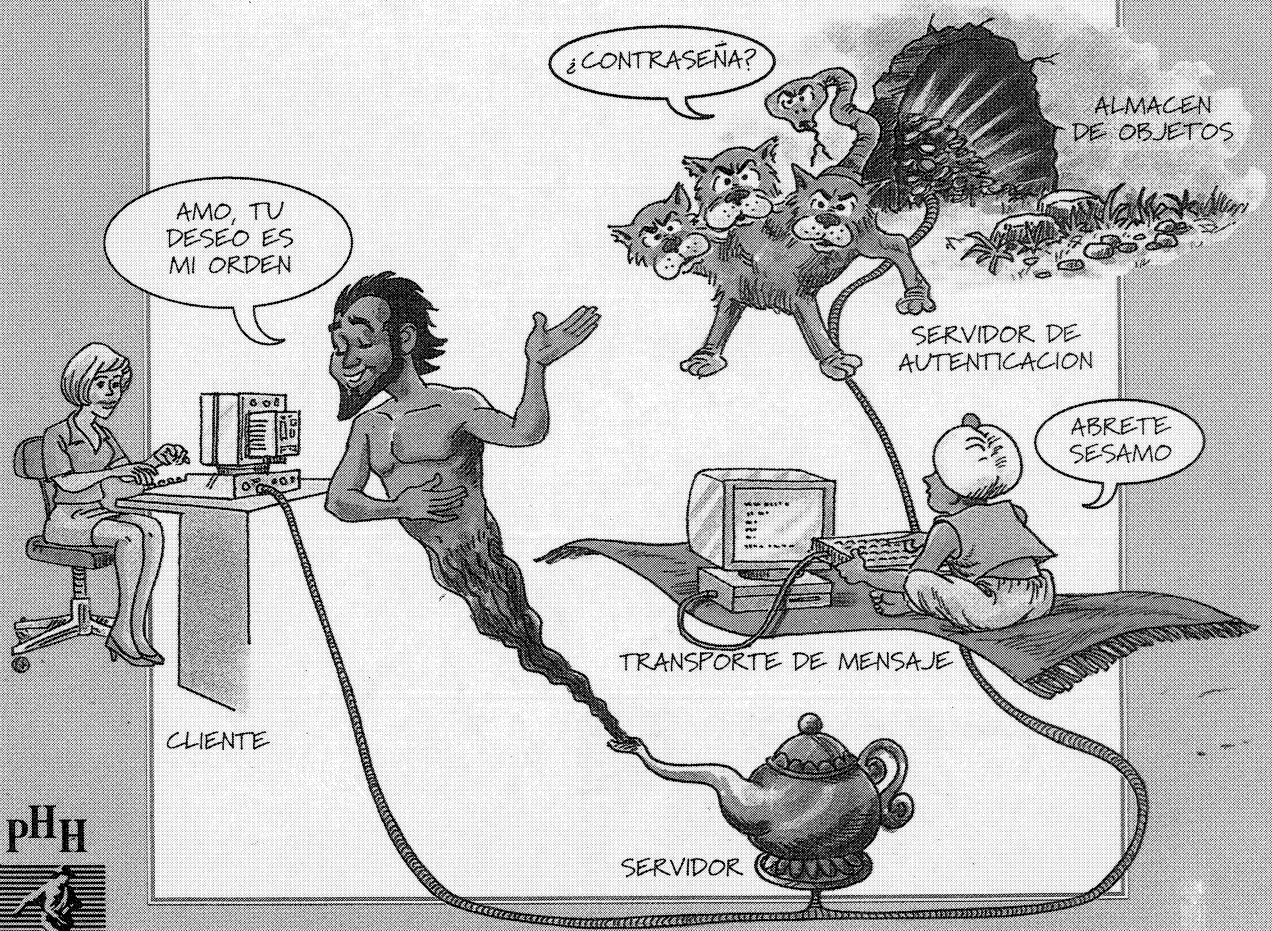
Sistemas Operativos Distribuidos



PHH
PRENTICE HALL

Andrew S. Tanenbaum

SISTEMAS OPERATIVOS DISTRIBUIDOS

PRIMERA EDICIÓN

Andrew S Tanenbaum

Vrije Universiteit
Amsterdam The Netherlands

TRADUCCIÓN

Óscar Alfredo Palmas Velasco
Facultad de Ciencias
Universidad Nacional Autónoma de México

REVISIÓN TÉCNICA

Gabriel Guerrero
Consultor SAXSA
Doctor en Informática
Universidad de París VI

PRENTICE HALL HISPANOAMERICANA S A

MÉXICO NUEVA YORK BOGOTÁ LONDRES SYDNEY
PARÍS MUNICH TORONTO NUEVA DELHI TOKIO
SINGAPUR RÍO DE JANEIRO ZURICH

EDICIÓN EN ESPAÑOL

PRESIDENTE DE LA DIVISION LATINO AMERICANA DE SIMON & SCHUSTER — RAYMUNDO CRUZADO GONZALEZ
DIRECTOR GENERAL — MOISES PEREZ ZAVALA
DIRECTOR DE EDICIONES — ALBERTO SIERRA OCHOA
GERENTE DIVISION UNIVERSITARIA — ENRIQUE IVAN GARCIA HERNANDEZ
GERENTE EDITORIAL — JOSE TOMAS PEREZ BONILLA
EDITOR — LUIS GERARDO CEDEÑO PLASCENCIA
GERENTE DE EDICIONES — JULIAN ESCAMILLA LIQUIDANO
SUPERVISOR DE TRADUCCION — JOAQUIN RAMOS SANTALLA
SUPERVISOR DE PRODUCCION — MAGDIEL GOMEZ MARINA

EDICIÓN EN INGLÉS

Acquisitions Editor Bill Zobrist
Production Supervisor Joe Scordato
Cover Designer DeLuca Design
Buyer Linda Behrens
Supplements Editor Alice Dworkin
Interior Designer Andrew S Tanenbaum

TANENBAUM SISTEMAS OPERATIVOS DISTRIBUIDOS 1 Ed

Traducción de la obra en inglés DISTRIBUTED OPERATING SYSTEMS

All rights reserved Authorized translation from English language edition published by Prentice Hall Inc

Todos los derechos reservados Traducción autorizada de la edición en inglés publicada por Prentice Hall Inc

All rights reserved No part of this book may be reproduced or transmitted in any form or by any means electronic or mechanical including photocopying recording, or by any information storage and retrieval system without permission in writing from the publisher

Prohibida la reproducción total o parcial de esta obra por cualquier medio o método sin autorización por escrito del editor

Derechos reservados © 1996 respecto a la primera edición en español publicada por
PRENTICE HALL HISPANOAMERICANA S A
Enrique Jacob 20 Col El Conde
53500 Naucalpan de Juárez Edo de México

ISBN 968 880 627 7

Miembro de la Cámara Nacional de la Industria Editorial Reg Núm 1524

Original English Language Edition Published by Prentice Hall Inc
Copyright © MCMXCV
All rights reserved

ISBN 0 13 219908 4

IMPRESO EN MÉXICO/PRINTED IN MEXICO

ENE
RA
AZ
OD NA 20
000 99

A Suzanne, Barbara, Marvin y el pequeño Brian

Contenido

PREFACIO xvii

1 INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS 1

1.1 ¿QUÉ ES UN SISTEMA DISTRIBUIDO? 2
1.2 OBJETIVOS 3
1.2.1 Ventajas de los sistemas distribuidos con respecto de los centralizados 3
1.2.2 Ventajas de los sistemas distribuidos con respecto de las PC independientes 6
1.2.3 Desventajas de los sistemas distribuidos 6
1.3 CONCEPTOS DE HARDWARE 8
1.3.1 Multiprocesadores con base en buses 10
1.3.2 Multiprocesadores con conmutador 12
1.3.3 Multicomputadoras con base en buses 13
1.3.4 Multicomputadoras con conmutador 14
1.4 CONCEPTOS DE SOFTWARE 15
1.4.1 Sistemas operativos de redes 16
1.4.2 Sistemas realmente distribuidos 18
1.4.3 Sistemas de multiprocesador con tiempo compartido 20
1.5 ASPECTOS DEL DISEÑO 22
1.5.1 Transparencia 22
1.5.2 Flexibilidad 25
1.5.3 Confiabilidad 27
1.5.4 Desempeño 28

1 5 5 Escalabilidad 29
1 6 RESUMEN 31

2 COMUNICACIÓN EN LOS SISTEMAS DISTRIBUIDOS 34

2 1 PROTOCOLOS CON CAPAS 35
2 1 1 La capa física 38
2 1 2 La capa de enlace de datos 38
2 1 3 La capa de red 40
2 1 4 La capa de transporte 40
2 1 5 La capa de sesión 41
2 1 6 La capa de presentación 41
2 1 7 La capa de aplicación 42
2 2 REDES CON MODO DE TRANSFERENCIA ASÍNCRONA 42
2 2 1 ¿Qué es el modo de transferencia asíncrona? 42
2 2 2 La capa física ATM 44
2 2 3 La capa ATM 45
2 2 4 La capa de adaptación ATM 46
2 2 5 Conmutación ATM 47
2 2 6 Algunas implicaciones del ATM para sistemas distribuidos 49
2 3 EL MODELO CLIENTE SERVIDOR 50
2 3 1 Clientes y servidores 51
2 3 2 Un ejemplo cliente servidor 52
2 3 3 Direccionamiento 56
2 3 4 Primitivas con bloqueo vs sin bloqueo 58
2 3 5 Primitivas almacenadas en buffer vs no almacenadas 61
2 3 6 Primitivas confiables vs no confiables 63
2 3 7 Implantación del modelo cliente servidor 65
2 4 LLAMADA A UN PROCEDIMIENTO REMOTO (RPC) 68
2 4 1 Operación básica de RPC 68
2 4 2 Transferencia de parámetros 72
2 4 3 Conexión dinámica 77
2 4 4 Semántica de RPC en presencia de fallas 80
2 4 5 Aspectos de la implantación 84
2 4 6 Áreas de problemas 95
2 5 COMUNICACIÓN EN GRUPO 99
2 5 1 Introducción a la comunicación en grupo 99
2 5 2 Aspectos del diseño 101
2 5 3 Comunicación en grupo en ISIS 110
2 6 RESUMEN 114

3 SINCRONIZACIÓN EN SISTEMAS DISTRIBUIDOS 118

3 1 SINCRONIZACIÓN DE RELOJES 119
3 1 1 Relojes lógicos 120
3 1 2 Relojes físicos 124
3 1 3 Algoritmos para la sincronización de relojes 127
3 1 4 Uso de relojes sincronizados 132
3 2 EXCLUSIÓN MUTUA 134
3 2 1 Un algoritmo centralizado 134
3 2 2 Un algoritmo distribuido 135
3 2 3 Un algoritmo de anillo de fichas 138
3 2 4 Comparación de los tres algoritmos 139
3 3 ALGORITMOS DE ELECCIÓN 140
3 3 1 El algoritmo del grandulón 141
3 3 2 Un algoritmo de anillo 143
3 4 TRANSACCIONES ATÓMICAS 144
3 4 1 Introducción a las transacciones atómicas 144
3 4 2 El modelo de transacción 145
3 4 3 Implantación 150
3 4 4 Control de concurrencia 154
3 5 BLOQUEOS EN SISTEMAS DISTRIBUIDOS 158
3 5 1 Detección distribuida de bloqueos 159
3 5 2 Prevención distribuida de bloqueos 163
3 6 RESUMEN 165

4 PROCESOS Y PROCESADORES EN SISTEMAS DISTRIBUIDOS 169

4 1 HILOS 169
4 1 1 Introducción a los hilos 170
4 1 2 Uso de hilos 171
4 1 3 Aspectos del diseño de paquetes de hilos 174
4 1 4 Implantación de un paquete de hilos 178
4 1 5 Hilos y RPC 184
4 2 MODELOS DE SISTEMAS 186
4 2 1 El modelo de estación de trabajo 186
4 2 2 Uso de estaciones de trabajo inactivas 189
4 2 3 El modelo de la pila de procesadores 193
4 2 4 Un modelo híbrido 197
4 3 ASIGNACIÓN DE PROCESADORES 197
4 3 1 Modelos de asignación 197
4 3 2 Aspectos del diseño de algoritmos de asignación de procesadores 199

4 3 3 Aspectos de la implantación de algoritmos de asignación de procesadores 201
4 3 4 Ejemplo de algoritmos de asignación de procesadores 203
4 4 PLANIFICACIÓN EN SISTEMAS DISTRIBUIDOS 210
4 5 TOLERANCIA DE FALLAS 212
4 5 1 Fallas de componentes 212
4 5 2 Fallas de sistema 213
4 5 3 Sistemas síncronos vs asíncronos 214
4 5 4 Uso de redundancia 214
4 5 5 Tolerancia de fallas mediante réplica activa 215
4 5 6 Tolerancia de fallas mediante respaldo primario 217
4 5 7 Acuerdos en sistemas defectuosos 219
4 6 SISTEMAS DISTRIBUIDOS DE TIEMPO REAL 223
4 6 1 ¿Qué es un sistema de tiempo real? 223
4 6 2 Aspectos del diseño 226
4 6 3 Comunicación en tiempo real 230
4 6 4 Planificación de tiempo real 234
4 7 RESUMEN 240

