton, NJ: Princeton University Press.

Fox y otros 1997 — Fox, A., Gribble, S., Chawathe, Y., Brewer, E. y Gauthier, P. (1997). Cluster-based scalable network services. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 78-91.

Garay y Moses 1993 — Garay, J., y Moses, Y. (1993). Fully polynomial Byzantine agreement in $t + 1$ rounds. En *Proceedings 25th ACM symposium on theory of computing*, ACM Press, pp. 31-41, mayo.

García-Molina 1982 — García-Molina, H. (1982). Elections in Distributed Computer Systems. *IEEE Transactions on Computers*, vol. C-31, n.° 1, pp. 48-59.

García-Molina y Spauster 1991 — García-Molina, H. y Spauster, A. (1991). Ordered and Reliable Multicast Communication. *ACM Transactions Computer Systems*, vol. 9, n.° 3, pp. 242-71.

Garfinkel 1994 — Garfinkel, S. (1994). *PGP: Pretty Good Privacy*, O'Reilly.

Gharachorloo y otros 1990 — Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. y Hennessy, J. (1990). Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. En *Proceedings 17th. Annual International Symposium on Computer Architecture*, Mayo, pp. 15-26.

Gibbs y Tsichritzis 1994 — Gibbs, S. J. y Tsichritzis, D. C. (1994). *Multimedia Programming*. Addison-Wesley.

Gifford 1979 — Gifford, D. K. (1979). Weighted voting for replicated data. En *Proceedings 7th Symposium on Operating Systems Principles*, ACM, pp. 150-62.

Glassman y otros 1995 — Glassman, S., Manasse, M., Abadi, M., Gauthier, P. y Sobalvarro, P. (1995). The Millicent Protocol for Inexpensive Electronic Commerce. *Fourth International WWW Conference*, diciembre. ωωω

Gokhale y Schmidt 1996 — Gokhale, A. y Schmidt, D. (1996). Measuring the Performance of Communication Middleware on High-Speed Networks. *Proceedings of SIGCOMM '96*, ACM, pp. 306-17.

Golding y Long 1993 — Golding, R. y Long, D. (1993). *Modeling replica divergence in a weak-consistency protocol for global-scale distributed databases*. Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz.

Gong 1989 — Gong, L. (1989). A Secure Identity-Based Capability System. En *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, Calif., mayo, pp. 56-63. ωωω

Goodman 1989 — Goodman, J. (1989). *Cache Consistency and Sequential Consistency*. Technical Report 61, SCI Committee.

Gordon 1984 — Gordon, J. (1984). *The Story of Alice and Bob*. ωωω

Govindan y Anderson 1991 — Govindan, R. y Anderson, D.P. (1991). Scheduling and IPC Mechanisms for Continuous Media. *ACM Operating Systems Review*, vol. 25, n.° 5, pp. 68-80.

Gray 1978 — Gray, J. (1978). Notes on database operating systems. En *Operating Systems: an Advanced Course. Lecture Notes in Computer Science*, vol. 60, pp. 394-481, Springer-Verlag.

Guerraoui y otros 1998 — Guerraoui, R., Felber, P., Garbinato, B. y Mazouni, K. (1998). System support for object groups. En *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications* (OOPSLA'98).

Gusella y Zatti 1989 — Gusella, R. y Zatti, S. (1989). The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions Software Engineering*, vol. 15, pp. 847-53.

Guttman 1999 — Guttman, E. (1999). Service Loocation Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, vol. 3, n.º 4, pp. 71-80.

Hadzilacos y Toueg 1994 — Hadzilacos, V. y Toueg, S. (1994). *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*, Technical report, Dept. of Computer Science, University of Toronto.

Härder 1984 — Härder, T. (1984). Observations on Optimistic Concurrency Control Schemes. *Information Systems*, vol. 9, n.º 2, pp. 111-20.

Härder y Reuter 1983 — Härder, T. y Reuter, A. (1983). Principles of Transaction- Oriented Database Recovery. *Computing Surveys*, vol. 15, n.º 4.

Härtig y otros 1997 — Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., y Wolter, J. (1997). The performance of kernel-based systems. En *Proceedings 16th ACM Symposium on Operating System Principles*, pp. 66-77.

Hartman y Ousterhout 1995 — Hartman, J. y Ousterhout, J. (1995). The Zebra Striped Network File System. *ACM Trans. on Computer Systems*, vol. 13, n.º 3, agosto, pp. 274-310.

Hayton y otros 1998 — Hayton R., Bacon J. y Moody K. (1998). OASIS: Access Control in an Open, Distributed Environment, en *Proceedings IEEE Symposium on Security and Privacy*, Oakland, Calif., pp 3- 14, mayo. ωωω

Hedrick 1988 — Hedrick, R. (1988). *Routing Information Protocol*, Internet RFC 1058. ωωω

Henning 1998 — Henning, M. (1998). Binding, Migration and Scalability in CORBA, *Comms. ACM*, Octubre, vol. 41, n.º 10. pp. 62-71.

Henning y Vinoski 1999 — Henning, M. y Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Reading, Mass.: Addison-Wesley.

Herlihy 1986 — Herlihy, M. (1986). A Quorum-Consensus Replication Method for Abstract Data Types. ACM *Transactions Computer Systems*, vol. 4, n.º 1, pp. 32-53.

Herlihy y Wing 1990 — Herlihy, M. y Wing, J. (1990). On Linearizability: a correctness condition for concurrent objects. *ACM Transactions on programming languages and systems*, vol. 12, n.º 3, (julio), pp. 463-92.

Herrtwich 1995 — Herrtwich, R .G. (1995). Achieving Quality of Service for Multimedia Applications. *ERSADS '95, European Research Seminar on Advanced Distributed Systems*, l'Alpe d'Huez, Francia, abril.

Hirsch 1997 — Hirsch, F. J. (1997). Introducing SSL and Certificates using SSLeay. *World Wide Web Journal*, vol. 2, n.º 3, verano. ωωω

Howard y otros 1988 — Howard, J. H., Kazar, M. L., Menees, S. G, Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N. y West, M. J. (1988). Scale and Performance in a Distributed File System. *ACM Transactions Computer Systems*, vol. 6, n.º 1, pp. 51-81.

Huang y otros 2000 — Huang, A., Ling, B., Barton, J. y Fox, A. (2000). Running the Web backwards: appliance data services. *Proceedings 9th international World Wide Web conference*. ωωω

Huitema 1995 — Huitema, C. (1995). *Routing in the Internet*. Englewood Cliffs, NJ: Prentice-Hall.

Huitema 1998 — Huitema, C. (1998). *IPv6 – the New Internet Protocol*. Upper Saddle River, NJ: Prentice-Hall.

Hunter y Crawford 1998 — Hunter, J. y Crawford, W. (1998). *Java Servlet Programming*, O'Reilly.

Hutchinson y Peterson 1991 — Hutchinson, N. y Peterson, L. (1991). The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, vol. 17, n.º 1, pp. 64-76.

Hutto y Ahamad 1990 — Hutto, P. y Ahamad, M. (1990). Slow memory: weakening consistency to enhance concurrency in distributed shared memories. En *Proceedings 10th International Conference on Distributed Computer Systems*, IEEE, pp. 302-11.

Hyman y otros 1991 — Hyman, J., Lazar, A. A. y Pacifici, G. (1991). MARS – The MAGNET-II Real-Time Scheduling Algorithm. *ACM SIGCOM '91*, Zurich.

IEEE 1985a — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – CSMA/CD Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.3, IEEE Computer Society.

IEEE 1985b — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Bus Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.4, IEEE Computer Society.

IEEE 1985c — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Ring Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.5, IEEE Computer Society.

IEEE 1990 — Institute of Electrical and Electronic Engineers (1990). *IEEE Standard 802: Overview and Architecture*. American National Standard ANSI/IEEE 802, IEEE Computer Society.

IEEE 1994 — Institute of Electrical and Electronic Engineers (1994). *Local and metropolitan area networks – Part 6: Distributed Queue Dual Bus (DQDB) access method and physical layer specifications*. American National Standard ANSI/IEEE 802.6, IEEE Computer Society.

IEEE 1999 — Institute of Electrical and Electronic Engineers (1999). *Local and metropolitan area networks – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, American National Standard ANSI/IEEE 802.11, IEEE Computer Society.

Iftode y otros 1996 — Iftode, L., Singh J. y Li, K. (1996). Scope consistency: a bridge between release consistency and entry consistency. En *Proceedings 8th annual ACM symposium on Parallel algorithms and architectures*. pp. 277-87.

info.isoc.org — Zakon, R. H. *Hobbes' Internet Timeline v5.0*, ωωω

international pgp — *The International PGP Home Page*, ωωω

ISO 1992 — International Standards Organization (1992). *Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use*. ISO/IEC JTC1/SC212/WG7 CD 10746-1, International Standards Organization, 1992.

ITU/ISO 1997 — ITU/ISO (1997). *Recommendation X.500 (08/97): Open Systems Interconnection – The Directory: Overview of concepts, models and services*. International Telecommunication Union. ωωω

java.sun.com I — Sun Microsystems. *Java Remote Method Invocation*. ωωω

java.sun.com II — Sun Microsystems. *Java Object Serialization Specification*. ωωω

java.sun.com III — Sun Microsystems. *Servlet Tutorial*. ωωω

java.sun.com IV — Jordan, M. y Atkinson, M. (1999). *Orthogonal Persistence for the Java Platform-Draft Specification*. Sun Microsystems Laboratories. Palo Alto, Ca. ωωω

java.sun.com V — Sun Microsystems, *Java Security API*. ωωω

java.sun.com VI — Sun Microsystems Inc. (1999). *JavaSpaces technology*. ωωω

Johnson y Zwaenepoel 1993 — Johnson, D. y Zwaenepoel, W. (1993). The Peregrine High- performance RPC System. *Software-Practice and Experience*, vol. 23, n.º 2, pp. 201-21.

Jordan 1996 — Jordan, M. (1996). Early Experiences with Persistent Java. En *Proceedings first international workshop on persistrence and Java*. Glasgow, Scotland. ωωω

Joseph y otros 1997 — Joseph, A., Tauber, J. y Kaashoek, M. (1997). Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, vol. 46, n.° 3, pp. 337-52.

Jul y otros 1988 — Jul, E., Levy, H., Hutchinson, N. y Black, A. (1988). Fine- grained Mobility in the Emerald System. *ACM Transactions Computer Systems*, vol. 6, n.° 1, pp. 109-33.

Kaashoek y Tanenbaum 1991 — Kaashoek, F. y Tanenbaum, A. (1991). Group Communication in the Amoeba Distributed Operating System. En *Proceedings 11th International Conference on Distributed Computer Systems*, pp. 222-30.

Kaashoek y otros 1989 — Kaashoek, F., Tanenbaum, A., Flynn Hummel, S. y Bal, H. (1989). An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, vol. 23, n.° 4, pp. 5-20.

Kaashoek y otros 1997 — Kaashoek, M., Engler, D., Ganzer, G., Briceño, H., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J. y Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 52-65.

Kahn 1967 — Kahn, D. (1967). *The Codebreakers: The Story of Secret Writing*. New York: Macmillan.

Kahn 1983 — Kahn, D. (1983). *Kahn on Codes*. New York: Macmillan.

Kahn 1991 — Kahn, D, (1991). *Seizing the Enigma*. Boston: Houghton Mifflin.

Kehne y otros 1992 — Kehne, A., Schonwalder, J. y Langendorfer, H. (1992). A Nonce-based Protocol for Multiple Authentications. *ACM Operating Systems Review*, vol. 26, n.° 4, pp. 84-9.

Keith y Wittle 1993 — Keith, B.E. y Wittle, M. (1993). LADDIS: The Next Generation in NFS File Server Benchmarking, *Summer USENIX Conference Proceedings*. USENIX Association, Berkeley, CA, junio.

Keleher y otros 1992 — Keleher, P., Cox, A. y Zwaenepoel, W. (1992). Lazy consistency for software distributed shared memory. En *Proceedings 19th Annual International Symposium on Computer Architecture*. pp. 13-21, mayo 1992.

Kessler y Livny 1989 — Kessler, R. E. y Livny, M. (1989). An Analysis of Distributed Shared Memory Algorithms, In *Proceedings 9th International Conference Distributed Computing Systems*. IEEE, pp. 98-104.

Kille 1992 — Kille, S. (1992). *Implementing X.400 and X.500: The PP and QUIPU Systems*. Artech House.

Kindberg 1995 — Kindberg, T. (1995). A Sequencing Service for Group Communication (abstract), En *Proceedings 14th annual ACM Symposium on Principles of Distributed Computing*, p. 260. Technical Report n.° 698, Queen Mary and Westfield College Dept. of CS, 1995. ωωω

Kindberg y otros 1996 — Kindberg, T., Coulouris, G., Dollimore, J. y Heikkinen, J. (1996). Sharing objects over the Internet: the Mushroom approach. En *Proceedings IEEE Global Internet 1996*, London, noviembre, pp. 67-71.

Kistler y Satyanarayanan 1992 — Kistler, J. J. y Satyanarayanan, M. (1992). Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, vol. 10, n.° 1, pp. 3-25.

Kleinrock 1961 — Kleinrock, L. (1961). *Information Flow in Large Communication Networks*, MIT, RLE Quarterly Progress Report, julio.

Kleinrock 1997 — Kleinrock, L. (1997). Nomadicity: anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, vol. 1, n.° 4, pp. 351-7.

Kohl y Neuman 1993 — Kohl, J. y Neuman, C. (1993). *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, septiembre. ωωω

Konstantas y otros 1997 — Konstantas, D., Orlarey, Y., Gibbs, S. y Carbonel, O. (1997). Distributed Music Rehearsal. En *Proceedings International Computer Music Conference 97*. ωωω

Kopetz y Verissimo 1993 — Kopetz, H. y Verissimo, P. (1993). Real Time and Dependability Concepts. en Mullender ed, *Distributed Systems*, segunda edición, Addison-Wesley.

Kopetz y otros 1989 — Kopetz, H. Damm, A., Koza, C., Mulazzani, M., Schwabl, W. Senft, C y Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems – The MARS Approach. *IEEE Micro*, vol. 9, n.° 1.

Kshemkalyani y Singhal 1991 — Kshemkalyani, A. y Singhal, M. (1991). Invariant-Based Verification of a Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*, vol. 17, n.° 8, agosto.

Kshemkalyani y Singhal 1994 — Kshemkalyani, A. y Singhal, M. (1994). On Characterisation and Correctness of Distributed Deadlock detection, *Journal of Parallel and Distributed Computing*, vol. 22, pp. 44-59.

Kung y Robinson 1981 — Kung, H. T. y Robinson, J. T. (1981). Optimistic methods for concurrency control. *ACM Transactions on Database Systems*, vol. 6, n.° 2, pp. 213-26.

Kurose y Ross 2000 — Kurose, J. F. y Ross, K. W. (2000). *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison Wesley Longman. ωωω

Ladin y otros 1992 — Ladin, R., Liskov, B., Shrira, L. y Ghemawat, S. (1992). Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, vol. 10, n.° 4, pp. 360-91.

Lai 1992 — Lai, X. (1992). On the Design and Security of Block Ciphers, *ETH Series in Information Processing*, vol. 1, Konstanz: Hartung-Gorre Verlag.

Lai y Massey 1990 — Lai, X. y Massey, J. (1990). A proposal for a new Block Encryption Standard. *Advances in Cryptology-Eurocrypt '90*, En *Proceedings*, Springer-Verlag, pp. 389-404.

Lamport 1978 — Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, vol. 21, n.° 7, pp. 558-65.

Lamport 1979 — Lamport, L. (1979). How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions Computers*, vol. C-28, n.° 9, pp. 690-1.

Lamport 1986 — Lamport, L. (1986). On interprocess communication, parts I and II. *Distributed Computing*, vol. 1, n.° 2, pp. 77-101.

Lamport y otros 1982 — Lamport, L., Shostak, R. y Pease, M. (1982). Byzantine Generals Problem. *ACM Transactions Programming Languages and Systems*, vol. 4, n.° 3, pp. 382-401.

Lampson 1971 — Lampson, B. (1971). Protection. En *Proceedings 5th Princeton Conference on Information Sciences and Systems*, Princeton, p. 437. Reeditado en *ACM Operating Systems Review*. vol. 8, n.° 1, enero, p. 18. ωωω

Lampson 1981a — Lampson, B. W. (1981). Atomic Transactions. En *Distributed systems: Architecture and Implementation. Lecture Notes in Computer Science* 105, pp. 254-9. Berlin: Springer-Verlag.

Lampson 1986 — Lampson, B. W. (1986). Designing a Global Name Service. En *Proceedings 5th ACM Symposium Principles of Distributed Computing*, pp. 1-10, agosto.

Lampson y otros 1992 — Lampson, B. W., Abadi, M., Burrows, M. y Wobber, E. (1992). Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, vol. 10, n.° 4, pp. 265-310.

Lea y otros 1993 — Lea, R., Jacquemot, C. y Pillevesse, E. (1993). COOL: system support for distributed programming. *Comms. ACM*, vol. 36, n.° 9, pp. 37-46.

Leach y otros 1983 — Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L. y Stumpf, B. L. (1983). The architecture of an integrated local network. *IEEE J. Selected Areas in Communications*, vol. SAC-1, n.° 5, pp. 842-56.

Lee y Thekkath 1996 — Lee, E. K. y Thekkath, C. A. (1996). Petal: Distributed Virtual Disks, en *Proc. 7th Intl. Conf. on Architectural Support for Prog. Langs. and Operating Systems*, octubre, pp. 84-96. ωωω

Lee y otros 1996 — Lee C., Rajkumar, R. y Mercer C. (1996). Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. En *Proceedings Multimedia Japan '96*.

Leffler y otros 1989 — Leffler, S., McKusick, M., Karels, M. y Quartermain J. (1989). *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Reading, Mass: Addison-Wesley.

Leiner 1997 — Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G. y Wolff, S. (1997). A Brief History of the Internet, *Comms. ACM*, vol. 40, n.° 1, febrero, pp. 102-108. ωωω

Leland y otros 1993 — Leland, W. E., Taqqu, M. S., Willinger, W. y Wilson, D. V. (1993). On the Self-Similar Nature of Ethernet Traffic. *ACM SIGCOMM '93*, San Francisco.

Lenoski y otros 1992 — Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. D., Gupta, A., Hennessy, J., Horowitz, M. y Lam, M. S. (1992). The Stanford Dash multiprocessor. *IEEE Computer*, vol. 25, n.° 3, pp. 63-79.

Leslie y otros 1996 — Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. y Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications, *ACM Journal of Selected Areas in Communication*, vol. 14, n.° 7, pp. 1280-97.

Li y Hudak 1989 — Li, K. y Hudak, P. (1989). Memory Coherence in Shared Virtual Memory Systems.*ACM Transactions on Computer Systems*, vol. 7, n.° 4, pp. 321-59.

Liedtke 1996 — Liedtke, J. (1996). Towards real microkernels, *Comms. ACM*, vol. 39, n.° 9, pp. 70-7.

Linux AFS — *The Linux AFS FAQ*. ωωω

Lipton y Sandberg 1988 — Lipton, R. y Sandberg, J. (1988). *PRAM: A scalable shared memory*. Technical Report CS-TR-180-88, Princeton University.

Liskov 1988 — Liskov, B. (1988). Distributed programming in Argus. *Comms. ACM*, vol. 31, n.° 3, pp. 300-12.

Liskov 1993 — Liskov, B. (1993). Practical uses of synchronized clocks in distributed systems, *Distributed Computing*, vol. 6, n.° 4, pp. 211-19.

Liskov y Scheifler 1982 — Liskov, B. y Scheifler, R. W. (1982). Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions Programming Languages and Systems*, vol. 5, n.° 3, pp. 381-404.

Liskov y Shrira 1988 — Liskov, B. y Shrira, L. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. En *Proceedings SIGPLAN '88 Conference Programming Language Design and Implementation*. Atlanta.

Liskov y otros 1991 — Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M. (1991). Replication in the Harp File System. En *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 226-38.

Liu y Albitz 1998 — Liu, C. y Albitz, P. (1998). *DNS and BIND*, tercera edición. O'Reilly.

Liu y Layland 1973 — Liu, C. L. y Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, vol. 20, n.° 1.

Loepere 1991 — Loepere, K. (1991). *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie-Mellon University.

Lundelius y Lynch 1984 — Lundelius, J. y Lynch, N. (1984). An Upper and Lower Bound for Clock Synchronization. *Information and Control* 62, 2/3 (agosto/septiembre), pp. 190-204.

Lynch 1996 — Lynch, N. (1996). *Distributed Algorithms*, Morgan Kaufmann.

Ma 1992 — Ma, C. (1992). *Designing a Universal Name Service*. Technical Report 270, University of Cambridge.

Macklem 1994 — Macklem, R. (1994). Not Quite NFS: Soft Cache Consistency for NFS. *Proceedings of the Winter '94 USENIX Conference*, San Francisco, Ca., enero, pp. 261-278. ωωω

Maekawa 1985 — Maekawa, M. (1985). A Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, vol. 3, n.° 2, pp. 145-159.

Maffeis 1995 — Maffeis, S. (1995). Adding group communication and fault tolerance to CORBA. En *Proceedings of the* 1995 *USENIX conference on object-oriented technologies*.

Malkin 1993 — Malkin, G. (1993). *RIP Version* 2 – *Carrying Additional Information*, Internet RFC 1388. ωωω

Marsh y otros 1991 — Marsh, B., Scott, M., LeBlanc, T. y Markatos, E. (1991). First-class User-level Threads. En *Proceedings 13th ACM Symposium on Operating System Principles*, pp. 110-21.

Marzullo y Neiger 1991 — Marzullo, K., y Neiger, G. (1991). Detection of global state predicates, In *Proceedings 5th International Workshop on Distributed Algorithms*, Toug, S., Spirakis, P. y Kirousis, L., eds, Springer-Verlag, pp. 254-72.

Mattern 1989 — Mattern, F. (1989). Virtual Time and Global States in Distributed Systems, In *Proceedings Workshop on Parallel and Distributed Algorithms*. Cosnard, M. y otros (eds), Amsterdam: North-Holland, pp. 215-26.

mbone — *MBone Software Archives*. ωωω

McGraw y Felden 1999 — McGraw, G. y Felden, E. (1999). *Securing Java*. John Wiley & Sons. ωωω

Melliar-Smith y otros 1990 — Melliar-Smith, P., Moser, L. y Agrawala, V. (1990). Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, n.° 1, pp. 17-25.

Menezes 1993 — Menezes, A. (1993). *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers. ωωω

Menezes y otros 1997 — Menezes, A., van Oorschot, O. y Vanstone, S. (1997). *Handbook of Applied Cryptography*. CRC Press. ωωω

Metcalfe y Boggs 1976 — Metcalfe, R. M. y Boggs, D. R. (1976). Ethernet: distributed packet switching for local computer networks. *Comms. ACM*, vol. 19, pp. 395-403.

mice.ed.ac.uk — Handley, M.. *The sdr Session Directory*. ωωω

Mills 1995 — Mills, D. (1995). Improved Algorithms for Synchronizing Computer Network Clocks, *IEEE Transactions Networks*, junio, pp. 245-54.

Milojicic y otros 1999 — Milojicic, J., Douglis, F. y Wheeler, R. (1999). *Mobility, Processes, Computers and Agents*, Reading: Addison-Wesley.

Minnich y Farber 1989 — Minnich, R. y Farber, D. (1989). The Mether System: a Distributed Shared Memory for SunOS 4.0. En *Proceedings Summer 1989 Usenix Conference*.

Mitchell y Dion 1982 — Mitchell, J. G. y Dion, J. (1982). A comparison of two network-based file servers. *Comms. ACM*, vol. 25, n.° 4, pp. 233-45.

Mitchell y otros 1992 — Mitchell, C. J., Piper, F. y Wild, P. (1992). Digital Signatures. En *Contemporary Cryptology*. Simmons, G. J. ed., New York: IEEE Press.

Mogul 1994 — Mogul, J. D. (1994). Recovery in Spritely NFS, *Computing Systems*, vol. 7, n.° 2.

Mok 1985 — Mok, A. K. (1985). SARTOR – A Design Environment for Real-Time Systems. *Ninth IEEE COMP-SAC*.

Morin 1997 — Morin, R. (ed.) (1997). *MkLinux: Microkernel Linux for the Power Macintosh*. Prime Time Freeware.

Morris y otros 1986 — Morris, J., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. y Smith, F. D. (1986). Andrew: a distributed personal computing environment. *Comms. ACM*, vol. 29, n.° 3, pp. 184-201.