5 SISTEMAS DISTRIBUIDOS DE ARCHIVOS 245

5 1 DISEÑO DE LOS SISTEMAS DISTRIBUIDOS DE ARCHIVOS 246
5 1 1 La interfaz del servicio de archivos 246
5 1 2 La interfaz del servidor de directorios 248
5 1 3 Semántica de los archivos compartidos 253
5 2 IMPLANTACIÓN DE UN SISTEMA DISTRIBUIDO DE ARCHIVOS 256
5 2 1 Uso de archivos 256
5 2 2 Estructura del sistema 258
5 2 3 Ocultamiento 262
5 2 4 Réplica 268
5 2 5 Un ejemplo el sistema de archivos de red (NFS) de Sun 272
5 2 6 Lecciones aprendidas 278
5 3 TENDENCIAS EN LOS SISTEMAS DISTRIBUIDOS DE ARCHIVOS 279
5 3 1 Hardware reciente 280
5 3 2 Escalabilidad 282
5 3 3 Redes de área amplia 283
5 3 4 Usuarios móviles 284
5 3 5 Tolerancia de fallas 284
5 3 6 Multimedia 285
5 4 RESUMEN 285

6 MEMORIA COMPARTIDA DISTRIBUIDA 289

6 1 INTRODUCCIÓN 290
6 2 ¿QUÉ ES LA MEMORIA COMPARTIDA? 292
6 2 1 Memoria en circuitos 293
6 2 2 Multiprocesadores basados en un bus 293
6 2 3 Multiprocesadores basados en un anillo 298
6 2 4 Multiprocesadores con conmutador 301
6 2 5 Multiprocesadores NUMA 307
6 2 6 Comparación de los sistemas con memoria compartida 311
6 3 MODELOS DE CONSISTENCIA 315
6 3 1 Consistencia estricta 315
6 3 2 Consistencia secuencial 317
6 3 3 Consistencia causal 321
6 3 4 Consistencia PRAM y consistencia del procesador 322
6 3 5 Consistencia débil 325
6 3 6 Consistencia de liberación 327
6 3 7 Consistencia de entrada 330
6 3 8 Resumen de modelos de consistencia 331
6 4 MEMORIA COMPARTIDA DISTRIBUIDA CON BASE EN PÁGINAS 333
6 4 1 Diseño básico 334
6 4 2 Réplica 334
6 4 3 Granularidad 335
6 4 4 Obtención de la consistencia secuencial 337
6 4 5 Busqueda del propietario 339
6 4 6 Busqueda de las copias 342
6 4 7 Reemplazo de página 343
6 4 8 Sincronización 344
6 5 MEMORIA COMPARTIDA DISTRIBUIDA CON VARIABLES COMPARTIDAS 345
6 5 1 Munin 346
6 5 2 Midway 353
6 6 MEMORIA COMPARTIDA DISTRIBUIDA BASADA EN OBJETOS 356
6 6 1 Objetos 356
6 6 2 Linda 358
6 6 3 Orca 365
6 7 COMPARACIÓN 371
6 8 RESUMEN 372

7 ESTUDIO 1: AMOEBA **376**

7 1 INTRODUCCIÓN A AMOEBA 376
7 1 1 Historia de Amoeba 376
7 1 2 Objetivos de investigación 377
7 1 3 La arquitectura del sistema Amoeba 378
7 1 4 El micronucleo de Amoeba 380
7 1 5 Los servidores de Amoeba 382
7 2 OBJETOS Y POSIBILIDADES EN AMOEBA 384
7 2 1 Posibilidades 384
7 2 2 Protección de objetos 385
7 2 3 Operaciones estándar 387
7 3 ADMINISTRACIÓN DE PROCESOS EN AMOEBA 388
7 3 1 Procesos 388
7 3 2 Hilos 391
7 4 ADMINISTRACIÓN DE MEMORIA EN AMOEBA 392
7 4 1 Segmentos 392
7 4 2 Segmentos asociados 393
7 5 COMUNICACIÓN EN AMOEBA 393
7 5 1 Llamada a un procedimiento remoto (RPC) 394
7 5 2 Comunicación en grupo en Amoeba 398
7 5 3 El protocolo Internet Fast Local (FLIP) 407
7 6 LOS SERVIDORES DE AMOEBA 415
7 6 1 El servidor de archivos 415
7 6 2 El servidor de directorios 420
7 6 3 El servidor de réplicas 425
7 6 4 El servidor de ejecución 425
7 6 5 El servidor de arranque 427
7 6 6 El servidor TCP/IP 427
7 6 7 Otros servidores 428
7 7 RESUMEN 428

8 ESTUDIO 2: MACH **431**

8 1 INTRODUCCIÓN A MACH 431
8 1 1 Historia de Mach 431
8 1 2 Objetivos de Mach 433
8 1 3 El micronucleo de Mach 433
8 1 4 El servidor BSD UNIX de Mach 435
8 2 ADMINISTRACIÓN DE PROCESOS EN MACH 436
8 2 1 Procesos 436

8 2 2 Hilos 439
8 2 3 Planificación 442
8 3 ADMINISTRACIÓN DE MEMORIA EN MACH 445
8 3 1 Memoria virtual 446
8 3 2 Memoria compartida 449
8 3 3 Administradores externos de la memoria 452
8 3 4 Memoria compartida distribuida en Mach 456
8 4 COMUNICACION EN MACH 457
8 4 1 Puertos 457
8 4 2 Envío y recepción de mensajes 464
8 4 3 El servidor de mensajes de la red 469
8 5 EMULACIÓN DE UNIX EN MACH 471
8 6 RESUMEN 472

9 ESTUDIO 3: CHORUS 475

9 1 INTRODUCCIÓN A CHORUS 475
9 1 1 Historia de Chorus 475
9 1 2 Objetivos de Chorus 477
9 1 3 Estructura del sistema 477
9 1 4 Abstracciones del nucleo 479
9 1 5 Estructura del nucleo 481
9 1 6 El subsistema UNIX 483
9 1 7 El subsistema orientado a objetos 483
9 2 ADMINISTRACIÓN DE PROCESOS EN CHORUS 483
9 2 1 Procesos 484
9 2 2 Hilos 485
9 2 3 Planificación 486
9 2 4 Señalamientos excepciones e interrupciones 487
9 2 5 Llamadas al nucleo para la administración de procesos 488
9 3 ADMINISTRACIÓN DE MEMORIA EN CHORUS 490
9 3 1 Regiones y segmentos 490
9 3 2 Asociadores 491
9 3 3 Memoria compartida distribuida 492
9 3 4 Llamadas al nucleo para la administración de memoria 493
9 4 COMUNICACIÓN EN CHORUS 495
9 4 1 Mensajes 495
9 4 2 Puertos 495
9 4 3 Operaciones de comunicación 496
9 4 4 Llamadas al nucleo para la comunicación 498

9 5 EMULACIÓN DE UNIX EN CHORUS 499
9 5 1 Estructura de un proceso en UNIX 500
9 5 2 Extensiones a UNIX 500
9 5 3 Implantación de UNIX en Chorus 501
9 6 COOL UN SUBSISTEMA ORIENTADO A OBJETOS 507
9 6 1 La arquitectura COOL 507
9 6 2 La capa base de COOL 508
9 6 3 El sistema genérico de tiempo de ejecución de COOL 509
9 6 4 El sistema de tiempo de ejecución de lenguaje 510
9 6 5 Implantación de COOL 510
9 7 COMPARACIÓN DE AMOEBA MACH Y CHORUS 510
9 7 1 Filosofía 511
9 7 2 Objetos 512
9 7 3 Procesos 513
9 7 4 Modelo de memoria 514
9 7 5 Comunicación 515
9 7 6 Servidores 516
9 8 RESUMEN 517

10 ESTUDIO 4: DCE **520**

10 1 INTRODUCCIÓN A DCE 520
10 1 1 Historia de DCE 520
10 1 2 Objetivos de DCE 521
10 1 3 Componentes de DCE 522
10 1 4 Celdas 525
10 2 HILOS 527
10 2 1 Introducción a los hilos de DCE 527
10 2 2 Planificación 529
10 2 3 Sincronización 530
10 2 4 Llamadas a hilos 531
10 3 LLAMADA A PROCEDIMIENTOS REMOTOS 535
10 3 1 Objetivos de la RPC de DCE 535
10 3 2 Escritura a un cliente y un servidor 536
10 3 3 Conexión de un cliente con un servidor 538
10 3 4 Realización de una RPC 539
10 4 SERVICIO DE TIEMPO 540
10 4 1 Modelo de tiempo DTS 541
10 4 2 Implantación de DTS 543
10 5 SERVICIO DE DIRECTORIOS 545

10.5.1 Nombres 546
10.5.2 El servicio de directorio de celda 547
10.5.3 El servicio de directorio global 549
10.6 SERVICIO DE SEGURIDAD 554
10.6.1 Modelo de seguridad 555
10.6.2 Componentes de seguridad 557
10.6.3 Boletos y autenticadores 558
10.6.4 RPC autenticada 559
10.6.5 ACL 562
10.7 SISTEMA DISTRIBUIDO DE ARCHIVOS 564
10.7.1 Interfaz DFS 565
10.7.2 Componentes DFS en el nucleo servidor 566
10.7.3 Componentes DFS en el nucleo cliente 569
10.7.4 Componentes DFS en el espacio del usuario 571
10.8 RESUMEN 573

11 LISTA DE LECTURAS Y BIBLIOGRAFÍA 577

11.1 SUGERENCIAS PARA LECTURA POSTERIOR 577
11.2 BIBLIOGRAFÍA EN ORDEN ALFABÉTICO 584

ÍNDICE 605

Prefacio

Con la publicación de *Sistemas operativos distribuidos* he completado mi trilogía relativa a los sistemas operativos Los tres volúmenes de esta trilogía son

Sistemas operativos Diseño e implantación
Sistemas operativos distribuidos
Sistemas operativos modernos

Sin embargo los tres volumenes no son por completo independientes Para las escuelas que tienen una serie de dos cursos de sistemas operativos (o un curso de licenciatura y otro de posgrado) una opción sería utilizar *Sistemas operativos Diseño e implantación* en el primer curso y *Sistemas operativos distribuidos* en el segundo

El primer libro abarca los principios usuales de los sistemas con un solo procesador incluyendo los procesos la sincronización la entrada/salida los bloqueos la administración de la memoria los sistemas de archivos la seguridad etc También ilustra estos principios con gran detalle mediante el uso de MINIX un clon de UNIX cuyo listado fuente aparece en un apéndice

El segundo libro (éste) abarca con detalle los sistemas operativos distribuidos incluyendo la comunicación la sincronización los procesos los sistemas de archivos y la administración de la memoria pero ahora en el contexto de los sistemas distribuidos Se proporcionan cuatro ejemplos detallados de sistemas distribuidos Amoeba Mach Chorus y DCE Amoeba está disponible en forma gratuita para universidades y usos educativos

Se puede ejecutar en los procesadores Intel 386/486 SPARC y Sun 3 Para mayor información acerca de la forma de obtener Amoeba por favor realice un FTP del archivo *amoeba/Intro ps Z* de *ftp cs vu nl* o comuníquese conmigo mediante correo electrónico en *ast@cs vu nl* Los usuarios potenciales deben advertir que Amoeba es más complejo que MINIX la sola documentación (disponible mediante FTP) tiene más de 1 000 páginas y el sistema necesita al menos cinco máquinas grandes y Ethernet para una buena ejecución

Al estudiar estos dos libros y utilizar MINIX y Amoeba los estudiantes tendrán un conocimiento cabal de los principios y prácticas de los sistemas operativos con un solo procesador y distribuidos Ahora que he terminado la trilogía planeo revisar MINIX y el libro que lo describe

Para las universidades o los profesionales de la computación con menos tiempo disponible *Sistemas operativos modernos* puede verse como una versión condensada de los otros dos libros Proporciona una introducción a los principios de los sistemas con un solo procesador y distribuidos pero sin el ejemplo detallado de MINIX También omite muchos de los temas avanzados presentes en este libro como una introducción a ATM los sistemas distribuidos tolerantes de fallas los sistemas distribuidos de tiempo real la memoria compartida distribuida Chorus DCE y otros temas En total cerca de 230 páginas de material relativo a los sistemas distribuidos correspondientes a este libro se han omitido en *Sistemas operativos modernos*

Muchas personas me ayudaron con este libro En particular quisiera agradecer a las siguientes personas por leer partes del manuscrito y darme muchas sugerencias utiles para mejorarlo Irina Athanasiu Henri Bal Saniya Ben Hassen David Black John Carter Randall Dean Wiebren de Jonge John Dugas Dick Grune Anoop Gupta Frans Kaashoek Marcus Koebler Hermann Kopetz Ed Lazowska Dan Lenoski Kai Li Marc Maathuis David Mosberger Douglas Orr Craig Partridge Carlton Pu Marc Rozier Rich Salz Mike Schroeder Karsten Schwan Greg Sharp Dennis Shasha Sol Shatz Jennifer Steiner Chuck Thacker John Tucker Walt Tuvell Leendert van Doorn Robbert van Renesse Kees Verstoep Ellen Zegura Willy Zwaenpoel y los revisores anónimos Mi editor Bill Zobrist toleró mis intentos por hacer todo perfecto y mis gimoteos

A pesar de toda esta ayuda no hay duda de que persisten algunos errores Esto parece inevitable sin importar la cantidad de personas que lean el manuscrito Las personas que deseen informar de errores pueden comunicarse conmigo mediante correo electrónico

Por último quiero agradecer nuevamente a Suzanne Después de ocho libros ella conoce las implicaciones de otro pero su paciencia y amor son infinitos También deseo agradecer a Barbara y Marvin por utilizar sus computadoras y dejar en paz la mía (excepto por la impresora) Al enseñarles a utilizar los programas de procesamiento de texto en PC he podido apreciar *troff* más que nunca Por ultimo agradezco al pequeño Bram el estar tranquilo mientras escribía

Andrew S Tanenbaum

1

Introducción a los sistemas distribuidos

Los sistemas de cómputo están sufriendo una revolución Desde 1945 cuando comenzó la era de la computadora moderna hasta cerca de 1985 las computadoras eran grandes y caras Incluso las minicomputadoras costaban por lo general cientos de miles de dólares cada una Como resultado la mayor parte de las organizaciones tenía tan sólo un puñado de computadoras y por carecer de una forma para conectarlas éstas operaban por lo general en forma independiente entre sí