Mosberger 1993 — Mosberger, D. (1993). *Memory Consistency Models*. Technical Report 93/1 1, University of Arizona.

Moser y otros 1994 — Moser, L., Amir, Y., Melliar-Smith, P. y Agarwal, D. (1994). Extended Virtual Synchrony. En *Proceedings 14th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 56-65.

Moser y otros 1996 — Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R., y Lingley-Papadopoulos, C. (1996). Totem: a Fault-Tolerant Multicast Group Communication System. *Comms. ACM*, vol. 39, n.° 4, pp. 54-63.

Moser y otros 1998 — Moser, L., Melliar-Smith, P. y Narasimhan, P. (1998). Consistent object replication in the Eternal system. *Theory and practice of object systems*, vol. 4, n.° 2.

Moss 1985 — Moss, E. (1985). *Nested Transactions, An Approach to Reliable Distributed Computing*. MIT Press.

multimedia index — Gibbs, S. y Szentivanyi, S., *Index to Multimedia Information Sources*. ωωω

Myers y Liskov 1997 — Myers, A. C. y Liskov, B. (1997). A Decentralized Model for Information Flow Control, *ACM Operating Systems Review*, vol. 31, n.° 5, pp. 129-42, diciembre.

Nagle 1984 — Nagle, J. (1984). Congestion Control in TCP/IP Internetworks, *Computer Communications Review*, vol. 14, pp. 11-17, octubre.

Nagle 1987 — Nagle, J. (1987). On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, vol. 35, n.° 4.

National Bureau of Standards 1977 — National Bureau of Standards (1977). *Data Encryption Standard (DES)*. Federal Information Processing Standards n.° 46, Washington DC: US National Bureau of Standards.

Needham 1993 — Needham, R. (1993). Names. En *Distributed Systems, an Advanced Course*. (Mullender, S., ed.), segunda edición. Wokingham, England: ACM Press/Addison-Wesley. pp. 315-26.

Needham y Schroeder 1978 — Needham, R. M. y Schroeder, M. D. (1978). Using encryption for authentication in large networks of computers. *Comms. ACM*, vol. 21, pp. 993-9.

Nelson y otros 1988 — Nelson, M. N., Welch, B. B. y Ousterhout, J. K. (1988). Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, vol. 6, n.° 1, pp. 134-154.

Netscape 1996 — Netscape Corporation (1996). *SSL 3.0 Specification*. ωωω

Neuman y otros 1999 — Neuman, B. C., Tung, B. y Wray, J. (1999). *Public Key Cryptography for Initial Authentication in Kerberos*, Internet Draft ietf-cat-kerberos-pk-init-09, julio. ωωω

Neumann y Ts'o 1994 — Neuman, B. C. y Ts'o, T. (1994). An Authentication Service for Computer Networks, *IEEE Communications*, vol. 32, no. 9, pp. 33-38. septiembre. ωωω

Nielson y otros 1997 — Nielsen, H., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. y Lilley, C. (1997). Network Performance Effects of HTTP/1.1, CSS1, and PNG. *Proceedings SIGCOMM '97*.

NIST 1995 — National Institute for Styards and Technology (1995). *Secure Hash Standard*. NIST FIPS PUP 180-1, US Department of Commerce. ωωω

NIST 1999 — National Institute for Standards and Technology (1999). *AES – a Crypto Algorithm for the Twenty-first Century*, US Department of Commerce. ωωω

now.cs.berkeley.edu — *The Berkeley NOW project home page*. ωωω

Oaks y Wong 1999 — Oaks, S. y Wong, H. (1999). *Java Threads* (segunda edición), O'Reilly.

OMG 1997a — Object Management Group. (1997). *Concurrency Control Service Specification*, Framingham, Mass: OMG. ωωω

OMG 1997b — Object Management Group (1997). *Naming Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1997c — Object Management Group (1997). *Event Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1997d — Object Management Group (1997). *The CORBA IDL Specification*. Framingham, Mass: OMG. ωωω

OMG 1997e — Object Management Group, (1997). *Object Transaction Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1998a — Object Management Group (1998). *CORBA/IIOP 2.3.1 Specification*. Framingham, Mass: OMG. ωωω

OMG 1998b — Object Management Group (1998). *CORBA Security Service Specification*. Framingham, Mass: OMG. ωωω

OMG 1998c — Object Management Group (1998). *Notification service Specification*. Framingham, Mass: OMG. Technical report telecom/98-06-15. ωωω

OMG 1998d — Object Management Group (1998). *CORBA Messaging*. Framingham, Mass: OMGωωω

OMG 1998e — Object Management Group(1998). *Objects by Value*. Framingham, Mass: OMG. ωωω

Omidyar y Aldridge 1993 — Omidyar, C. G. y Aldridge, A. (1993). Introduction to SDH/SONET. *IEEE Communications Magazine*, vol. 31, pp. 30-3, septiembre.

Oppen y Dalal 1983 — Oppen, D. C. y Dalal Y. K. (1983). The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Trans. on Office Systems*, vol. 1, pp. 230-53.

Orfali y otros 1996 — Orfali, R., Harkey, D. y Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. New York: Wiley.

Organick 1972 — Organick, E. I. (1972). *The MULTICS System: An Examination of its Structure*. Cambridge, Mass: MIT Press.

Orman y otros 1993 — Orman, H., Menze, E., O'Malley, S. y Peterson, L. (1993). A fast and general implementation of Mach IPC in a Network. En *Proceedings Third USENIX Mach Conference*, abril.

OSF 1997 — *Introduction to OSF DCE*. The Open Group. ωωω

Ousterhout y otros 1985 — Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. y Thompson, J. (1985). A Trace-driven analysis of the UNIX 4.2 BSD file system. En *10th ACM Symposium Operating System Principles*.

Ousterhout y otros 1988 — Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. y Welch, B. (1988). The Sprite Network Operating System. *IEEE Computer*, vol. 21, n.° 2, pp. 23-36.

Parker 1992 — Parker, B. (1992). *The PPP AppleTalk Control Protocol (ATCP)*. Internet RFC 1378. ωωω

Parrington y otros 1995 — G. D. Parrington, S. K. Shrivastava, Wheater, S.M. y Little, M. C. (1995). The Design and Implementation of Arjuna, *USENIX Computing Systems Journal*, vol. 8, n.° 3.

Partridge 1992 — Partridge, C. (1992). *A Proposed Flow Specification*. Internet RFC 1363. ωωω

Patterson y otros 1988 — Patterson, D., Gibson, G y Katz, R. (1988). A Case for Redundant Arrays of Interactive Disks, *ACM International Conf. on Management of Data (SIGMOD)*, pp. 109-116, mayo.

Pease y otros 1980 — Pease, M., Shostak, R. y Lamport, L.(1980). Reaching agreement in the presence of faults. *Journal of the ACM*, vol. 27, n.° 2, abril, pp. 228-34.

Pedone y Schiper 1999 — Pedone, F. y Schiper, A. (1999). Generic Broadcast. En *Proceedings of the 13th International Symposium on Distributed Computing* (DISC '99), septiembre. ωωω

Petersen y otros 1997 — Petersen, K., Spreitzer, M., Terry, D., Theimer, M. y Demers, A. (1997). Flexible update propagation for weakly consistent replication. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 288-301.

Peterson 1988 — Peterson, L. (1988). The Profile Naming Service. *ACM Transactions Computer Systems*, vol. 6, n.° 4, pp. 341-64.

Peterson y otros 1989 — Peterson, L. L., Buchholz, N. C. y Schlichting, R. D. (1989). Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, vol. 7, n.° 3, pp. 217-46.

Pike y otros 1993 — Pike, R., Presotto, D., Thompson, K., Trickey, H. y Winterbottom, P. (1993). The Use of Name Spaces in Plan 9. *Operating Systems Review*, vol. 27, .° 2, abril 1993, pp. 72-76. ωωω

Popek y Walker 1985 — Popek, G. y Walker, B. (eds.). (1985). *The LOCUS Distributed System Architecture*. Cambridge Mass: MIT Press.

Postel 1981a — Postel, J. (1981). *Internet Protocol*. Internet RFC 791. ωωω

Postel 1981b — Postel, J. (1981). *Transmission Control Protocol*. Internet RFC 793. ωωω

Powell 1991 — Powell, D. (ed.) (1991). *Delta-4: a Generic Architecture for Dependable Distributed Computing*. Berlin and New York: Springer-Verlag.

Preneel y otros 1998 — Preneel, B., Rijmen, V. y Bosselaers, A. (1998). Recent developments in the design of conventional cryptographic algorithms, In Computer Security and Industrial Cryptography, State of the Art and Evolution, *Lecture Notes in Computer Science*, n.° 1528, Springer-Verlag, pp. 106-131. ωωω

privacy.nb.ca — *International Cryptography Freedom*. ωωω

Radia y otros 1993 — Radia, S., Nelson, M. y Powell, M. (1993). *The Spring Naming Service*. Technical Report 93-16, Sun Microsystems Laboratories, Inc.

Rashid 1985 — Rashid, R. F. (1985). Network operating systems. En Local Area Networks: An Advanced Course, *Lecture Notes in Computer Science*, 184, Springer-Verlag, pp. 314-40.

Rashid 1986 — Rashid, R. F. (1986). From RIG to Accent to Mach: the evolution of a network operating system. En *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, ACM, noviembre.

Rashid y Robertson 1981 — Rashid, R. y Robertson, G. (1981). Accent: a communications oriented network operating system kernel. *ACM Operating Systems Review*, vol. 15, n.° 5, pp. 64-75.

Rashid y otros 1988 — Rashid, R., Tevanian Jr, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J. y Chew, J. (1988). Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions Computers*, vol. 37, n.° 8, pp. 896-907.

Raynal 1988 — Raynal, M. (1988). *Distributed Algorithms and Protocols*. Wiley.

Raynal 1992 — Raynal, M. (1992). About Logical Clocks for Distributed Systems. *ACM Operating Systems Review*, vol. 26, n.° 1, pp. 41-8.

Raynal y Singhal 1996 — Raynal, M. y Singhal, M. (1996). Capturing Causality in Distributed Systems. *IEEE Computer*, febrero, pp. 49-56.

Redmond 1997 — Redmond, F. E. (1997). *DCOM: Microsoft Distributed Component Model*. IDG Books Worldwide.

Reed 1983 — Reed, D. P. (1983). Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, vol. 1, n.° 1, pp. 3-23.

Ricart y Agrawala 1981 — Ricart, G. y Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Comms. ACM*, vol. 24, n.° 1, pp. 9-17.

Richardson y otros 1998 — Richardson, T., Stafford-Fraser, Q., Wood, K. R. y Hopper, A. (1998). Virtual Network Computing, *IEEE Internet Computing*. vol. 2, n.° 1, enero/febrero, pp. 33-8. ωωω

Ritchie 1984 — Ritchie, D. (1984). A Stream Input Output System. *AT&T Bell Laboratories Technical Journal*, vol. 63, n.° 8, pt. 2, pp. 1897-910.

Rivest 1992 — Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. Internet RFC 1321. ωωω

Rivest y otros 1978 — Rivest, R. L., Shamir, A. y Adelman, L. (1978). A method of obtaining digital signatures and public key cryptosystems. *Comms. ACM*, vol. 21, n.° 2, pp. 120-6.

Rodrigues y otros 1998 — Rodrigues, L., Guerraoui, R., y Schiper, A. (1998). Scalable Atomic Multicast. En *Proceedings IEEE IC3N '98*. Technical Report 98/257. École polytechnique fédérale de Lausanne. ωωω

Rose 1992 — Rose, M. T. (1992). *The Little Black Book: Mail Bonding with OSI Directory Services*. Englewood Cliffs, NJ: Prentice-Hall.

Rosenblum y Ousterhout 1992 — Rosenblum, M. y Ousterhout, J. (1992). The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, vol. 10, n.° 1, febrero, pp. 26-52. ωωω

Rosenblum y Wolf 1997 — Rosenblum, D. S. y Wolf, A. L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. En *Proceedings sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, Zurich, Switzerland.

Rowley 1998 — Rowley, A. (1998). *A Security Architecture for Groupware*, Doctoral Thesis, Queen Mary and Westfield College, University of London. ωωω

Rozier y otros 1988 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. y Neuhauser, W. (1988). Chorus Distributed Operating Systems. *Computing Systems Journal*, vol. 1, n.° 4, pp. 305-70.

Rozier y otros 1990 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. y Neuhauser, W. (1990). *Overview of the Chorus Distributed Operating System*. Technical Report CS/TR-90-25.1, Chorus Systèmes, France.