Sin embargo a partir de la mitad de la década de 1980 dos avances tecnológicos comenzaron a cambiar esta situación El primero fue el desarrollo de poderosos microprocesadores En principio se disponía de máquinas de 8 bits pero pronto se volvieron comunes las CPU de 16 32 e incluso 64 bits Muchos de ellos tenían el poder de cómputo de una computadora mainframe de tamaño respetable (es decir grande) pero por una fracción de su precio

La cantidad de mejoras ocurridas en la tecnología de cómputo en el ultimo medio siglo es de verdad impresionante y sin precedentes en otras industrias Desde la máquina que costaba 10 millones de dólares y ejecutaba una instrucción por segundo hemos llegado a máquinas que cuestan 1 000 dólares y ejecutan 10 millones de instrucciones por segundo una ganancia precio/rendimiento de 10^{11} Si los automóviles hubieran mejorado con esta razón en el mismo periodo un Rolls Royce costaría 10 dólares y daría mil millones de millas por galón (Por desgracia es probable que tuviera un manual de 200 páginas indicando cómo abrir la puerta)

El segundo desarrollo fue la invención de redes de área local de alta velocidad (LAN) Las **redes de área local (LANs[Local area networks])** permiten conectar docenas e incluso cientos de máquinas dentro de un edificio de tal forma que se pueden transferir pequeñas cantidades de información entre ellas en un milisegundo o un tiempo parecido

Las cantidades mayores de datos se pueden desplazar entre las máquinas a razón de 10 a 100 millones de bits/segundo o más Las **redes de área amplia (WANs [wide area networks])** permiten que millones de máquinas en toda la Tierra se conecten con velocidades que varían de 64 Kbps (kilobits por segundo) a gigabits por segundo para ciertas redes experimentales avanzadas

El resultado neto de estas tecnologías es que hoy en día no sólo es posible sino fácil reunir sistemas de cómputo compuestos por un gran número de CPU conectados mediante una red de alta velocidad Estos reciben el nombre genérico de **sistemas distribuidos** en contraste con los **sistemas centralizados** anteriores (o **sistemas con sólo un procesador**) que constan de un CPU su memoria sus periféricos y algunas terminales

Sólo existe una mosca en la sopa el software Los sistemas distribuidos necesitan un software radicalmente distinto al de los sistemas centralizados En particular los sistemas operativos necesarios para estos sistemas distribuidos están apenas en una etapa de surgimiento Se han dado algunos primeros pasos pero todavía existe un largo camino por recorrer Sin embargo ya se sabe bastante de estos sistemas por lo que podemos presentar las ideas básicas El resto del libro se dedica al estudio de los conceptos implantación y ejemplos de los sistemas operativos distribuidos

1 1 ¿QUÉ ES UN SISTEMA DISTRIBUIDO?

Se han dado varias definiciones de sistema distribuido en la bibliografía pero ninguna de ellas es satisfactoria ni está de acuerdo con las demás Para nuestros propósitos es suficiente dar una vaga caracterización

> *Un sistema distribuido es una colección de computadoras independientes que aparecen ante los usuarios del sistema como una unica computadora*

Esta definición tiene dos aspectos El primero se refiere al hardware las máquinas son autónomas El segundo se refiere al software los usuarios piensan que el sistema es como una unica computadora Ambos son esenciales Regresaremos a este punto en una sección posterior de este capítulo después de revisar algunos conceptos básicos del hardware y el software

En vez de continuar con más definiciones tal vez sea más útil dar varios ejemplos de sistemas distribuidos Como primer ejemplo considere una red de estaciones de trabajo en un departamento de una universidad o compañía Además de cada estación de trabajo personal podría existir una pila de procesadores en el cuarto de máquinas que no estén asignados a usuarios específicos sino que se utilicen de manera dinámica conforme sea necesario Tal sistema podría tener un sistema de archivos único con todos los archivos accesibles desde todas las máquinas de la misma forma y con el mismo nombre de ruta de acceso Además cuando el usuario escriba un comando el sistema podría buscar el mejor lugar para ejecutarlo tal vez en la propia estación de trabajo del usuario o en una estación

de trabajo inactiva que pertenezca a otra persona o en uno de los procesadores no asignados en el cuarto de máquinas Si el sistema se ve como un todo y actúa como un sistema de tiempo compartido clásico con un único procesador podría considerarse como un sistema distribuido

Como segundo ejemplo considere una fábrica de robots cada uno de los cuales contiene una poderosa computadora para el manejo de visión planeación comunicación y otras tareas Cuando un robot de la línea de ensamble nota que una parte por instalar es defectuosa le pide al robot del departamento de partes que le traiga una refacción Si todos los robots actuan como dispositivos periféricos unidos a la misma computadora central y el sistema se puede programar de esta manera también se considera como un sistema distribuido

Como último ejemplo piense en un enorme banco con cientos de sucursales por todo el mundo Cada oficina tiene una computadora maestra para guardar las cuentas locales y el manejo de las transacciones locales Además cada computadora tiene la capacidad de comunicarse con las de otras sucursales y con una computadora central en las oficinas centrales Si las transacciones se pueden realizar sin importar dónde se encuentre el cliente o la cuenta y si los usuarios no observan diferencia alguna entre este sistema y el antiguo centralizado que ha remplazado también se podría considerar como un sis tema distribuido

1 2 OBJETIVOS

El simple hecho de poder construir sistemas distribuidos no significa necesariamente que sean buena idea Después de todo con la tecnología actual es posible colocar 4 unidades de disco flexible en una computadora personal El hecho es que esto no tendría sentido En esta sección analizaremos la motivación y los objetivos de los sistemas distribuidos típicos y revisaremos sus ventajas y desventajas en comparación con los sistemas centralizados tradicionales

1 2 1 Ventajas de los sistemas distribuidos con respecto de los centralizados

La fuerza motriz real detrás de la tendencia hacia la descentralización es la economía Hace un cuarto de siglo una persona experta en computadoras Herb Grosch enunció lo que se conocería después como la ley de Grosch el poder de cómputo de CPU es propor cional al cuadrado de su precio Si se paga el doble se obtiene cuatro veces el desempeño Esta observación encajó bien en la tecnología mainframe de su tiempo y provocó que muchas organizaciones compraran una sola máquina la más grande que pudieran conse guir

Con la tecnología del microprocesador la ley de Grosch ya no es válida Por unos cuantos cientos de dólares es posible comprar un microcircuito de CPU que puede ejecutar más instrucciones por segundo de las que realizaba una de las más grandes mainframes de la década de 1980 Si uno está dispuesto a pagar el doble se obtiene el mismo CPU sólo

que con una velocidad un poco mayor Como resultado la solución más eficaz en cuanto a costo es limitarse a un gran numero de CPU baratos reunidos en un mismo sistema Así la razón numero uno de la tendencia hacia los sistemas distribuidos es que estos sistemas tienen en potencia una proporción precio/desempeño mucho mejor que la de un sistema centralizado En efecto un sistema distribuido da mejor en el clavo

Una ligera variación con respecto a este tema es la observación de que una colección de microprocesadores no sólo proporciona mejor precio/rendimiento que un mainframe sino que puede producir mejor rendimiento del que podría dar cualquier mainframe a cual quier precio Por ejemplo con la tecnología actual es posible construir un sistema a partir de 10 000 microcircuitos de un CPU moderno cada uno de los cuales tiene una ejecución de 50 MIPS (millones de instrucciones por segundo) para un rendimiento total de 500 000 MIPS Para que un unico procesador (es decir CPU) logre esto tendría que ejecutar una instrucción en 0 002 nanosegundos (2 picosegundos) Ninguna máquina existente llega a acercarse a esta cifra además de que consideraciones teóricas y de ingeniería consideran improbable que alguna máquina lo logre Teóricamente la teoría de la relatividad de Einstein establece que nada puede viajar más rápido que la luz que sólo cubre una distancia de 0 6 mm en 2 picosegundos Desde el punto de vista práctico una computadora de esa velocidad contenida en totalidad en un cubo de 0 6 mm generaría un calor tal que se fundiría Así si el objetivo es un rendimiento normal a bajo costo o un alto rendimiento con mayor costo los sistemas dis tribuidos tienen mucho que ofrecer

Sea de paso algunos autores distinguen entre los *sistemas distribuidos* diseñados para que muchos usuarios trabajen en forma conjunta y los *sistemas paralelos* cuya meta es lograr la máxima rapidez en un problema como lo haría nuestra máquina de 500 000 MIPS Creemos que esta distinción es difícil de sostener puesto que el espectro del diseño es en realidad un continuo Preferimos utilizar el término sistemas distribuidos en el sentido amplio para denotar cualquier sistema en el que varios CPU conectados entre sí trabajan de manera conjunta

Otra razón para la construcción de un sistema distribuido es que ciertas aplicaciones son distribuidas en forma inherente Una cadena de supermercados podría tener muchas tiendas las cuales reciben los artículos de manera local (tal vez de las granjas locales) realiza ventas locales y toma decisiones locales acerca de las verduras que están viejas o podridas y que deben desecharse Por lo tanto tiene sentido mantener un inventario en cada tienda dentro de una computadora local en vez de tenerlo de manera central en las oficinas de la compañía Después de todo la mayoría de las solicitudes y actualizaciones se harían de forma local Sin embargo de vez en cuando la administración central podría tratar de determinar la cantidad de nabos que posee en cierto momento Una forma de lograr este objetivo es hacer que todo el sistema se vea como una computadora para los programas de aplicación pero implantado de manera descentralizada con una computadora por tienda como ya hemos descrito Éste sería entonces un sistema distribuido comercial

Otro sistema inherente distribuido es lo que se denomina con frecuencia un **trabajo cooperativo apoyado por computadora** en donde un grupo de personas localizados a cierta distancia entre sí trabajan juntos por ejemplo para producir un informe conjunto

Dadas las tendencias a largo plazo en la industria de la computación uno puede imaginar con facilidad una nueva área los **juegos cooperativos apoyados por computadora** en donde los jugadores de diversos lugares juegan entre ellos en tiempo real Uno puede imaginar a las personas escondiéndose y buscándose de forma electrónica en un enorme laberinto multidimensional e incluso combates electrónicos en los que cada jugador utilice un simulador de vuelo local para intentar derribar a los demás jugadores de modo que la pantalla de cada jugador muestre una vista del plano del jugador incluyendo los demás aviones que vuelen dentro de su alcance visual

Otra ventaja potencial de un sistema distribuido sobre uno centralizado es una mayor confiabilidad Al distribuir la carga de trabajo en muchas máquinas la falla de un circuito descompondrá a lo más a una máquina y el resto seguirá intacto En forma ideal si el 5% de las máquinas están descompuestas en cierto momento el sistema podría continuar su trabajo con una pérdida de 5 / del rendimiento Para el caso de aplicaciones críticas como el control de los reactores nucleares o la aviación el uso de un sistema distribuido para lograr mayor confiabilidad puede ser el factor dominante

Por ultimo el crecimiento por incrementos también es una ventaja potencial Con frecuencia ocurre que una compañía compra un mainframe con la intención de hacer todo su trabajo en él Si la compañía prospera y la carga de trabajo aumenta el mainframe no será adecuado en cierto momento Las únicas soluciones posibles son el reemplazo del mainframe con otro más grande (si existe) o añadir un segundo mainframe Ambas ideas pueden representar un duro golpe a las operaciones de la compañía Por el con trario con un sistema distribuido podrían añadirse sólo más procesadores al sistema lo que permite un desarrollo gradual conforme surjan las necesidades Estas ventajas se resumen en la figura 1 1

El m t	D ip ió
E mí	L mi p d f m j p p ió p i / dimi t q l m i f m
V l id d	U i t m di t ib id p d t m y p d d ómp t q m i f m
Di t ib ió i h t	Alg pli i tili máq i q tá p d i t di t i
C fi bilid d	Si máq i d mp l i t m p d b i i m t d
C imi t p i m t	S p d ñ di p d d ómp t p q ñ i m t

Fig a 1 1 V tajas de l i temas dist ib id obre l i te as centrali ados

A largo plazo la principal fuerza motriz será la existencia de un gran número de computadoras personales y la necesidad de que las personas trabajen juntas y compartan información de manera conveniente sin preocuparse por la geografía o la distribución física de las personas los datos o las máquinas

1 2 2 Ventajas de los sistemas distribuidos con respecto de las PC independientes

Puesto que los microprocesadores constituyen una forma económica de trabajo ¿por qué no se ofrece a cada persona su propia PC y se le deja trabajar de manera independiente? El asunto aquí es que muchos usuarios necesitan compartir ciertos datos Por ejemplo los empleados de reservaciones en las líneas aéreas necesitan tener acceso a la base de datos maestra de los vuelos y reservaciones existentes Si se le diera a cada empleado una copia particular de toda la base de datos eso no funcionaría puesto que nadie conocería los asientos vendidos por los demás empleados Los datos compartidos son absolutamente esenciales para ésta y otras aplicaciones de modo que las máquinas deben estar conectadas entre sí La conexión de las máquinas conduce a un sistema distribuido

Los datos no son los únicos elementos que se pueden compartir Otros candidatos son también los periféricos caros como las impresoras láser de color equipos de fotocomposición y los dispositivos de almacenamiento masivo (por ejemplo las cajas ópticas)

Una tercera razón para la conexión de un grupo de computadoras aisladas en un sistema distribuido es lograr una mejor comunicación entre las personas Para mucha gente el correo electrónico tiene numerosos atractivos con respecto del correo con cartas el teléfono o el FAX Es mucho más rápido que el correo con cartas no requiere que ambas partes estén disponibles al mismo tiempo como el teléfono y a diferencia del FAX produce documentos que se pueden editar reordenar almacenar en la computadora y manejar mediante programas para procesamiento de texto

Por último un sistema distribuido tiene mayor flexibilidad potencial que el hecho de darle a cada usuario una computadora personal aislada Aunque un modelo consiste en darle a cada persona una computadora personal y conectarlas mediante LAN ésta no es la unica posibilidad Otra es tener una mezcla de computadoras personales y compartidas tal vez con distintos tamaños y dejar que los trabajos se ejecuten de la forma más adecuada en vez de ejecutarlos siempre en la computadora del propietario De esta manera la carga de trabajo se puede difundir entre las computadoras de forma más eficaz y la pérdida de unas cuantas máquinas se puede compensar si se permite a las personas que ejecuten sus trabajos en otra parte La figura 1 2 resume estos puntos

1 2 3 Desventajas de los sistemas distribuidos

Aunque los sistemas distribuidos tienen sus aspectos fuertes también tienen sus debilidades En esta sección señalaremos algunas de ellas Ya hemos señalado el peor de los problemas el software Con el actual estado de las cosas no tenemos mucha experiencia en el diseño implantación y uso del software distribuido ¿Qué tipo de sistemas operativos lenguajes de programación y aplicaciones son adecuados para estos sistemas? ¿Cuánto deben

El m t	D ip ió
D t mp rtid	P mit q i i t g b d d t mú
Di p iti mp rtid	P mit q i i mp rt p ifé i m l imp l
C m i ió	F ilit l m i ió d p p ; p j mpl m di t l l t ó i
Fl ibilid d	Dif de l g d t b j t l máq i di p ibl l f m má fi t l t

Figura 1 2 Ventajas d los sist mas dist ib idos sob las comp t do as aisladas (pe so l s)

saber los usuarios de la distribución? ¿Qué tanto debe hacer el sistema y qué tanto deben hacer los usuarios? Los expertos tienen sus diferencias (no es que esto sea poco usual entre los expertos pero cuando se entra al tema de los sistemas distribuidos pocas veces se ponen de acuerdo) Mientras se realice más investigación este problema disminuirá pero por el momento no puede subestimarse

Un segundo problema potencial es el debido a las redes de comunicación Éstas pueden perder mensajes lo cual requiere un software especial para su manejo y puede verse so brecargado Al saturarse la red ésta debe remplazarse o añadir una segunda En ambos casos hay que tender cables en una parte de uno o más edificios con gran costo o bien hay que remplazar las tarjetas de interfaz de la red (por ejemplo por fibras ópticas) Una vez que el sistema llega a depender de la red la pérdida o saturación de ésta puede negar algunas de las ventajas que el sistema distribuido debía conseguir

Por último el hecho ya descrito de que los datos sean fácil de compartir es una ventaja pero se puede convertir en un arma de dos filos Si las personas pueden tener acceso a los datos en todo el sistema entonces también pueden tener acceso a datos con los que no tienen nada que ver En otras palabras la seguridad es con frecuencia un problema Para que los datos se mantengan en secreto a toda costa es preferible tener una computadora personal aislada sin conexiones de red con las demás máquinas y mantenerla en un cuarto cerrado con un mueble seguro donde guardar todos los discos flexibles Las desventajas de los sistemas distribuidos se resumen en la figura 1 3

A pesar de estos problemas potenciales muchas personas sienten que las ventajas tienen mayor peso que las desventajas y se espera que los sistemas distribuidos tengan cada vez mayor importancia en los años venideros De hecho es probable que en unos cuantos años gran parte de las organizaciones conecten la mayoría de sus computadoras a extensos sistemas distribuidos para proporcionar un servicio mejor más barato y conveniente a sus usuarios Es probable que una computadora aislada en una empresa de tamaño medio o grande o alguna otra organización ya no exista dentro de diez años

El m t	D ip ió
S ftw	E i t p ftw p l i t m di t ib id l t lid d
R d	L d p d t t p bl m
S g id d	U ill t mbié pli d t t

Fig a 1 3 D taj d l i t distrib id

1 3 CONCEPTOS DE HARDWARE

Aunque todos los sistemas distribuidos constan de varios CPU existen diversas formas de organizar el hardware en particular en la forma de interconectarlos y comunicarse entre sí En esta sección analizaremos de manera breve el hardware de los sistemas distribuidos en particular la forma en que se conectan entre sí las máquinas En la siguiente sección exami naremos algunos de los aspectos del software relacionados con los sistemas distribuidos

Con el paso de los años se han propuesto diversos esquemas de clasificación para los sistemas de cómputo con varios CPU pero ninguno de ellos ha tenido un éxito completo ni se ha adoptado de manera amplia Es probable que la taxonomía más citada sea la de Flynn (1972) aunque es algo rudimentaria Flynn eligió dos características consideradas por él como esenciales el número de flujos de instrucciones y el numero de flujos de datos Una computadora con un flujo de instrucciones y uno de datos se llama SISD (Single Instruction Single Data) Todas las computadoras tradicionales de un procesador (es decir aquellas que tienen un CPU) caen dentro de esta categoría desde las computadoras perso nales hasta las grandes mainframes

La siguiente categoría es SIMD (Single Instruction Multiple Data) con un flujo de instrucciones y varios flujos de datos Este tipo se refiere a ordenar procesadores con unidad de instrucción que busca una instrucción y después instruye a varias unidades de datos para que la lleven a cabo en paralelo cada una con sus propios datos Estas máquinas son utiles para los cómputos que repiten los mismos cálculos en varios conjuntos de datos por ejemplo sumando todos los elementos de 64 vectores independientes Ciertas supercomputadoras son SIMD

La siguiente categoría es MISD (Multiple Instruction Single Data) con un flujo de varias instrucciones y un flujo de datos Ninguna de las computadoras conocidas se ajusta a este modelo Por último está MIMD (Multiple Instruction Multiple Data) que significa un grupo de computadoras independientes cada una con su propio contador del programa y datos Todos los sistemas distribuidos son MIMD por lo que este sistema de clasificación no es muy útil para nuestros fines

Aunque Flynn se detuvo en este punto nosotros avanzaremos un poco más En la figura 1 4 dividimos todas las computadoras MIMD en dos grupos aquellas que tienen memoria compartida que por lo general se llaman **multiprocesadores** y aquellas que no que a veces reciben el nombre de **multicomputadoras** La diferencia esencial es ésta en un multipro

cesador existe un espacio de direcciones virtuales compartido por todos los CPU Por ejemplo si algún CPU escribe el valor 44 en la dirección 1000 cualquier otro CPU que haga una lectura posterior de *su* dirección 1 000 obtendıá el valor 44 Todas las máquinas comparten la misma memoıia

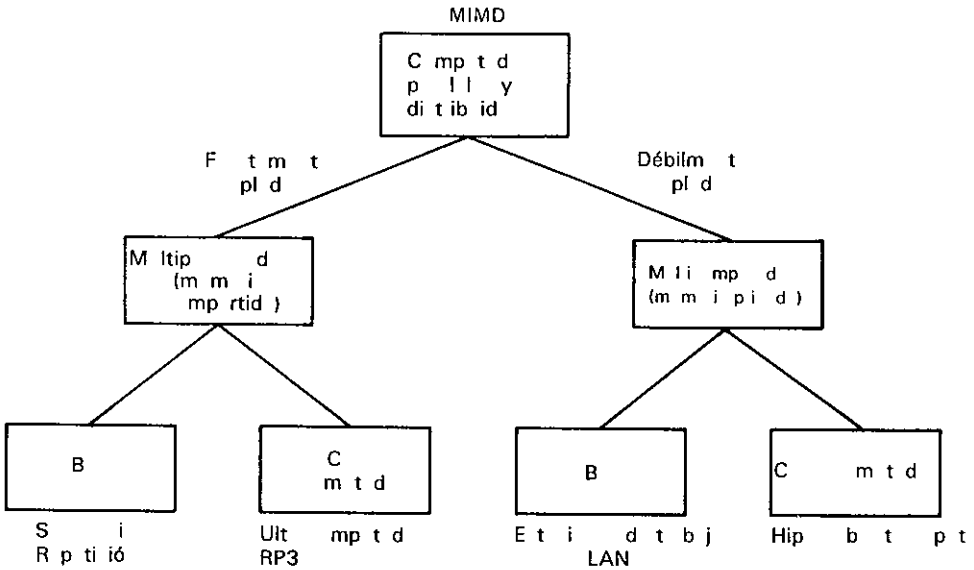


Fig a 1 4 U ta o mí de l i t nas d ó p to p l l y dist ib id

En contraste en una multicomputadora cada máquina tiene su propia memoria Si un CPU escribe el valoı 44 en la dirección 1 000 y otro CPU lee su dirección 1 000 obtendrá el valor que se encontıaba ahí antes La escritura de 44 no afecta *su* memoria de manera alguna Un ejemplo común de multicomputadora es una colección de computadoras per sonales conectadas mediante una ıed

Cada una de estas categorías se puede subdividir con base en la arquitectura de la red de interconexión En la figura 1 4 describimos esas dos categorías como **bus** y con **con mutador** En la pıimera queremos indicar que existe una red plano de base bus cable u otro medio que conecta todas las máquinas La televisión comercial por cable utiliza un esquema como éste la compañía tiende un cable en la calle y todos los suscıiptores tienen una conexión hasta sus televisoıes

Los sistemas con conmutador no tienen sólo una columna vertebıal como en la tele visión por cable sino que tienen cables individuales de una máquina a otra y utilizan varios patrones diferentes de cableado Los mensajes se mueven a tıavés de los cables y se toma una decisión explícita de conmutación en cada etapa para dirigir el mensaje a lo largo de uno de los cables de salida El sistema mundial de teléfonos públicos está organizado de esta manera

Otra dimensión de nuestra taxonomía es que en ciertos sistemas las máquinas están **fuertemente acopladas** y en otras están **débilmente acopladas** En un sistema fuertemente acoplado el retraso que se experimenta al enviar un mensaje de una computadora a otra es corto y la tasa de trasmisión de los datos es decir el numero de bits por segundo que se pueden transferir es alta En un sistema débilmente acoplado ocurre lo contrario el retraso de los mensajes entre las máquinas es grande y la tasa de transmisión de los datos es baja Por ejemplo es probable que dos circuitos de CPU de la misma tarjeta de circuito impreso conectados mediante cables insertados en la tarjeta estén fuertemente acoplados mientras que dos computadoras conectadas mediante un módem de 2400 bits/seg a través del sistema telefónico están débilmente acopladas

Los sistemas fuertemente acoplados tienden a utilizarse más como sistemas paralelos (para trabajar con un problema) y los débilmente acoplados tienden a utilizarse como sistemas dis tribuidos (para trabajar con varios problemas no relacionados entre sí) aunque esto no siempre es cierto Un contraejemplo famoso es un proyecto en el que cientos de computadoras en todo el mundo trabajaron en forma conjunta para factorizar un enorme número de cerca de 100 dígitos A cada computadora se le asignó un límite distinto de divisores para su análisis y todas ellas trabajaron en el problema durante su tiempo correspondiente y reportaban sus resultados mediante el correo electrónico

En general los multiprocesadores tienden a estar más fuertemente acoplados que las multicomputadoras puesto que pueden intercambiar datos a la velocidad de sus memorias pero algunas multicomputadoras basadas en fibras ópticas pueden funcionar también con velocidad de memoria A pesar de lo vago de los términos débilmente acoplados y fuer temente acoplados son conceptos utiles de la misma forma que decir Jack es gordo y Jill es flaca proporciona información aunque uno podría discutir con amplitud los con ceptos de gordo y flaco

En las siguientes cuatro secciones analizaremos las cuatro categorías de la figura 1 4 con mayor detalle los multiprocesadores de bus multiprocesadores con conmutador mul ticomputadoras de bus y multicomputadoras con conmutador Aunque estos temas no tienen relación directa con nuestro interés principal los sistemas operativos distribuidos arrojan cierta luz sobre el tema ya que como veremos distintas categorías de máquinas utilizan diversos tipos de sistemas operativos

1 3 1 Multiprocesadores con base en buses

Los multiprocesadores con base en buses constan de cierta cantidad de CPU conectados a un bus común junto con un módulo de memoria Una configuración sencilla consta de un plano de base (backplane) de alta velocidad o tarjeta madre en el cual se pueden insertar las tarjetas de memoria y el CPU Un bus típico tiene 32 o 64 líneas de direcciones 32 o 64 líneas de datos y 32 o más líneas de control todo lo cual opera en paralelo Para leer una palabra de memoria un CPU coloca la dirección de la palabra deseada en las líneas de direcciones del bus y coloca una señal en las líneas de control adecuadas para indicar que desea leer La memoria responde y coloca el valor de la palabra en las líneas de datos para permitir la lectura de ésta por parte del CPU solicitante La escritura funciona de manera similar

Puesto que sólo existe una memoria si el CPU *A* escribe una palabra en la memoria y después el CPU *B* lee esa palabra un microsegundo después *B* obtendrá el valor recién escrito Una memoria con esta propiedad es **coherente** La coherencia juega un papel importante en los sistemas operativos distribuidos en una variedad de formas que estudiaremos más adelante

El problema con este esquema es que si sólo se dispone de 4 o 5 CPU el bus estará por lo general sobrecargado y el rendimiento disminuirá en forma drástica La solución es añadir una **memoria caché** de alta velocidad entre el CPU y el bus como se muestra en la figura 1 5 El caché guarda las palabras de acceso reciente Todas las solicitudes de la memoria pasan a través del caché Si la palabra solicitada se encuentra en el caché éste responde al CPU y no se hace solicitud alguna al bus Si el caché es lo bastante grande la probabilidad de éxito (la **tasa de encuentros**) será alta y la cantidad de tráfico en el bus por cada CPU disminuirá en forma drástica lo que permite un número mayor de CPU en el sistema Los tamaños comunes del caché van desde los 64K hasta 1M lo que da como resultado una tasa de encuentros del 90 / o más