Saltzer y otros 1984 — Saltzer, J. H., Reed, D. P. y Clarke, D. (1984). End-to-End Arguments in System Design, *ACM Transactions on Computer Systems* vol. 2, n.° 4, pp. 277-88. ωωω

Sandberg 1987 — Sandberg, R. (1987). *The Sun Network File System: Design, Implementation and Experience*. Technical Report. Mountain View Calif.: Sun Microsystems.

Sandberg y otros 1985 — Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. y Lyon, B. (1985). The Design and Implementation of the Sun Network File System. En *Proceedings Usenix Conference*, Portland, Ore.

Sandhu y otros 1996 — Sandhu, R., Coyne, E., Felstein, H. y Youman, C. (1996). Role-Based Access Control Models, *IEEE Computer*, vol. 29, n.° 2, febrero. ωωω

Sane y otros 1990 — Sane, A., MacGregor, K. y Campbell, R. (1990). Distributed Virtual Memory Consistency Protocols: Design and Performance. *Second IEEE Workshop on Experimental Distributed Systems*, pp. 91-6, octubre.

Sansom y otros 1986 — Sansom, R. D., Julin, D. P. y Rashid, R. F. (1986). *Extending a capability based system into a network environment*. Technical Report CMU-CS-86-116, Carnegie-Mellon University.

Santifaller 1991 — Santifaller. M. (1991). *TCP/IP and NFS, Internetworking in a Unix Environment*. Reading, Mass: Addison-Wesley.

Satyanarayanan 1981 — Satyanarayanan, M. (1981). A study of file sizes and functional lifetimes. En *Proceedings 8th ACM Symposium on Operating System Principles*, Asilomar, Calif.

Satyanarayanan 1989a — Satyanarayanan, M. (1989). Distributed File Systems. En *Distributed Systems, an Advanced Course*. (Mullender, S. ed.), segunda edición, Wokingham: ACM Press/Addison-Wesley. pp 353-83.

Satyanarayanan 1989b — Satyanarayanan, M. (1989). Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, vol. 7, n.° 3, pp. 247-80.

Satyanarayanan y otros 1990 — Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H. y Steere, D. C. (1990). Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, vol. 39, n.° 4, pp. 447-59.

Saunders 1987 — Saunders, B. (1987). The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems*, vol. 3, n.° 2, pp. 145-59.

Scheifler y Gettys 1986 — Scheifler, R. W. y Gettys, J. (1986). The X window system. *ACM Transactions on Computer Graphics*, vol. 5, n.° 2, pp. 76-109.

Schiper y Raynal 1996 — Schiper, A. y Raynal, M. (1996). From Group Communication to Transactions in Distributed Systems. *Comms. ACM*, vol. 39, n.º 4, pp. 84-7.

Schiper y Sandoz 1993 — Schiper, A. y Sandoz, A. (1993). Uniform reliable multicast in a virtually synchronous environment. *Proceedings 13th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 561-8.

Schlageter 1982 — Schlageter, G. (1982). Problems of Optimistic Concurrency Control in Distributed Database Systems. *SigMOD Record*. vol. 13, n.º 3, pp. 62-6.

Schmidt 1998 — Schmidt, D. (1998). Evaluating architectures for multithreaded object request brokers, *Comms. ACM*, vol. 44, n.º 10, pp. 54-60.

Schneider 1990 — Schneider, F. B. (1990). Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, vol. 22, n.º 4, pp. 300-19.

Schneider 1996 — Schneider, S. (1996). Security properties and CSP. En *IEEE Symposium, on Security and Privacy*, pp. 174-187.

Schneier 1996 — Schneier, B. (1996). *Applied Cryptography*, segunda edición. New York: John Wiley.

Schroeder y Burrows 1990 — Schroeder, M. y Burrows, M. (1990). The Performance of Firefly RPC. *ACM Transactions on Computer Systems*, vol. 8, n.º 1. pp. 1-17.

Schulzrinne y otros 1996 — Schulzrinne, H., Casner, S., Frederick, D. y Jacobson, V. (1996). *RTP:A Transport Protocol for Real-Time Applications*, Internet RFC 1889, enero. ωωω

Seetharamanan 1998 — Seetharamanan, K. (ed.) (1998). Special Issue: The CORBA Connection, *Comms. ACM*, octubre, vol. 41, n.º 10.

session directory — *User Guide to sd (Session Directory)*. ωωω

Shannon 1949 — Shannon, C. E. (1949). Communication Theory of Secrecy Systems, *Bell System Technical Journal*, vol. 28, n.º 4, pp. 656-715.

Shepler 1999 — Shepler, S. (1999). *NFS Version 4 Design Considerations*, Internet RFC 2624, Sun Microsystems, junio. ωωω

Shoch y Hupp 1980 — Shoch, J. F. y Hupp, J. A. (1980). Measured performance of an Ethernet local network. *Comms. ACM*, vol. 23, n.º 12, pp. 711-21.

Shoch y Hupp 1982 — Shoch, J. F. y Hupp, J. A. (1982). The «Worm» programs – early experience with a distributed computation. *Comms. ACM*, vol. 25, n.º 3, pp. 172-80.

Shoch y otros 1982 — Shoch, J. F., Dalal, Y. K. y Redell, D. D. (1982). The evolution of the Ethernet local area network. *IEEE Computer*, vol. 15, n.º 8, pp. 10-28.

Shoch y otros 1985 — Shoch, J. F., Dalal, Y. K., Redell, D. D. y Crane, R. C. (1985). The Ethernet. En Local Area Networks: an Advanced Course, *Lecture Notes in Computer Science*. n.º 184, Springer-Verlag, pp. 1-33.

Shrivastava y otros 1991 — Shrivastava, S., Dixon, G. N. y Parrington, G. D. (1991). An Overview of the Arjuna Distributed Programming System. *IEEE Software*, enero. pp. 66-73.

Singh 1999 — Singh, S. (1999). *The Code Book*. London: Fourth Estate.

Sinha y Natarajan 1985 — Sinha, M. y Natarajan, N. (1985). A Priority Based Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*. vol. 11, n.º 1, pp. 67-80.

Spafford 1989 — Spafford, E. H. (1989). The Internet Worm: Crisis and Aftermath. *Comms. ACM*, vol. 32, n.º 6, pp. 678-87.

Spasojevic y Satyanarayanan 1996 — Spasojevic, M. y Satyanarayanan, M. (1996). An Empirical Study of a Wide-Area Distributed File System, *ACM Transactions on Computer Systems*, vol. 14, n.º 2, mayo, pp. 200-222.

Spector 1982 — Spector, A. Z. (1982). Performing remote operations efficiently on a local computer network. *Comms. ACM*, vol. 25, n.º 4, pp. 246-60.

Spurgeon 2000 — Spurgeon, C. E. (2000). *Ethernet: The Definitive Guide*. O'Reilly.

Srikanth y Toueg 1987 — Srikanth, T. y Toueg, S. (1987). Optimal Clock Synchronization. *Journal ACM*. 34, 3 (julio), pp. 626-45.

Srinivasan 1995a — Srinivasan, R. (1995). *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems. Internet RFC 1831, agosto. ωωω

Srinivasan 1995b — Srinivasa, R. (1995). *XDR: External Data Representation Standard*. Sun Microsystems. RFC 1832. Network Working Group, agosto. ωωω

Srinivasan y Mogul 1989 — Srinivasan, V. y Mogul, J. D. (1989). Spritely NFS: Experiments with Cache-Consistency Protocols, *12th ACM Symposium on Operating System Principles*, Litchfield Park, Az., diciembre, pp. 45-57.

Stallings 1998a — Stallings, W. (1998). *High Speed Networks – TCP/IP and ATM Design Principles*. Upper Saddle River, NJ: Prentice-Hall.

Stallings 1998b — Stallings, W. (1998). *Operating Systems*, tercera edición. Prentice-Hall International.

Stallings 1999 — Stallings, W. (1999). *Cryptography and Network Security – Principles and Practice*, segunda edición, Upper Saddle River, NJ: Prentice-Hall.

Steiner y otros 1988 — Steiner, J., Neuman, C. y Schiller, J. (1988). Kerberos: an authentication service for open network systems. En *Proceedings Usenix Winter Conference*, Berkeley: Calif.

Stelling y otros 1998 — Stelling, P., Foster, I., Kesselman, C., Lee, C. y von Laszewski, G. (1998). A Fault Detection Service for Wide Area Distributed Computations, *Proceedings 7th IEEE Symposium on High Performance Distributed Computing*, pp. 268-78.

Stoll 1989 — Stoll, C. (1989). *The Cuckoo's Egg: Tracking a Spy Through a Maze of Computer Espionage*. New York: Doubleday.

Stone 1993 — Stone, H. (1993). *High-performance Computer Architecture*, tercera edición. Addison-Wesley.AAAAAA

Sun 1989 — Sun Microsystems Inc. (1989). *NFS: Network File System Protocol Specification*. Internet RFC 1094. ωωω

Sun y Ellis 1998 — Sun, C. y Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proceedings Conference on Computer Supported Cooperative Work Systems*, ACM Press, pp. 59- 68.

Tanenbaum 1992 — Tanenbaum, A. S. (1992). *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum 1996 — Tanenbaum, A. S. (1996). *Computer Networks*, tercera edición. Prentice-Hall International.

Tanenbaum y van Renesse 1985 — Tanenbaum, A. y van Renesse, R. (1985). Distributed Operating Systems, *Computing Surveys*, ACM, vol. 17, n.º 4, pp. 419-70.

Tanenbaum y otros 1990 — Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, J. y van Rossum, G. (1990). Experiences with the Amoeba Distributed Operating System. *Comms. ACM*, vol. 33, n.º 12, pp. 46-63.

Terry y otros 1995 — Terry, D., Theimer, M., Petersen, K., Demers, A, Spreitzer, M. y Hauser, C. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 172-183.

TFCC — *IEEE Task Force on Cluster Computing*. ωωω

Thayer 1998 — Thayer, R. (1998). *IP Security Document Roadmap*, Internet RFC 2411, noviembre ωωω

Thekkath y otros 1997 — Thekkath, C. A., Mann, T. y Lee, E. K. (1997). Frangipani: A Scalable Distributed File System, En *Proc. 16th ACM Symposium on Operating System Principles*, St. Malo, Francia, octubre, pp. 224-237. ωωω

Tokuda y otros 1990 — Tokuda, H., Nakajima, T. y Rao, P. (1990). Real-time Mach: towards a predictable real-time system. En *Proceedings USENIX Mach Workshop*, pp. 73-82, octubre.

Topolcic 1990 — Topolcic, C. (ed.) (1990). *Experimental Internet Stream Protocol, Version* 2. Internet RFC 1190. ωωω

Tzou y Anderson 1991 — Tzou, S.-Y. y Anderson, D. (1991). The performance of message-passing using restricted virtual memory remapping. *Software-Practice and Experience*, vol. 21, pp. 251- 67.

van Renesse y otros 1989 — van Renesse, R., van Staveran, H. y Tanenbaum, A. (1989). The Performance of the Amoeba Distributed Operating System. *Software – Practice and Experience*, vol. 19, n.° 3, pp. 223-34.

van Renesse y otros 1995 — van Renesse, R., Birman, K., Friedman, R., Hayden, M. y Karr, D. (1995). A Framework for Protocol Composition in Horus. *Proceedings PODC* 1995, pp. 80-9.

van Renesse y otros 1996 — van Renesse, R., Birman, K. y Maffeis, S. (1996). Horus: a Flexible Group Communication System. *Comms. ACM*, vol. 39, n.° 4, pp. 54-63.

van Steen y otros 1998 — van Steen, M., Hauck, F., Homburg, P. y Tanenbaum, A. (1998). Locating objects in wide-area systems. *IEEE Communication*, vol. 36, n.° 1, pp. 104-109.

Vinoski 1998 — Vinoski, S. (1998). New Features for CORBA 3.0, *Comms. ACM*, octubre 1998, vol. 41, n.° 10. pp. 44-52.

Vogt y otros 1993 — Vogt, C., Herrtwich R.G. y Nagarajan, R. (1993). HeiRAT – The Heidelberg Resource Administration Technique: Design Philosophy and Goals. *Kommunikation in verteilten Systemen*, Munich, Informatik aktuell, Springer.