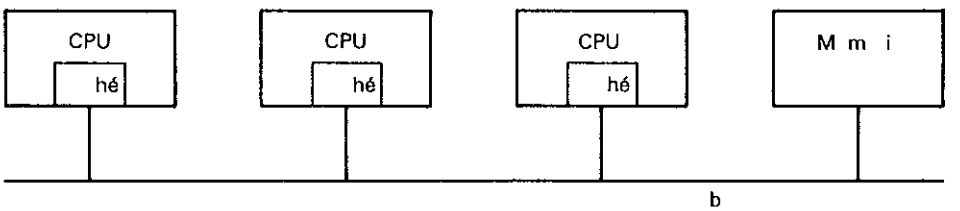


Fig 1 5 Multiprocesador b b

Sin embargo el uso de cachés también acarrea un serio problema Supongamos que dos CPU *A* y *B* leen la misma palabra en sus respectivos cachés Después *A* escribe sobre la palabra Cuando *B* lee esa palabra obtiene el valor anterior en su caché y no el valor recién escrito por *A* La memoria es entonces incoherente y el sistema es difícil de programar

Muchos investigadores han estudiado este problema y se conocen varias soluciones de las cuales esbozaremos una de ellas Supongamos que las memorias caché estén diseñadas de tal forma que cuando una palabra sea escrita al caché también sea escrita a la memoria Tal caché recibe el nombre poco sorprendente de **caché de escritura** En este diseño el uso del caché para la lectura no provoca un tráfico en el bus pero al no utilizar el caché para una lectura así como todas las escrituras sí provocan un tráfico en el bus

Además todos los cachés realizan un monitoreo constante del bus Cada vez que un caché observa una escritura a una dirección de memoria presente en él puede eliminar ese dato o actualizarlo con el nuevo valor Tal caché recibe el nombre de **caché monitor** puesto que siempre realiza un monitoreo en el bus Un diseño consistente en cachés monitores y de escritura es coherente e invisible para el programador Casi todos los multiprocesadores basados en buses utilizan esta arquitectura u otra muy relacionada con ésta

Mediante su uso, es posible colocar de 32 hasta 64 CPU en el mismo bus. Para mayor información acerca de los multiprocesadores basados en buses, véase Lilja (1993).

1.3.2. Multiprocesadores con conmutador

Para construir un multiprocesador con más de 64 procesadores, es necesario un método distinto para conectar cada CPU con la memoria. Una posibilidad es dividir la memoria en módulos y conectarlos a las CPU con un **conmutador de cruceta**, como se muestra en la figura 1-6(a). Cada CPU y cada memoria tiene una conexión que sale de él, como se muestra en la figura. En cada intersección está un delgado **conmutador de punto de cruce** electrónico que el hardware puede abrir y cerrar. Cuando un CPU desea tener acceso a una memoria particular, el conmutador del punto de cruce que los conecta se cierra de manera momentánea, para permitir dicho acceso. La virtud del conmutador de cruceta es que muchos CPU pueden tener acceso a la memoria al mismo tiempo, aunque si dos CPU intentan tener acceso a la misma memoria en forma simultánea, uno de ellos deberá esperar.

La desventaja del conmutador de cruceta es que con n CPU y n memorias, se necesitan n^2 conmutadores en los puntos de cruce. Si n es grande, este número puede ser prohibido. Como

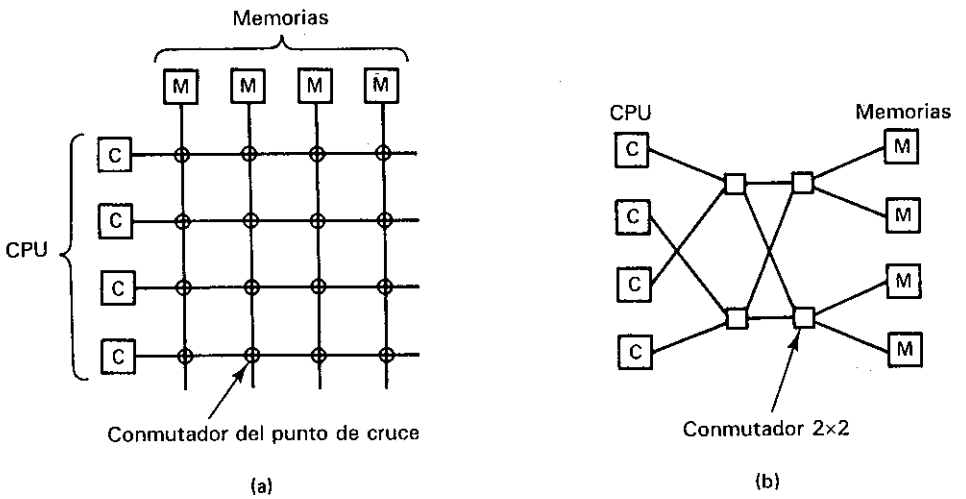


Figura 1-6. (a) Un conmutador de cruceta. (b) Una red omega de conmutación.

resultado, las personas han buscado y encontrado otras redes de conmutación que necesiten menos conmutadores. La **red omega** de la figura 1-6(b) es un ejemplo. Esta red contiene conmutadores 2 × 2, cada uno de los cuales tiene dos entradas y dos salidas. Cada conmutador puede dirigir cualquiera de las entradas en cualquiera de las salidas. Un análisis cuidadoso de la figura mostrará que si se eligen los estados adecuados de los conmutadores, cada CPU podrá tener acceso a cada memoria. Estos conmutadores se pueden activar en cuestión de nanosegundos.

En el caso general, con n CPU y n memorias, la red omega necesita $\log_2 n$ etapas de conmutación, cada una de las cuales tiene $n/2$ conmutadores, para un total de $(n \log_2 n)/2$ conmutadores. Aunque este número es mejor que n^2, sigue siendo considerable.

Además, existe otro problema: el retraso. Por ejemplo, si $n = 1024$, existen 10 etapas de conmutación del CPU a la memoria y otras 10 para que la palabra solicitada regrese. Supongamos que el CPU es un moderno circuito RISC de 100 MIPS; es decir, que el tiempo de ejecución de una instrucción es de 10 nseg. Si una solicitud de la memoria debe recorrer un total de 20 etapas de conmutación (10 de ida y 10 de regreso) en 10 nseg, el tiempo de conmutación debe ser de 500 picosegundos (0.5 nseg). Todo el multiprocesador necesitará 5 120 conmutadores de 500 picosegundos. Esto no será barato.

Se ha intentado reducir el costo mediante los sistemas jerárquicos. Cada CPU tiene asociada cierta memoria. Cada CPU puede tener un rápido acceso a su propia memoria local, pero será más lento el acceso a la memoria de los demás. Este diseño da lugar a la llamada máquina **NUMA** (Non Uniform Memory Access [**Acceso no uniforme a la memoria**]). Aunque las máquinas NUMA tienen mejor tiempo promedio de acceso que las máquinas basadas en redes omega, tienen una nueva complicación: la colocación de los programas y los datos se convierten en un factor crítico, para lograr que la mayoría de los accesos sean hacia la memoria local.

En resumen, los multiprocesadores basados en buses, incluso con cachés monitores, quedan limitados a lo más a 64 CPUs por la capacidad del bus. Para rebasar estos límites, es necesaria una red con conmutador, como uno de cruceta, una red omega o algo similar. Los grandes conmutadores de cruceta y las grandes redes omega son muy caros y lentos. Las máquinas NUMA necesitan complejos algoritmos para la buena colocación del software. La conclusión es clara: la construcción de un multiprocesador grande, fuertemente acoplado y con memoria compartida es difícil y cara.

1.3.3. Multicomputadoras con base en buses

Por otro lado, la construcción de una multicomputadora (es decir, sin memoria compartida) es fácil. Cada CPU tiene conexión directa con su propia memoria local. El único problema restante es la forma en que los CPU se comunicarán entre sí. Es claro que aquí también se necesita cierto esquema de interconexión, pero como sólo es para la comunicación entre un CPU y otro, el volumen del tráfico será de varios órdenes menor en relación con el uso de una red de interconexión para el tráfico CPU-memoria.

En la figura 1-7 vemos una multicomputadora con base en un bus. Es similar, desde el punto de vista topológico, al multiprocesador basado en un bus, pero como tendrá menor tráfico, no necesita ser un bus con un plano de base de alta velocidad. De hecho, puede ser una LAN de menor velocidad (por lo general de 10-100 Mb/seg, en comparación con 300 Mb/seg o más para un bus con un plano de base). Así, la figura 1-7 es más a menudo una colección de estaciones de trabajo en una LAN que una colección de tarjetas de CPU que se insertan en un bus rápido (aunque esto último definitivamente es un diseño posible).

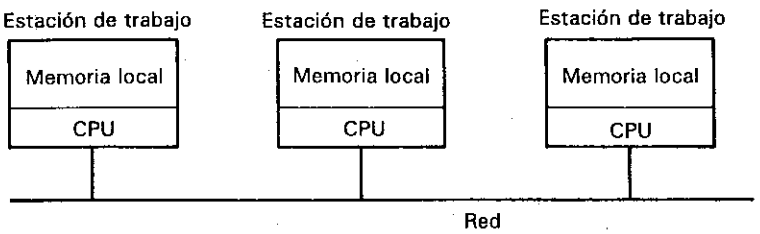


Figura 1-7. Una multicomputadora que consta de estaciones de trabajo en una LAN.

1.3.4. Multicomputadoras con conmutador

Nuestra última categoría es la de las multicomputadoras con conmutador. Se han propuesto y construido varias redes de interconexión, pero todas tienen la propiedad de que cada CPU tiene acceso directo y exclusivo a su propia memoria particular. La figura 1-8 muestra dos topologías populares, una retícula y un hipercubo. Las retículas son fáciles de comprender y se basan en las tarjetas de circuitos impresos. Se adecuan mejor a problemas con naturaleza bidimensional inherente, como la teoría de gráficas o la visión (por ejemplo, los ojos de un robot o el análisis de fotografías).

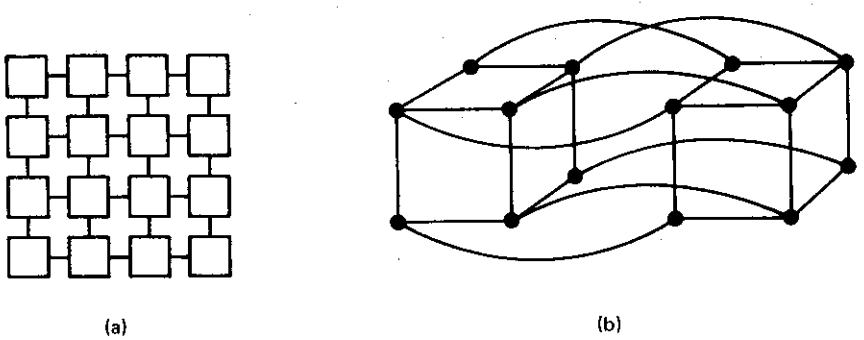


Figura 1-8. (a) Retícula. (b) Hipercubo.

Un **hipercubo** es un cubo n-dimensional. El hipercubo de la figura 1-8(b) es de dimensión 4. Se puede pensar como dos cubos ordinarios, cada uno de los cuales cuenta con 8 vértices y 12 aristas. Cada vértice es un CPU. Cada arista es una conexión entre dos CPU. Se conectan los vértices correspondientes de cada uno de los cubos.

Para extender el cubo a 5 dimensiones, podríamos añadir a la figura otro conjunto de dos cubos conectados entre sí y conectar las aristas correspondientes en las dos mitades, y así en lo sucesivo. Para el caso de un hipercubo n-dimensional, cada CPU tiene n conexiones con otras CPU. Así, la complejidad del cableado aumenta en proporción logarítmica con

el tamaño. Puesto que sólo se conectan los vecinos más cercanos, muchos mensajes deben realizar varios saltos antes de llegar a su destino. Sin embargo, la trayectoria de mayor longitud también crece en forma logarítmica junto con el tamaño, en contraste con la retícula, donde ésta crece conforme la raíz cuadrada del número de CPU. Los hipercubos con 1 024 CPU están disponibles en el mercado desde hace varios años y ya comienzan a estar disponibles los hipercubos con hasta 16 384 CPU.

1.4. CONCEPTOS DE SOFTWARE

Aunque el hardware es importante, el software lo es más. La imagen que presenta y la forma de pensar de los usuarios de un sistema, queda determinada en gran medida por el software del sistema operativo, no por el hardware. En esta sección daremos una introducción a los distintos tipos de sistemas operativos para los multiprocesadores y multicomputadoras que hemos estudiado y analizaremos qué tipo de software va de acuerdo con el hardware.

Los sistemas operativos no se pueden clasificar tan fácil como el hardware. Por su propia naturaleza, el software es vago y amorfo. Aun así, es más o menos posible distinguir dos tipos de sistemas operativos para los de varios CPU: los débilmente acoplados y los fuertemente acoplados. Como veremos, el software débil o fuertemente acoplado es un tanto análogo al hardware débil o fuertemente acoplado.

El software débilmente acoplado permite que las máquinas y los usuarios de un sistema distribuido sean independientes entre sí en lo fundamental, pero que interactúen en cierto grado cuando sea necesario. Consideremos un grupo de computadoras personales, cada una de las cuales tiene su propio CPU, su propia memoria, su propio disco duro y su propio sistema operativo, pero que comparten ciertos recursos, como las impresoras láser y las bases de datos en una LAN. Este sistema está débilmente acoplado, puesto que las máquinas individuales se distinguen con claridad, cada una de las cuales tiene su propio trabajo por realizar. Si la red falla por alguna razón, las máquinas individuales continúan su ejecución en cierto grado considerable, aunque se puede perder cierta funcionalidad (por ejemplo, la capacidad de imprimir archivos).