Volpano y Smith 1999 — Volpano, D. y Smith, G. (1999). Language Issues in Mobile Program Security. To appear in *Lecture Notes in Computer Science*, Springer.

von Eicken y otros 1995 — von Eicken, T., Basu, A., Buch, V. y Vogels, V. (1995). U- Net: a user-level network interface for parallel and distributed programming. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 40-53.

Wahl y otros 1997 — Wahl, M., Howes, T. y Kille, S. (1997). *The Lightweight Directory Access Protocol (v3)*. Internet RFC 2251. ωωω

Waldo 1999 — Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Comms. ACM*. vol. 42, n.° 7, pp. 76-82.

Waldo y otros 1994 — Waldo, J, Wyant, G., Wollrath, A. y Kendall, S. (1994). A Note on Distributed Computing. En Arnold y otros 1999, pp. 307-26. ωωω

Waldspurger y otros 1992 — Waldspurger, C., Hogg, T., Huberman, B., Kephart, J. y Stornetta, W. (1992). Spawn: A Distributed Computational Economy. *Transactions on Software Engineering*, vol. 18, n.° 2, pp. 103-17.

Want y otros 1992 — Want, R., Hopper, A., Falcao, V y Gibbons, V. (1992). The Active Badge Location System. ACM *Transactions on Information Systems*, vol. 10, n.° 1, enero, pp. 91-102. ωωω

web.mit.edu I — *Kerberos: The Network Authentication Protocol*. ωωω

web.mit.edu II — *The Three Myths of Firewalls*. ωωω

Weikum 1991 — Weikum, G. (1991). Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions Database Systems*, vol. 16, n.° 1, pp. 132-40.

Weiser 1993 — Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Comms. ACM*, vol. 36, n.° 7, pp. 74-84.

Wheeler y Needham 1994 — Wheeler, D. J. y Needham, R. M. (1994). *TEA, a Tiny Encryption Algorithm*. Technical Report 355, Two Cryptographic Notes, Computer Laboratory, University of Cambridge, diciembre, pp. 1-3. ωωω

Wheeler y Needham 1997 — Wheeler, D. J. y Needham, R. M. (1997). *Tea Extensions*, octubre 1994, pp. 1-3. ωωω

Wiesmann y otros 2000 — Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. y Alonso, G. (2000). Understanding replication in databases and distributed systems. En *Proceedings 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, Taipei, Republic of China, IEEE.

Wobber y otros 1994 — Wobber, E., Abadi, M., Burrows, M. y Lampson, B. (1994). Authentication in the Taos operating system. *ACM Transactions Computer Systems*. 12, 1 (febrero), pp. 3-32. ωωω

Wuu y Bernstein 1984 — Wuu, G. T. y Bernstein, A. J. (1984). Efficient Solutions to the Replicated Log and Dictionary Problems. *ACM Proceedings Third Annual Symposium Principles of Distributed Computing*, pp. 233-42.

www.bluetooth.com — *The Official Bluetooth SIG Website*. ωωω

www.bxa.doc.gov — Bureau of Export Administration, US Department of Commerce, *Commercial Encryption Export Controls*. ωωω

www.cdk3.net — Coulouris, G., Dollimore, J. y Kindberg, T. (Eds), *Distributed Systems, Concepts and Design: Supporting material*. ωωω

www.citrix.com — Citrix Corporation. *Server-based Computing White Paper*. ωωω

www.cooltown.hp.com — Hewlett-Packard Corporation, *CoolTown nomadic computing project pages*. ωωω

www.cren.net — Corporation for Research and Educational Networking, *CERN Certificate Authority*. ωωω

www.cuseeme.com — CU-SeeMe Networks Inc, *Home page*. ωωω

www.doi.org — International DOI Foundation, *Pages on digital object identifiers*. ωωω

www.handle.net — Handle system, *Home page*. ωωω

www.iana.org — Internet Assigned Numbers Authority, *IANA Home Page*. ωωω

www.ietf.org — Internet Engineering Task Force, *Internet RFC Index Page*. ωωω

www.isoc.org — Robert Hobbes Zakon. *Hobbes' Internet Timeline*. ωωω

www.microsoft.com — Microsoft Corporation, *NetShow Theater Server Web Page*. ωωω

www.microsoft.com I — Microsoft Corporation, *Active Directory Services*. ωωω

www.microsoft.com II — Microsoft Corporation, *Windows 2000 Kerberos Authentication, White Paper*. ωωω

www.microsoft.com III — Microsoft Corporation, *NetMeeting home page*. ωωω

www.mids.org — Matrix Information and Directory Services Inc. *Internet Performance*. ωωω

www.millicent.com — Compaq Corporation, *Millicent MicroCommerce system*. ωωω

www.omg.org — Object Management Group, *Index to CORBA services*. OMG: Framingham, Mass. ωωω

www.opengroup.org — Open Group, *Portal to the World of DCE*. ωωω

www.openssl.org — OpenSSL Project, *OpenSSL: The Open Source toolkit for SSL/TLS*. ωωω

www.pgp.com — *PGP Home*. ωωω

www.reed.com — Read, D. P. (2000). *The End of the End-to-End Argument*. ωωω

www.rsasecurity.com — RSA Security Inc., *Home page*. ωωω

www.rsasecurity.com I — RSA Corporation (1997). *DES Challenge*. ωωω

www.rsasecurity.com II — RSA Corporation (1999). *RSA Factoring Challenge*. ωωω

www.rtj.org — *Real-Time for Java TM Experts Group*. ωωω

www.spec.org — *SPEC SFS97 Benchmark*. ωωω

www.upnp.com — *Universal Plug and Play home page*. ωωω

www.verisign.com — Verisign Inc., *Home page*. ωωω

www.w3.org I — World Wide Web Consortium, *Home page*. ωωω

www.w3.org II — World Wide Web Consortium, *Pages on the HyperText Markup Language*. ωωω

www.w3.org III — World Wide Web Consortium, *Pages on Naming and Addressing*. ωωω

www.w3.org IV — World Wide Web Consortium, *Pages on the HyperText Transfer Protocol*. ωωω

www.w3.org V — World Wide Web Consortium, *Pages on the Resource Description Framework and other metadata schemes*. ωωω

www.w3.org VI — World Wide Web Consortium, *Pages on the Extensible Markup Language.* ωωω

www.w3.org VII — World Wide Web Consortium, *Pages on the Extensible Stylesheet Language.* ωωω

www.wapforum.org — WAP Forum, *White Papers and Specifications.* ωωω

www.wlana.com — *The IEEE 802.11 Wireless LAN Standard.* ωωω

Wyckoff y otros 1998 — Wyckoff, P., McLaughry, S., Lehman, T. y Ford, D. (1998). T Spaces. *IBM Systems Journal*, vol. 37, n.° 3.

Zhang y otros 1993 — Zhang, L., Deering, S. E., Estrin, D., Shenker, S. y Zappala, D. (1993). RSVP – A New Resource Reservation Protocol. *IEEE Network Magazine*, vol. 9, n.° 5.

Zimmermann 1995 — Zimmermann, P. R. (1995). *The Official PGP User's Guide*, Cambridge, Mass.: MIT Press.

ÍNDICE ALFABÉTICO

A

aborta, 450
aborto en cascada o encadenado, 456
acceso
  control de, 250-252
    derechos de, 54
    lista de control de, 251
  transparencia, *véase* transparencia de acceso
ACID, propiedades, 449
ack-implosion, 417
activa, replicación, *véase* replicación activa
activación, *véase* objeto, activación
activador, *véase* objeto, activador
activo, objeto, *véase* objeto activo
actualización
  perdida, 451
  propagación diferida o perezosa, 565
acuerdo
  de la entrega por multidifusión, 417
  en el consenso y problemas relacionados, 429-430
  problemas, 428
  uniforme, 420
adaptador de objeto, 645
AES (Advanced Encryption Standard), 261
agente móvil, 35
alcanzabilidad, 384
aleatoriedad, 438
algoritmo
  de encriptación triple-DES, 271
  de estado de enlace, 80
  de Nagle, 99
  de rutado por distancia entre vectores, 78
  del depósito agujereado, 590
alias, *véase* suplantación
almacenamiento de objetos persistentes
  Java persistente, 293
  sistema Khazana, 293
  sistema PerDiS, 293
al-menos-una-vez, semántica de invocación, *véase* invocación, semántica
Amoeba
  protocolo de multidifusión, 424
  servidor de ejecución, 203
ancho de banda
Andrew, Sistema de Archivos (AFS), 316-325
  en DCE/DFS, 328
  para Linux, 299
  prestaciones, 325
  soporte de área extensa, 325
anti-entropía, protocolo, *véase* protocolo anti-entropía
API de Java
  DatagramPacket, 123
  DatagramSocket, 123
  InetAddres, 121
  MulticastSocke, 145
  ServerSocket, 127
  Socket, 127
aplicación
  de telefonía en red, 582
  nivel de, *véase* niveles
Apollo Domain, 606
Applet, 14
  hilos en, 212
archivo
  correlacionado en memoria, 202
  de recuperación, 513-524
    reorganización, 517
  indexados por posición, *véase* correlacionado en memoria
  replicado, 557
ARP (protocolo de resolución de direcciones), *véase* protocolo ARP

arquitectura
cotilla (de cotilleo), 545-554
mensaje cotilla, 547
procesamiento, 551-552
propagación, 552-553
procesamiento de actualizaciones, 550
procesamiento de consultas, 549
de procesamiento simétrico, 197
asíncrona
comunicación, *véase* comunicación asíncrona
invocación, *véase* invocación asíncrona
operación, 224-226
RMI, en CORBA, *véase* CORBA
asíncrono, sistema distribuido, 47-48
ataque
de denegación de servicio, 18
véase seguridad, ataque
ATM (Asynchronous Transfer Mode), 67, 112-114
atómica
consistencia, *véase* consistencia atómica
protocolo de consumación, *véase* protocolo de consumación atómica
transacción, *véase* transacción
atomicidad del fallo, 513
atrapar excepción, *véase* excepción, atrapar
audio, aplicación de conferencia, 582
autenticación, 246-248
servicio de, 275
servidor de, 273
autentificación, *véase* autenticación
autoridad de certificación, 270

B

backbone, *véase* troncal
balance de carga, 42
Bayou, 554-556
chequeo de dependencia, 555
procedimiento de fusión, 555
Bellman-Ford, protocolo de rutado de, *véase* protocolo, Bellman-Ford
big-endian, ordenamiento, *véase* ordenamiento de bits
binder, *véase* enlazador
bizantino
fallo, 52
generales, 433-436
bloque, cifrador de, *véase* cifrador de bloque
bloqueante, operación, 120
bloqueo, 459-472
actualilzación perdida, 451
compartido, 461
consumación de lectura-escritura, 470
de dos versiones, 470
en dos fases, 459
estricto, 460
en transacciones
anidadas, 465
distribuidas, *véase* transacciones distribuidas, control de concurrencia
exclusivo, 459
gestor de bloqueos, 463
granularidad, 460
implementación, 463
jerárquico, 470-471
lectura-escritura, 461
promoción, 462
provocando interbloqueos, *véase* interbloqueo
reglas de operaciones conflictivas, 460-461
tiempo de espera límite, 469
BlueTooth, red inalámbrica, *véase* red inalámbrica
Bridge, 83
broadcast, *véase* difusión global

C

caché, 33
coherencia de archivos en, 561
empleo de,
de archivos
completos, 317
en el cliente, 311
en el servidor, 311
escritura a través (directa) de archivo, 311
procedimiento de validación, 312
caja de arena, modelo de protección, 239
calidad de servicio, 42
gestión de, 585-594
negociación de, 587-593
parámetros de, 587
callback, *véase* devolución de llamada
cambio de contexto, 209
camino virtual (en redes ATM), 112
canal
fiable, 401
seguro, 57
virtual (en redes ATM), 112
canales de comunicación
amenazas, 56
prestaciones, 46
capabílity, *véase* habilitación
capacidad de respuesta, 41
casos de estudio
Andrew File System (AFS), 316-325
ATM (Asynchronous Transfer Mode), 112-114
Bayou, 554-556
Coda, sistema de archivos, 556-563
Comunicación entre procesos en UNIX, 147-150
CORBA, 637-660
DNS (Sistema de Nombres de Dominio), 346-352
Ethernet, 105-109
GNS (Servicio de Nombres Global), 356-359
Gossip, arquitectura, 545-554
Invocación de métodos remotos en Java, 182-190
Ivy, 622-626
Kerberos, 275-280
LAN inalámbrica IEEE 802.11, 109-112
Mach, 665-685
Millicent, protocolo, 284-287
Munin, 630-632
Needham-Schroeder, protocolo, 273-275
NFS (Network File System), 305-316
NTP (Network Time Protocol), 375-377
SSL (Secure Sockets Layer), 280-284
Sun RPC, 173-175
X.500, Servicio de directorio, 359-363
catálogo (suite) de cifrado, 282
caudal
adaptación de, 596
multimedia, 584
anchos de banda usuales, 583
ráfagas de carga, 589
caudales de datos
isócronos, 584
temporizados, 584
véase red, caudales de datos
causal, consistencia, *véase* consistencia causal
CDR, *véase* CORBA Common Data Representation
certificado, 248-250
autoridad, *véase* autoridad de certificación
formato estándar X.509, 269
Chorus, 229
CIDR (rutado entre dominios sin clase), 93
cifrador
de bloque, 256-257
de encadenamiento de bloques, 256
de flujo, 257
circuito virtual, 76
clave
de sesión, 246
pública, 245
certificado de, 249
criptografía de, 261-264
privada, 245
cliente
ligero (delgado), 36
servidor
comunicación, 135-143
modelo, 32
variaciones, 34-40
clientes, 8
cluster de computadores, 203
Coda, sistema de archivos, 556-563
disponible (AVSG), 557
vector de versión (CVV), 558
grupo de volúmenes de almacenamiento (VSG), 557
codec (codificador/descodificador), 585
código
de autenticación de mensaje (MAC), 266
descargado, 14
móvil, 35
amenazas a la seguridad, 240
coherencia de la memoria compartida distribuida, 615-616
cola de retención, 420
comercio electrónico, necesidades de seguridad, 241
commit, *véase* consumado
como-máximo-una-vez, semántica de invocación, *véase* invocación, semántica
compactación automática de memoria, 160-161
distribuida, 171-172
en Java, 171
utilización de concesiones, 172
en un sistema de objetos distribuido, 163
local, 160
compartición
de carga, 207
de recursos, 7-8
falsa, 618
competencia de accesos a memoria, 627
compresión, 584
de datos para multimedia, 584
computación
independiente de posición, 6
móvil, 5-7
nómada, 6
ubícua, 6
computadora de red, 36
comunicación
asíncrona, 119
cliente-servidor, 135-143
de grupo, 143-147
en grupo virtualmente síncrona, 537
entre procesos
características, 119
en UNIX, 147-150
fiable, 53
integridad, 54
síncrona, 119
soporte del sistema operativo para la, 216-223
validez, 54
concentrador, 83
Ethernet, 83
concesiones, 187
en compactación automática de memoria distribuida en Jini, *véase* Jini
para devoluciones de llamada, 187

concurrencia, 21-22
control de, 451-455
comparación de métodos, 483
con bloqueos, *véase* actualización de bloqueos perdidos
en el servicio de control de concurrencia de CORBA, 462
en transacciones distribuidas, 503-506
operaciones conflictivas, 453
optimista, *véase* optimista, control de concurrencia
por ordenamiento de marcas temporales, *véase* ordenamiento por marcas temporales
recuperación inconsistente, 452
reglas de conflicto de operaciones, 453
de actualizaciones sobre archivos, 297
conectar y ejecutar, universal, *véase* plug and play universal
conexión persistente, 141
confusión (en criptografía), 258
conmutación
de paquetes, red de, *véase* red, conmutación
operaciones de, 544
conmutador Ethernet, 83
consenso, 428-438
con relación a otros problemas, 431
en un sistema síncrono, 432
por quorum, 570-572
resultado de imposibilidad para un sistema asíncrono, 436
consistencia
atómica, 614
causal, 632
de admisión, 632
de alcance, 633
de datos
compartidos, 613
replicados, 529
de procesador, 632
débil, 633
interactiva, 431
secuencial, 540
de la memoria compartida distribuida, 615
consumado, 449
contexto de nombres (nominación), 343
control
de admisión, 593-594
de congestión, *véase* red, control de congestión
de flujo, 99
cookie, 308
copia
en escritura, 204
en Mach, 680-681
primaria, 316
CORBA
arquitectura, 644-647
adaptador de objeto, 645
esqueleto, 646
intermediario de petición de objetos (ORB), 645
proxy, 646
caso de estudio, *véase* casos de estudio, CORBA
CDR (Common Data Representation), 130-132
correspondencia con el lenguaje, 652
ejemplo de cliente y servidor, 641-644
empaquetado, 132
extensiones, 650
objetos por valor, 650
RMI asíncrono, 651
interfaz
de esqueleto dinámico, 647
de invocación dinámica, 647
intermediario de petición de objetos (ORB), 646
invocación de método remoto, 638-644
asíncrona, 651
devolución de llamada, *véase* devolución de llamada
lenguaje de definición de interfaces, 647-651
atributo, 649
herencia, 650
interfaz, 647
método, 648
módulo, 647
parámetros y resultados, 639
pseudo-objeto, 641
tipo, 649
marshalling, *véase* CORBA, empaquetado
modelo de objeto, 639
protocolo
general Inter-ORB (GIOP), 638
interoperable Inter-ORB (IIOP), 638
referencia
a objeto remoto, *véase* CORBA, IOR
interoperable de objeto (IOR), 651
persistente, 652
transitoria, 652
repositorio
de implementación, 646
de interfaz, 651
servicios, 653-660
de control de concurrencia, 653
de eventos, 657-658
de intercambio (comercio), 653
de nombres, 654-657
de notificación, 658-659
de objetos persistentes, 654
de seguridad, 660
de transacciones, 653
correlacionador de puertos, *véase* también enlazador, portmapper
cortafuegos, 254

corte, 382
consistente, 382
frontera, 383
credenciales, 252-254
criptografía, 254-264
asimétrica, 255
prestaciones de los algoritmos, 271
simétrica, 255
y política, 271
CSMA/CA, *véase* red, CSMA/CA
CSMA/CD, *véase* red, CSMA/CD

D

datagrama, 76
delegación (de derechos), 253
denegación de servicio, 57
depuración de programas distribuidos, 382
DES (Data Encryption Standard), 259
desafío de autenticación, 247
descarga de código en Java RMI, *véase* Java, RMI, descarga de clases
descubrimiento, servicio, *véase* servicio de descubrimiento
desempaquetado, 130
detección
de colisiones, 107
de fallo, 20
detector de fallo, 401-403
para resolver el consenso, 437
determinación de terminación, 381
devolución de llamada
en invocación de métodos remotos en
CORBA, 644
Java, 187
promesa de, 321
difusión
(en criptografía), 256
global, 415
dirección
de Internet, API de Java, 121
de socket, *véase* socket
de transporte, 75
disponibilidad, 21
dispositivos móviles, 37
distribuidor, 168
DNS, *véase* Sistema de Nombres de Dominio
dominio
de nombres, 341
de protección, 250
dos-fases, protocolo de consumación en, *véase* protocolo de consumación atómica
DSL (línea de subscriptor digital), 67
DSM, *véase* memoria compartida distribuida

E

ejecución, una, 384
ejecuciones estrictas, 456
elección, 411-415
algoritmo del abusón, 413
para procesos en un anillo, 411
empaquetado, 130
emulación de un sistema operativo, 228
encaminador, *véase* router
encapsulación, 72
encriptación, 56
por curvas elípticas, 263
enemigo, 55
amenazas de seguridad por un, 56
enlace
de datos, nivel de, *véase* niveles
de un socket, 147
enlaces (vínculos), 9
enlazador, 169
portmapper, 175
rmiregistry, 184
enmascaramiento de fallo, 20
ensamblado de paquetes, 74
entorno de ejecución, 200
envío no bloqueante, 120
equidad, condición de, 404
equivalencia secuencial, 452
escalabilidad, 19-20
escritura prematura, 456
espacio
de direcciones, 199
herencia, 204
región, 201
compartida, 202
suplantación, 211
de nombres, 340
de tuplas, 611
especificación de caudal, 591
espontánea, red, 37-40
esqueleto, 168
en CORBA
estado
de transacción, 514
global, 381-389
consistente, 384
estable, 384
instantánea, 385-389
predicado, 384
estándares IEEE 802, 104
Ethernet, 105-109
evento, 175-182
anunciante, 179
concurrencia, 378-379
en el servicio de eventos de CORBA, 657-658

en el servicio de notificación de CORBA, 658-659
en Jini, 180-182
encaminamiento, 180
filtrado, 180
notificación, 175-182
objeto de interés, 178
observador, 179
ordenamiento, 48
patrones, 180
semántica de reparto, 179
suscriptor, 179
tipo, 177
exactamente una vez, 164
excepción, 160
captura, 160
en la invocación de métodos remotos en Java, 166
en la invocación remota en CORBA, 166
lanza, 160
exclusión mutua, 403-410
algoritmo de Maekawa, 408
entre procesos en un anillo, 406
mediante multidifusión, 407-408
por un servidor central, 405
testigo, 406
Exokernel, 230
extensibilidad, 17

F

fallo
arbitrario, 52
bizantino, *véase* bizantino, fallo
independiente, 2
parada, 51
por caída, 402
por omisión, 50
en comunicación, 51
en proceso, 50
temporización, 53
fallos
de entrega, 137
por omisión
de envío, 51
de recepción, 51
falta de página, 205
fantasma, interfloqueo, *véase* interbloqueo fantasma
fiabilidad, 43
finalización del consenso y problemas relacionados, 429-430
Firefly RPC, 220
firewall, *véase* cortafuegos
firma digital, 264-270
físico, nivel, *véase* niveles
fluctuación, 47
frame relay, *véase* red, frame relay
Frangipiani, sistema de archivos distribuido, 331
frontal, 532
FTP, 74
fuga de información como amenaza a la seguridad, 240
función
de dispersión segura, 265
de hash de un solo sentido, 268
de puerta falsa, 255
de resumen, 265
segura, 248
de un solo sentido, 255

G

galletita, *véase* cookie
garbage collection, *véase* compactación automática de memoria
Gateway, 68
generador de flujo de claves, 257
generales bizantinos, *véase* bizantino, generales
gestión
de fallos, 20-21
de recursos (para multimedia), 594-596
gestor
de bloqueos, *véase* bloqueo, gestor de bloqueos
de recuperación, 513
de réplicas, 530
Global
Name Service, 356-359
identificador de directorio, 356
raíz de trabajo, 357
Positioning System (GPS), 371
GNS, *véase* Global Name Service
GPS, *véase* Global Positioning System
grafo de precedencia, 569
grupo
abierto, 417
cerrado, 416
comunicación en, 533-538
con vistas síncronas, 536-537
de objetos, 536
servicio de pertenencia, 533
solapamiento de, 427
vista, 535-536
GSM, red de telefonía celular, 67

H

habilitación, 251
hablar por, relación (en seguridad), 253
Handle, sistema, 339
handsake, protocolo de, en SSL, 280

heterogeneidad, 15
servicio de nombres, 342
híbrido
modelo de consistencia de memoria, *véase* modelo de consistencia de memoria híbrido
protocolo criptográfico, *véase* protocolo criptográfico híbrido
hilo, 205-216
arquitectura multi-hilo, 207
asociación de trabajadores, 207
C, 210
comparación con proceso, 208
conmutación entre, 209
coste de creación, 210
de nivel de núcleo, 214
en el cliente, 208
en el servidor, 206
en un multiprocesador, 208
implementación, 213-216
Java, 210-211
planificación, 213
POSIX, 210
programación con, 210-211
recibe bloqueante, 120
sincronización, 212
historial, 369
de operaciones del servidor en petición-respuesta, 138
global, 384
histórico
almacén de archivo estructurado en, 329
registro, 515-518
HTML, 11
HTTP, 14
prestaciones de, 221
sobre una conexión persistente, 225
hub
Ethernet, *véase* concentrador Ethernet