Para mostrar lo difícil que resulta establecer definiciones en esta área, consideremos ahora el mismo sistema anterior, pero sin la red. Para imprimir un archivo, el usuario escribe un archivo en un disco flexible, lo lleva hasta la máquina que tiene la impresora, lo lee en ella y después lo imprime. ¿Es todavía un sistema distribuido, sólo que ahora más débilmente acoplado? Esto es difícil de decir. Desde un punto de vista fundamental, no existe una diferencia real entre la comunicación a través de una LAN y la comunicación mediante el traslado físico de los discos flexibles. Lo más que se puede decir es que las tasas de retraso y transmisión de los datos son peores en el segundo ejemplo.

En el otro extremo, podríamos tener el caso de un multiprocesador dedicado a la ejecución de un programa de ajedrez en paralelo. A cada CPU se le asigna un tablero para su evaluación y éste ocupa su tiempo en la evaluación de este tablero y los que se pueden generar a partir de él. Al terminar la evaluación, el CPU informa de sus resultados y se le proporciona un nuevo tablero para trabajar con él. El software para este sistema, es decir,

el programa de aplicación y el sistema operativo necesario para soportarlo, están mejor acoplados que el ejemplo anterior.

Hemos visto entonces cuatro tipos de hardware distribuido y dos de software. En teoría, deberían existir ocho combinaciones de hardware y software. De hecho, sólo existen cuatro, puesto que para el usuario, la interconexión de la tecnología no es visible. Para la mayor parte de los propósitos, un multiprocesador lo es, sin importar si utiliza un bus con cachés monitores o una red omega. En las secciones siguientes analizaremos algunas de las combinaciones más comunes de hardware y software.

1.4.1. Sistemas operativos de redes

Comenzaremos con el software débilmente acoplado en hardware débilmente acoplado, puesto que tal vez ésta sea la combinación más común en muchas organizaciones. Un ejemplo típico es una red de estaciones de trabajo de ingeniería conectadas mediante una LAN. En este modelo, cada usuario tiene una estación de trabajo para su uso exclusivo. Puede o no tener un disco duro. En definitiva, tiene su propio sistema operativo. Lo normal es que todos los comandos se ejecuten en forma local, justo en la estación de trabajo.

Sin embargo, a veces es posible que un usuario se conecte de manera remota con otra estación de trabajo mediante un comando como

```
rlogin machine
```

El efecto de este comando es convertir la propia estación de trabajo del usuario en una terminal enlazada con la máquina remota. Los comandos escritos en el teclado se envían a la máquina remota y la salida de la máquina remota se exhibe en la pantalla. Para pasar a otra máquina remota, primero es necesario desconectarse de la primera, y utilizar después el comando *rlogin* para conectarse a la otra. En cualquier instante, sólo se puede utilizar una máquina y la selección de ésta se realiza de forma manual.

Las redes de estaciones de trabajo también tienen un comando de copiado remoto para copiar archivos de una máquina a otra. Por ejemplo, un comando como

```
rcp machine1:file1 machine2:file2
```

copiaría el archivo *file1* de *machine1* a *machine2*, con el nombre de *file2*. De nuevo, el movimiento de los archivos es explícito y se requiere que el usuario esté por completo consciente de la posición de todos los archivos y el sitio donde se ejecutan todos los comandos.

Aunque es mejor que nada, esta forma de comunicación es primitiva en extremo y ha provocado que los diseñadores de sistemas busquen formas más convenientes de comunicación y distribución de la información. Un método consiste en proporcionar un sistema de archivos global, compartido, accesible desde todas las estaciones de trabajo. Una o varias máquinas, llamadas **servidores de archivos**, soportan al sistema de archivos. Los servidores

de archivo aceptan solicitudes para la lectura y escritura de archivos por parte de los programas usuarios que se ejecutan en las otras máquinas (no servidoras), llamadas **clientes**. Cada una de las solicitudes que llegue se examina, se ejecuta y la respuesta se envía de regreso, como se ilustra en la figura 1-9.

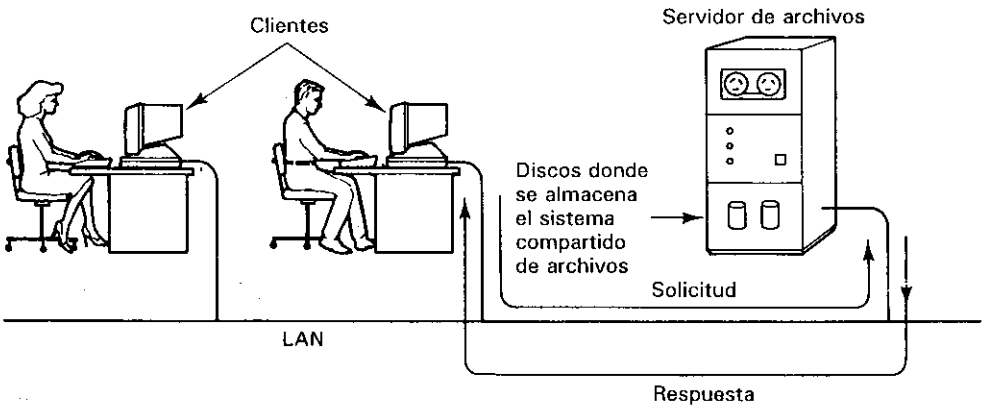


Figura 1-9. Dos clientes y un servidor en un sistema operativo de red.

Los servidores de archivos tienen por lo general un sistema jerárquico de archivos, cada uno de los cuales tiene un directorio raíz, con subdirectorios y archivos. Las estaciones de trabajo pueden importar o montar estos sistemas de archivos, lo que aumenta sus sistemas locales de archivos con aquellos localizados en los servidores. Por ejemplo, en la figura 1-10 se muestran dos servidores de archivos. Uno tiene un directorio de nombre *juegos*, mientras que el otro tiene un directorio de nombre *trabajo*. Cada uno de estos directorios contiene varios archivos. Los clientes que se muestran tienen montados ambos servidores, pero los han montado en lugares diferentes en sus respectivos sistemas de archivos. El cliente 1 los montó en su directorio raíz y tiene acceso a ellos como */juegos* y */trabajo*, respectivamente. El cliente 2, como el cliente 1, montó *juegos* en su directorio raíz, pero como considera la lectura del correo y las noticias como un tipo de juego, creó un directorio */juegos/trabajo* y montó *trabajo* en él. En consecuencia, puede tener acceso a *noticias* mediante la ruta de acceso */juegos/trabajo/noticias* en vez de */trabajo/noticias*.

Aunque no importa la posición de la jerarquía de directorios donde un cliente coloque a un servidor, es importante observar que los diversos clientes tienen un punto de vista distinto del sistema de archivos. El nombre de un archivo depende del lugar desde el cual se tiene acceso a él y de la configuración del sistema de archivos por parte de la máquina. Puesto que cada una de las estaciones de trabajo opera en forma relativamente independiente del resto, no existe garantía alguna de que todas presenten la misma jerarquía de directorios a sus programas.

El sistema operativo a utilizar en este tipo de ambiente debe controlar las estaciones de trabajo en lo individual, a los servidores de archivo y también debe encargarse de la

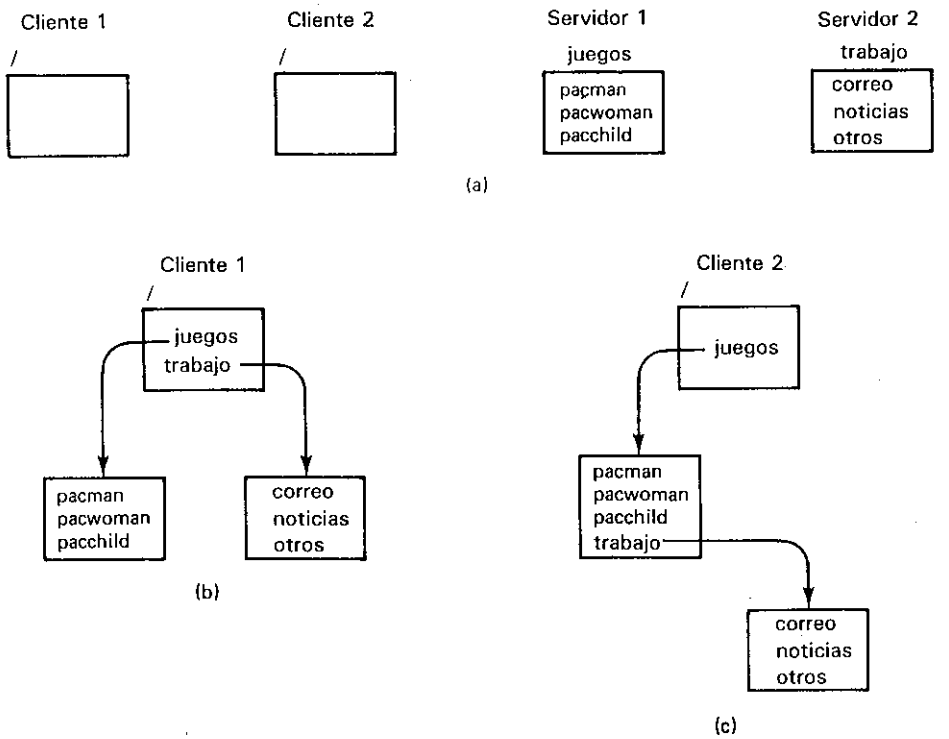


Figura 1-10. Los diversos clientes pueden montar los servidores en diversos lugares.

comunicación entre ellos. Es posible que todas las máquinas ejecuten el mismo sistema operativo, pero esto no es necesario. Si los clientes y los servidores ejecutan diversos sistemas, entonces, como mínimo, deben coincidir en el formato y significado de todos los mensajes que podrían intercambiar. En una situación como ésta, en la que cada máquina tiene un alto grado de autonomía y existen pocos requisitos a lo largo de todo el sistema, las personas se refieren a ella como un **sistema operativo de red**.

1.4.2. Sistemas realmente distribuidos

Los sistemas operativos de red están formados por un software débilmente acoplado en un hardware débilmente acoplado. De no ser por el sistema compartido de archivos, a los usuarios les parecería que el sistema consta de varias computadoras. Cada una puede ejecutar su propio sistema operativo y hacer lo que el propietario quiera. En esencia, no hay coordinación alguna, excepto por la regla de que el tráfico cliente-servidor debe obedecer los protocolos del sistema.

El siguiente paso en la evolución es el del software fuertemente acoplado en hardware débilmente acoplado (es decir, en multicomputadoras). El objetivo de un sistema de este

tipo es crear la ilusión en las mentes de los usuarios que toda la red de computadoras es un sistema de tiempo compartido, en vez de una colección de máquinas diversas. Algunos autores se refieren a esta propiedad como la **imagen de único sistema**. Otros tienen un punto de vista diferente y dicen que un sistema distribuido es aquel que se ejecuta en una colección de máquinas enlazadas mediante una red pero que actúan como un **uniprocesador virtual**. No importa la forma en que se exprese, la idea esencial es que los usuarios no deben ser conscientes de la existencia de varios CPU en el sistema. Ningún sistema en la actualidad cumple en su totalidad este requisito, pero hay varios candidatos promisorios en el horizonte. Analizaremos algunos de ellos en secciones posteriores de este libro.

¿Cuáles son algunas de las características de un sistema distribuido? Para comenzar, debe existir un mecanismo de comunicación global entre los procesos, de forma que cualquier proceso pueda comunicarse con cualquier otro. No tiene que haber distintos mecanismos en distintas máquinas o distintos mecanismos para la comunicación local o la comunicación remota. También debe existir un esquema global de protección. La mezcla del acceso a las listas de control, los bits de protección de UNIX® y las diversas capacidades no producirán una imagen de único sistema.

La administración de procesos también debe ser la misma en todas partes. La forma en que se crean, destruyen, inician y detienen los procesos no debe variar de una máquina a otra. En resumen, la idea detrás de los sistemas operativos de red, en el sentido de que cualquier máquina puede hacer lo que desee mientras obedezca los protocolos estándar cuando participe en una comunicación cliente-servidor, no es suficiente. No sólo debe existir un conjunto de llamadas al sistema disponible en todas las máquinas, sino que estas llamadas deben ser diseñadas de manera que tengan sentido en un ambiente distribuido.

También el sistema de archivos debe tener la misma apariencia en todas partes. El hecho de tener nombres de archivo restringidos a 11 caracteres en ciertos lugares y no tener restricciones en otros no es deseable. Además, todo archivo debe ser visible desde cualquier posición, sujeto, por supuesto, a restricciones de protección y seguridad.

Como consecuencia lógica del hecho de tener una misma interfaz de llamadas al sistema en todas partes, es normal que se ejecuten núcleos idénticos en todos los CPU del sistema. Esto facilita la coordinación de actividades globales. Por ejemplo, cuando se inicie un proceso, todos los núcleos deben cooperar en la búsqueda del mejor lugar para ejecutarlo. Además, se necesita un sistema global de archivos.

Sin embargo, cada núcleo debe tener un control considerable sobre sus propios recursos locales. Por ejemplo, puesto que no existe memoria compartida, es lógico permitir que cada núcleo controle su memoria. Por ejemplo, si se utiliza el intercambio o la paginación, el núcleo de cada CPU es el sitio lógico donde se determina aquello que se deba intercambiar o paginar. No existe razón para centralizar esta autoridad. De manera similar, si se ejecutan varios procesos en ciertos CPU, también tiene sentido hacer la planificación en ese lugar.

En la actualidad se dispone de un considerable conocimiento acerca del diseño de los sistemas operativos distribuidos. En vez de pasar a estos aspectos, primero terminaremos nuestro análisis de las distintas combinaciones de hardware y software y regresaremos a esto en la sección 1.5.