I

IANA (Internet Assigned Numbers Authority), 75
IDEA (International Data Encryption Algorithm), 261
identificador, 336
de grupo de archivo, 305
único de archivo (UFID), 300
IDL, *véase* lenguaje de definición de interfaz
IIOP (protocolo interoperable Inter-ORB), *véase* CORBA, IIOP
inalámbrica, red, *véase* red, inalámbrica
inanición, 404
i-nodo, número, *véase* número de i-nodo
integridad
de la entrega por multidifusión, 417
de las comunicaciones fiables, 53
en el consenso y los problemas relacionados, 429-430
intenciones, lista de, 514
interacción, modelo de, 45-50
interbloqueo, 466-469
con bloqueos lectura-escritura, 466
definición, 467
detección, 468
distribuido, 506-513
caza de arcos, 509-511
prioridades de trasacciones, 511
interbloqueos fantasma, 508
grafo espera-por, 467
prevención, 468
tiempos límite de espera, 469
interconexión de sistemas abiertos, *véase* Modelo de Referencia OSI
interfaz, 155-157
de red, nivel de, *véase* niveles
del servicio, 158
en un sistema distribuido, 157
lenguaje de definición, *véase* lenguaje de definición de interfaz
remota, 158
en Java RMI, 182
intermediario de petición de objetos (ORB), *véase* CORBA
Internet, 3
estadísticas de tráfico, 64
Protocol, *véase* IP
protocolos de encaminado, 91
interredes, 81-84
interrupción
interna de llamada al sistema, 199
software, 209
Intranet, 4
invoca una operación, 8
invocación
de métodos remotos, 8
caso de estudio Java, 182-190
CORBA, *véase* CORBA
distribuidor, 168
esqueleto, 168
filtrado de duplicados, 164
implementación, 166-171
módulo
de comunicación, 167
de referencia remota, 167
nula, 218
paso de parámetros y de resultados en Java, 183
prestaciones de, *véase* invocación, mecanismo de proxy, 167
reintento de mensaje de petición, 164
retransmisión de respuestas, 164

semántica, 164
transparencia, 165
mecanismo de, 196
asíncrona, 224
persistente, 225
dentro de un computador, 222-223
latencia, 218
planificación y comunicación como parte la, 196
prestaciones, 217-223
soporte del sistema operativo, 216-226
tasa de productividad, 219
semántica de
al menos una vez, 165
como máximo una vez, 165
pudiera ser, 164
IOR, *véase* CORBA, referencia interoperable de objeto
IP, 89-97
API, 119-128
direccionamiento, 86-89
multidifusión, 144-146
API de Java, 144
asignación de direcciones, 145
modelo de fallo, *véase* modelo de fallo de la multidifusión IP
router, 144
trucado, 90
IPC, *véase* comunicación entre procesos
IPv4, 86
IPv6, 93-96
ISDN, 81
ISIS, 537
isócronos, caudales de datos, *véase* caudales de datos isócronos
Ivy, 622-626

J

Java
hilo, *véase* hilo, Java
invocación de métodos remotos (RMI), 182-190
devolución de llamada, *véase* devolución de llamada
descarga de clases, 184
diseño e implementación, 188-189
empleo de la reflexión, 188
excepción, 166
interfaz remota, 182
paso de parámetros y resultados, 183
programa cliente, 186
programa servidor, 184
RMIregistry, 184
reflexión, 133
seguridad en, 240
serialización de objetos, 132-134
Jini
concesiones, 172
especificación de eventos distribuidos, 180-182
servicio de descubrimiento, 353-356
concesiones, 356
jitter, *véase* fluctuación

K

Kerberos, 275-280
autenticación para NFS, 313

L

L4, micronúcleo, 230
Lamport, marca temporal, *véase* marca temporal, Lamport
LAN
inalámbrica IEEE 802.11, *véase* red, LAN inalámbrica IEEE 802.11
véase red, área local
lanza excepción, 160
latencia, 46
LBAP, *véase* modelo de llegada lineal de procesos
LDAP, *véase* protocolo de acceso a directorio de peso ligero
lectura sucia, 455
lenguaje de definición de interfaz, 157
CORBA, *véase* CORBA, lenguaje de definición de interfaz
ejemplo
CORBA IDL, 159
Sun RPC, 173
liberación de consistencia, 626-632
Linda, 611
linealizabilidad, 539
linealización, 384
little-endian, ordenamiento, *véase* ordenamiento de bits
llamada
a procedimiento remoto, 172-175
de peso ligero, 222
con cola, 226
nulo, 218
prestaciones de, *véase* invocación, mecanismo de protocolos de intercambio, 139
hacia atrás, *véase* upcall

M

MAC, *véase* código de autenticación de mensaje
Mach, 665-686
comunicación, 672-675
en red, 676-678

conjunto de puertos, 669
derecho de puerto, 669
gestión de memoria, 678-684
memoria virtual, 668
mensaje, 669
nominación, 669
objeto de memoria caché, 669
paginador externo, 681-682
puerto, 669
de red, 677
servidor de red, 677
tarea, 668
MAN, *véase* red, área metropolitana
máquina
de estado, 531
virtual, 16
marca temporal
de Lamport, 379
vectorial, 380
ordenamiento de, 476-482
conflictos de operaciones, 477
en transacciones distribuidas, 504
multiversión, 480-481
regla de escritura, 477
regla de lectura, 478
máxima unidad de transferencia (MTU), 106
MD5, algoritmo de resumen de mensajes, 269
mecanismo de seguridad, *véase* seguridad
memoria
compartida distribuida, 605-636
cuestiones de diseño e implementación, 610-619
datos inmutables alojados en, 611
escritura
actualizante, 616
invalidante, 616
gestor
centralizado, 623
distribuido, 624-626
granularidad, 617-618
modelo
de consistencia, 612-616
de sincronización, 612
opciones de actualización, 616-617
orientada
a bytes, 610
al objeto, 611
thrashing (fustigamiento), 619
virtual
en Mach, 678-684
paginador externo, 681-681
mensaje
de petición, 139
de respuesta, 139
perdida, 138
destino de, 120
metadatos, 295
método factoría, 168
micronúcleo, 227-228
comparación con el núcleo monolítico, 229
Middleware, 16
soporte del sistema operativo para el, 195
y heterogeneidad, 157
migración de proceso, *véase* proceso, migración
Millicent, protocolo, *véase* protocolo Millicent
mime, tipo, 141
mobileIP, *véase* red, mobileIP
modelo
de consistencia de memoria
híbrido, 632
uniforme, 632
de fallo, 50-54
de la multidifusión IP, 145
de TCP, 127
de transacciones, 446
de UDP, 123
del protocolo
de consumación atómica, 496
de petición-respuesta, 138
de interacción, 45-50
de llegada lineal de procesos (LBAP), 589
de objetos distribuidos, *véase* objeto, modelo de referencia OSI, 73
de seguridad, 54-58, *véase* seguridad fundamental, 44-48
modelos
arquitectónicos, 28-44
fundamentales, 44-58
modo
supervisor, 199
usuario, 199
módulo de referencia remota, *véase* invocación de métodos remotos
moldeado del tráfico (para datos multimedia), 590
móvil, computación, *véase* computación móvil
MPEG, compresión, 589
de vídeo, 584
MTU, *véase* máxima unidad de transferencia
multicast, *véase* multidifusión
multidifusión, 415-428
atómica, 421
básica, 417
de grupo, 144, 415
fiable, 417-421
IP, *véase* IP, multidifusión
no fiable, 145
ordenado, 421-428
para datos
con alta disponibilidad, 143
replicados, 147
para grupos solapados, 427

para notificación de eventos, 143
para servicios de descubrimiento, 143
para tolerancia a fallos, 143
reparto ordenado
causalmente, 421
en forma FIFO, 421
totalmente, 421
multimedia
aplicaciones, 112
basada en web, 582
multiprocesador
memoria
compartida, 197
NUMA (Acceso a Memoria No Uniforme), 607
distribuida, 607
soporte de Mach para, 666
Munin, 630-632

N

navegación
por multidifusión, 344
controlada por servidor, 344
Needham-Schroeder, protocolo, 273-275
Nemesis, 229
netmsgserver, *véase* Mach, servidor de red
Network
File System (NFS), 305-316
autenticación Kerberos, 313
automounter, 310
mejoras, 326
montado estático y flexible, 309
prestaciones, 314
pruebas de laboratorio, 314
servicio de montado, 307
Spritely NFS, 326
VFS (sistema de archivos virtual), 306
v-nodo, 306
WebNFS, 327
Information Service (NIS), 542
NIS, *véase* Network Information Service
niveles (capas), *véase* también protocolos
de aplicación, 72
de enlace de datos, 74
de interfaz de red, 74
de presentación, 74
de red, 74
de sesión, 74
de transporte, 74
físico, 73
NNTP, 84
no repudio, 242
nombre, 336
componente, 340
de host, 338
no ligado, 340
puro, 336
prefijo, 340
nonce, *véase* ocasión
notas históricas
aparición de la criptografía moderna, 237
redes, 62
seguridad, evolución de las necesidades de, 236
sistemas de archivos distribuidos, 292
notificación, *véase* evento, notificación
NQNFS (Not Quite NFS), 327
NTP (Network Time Protocol), 375-377
núcleo, 199
monolítico, 227
número de i-nodo, 306

O

objeto, 40
activación de, 169
activo, 169
CORBA, *véase* CORBA
de memoria, *véase* Mach, objeto de memoria
distribuido, 161
comunicación, 159-172
modelo, 161
factoría, 168
interfaz, 40
modelo de, 159
pasivo, 169
persistente, 169
en el servicio de objetos persistentes de CORBA, 654
protección, 54
referencia a, 160
referencia remota, 162
remoto, 162
ubicación, 170
ocasión, 274
ocurrió antes, 378
OMG (Object Management Group), 638
open system, *véase* sistema abierto
openness, *véase* extensibilidad
operación
atómica, 444
idempotente, 138
multidifunde, 143
operaciones
conflictivas, *véase* concurrencia, control de
sobre archivos en
el modelo de servicio
de achivos plano, 300

de directorio, 304
un servidor NFS, 308
UNIX, 296
optimista, control de concurrencia, 472-476
comparación de la validación hacia delante y hacia atrás, 476
en transacciones distribuidas, 505
fase de actualización, 473
fase de trabajo, 473
inanición, 476
validación, 473
hacia atrás, 474
hacia delante, 475
orca, 609
ordenamiento
causal, 378
de la gestión de peticiones, 532
del reparto por multidifusión, 421
de bits
big-endian, 129
little-endian, 129
del emisor, 121
FIFO
de la entrega por multidifusión, 420
de la gestión de peticiones, 532
total, 369
de la entrega por multidifusión, 421
de la gestión de peticiones, 532
OSI Modelo de Referencia, *véase* Modelo de Referencia OSI

P

paginador externo, 681-682
paquetes, *véase* red, paquetes
parámetros de entrada y de salida, 157
partición de red, 401
primaria, 534
virtual, 572-575
pasarela, *véase* gateway
paso de mensajes, 118
Petal, sistema de disco virtual distribuido, 331
petición-respuesta, protocolo de, 136-137
damePetición, 136
empleo de TCP, 139
envíaRespuesta, 136
hazOperación, 136
historia de operaciones del servidor, 138
mensaje de, 137
de respuesta perdidos, 138
petición duplicado, 138
modelo de fallo, 138
tiempo límite de espera, 138
Plan 9, 343
planificación
apropiativa, 213
de recursos por el mejor esfuerzo, 585
de tiempo real, 595
no apropiativa, 213
planificador, activación, 215
plataforma, 195
de computación fiable, 243
plug and play universal, 353
política de seguridad, *véase* seguridad
POP, 84
port mapper, *véase* correlacionador de puertos
POTS (antiguo sistema telefónico simple), 70
PPP, 86
presentación, nivel de, *véase* niveles
Pretty Gook Privacy (PGP), 272
primer camino abierto más corto, 91
principal, 54
procesador virtual, 214
proceso, 199-216
amenazas a, 55
correcto, 402
coste de creación, 210
creación, 202-205
de nivel de usuario, 199
ligero de Solaris, 214
migración, 204
multi-hilo, 200
promedio de tolerancia a fallos, 374
promesa, 225
de devolución de llamada, *véase* devolución de llamada
propagación de actualizaciones ansiosa o impaciente, 565
propiedad
de seguridad, 385
de uniformidad, 420
protección, 198-199
por el núcleo, 199
y lenguaje de tipos seguro, 199
protocolo, 71-77
anti-entropía, 554
ARP, 90
Bellman-Ford, protocolos de rutado, 78
composición dinámica, 217
conjunto de, 73
criptográfico híbrido, 264
de acceso a directorio de peso ligero, 363
de consumación atómica, 465-503
modelo de fallo, *véase* también modelo de fallo
en dos fases, 496-499
acciones de tiempo de espera límite, 497
prestaciones, 498
recuperación, 520-523
transacción distribuida, 499-503

consumación jerárquica, 501
consumación plana, 502
FTP, 74
HTTP, 74
IP, 74
IPv4, 86
IPv6, 86
mobileIP, 96-97
niveles (capas), *véase* también niveles
NNTP, 84
pila de, 73
POP, 84
PPP, 86
simple de servicio de descubrimiento, 353
SMTP, 8
soporte del sistema operativo para los, 217
SSL, 74
TCP, 86
TCP/IP, 84
transporte, 74
UDP, 86
WAP, 85
Proxy, 167
en CORBA, 645
publica-suscribe, 176
pudiera ser, semántica de invocación, *véase* invocación, semántica
puente, *véase* bridge
puerto, 75
de servidor, 122
en Mach, *véase* Mach, puerto
punto de control, establecimiento de, 517

Q

QoS, *véase* calidad de servicio

R

RAID (conjuntos redundantes de discos asequibles), 330
RAM encauzada, 632
RDSI, *véase* ISDN
Real Time Transport Protocol (RTP), 70
realización, de accesos a memoria, 628
recolección de basura, *véase* compactación automática de memoria
reconocimiento negativo, 419
recuperación, 513-523
abortos encadenados 456
de un aborto, 455
del protocolo de consumación en dos fases, 520-523
desde un fallo, 21
ejecuciones estrictas, 456
escritura prematura, 456
estado de transacción, 514
históricos, 515-518
inconsistente, 452
lectura sucia, 455
lista de intenciones, 514
transacciones anidadas, 521
versiones sombra, 518-519
recurso, 2
ancho de banda, 581
invocación sobre un, 196
red
ancho de banda total del sistema, 63
área
extensa, 66
local, 66
metropolitana, 67
ATM (Asynchronous Transfer Mode), 112-114
BlueTooth, red inalámbrica, 67
bridge, 83
conmutación
de circuitos, 70
de paquetes, 70
control de congestión, 80
CSMA/CA, 111
CSMA/CD, 106
de área local, *véase* red, área local
dirección de transporte, 75
ensamblado de paquetes, 74
estándares, 104
Ethernet, 105-109
flujos de datos, 69
frame relay, 71
gateway, 68
inalámbrica, 67
Internet, 84-104
LAN inalámbrica IEEE 802.11, 109-112
latencia, 63
multidifusión IP, 83
nivel de, *véase* niveles
paquetes, 69
parámetros de prestaciones, 63
privada virtual (VPN), 103
protocolo, *véase* protocolo
puerto, 75
requisitos, 63-65
de escalabilidad, 64
de fiabilidad, 64
de seguridad, 64
router, 82
rutado, 77-80
tasa de transferencia de datos, 63
TCP/IP, 84
túneles, 83

WaveLAN, 104
redundancia, 21
referencia a objeto remoto (ROID), 162
en CORBA, 651
reflexión en Java, 134
RMI, 188
región, *véase* espacio de direcciones
registrar el interés, 176
reloj
acuerdo, 371
del computador, 369
deriva, 370
erróneo, 372
exactitud, 371
global, 2
lógico, 379
matriz de, 381
monotonicidad, 371
precisión, 371
resolución, 370
sesgo, 370
sincronización, *véase* sincronización de relojes
vector de, 380
replicación, 527-577
activa, 543-545
consenso del quórum, 570-572
copias disponible, 566-568
con validación, 568-570
de archivos, 297
partición virtual, 572-575
pasiva, *véase* replicación primario-respaldo
primario-respaldo, 541-543
transaccional, 563-567
transparencia, 529
repositorio
de implementación en CORBA, *véase* CORBA
de interfaz, en CORBA, *véase* CORBA, repositorio de interfaz
representación externa de datos, 128-135
requisitos de diseño, 41
resguardo, procedimiento de, 172
en CORBA, 645
resolución de nombres, 336
Resource Reservation Protocol (RSVP), 70
respuesta, mensaje de, 138
resumen de un mensaje, 247
retransmisión de tramas, *véase* red, frame relay
retrollamada, *véase* devolución de llamada
RFC, 17
RIP
(protocolo de información de router), 80
-1, 80
-2, 91
RMI, *véase* invocación de métodos remotos
RMIregistry, 184
Router 81, 82
RPC
de peso ligero, 222
protocolos de intercambio de, 139
véase llamada a procedimiento remoto
RR, 139
RRA, 139
RSA, algoritmo de encriptación de clave pública, 262
RSVP, protocolo para reserva del ancho de banda, 593
rutado, *véase* red, rutado

S

sala de contratación, 177
scrip, *véase* vale
sección crítica, 403
secuenciabilidad de una copia, 563
secuenciador, 424
Secure Sockets Layer (SSL), 280-284
seguridad, 18
«Alice y Bob», nombres para los protagonistas, 238
amenazas
desde el código móvil, 239
fugas, modificación, vandalismo, 238
ataque
del cumpleaños, 268
del hombre entre medias, 239
por denegación de servicio, 239
por fisgoneo, 239
por fuerza bruta, 255
por modificación de mensaje, 239
por retransmisión, 239
por suplantación, 238
por texto en claro escogido, 263
en el servicio de seguridad de CORBA, 660
en Java, 239
guías de diseño, 242
mecanismo de, 236
modelo de, 54-58
modelos de fuga de información, 240
plataforma de computación fiable, 244
política de, 236
Transmission Layer Security (TLS), *véase* Secure Sockets Layer, 280
semántica de actualización, 298
serialización, 132
de objetos en Java, *véase* Java
servicio, 8
con alta disponibilidad, 545-563
de archivos planos, 300
de descubrimiento, 352
alcance, 353
búsqueda, 353
Jini, 354-356

protocolo simple de servicio de descubrimiento, 353
registro, 353
de directorio, 352
atributo, 352
de localización, 170
de nombres, 335-365
empleo de caché, 345
heterogeneidad, 342
navegación, 343-345
replicación, 348
servicio de nombres de CORBA, *véase* CORBA, servicio
tolerante a fallos, 538-545
servidor, 8
arquitectura multi-hilo, 207
multi-hilo, 206
proxy, 33
sin estado, 302
tasa de servicio, 206
Servlet, 212
sesión, nivel de, *véase* niveles
SHA, algoritmo de dispersión seguro, 269
signatura de un método, 160
Simple Public-key Infrastructure (SPKI), 270
sin conexión, funcionamiento, 529
sincronización
accesos de, a memoria, 627
de medios, 582
de operaciones del servidor, 445
de relojes, 371-377
algoritmo de Berkeley, 374
algoritmo de Cristian, 373
en un sistema síncrono, 372
externa, 371
interna, 371
Network Time Protocol, *véase* Network Time Protocol
sistema
abierto, 17
de archivo sin servidor (xFS), 329
de archivos virtual (VFS), *véase* Network File System
de Nombres de Dominio (DNS), 346-352
consulta, 346
implementación BSD, 351
navegación, 349
nombre de dominio, 346
registro de recurso, 350
servidor de nombres, 347-352
zona, 348
distribuido síncrono, 432
operativo
arquitectura, 226-230
distribuido, 195
en red, 194
política y mecanismo, 227
soporte
a la invocación y a la comunicación, 216-226
a los procesos y a los hilos, 199-216
sistemas distribuidos abiertos, 17
SMTP, 84
socket
bind, 147
close, 147
comunicación
con stream, 149
por datagrama, 148
connect, 129
dirección, 147
llamada al sistema, 150
pareja de, 147
SPIN, 229
Sprint, servicio de nombres, 342
Spritely NFS, *véase* Network File System
SSL, *véase* Secure Sockets Layer
subsistema, 228
Sun Network File System (NFS), *véase* Network File System
Sun RPC, 173-175
lenguaje de definición de interfaz, 173
representación externa de datos, 175
rpcgen, 174
supervisor, 198
suplantación, 341
suscribe, 176
switch Ethernet, *véase* conmutador ethernet

T

tabla de objetos remotos, 167
tablero compartido, 182
tarea, *véase* Mach, tarea
tasa de servicio, 41
TCP, 125-128
API, 125
de Java, 127-128
de UNIX, 149
modelo de fallo, 27
y protocolos de petición-respuesta, 139
TCP/IP, *véase* protocolo, TCP/IP
TEA (Tiny Encryption Algorithm), 258-259
temas de prestaciones, 41
temporización crítica, datos de, 42
texto
en claro, 254
en clave, 254
thrashing (fustigamiento), 619
Ticket
de autenticación, 246
Granting Service (TGS), 276
tiempo, 367-381
Atómico Internacional, 370
de ejecución, para datos multimedia, 69
lógico, 49
Universal Coordinado (UTC), 371
tiempos de espera límite, 50
Tiger, servidor de archivos de vídeo, 598
TLS, *véase* Secure Sockets Layer
timeouts, *véase* tiempos de espera límite
tolerancia a fallos, 21
totalmente ordenada, multidifusión, *véase* multidifusión
Trampolining, 668
transacción, 447-459
aborta, 448
abortaTransacción, 450
abreTransacción, 450
ACID, propiedades, 449
anidada, 457-459
bloqueo, 465
protocolo de consumación en dos fases, 499-503
acciones por timeout, 503
recuperación, 521
cierraTransacción, 450
con datos replicados, 563-575
consuma, 449
control de concurrencia, *véase* control de concurrencia
distribuida
anidada, 492
control de concurrencia, 503-506
bloqueo, 503
optimista, 505
por ordenamiento de marcas temporales, 504
coordinador, 493-495
plana, 492
protocolo de consumación
atómica, 495-503
en dos fases, *véase* protocolo de consumación atómica, en dos fases
en una fase, 495
en el servicio de transacciones de CORBA, 653
equivalencia serie, 452
modelo de fallo, 446
recuperación, *véase* recuperación
transferencia de estado, 536
transformación operacional, 555
transición de dominio, 210
Transmission Layer Security, *véase* Secure Sockets Layer
transparencia, 22-24
de acceso, 22
de concurrencia, 22
de escalado, 23
de invocación a método remoto, 165
de movilidad, 22
de prestaciones, 22
de red, 23
de replicación, 22
de ubicación, 22
en el middleware, 156
frente a fallo, 22
transporte, nivel de, *véase* niveles
troncal, 90
túnel web, 254
Tunneling, 83

U

UDP, 122-125
API
de Java, 123-125
de UNIX, 148
empleo de, 123
modelo de fallo, 123
para la comunicación petición-respuesta, 221
UFID, *véase* identificador único de archivo
Uniform Resource
Characteristic, 338
Identifier, 338
Locator, 338
Name, 338
Universal Transfer Format, 133
UNIX
i-nodo, 306
llamadas al sistema
accept, 150
bind, 148
connect, 150
exec, 202
fork, 202
listen, 150
read, 150
recvfrom, 148
sendto, 148
socket, 148
write, 150
señal, 209
Upcall, 215
URC, *véase* Uniform Resource Characteristic
URI, *véase* Uniform Resource Identifier
URL, *véase* Uniform Resource Locator
URN, *véase* Uniform Resource Name
UTC, *véase* Tiempo Universal Coordinado
UTF, *véase* Universal Transfer Format

V, sistema
ejecución remota, 203
soporte para grupos, 538
vale, 285
validez
de la comunicación fiable, 54
de la entrega por multidifusión, 418
vector
de inicialización (para un cifrador), 256
de marcas temporales, 380
comparación, 381
operación de fusionado, 380
reloj, *véase* reloj
ventana de escasez (de recursos para aplicaciones multimedia), 583
Verisign Corporation, 270
versiones sombra, 518-519
vídeo bajo demanda, servicio, 582
videoconferencia, 582
aplicación
CU-SeeMe, 582
NetMeeting, 582
vista, de grupo, *véase* grupo, vista
vistas síncronas, comunicación de grupo por, *véase* grupo, comunicación en
vitalidad, propiedad, 385
votación, 408

W

WAN, *véase* red, área extensa
WAP (Wireless Application Protocol), *véase* protocolo
WaveLan, *véase* red
Web, *véase* World Wide Web
empleo de caché en, 43
WebNFS, *véase* Network File System
WLAN (red de área local inalámbrica), *véase* red, inalámbrica
World Wide Web, 8-15
WPAN (red de área personal inalámbrica), *véase* red, inalámbrica

X

X.500, servicio de directorio, 359-363
árbol de información de directorio, 359
LDAP, 363
X.509, certificado, *véase* certificado
XDR, *véase* Sun RPC, representación externa de datos
xFS, sistema de archivo sin servidor, *véase* sistema de archivo sin servidor xFS