1.4.3. Sistemas de multiprocesador con tiempo compartido

La última combinación que queremos analizar es el sóftware y hardware fuertemente acoplados. Aunque existen varias máquinas de propósito especial en esta categoría (como las máquinas dedicadas a las bases de datos), los ejemplos más comunes de propósito general son los multiprocesadores que operan como un sistema de tiempo compartido de UNIX, sólo que con varios CPU en vez de uno. Para el mundo exterior, un multiprocesador con 32 CPU de 30 MIPS actúa de manera muy parecida a un solo CPU de 960 MIPS (ésta es la imagen de único sistema analizada anteriormente), excepto por el hecho de que la implantación de ésta en un multiprocesador hace más sencilla la vida, puesto que todo el diseño se puede centralizar.

La característica clave de este tipo de sistema es la existencia de una cola de ejecución: una lista de todos los procesos en el sistema que no están bloqueados en forma lógica y listos para su ejecución. La cola de ejecución es una estructura de datos contenida en la memoria compartida. Como ejemplo, consideremos el sistema de la figura 1-11, con 3 CPU y 5 procesos listos para su ejecución. Los cinco procesos están dentro de la memoria compartida y por el momento se ejecutan tres de ellos: el proceso *A* en el CPU 1, el proceso *B* en el CPU 2 y el proceso *C* en el CPU 3. Los otros dos procesos, *D* y *E*, también están en la memoria y esperan su turno.

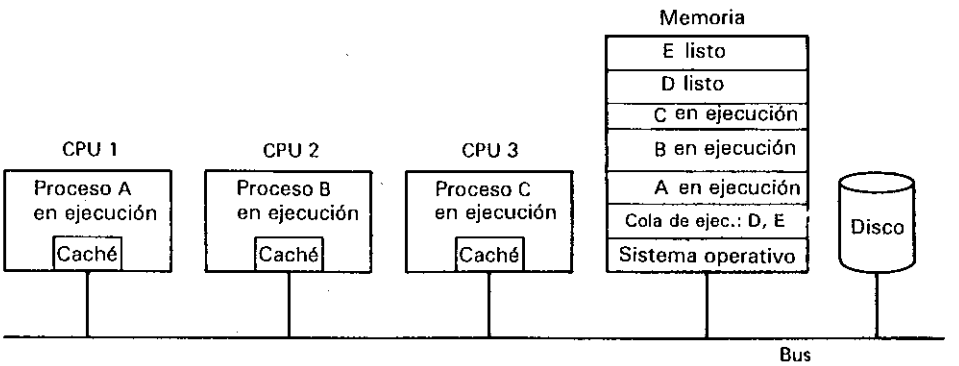


Figura 1-11. Un multiprocesador con una cola de ejecución.

Ahora supongamos que el proceso *B* se bloquea en espera de E/S* o que su quantum se agota. De cualquier forma, el CPU 2 debe suspenderlo y encontrar otro proceso para ejecutarlo. Lo normal sería que el CPU 2 comenzara la ejecución del código del sistema operativo (localizado en la memoria compartida). Después de resguardar todos los registros de *B*, entrará a una región crítica para ejecutar el planificador y buscar otro proceso para su ejecución. Es esencial que el planificador se ejecute como región crítica, con el fin de evitar que dos CPU elijan el mismo proceso para su ejecución inmediata. La exclusión mutua necesaria se puede lograr mediante el uso de monitores, semáforos, o cualquiera de las demás construcciones estándar utilizadas en los sistemas con un procesador.

* E/S = I/O (entrada/salida).

Una vez que el CPU 2 logra el acceso exclusivo a la cola de ejecución, puede eliminar la primera entrada, *D*, salir de la región crítica y comenzar la ejecución de *D*. Al principio, la ejecución será lenta, puesto que el caché del CPU 2 está ocupado por palabras que pertenecen a aquella parte de la memoria compartida que contiene al proceso *B*, pero después de un pequeño lapso, estas palabras se habrán purgado y el caché estará ocupado por el código y los datos de *D*, de modo que la ejecución se acelerará.

Puesto que ninguno de los CPU tiene memoria local y todos los programas se guardan en la memoria global compartida, no importa el CPU donde se ejecute un proceso. Si un proceso con un tiempo de ejecución grande se planifica muchas veces antes de terminar, en promedio, gastará la misma cantidad de tiempo que si se ejecutase en cada CPU. El único factor que no tiene efecto alguno en la elección del CPU es la ligera ganancia en el rendimiento obtenida cuando un proceso se inicia en un CPU que oculta en ese momento parte de su espacio de direcciones. En otras palabras, si todos los CPU están inactivos, en espera de E/S y un proceso está listo para su ejecución, es ligeramente preferible asignarlo al CPU que utilizó por última vez, bajo la hipótesis de que ningún otro proceso ha utilizado ese CPU desde entonces (Vaswani y Zahorjan, 1991).

Un aspecto colateral es que si un proceso se bloquea en espera de E/S en un multiprocesador, el sistema operativo tiene la opción de suspenderlo o bien dejarlo que realice una espera ocupada. Si la mayoría de la E/S se lleva a cabo en menos tiempo del que tarda un cambio entre los procesos, entonces es preferible una espera ocupada. Algunos sistemas dejan que el proceso conserve su procesador por unos cuantos milisegundos, con la esperanza de que la E/S termine pronto, pero si esto no ocurre antes de que se agote el tiempo, se realiza una conmutación de procesos (Karlin *et al.*, 1991). Si la mayor parte de las regiones críticas son cortas, este método puede evitar muchos intercambios de proceso caros.

Un área donde este tipo de multiprocesador difiere de manera apreciable de una red o un sistema distribuido es la organización del sistema de archivos. Lo normal es que el sistema operativo contenga un sistema de archivos tradicional, con un bloque caché unificado. Cuando cualquier proceso ejecuta una llamada al sistema, se hace un señalamiento al sistema operativo, el cual lo lleva a cabo, mediante semáforos, monitores, o cualquier otro equivalente para bloquear los demás CPU, mientras se ejecutan las secciones críticas o se tiene acceso a las tablas centrales. De este modo, al hacer una llamada WRITE, el caché central se bloquea, los nuevos datos entran al caché y se retira el bloqueo. Cualquier llamada posterior READ verá los nuevos datos, al igual que en un sistema con un procesador. Como un todo, el sistema de archivos no es muy distinto del sistema de archivos de un procesador. De hecho, en ciertos multiprocesadores se dedica uno de los CPU para la ejecución del sistema operativo; los demás ejecutan programas del usuario. Sin embargo, esta situación no es deseable, ya que la máquina con el sistema operativo se convierte a menudo en un cuello de botella. Este punto se analiza con detalle en Boykin y Langerman (1990).

Debe ser claro que los métodos utilizados en el multiprocesador para lograr la apariencia de un uniprocesador virtual no se pueden aplicar a máquinas sin memoria compartida. Las colas centralizadas de ejecución y los cachés de bloque sólo funcionan cuando todos los CPU tienen acceso a ellos con poco retraso. Aunque estas estructuras de datos se podrían simular en una red de máquinas, los costos de comunicación hacen que este método sea demasiado caro.

La figura 1-12 muestra algunas de las diferencias entre los tres tipos de sistemas analizados hasta ahora.

Elemento	Sistema operativo de red	Sistema operativo distribuido	Sistema operativo de multiprocesador
¿Se ve como un uniprocesador virtual?	No	Sí	Sí
¿Todos tienen que ejecutar el mismo sistema operativo?	No	Sí	Sí
¿Cuántas copias del sistema operativo existen?	N	N	1
¿Cómo se logra la comunicación?	Archivos compartidos	Mensajes	Memoria compartida
¿Se requiere un acuerdo en los protocolos de la red?	Sí	Sí	No
¿Existe una cola de ejecución?	No	No	SI
¿Existe una semántica bien definida para los archivos compartidos?	Por lo general no	SI	SI

Figura 1-12. Comparación de tres formas distintas de organizar *n* CPU.

1.5. ASPECTOS DEL DISEÑO

En las secciones anteriores analizamos los sistemas distribuidos y algunos temas relacionados con éstos, tanto desde el punto de vista del hardware como del software. En el resto de este capítulo analizaremos en forma breve algunos de los aspectos claves del diseño con los que deben trabajar las personas que piensan construir un sistema operativo distribuido. Regresaremos a estos aspectos con mayor detalle en secciones posteriores de este libro.

1.5.1. Transparencia

Tal vez el aspecto más importante sea la forma de lograr la imagen de un sistema. En otras palabras, ¿cómo engañan los diseñadores del sistema a todas las personas, de forma que piensen que la colección de máquinas es tan sólo un sistema de tiempo compartido de un procesador, a la manera antigua? Un sistema que logre este objetivo se conoce como **transparente**.

La transparencia se puede lograr en dos niveles distintos. Lo más fácil es ocultar la distribución a los usuarios. Por ejemplo, cuando un usuario de UNIX escribe *make* para volver a compilar un gran número de archivos en un directorio, no hay que decirle que

todas las compilaciones se realizan en paralelo en distintas máquinas y que éstas utilizan una variedad de servidores de archivos con este fin. Para el usuario, lo único poco usual es que el desempeño del sistema tiene un cambio decente. En términos de los comandos proporcionados a través de la terminal y los resultados exhibidos en ella, se puede lograr que el sistema distribuido aparezca como un sistema con un procesador.

En un nivel inferior, también es posible, aunque más difícil, hacer que el sistema sea transparente para los programas. En otras palabras, se puede diseñar la interfaz de llamadas al sistema de modo que no sea visible la existencia de varios procesadores. Sin embargo, es más difícil colocar una venda sobre los ojos del programador que sobre los ojos del usuario de la terminal.

¿Qué significa en realidad la transparencia? Es uno de esos conceptos escurridizos que suenan razonables, pero más sutiles de lo que aparentan. Como ejemplo, imaginemos un sistema distribuido consistente en estaciones de trabajo, las cuales ejecutan cierto sistema operativo estándar. Por lo general, los servicios de sistema (por ejemplo, la lectura de archivos) se obtienen al proporcionar una llamada al sistema, la cual hace un señalamiento al núcleo. En este sistema, el acceso a los archivos remotos debe realizarse de la misma manera. Un sistema donde el acceso a los archivos remotos se realice mediante el establecimiento explícito de una conexión en la red con un servidor remoto para después enviarle mensajes no es transparente, ya que entonces el acceso a los servicios remotos será distinto al acceso a los servicios locales. El programador podría notar la participación de varias máquinas, lo cual no está permitido.

El concepto de transparencia se puede aplicar a varios aspectos de un sistema distribuido, como se muestra en la figura 1-13. La **transparencia de localización** se refiere al hecho de que en un verdadero sistema distribuido, los usuarios no pueden indicar la localización de los recursos de hardware y software, como los CPU, impresoras, archivos y bases de datos. El nombre del recurso no debe codificar de manera secreta la localización de éste, por lo que no se permiten nombres tales como *machine1:prog.c* o */machine1/prog.c*.

Tipo	Significado
Transparencia de localización	Los usuarios no pueden indicar la localización de los recursos
Transparencia de migración	Los recursos se pueden mover a voluntad sin cambiar sus nombres
Transparencia de réplica	Los usuarios no pueden indicar el número de copias existentes
Transparencia de concurrencia	Varios usuarios pueden compartir recursos de manera automática
Transparencia de paralelismo	Las actividades pueden ocurrir en paralelo sin el conocimiento de los usuarios

Figura 1-13. Distintos tipos de transparencia en un sistema distribuido.

La **transparencia de migración** significa que los recursos deben moverse de una posición a otra sin tener que cambiar sus nombres. En el ejemplo de la figura 1-10 vimos la

forma en que los directorios servidores se podían montar en lugares arbitrarios dentro de la jerarquía de directorios de los clientes. Puesto que una ruta de acceso del tipo */trabajo/noticias* no revela la posición del servidor, es transparente con respecto de la localización. Sin embargo, supongamos ahora que la persona que ejecutó el servidor decide que la lectura de noticias en la red cae en realidad dentro de la categoría "juegos" en vez de la categoría "trabajo". De acuerdo con esto, el usuario desplaza *noticias* del servidor 2 al servidor 1. La siguiente vez que el cliente 1 arranque y monte los servidores en la forma acostumbrada por él, notará que ya no existe */trabajo/noticias*. En vez de esto, existe una nueva entrada */juegos/trabajo*. Así, el simple hecho de que un archivo o directorio ha emigrado de un servidor a otro lo ha obligado a adquirir un nuevo nombre, puesto que el sistema de montajes remotos no es transparente con respecto de la migración.

Si un sistema distribuido tiene **transparencia de réplica**, entonces el sistema operativo es libre de fabricar por su cuenta copias adicionales de los archivos y otros recursos sin que lo noten los usuarios. En el ejemplo anterior, es claro que la réplica automática es imposible, puesto que los nombres y posiciones están muy ligados entre sí. Para ver la forma de lograr la transparencia de réplica, consideremos una colección de *n* servidores, conectados de manera lógica para formar un anillo. Cada servidor mantiene toda la estructura del árbol de directorios, pero sólo conserva un subconjunto de los propios archivos. Para leer un archivo, un cliente envía un mensaje a alguno de los servidores con el nombre completo de la ruta de acceso. Ese servidor verifica si tiene el archivo. En ese caso, regresa los datos solicitados. En caso contrario, turna la solicitud al siguiente servidor del anillo, el cual repite a su vez el algoritmo. En este sistema, los servidores pueden decidir por sí mismos si realizan una réplica de un archivo en alguno o en todos los servidores, sin que los usuarios deban saber esto. Tal esquema es transparente con respecto a la réplica, puesto que permite al sistema realizar copias de archivos de uso intenso, en forma transparente y sin que los usuarios observen que esto ocurre.

Los sistemas distribuidos tienen por lo general varios usuarios independientes. ¿Qué debe hacer el sistema cuando dos o más usuarios intenten tener acceso al mismo recurso al mismo tiempo? Por ejemplo, ¿qué ocurre si dos usuarios intentan actualizar el mismo archivo al mismo tiempo? Si el sistema es **transparente con respecto a la concurrencia**, los usuarios no notarán la existencia de otros usuarios. Un mecanismo para lograr esta forma de transparencia sería que el usuario cerrara en forma automática un recurso, una vez que alguien haya comenzado a utilizarlo, elimina el bloqueo sólo hasta que termine el acceso. De esta forma, todos los recursos tendrían un acceso secuencial, nunca concurrente.

Por último, tenemos la más difícil, la **transparencia con respecto al paralelismo**. En principio, se supone que un sistema distribuido debe aparecer ante los usuarios como un sistema tradicional de tiempo compartido con un procesador. ¿Qué ocurre si un programador sabe que su sistema distribuido tiene 1 000 CPU y desea utilizar una parte esencial de ellos para un programa de ajedrez, el cual evalúa posiciones en forma paralela? La respuesta teórica es que el compilador, el sistema de tiempo de ejecución y el sistema operativo deben, en forma conjunta, aprovechar este paralelismo potencial sin que el programador se dé cuenta de ello. Por desgracia, el estado actual de las cosas es que nada de esto está por ocurrir. Los programadores que en realidad desean utilizar varios CPU para un problema deben progra-

mar esto en forma explícita, al menos en el futuro próximo. La transparencia con respecto al paralelismo se puede considerar como el cáliz sagrado para los diseñadores de sistemas distribuidos. Cuando esto se logre, la obra habrá terminado y será hora de desplazarse a nuevos campos.

A pesar de todo esto, hay ocasiones en que los usuarios *no* desean la transparencia completa. Por ejemplo, cuando un usuario solicita la impresión de un documento, con frecuencia prefiere que la salida aparezca en la impresora local, no una que esté a 100 km de distancia, aunque la impresora distante sea rápida, barata, maneje color y olor y esté inactiva por el momento.

1.5.2. Flexibilidad

El segundo aspecto clave del diseño es la flexibilidad. Es importante que el sistema sea flexible, ya que apenas estamos aprendiendo a construir sistemas distribuidos. Es probable que este proceso tenga muchas salidas falsas y una considerable retroalimentación. Las decisiones de diseño que ahora parezcan razonables podrían demostrar ser incorrectas posteriormente. La mejor forma de evitar los problemas es mantener abiertas las opciones.

La flexibilidad, junto con la transparencia es como la paternidad y los pasteles de manzana: ¿quién podría estar en contra de ellos? Es difícil imaginar a alguien que argumente a favor de un sistema inflexible. Sin embargo, las cosas no son tan simples como parecen. Existen dos escuelas de pensamiento en cuanto a la estructura de los sistemas distribuidos. Una escuela mantiene que cada máquina debe ejecutar un núcleo tradicional que proporcione la mayoría de los servicios. La otra sostiene que el núcleo debe proporcionar lo menos posible y que el grueso de los servicios del sistema operativo se obtenga a partir de los servidores al nivel usuario. Estos dos modelos, conocidos como el núcleo monolítico y el micronúcleo, respectivamente, se ilustran en la figura 1-14.

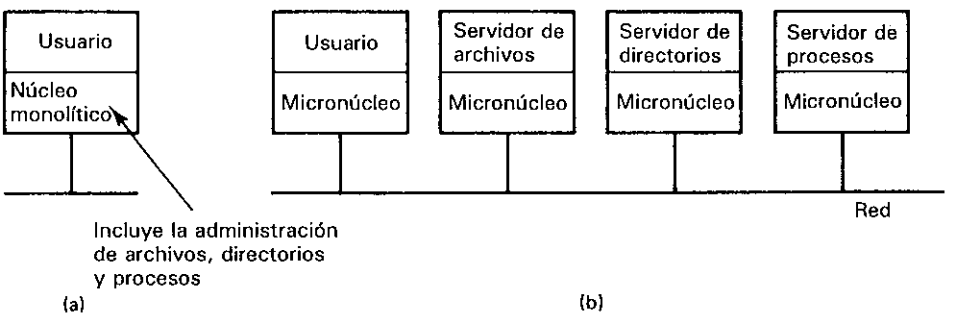


Figura 1-14. (a) Núcleo monolítico. (b) Micronúcleo.

El núcleo monolítico es el sistema operativo centralizado básico actual, aumentado con capacidades de red y la integración de servicios remotos. La mayoría de las llamadas al sistema se realizan mediante señalamientos al núcleo, en donde se realiza el trabajo, para

que después el núcleo regrese el resultado deseado al proceso del usuario. Con este método, la mayoría de las máquinas tiene discos y administra sus propios sistemas locales de archivos. Muchos de los sistemas distribuidos que son extensiones o imitaciones de UNIX utilizan este método, puesto que el propio UNIX tiene un gran núcleo monolítico.

Si el núcleo monolítico es el campeón reinante, el micronúcleo es el retador en ascenso. La mayoría de los sistemas distribuidos diseñados a partir de cero utilizan este método. El micronúcleo es más flexible, ya que casi no hace nada. En términos básicos, proporciona sólo cuatro servicios mínimos:

1. Un mecanismo de comunicación entre procesos.
2. Cierta administración de la memoria.
3. Una cantidad limitada de planificación y administración de procesos de bajo nivel.
4. Entrada/salida de bajo nivel.

En particular, a diferencia del núcleo monolítico, no proporciona el sistema de archivos, el sistema de directorios, toda la administración de procesos o gran parte del manejo de las llamadas al sistema. Los servicios que presta el micronúcleo están incluidos debido a que sería difícil o caro proporcionarlos en alguna otra parte. El objetivo es mantenerlo pequeño.

Todos los demás servicios del sistema operativo se implantan por lo general como servidores a nivel usuario. Para buscar un nombre, leer un archivo u obtener algún otro servicio, el usuario envía un mensaje al servidor apropiado, el cual realiza el trabajo y regresa el resultado. La ventaja de este método es que es altamente modular: existe una interfaz bien definida con cada servicio (el conjunto de mensajes que comprende el servidor) y cada servicio es igual de accesible para todos los clientes, en forma independiente de la posición. Además, es fácil implantar, instalar y depurar nuevos servicios, puesto que la adición o modificación de un servicio no requiere el alto total del sistema y el arranque de un nuevo núcleo, como en el caso de un núcleo monolítico. Es precisamente esta capacidad de añadir, eliminar y modificar servicios lo que le da al micronúcleo su flexibilidad. Además, los usuarios que no estén satisfechos con alguno de los servicios oficiales son libres de escribir el suyo propio.

Como ejemplo sencillo de esta fuerza, es posible tener un sistema distribuido con varios servidores de archivo donde uno soporte el servicio de archivos de MS-DOS y otro soporte el servicio de archivos de UNIX. Los programas individuales pueden utilizar uno de ellos o ambos, si así lo desean. Por el contrario, con un núcleo monolítico, el sistema de archivos se integra al núcleo y los usuarios no tienen más opción que utilizarlo.

La única ventaja potencial del núcleo monolítico es el rendimiento. Los señalamientos al núcleo y la realización de todo el trabajo ahí puede ser más rápido que el envío de mensajes a los servidores remotos. Sin embargo, una comparación detallada de dos sistemas operativos distribuidos, uno con un núcleo monolítico (Sprite) y otro con un micronúcleo (Amoeba), ha mostrado que en la práctica esta ventaja no existe (Douglis *et al.*, 1991). Otros factores tienden a dominar, y la pequeña cantidad de tiempo necesaria para enviar un men-

saje y obtener una respuesta (por lo general de 1 mseg) es a menudo insignificante. Como consecuencia, es probable que los sistemas de micronúcleo lleguen a dominar de manera gradual el esquema de los sistemas distribuidos y que los núcleos monolíticos desaparezcan o evolucionen en micronúcleos. Tal vez las futuras ediciones del libro de Silberschatz y Galvin de sistemas operativos (1994) presentarán colibríes y vencejos en vez de estegosaurios y triceratopos.

1.5.3. Confiabilidad

Uno de los objetivos originales de la construcción de sistemas distribuidos fue el hacerlos más confiables que los sistemas con un procesador. La idea es que si una máquina falla, alguna otra máquina se encargue del trabajo. En otras palabras, en teoría, la confiabilidad global del sistema podría ser el OR Booleano de la confiabilidad de los componentes. Por ejemplo, con cuatro servidores de archivos, cada uno con una probabilidad de 0.95 de funcionar en un instante dado, la probabilidad de que los cuatro fallen de manera simultánea es $0.05^4 = 0.000\ 006$, de modo que la probabilidad de que al menos uno esté disponible es 0.999 994, mucho mejor que la de cualquier servidor individual.

Ésa es la teoría. La práctica es que los sistemas distribuidos actuales cuentan con que un número de ciertos servidores sirvan para que el todo funcione. Como resultado, algunos de ellos tienen disponibilidad más relacionada con el AND Booleano de las componentes que con el OR Booleano. En una observación muy citada, Leslie Lamport definió alguna vez un sistema distribuido como: "aquel del cual no puedo obtener un trabajo debido a que cierta máquina de la cual nunca he oído se ha descompuesto". Aunque esta observación fue hecha (supuestamente) de manera irónica, es claro que hay mucho por mejorar.

Es importante distinguir entre varios aspectos de la confiabilidad. La **disponibilidad**, como acabamos de ver, se refiere a la fracción de tiempo en que se puede utilizar el sistema. El sistema de Lamport no tiene buena calificación a este respecto. La disponibilidad se puede mejorar mediante un diseño que no exija el funcionamiento simultáneo de un número sustancial de componentes críticos. Otra herramienta para mejorar la disponibilidad es la redundancia: se pueden duplicar piezas clave del hardware y del software, de modo que si una de ellas falla, las otras puedan llenar su hueco.

Un sistema ampliamente confiable debe ser muy disponible, pero eso no es suficiente. Los datos confiados al sistema no deben perderse o mezclarse de manera alguna; si los archivos se almacenan de manera redundante en varios servidores, todas las copias deben ser consistentes. En general, mientras se tengan más copias, será mejor la disponibilidad, pero también aumentará la probabilidad de que sean inconsistentes, en particular si las actualizaciones son frecuentes. Los diseñadores de todos los sistemas distribuidos deben tener en mente este dilema en todo momento.

Otro aspecto de la confiabilidad general es la seguridad. Los archivos y otros recursos deben ser protegidos contra el uso no autorizado. Aunque este mismo aspecto aparece en los sistemas con un único procesador, es más severo en los sistemas distribuidos. En un sistema con un procesador, el usuario se conecta y pasa por un proceso de autenticación. A partir de ese momento, el sistema sabe quién es el usuario y puede verificar la validez

de cada intento de acceso. En un sistema distribuido, cuando llega un mensaje a un servidor con cierta solicitud, éste no tiene una forma sencilla para determinar de quién proviene. No se puede confiar en un nombre o campo de identificación en el mensaje, puesto que el emisor puede mentir. Al menos, se requiere de un cuidado considerable.

Otro aspecto relacionado con la confiabilidad es la **tolerancia de fallas**. Supongamos que un servidor falla y vuelve a arrancar de manera súbita. ¿Qué ocurre? ¿Se extiende esta falla del servidor a una falla con los usuarios? Si el servidor tiene tablas con importante información acerca de actividades en proceso, lo menos que ocurrirá es que la recuperación será difícil.

En general, los sistemas distribuidos se pueden diseñar de forma que escondan las fallas; es decir, ocultarlos de los usuarios. Si un servicio de archivo o algún otro servicio se construye a partir de un grupo de servidores con una cooperación cercana, entonces sería posible construirlo de forma que los usuarios no noten la pérdida de uno o dos servidores, de no ser por cierta degradación del desempeño. Por supuesto, el truco es arreglar esta cooperación de modo que no añada un costo sustancial al sistema en el caso normal, cuando todo funciona de manera correcta.

1.5.4. Desempeño

El tema del desempeño siempre está oculto, al acecho. La construcción de un sistema distribuido transparente, flexible y confiable no hará que usted gane premios si es tan lento como la miel. En particular, cuando se ejecuta una aplicación en un sistema distribuido, no debe parecer peor que su ejecución en un procesador. Por desgracia, esto es más difícil de lograr que de decir.

Se pueden utilizar diversas métricas del desempeño. El tiempo de respuesta es una, pero también lo son el rendimiento (número de trabajos por hora), uso del sistema y cantidad consumida de la capacidad de la red. Además, es frecuente que el resultado de cualquier parámetro dependa de la naturaleza de éste. Un parámetro relacionado con gran número de cálculos independientes pero con límites en el uso del CPU puede producir resultados radicales distintos de los que constan del rastreo de un archivo de gran tamaño en búsqueda de un patrón.

El problema del desempeño se complica por el hecho de que la comunicación, factor esencial en un sistema distribuido (y ausente en un sistema con un procesador) es algo lenta por lo general. El envío de un mensaje y la obtención de una respuesta en una LAN tarda cerca de 1 mseg. La mayor parte de este tiempo se debe a un manejo inevitable del protocolo en ambos extremos, más que el tiempo que pasan los bits en los cables. Así, para optimizar el desempeño, con frecuencia hay que minimizar el número de mensajes. La dificultad con esta estrategia es que la mejor forma de mejorar el desempeño es tener muchas actividades en ejecución paralela en distintos procesadores, pero esto requiere el envío de muchos mensajes. (Otra solución es hacer todo el trabajo en una máquina, pero esto es poco apropiado para un sistema distribuido.)

Una posible salida es prestar atención al **tamaño de grano** de todos los cálculos. El desarrollo de un cálculo pequeño de manera remota, como la suma de dos enteros, rara vez

vale la pena, puesto que el costo excesivo de la comunicación excede los ciclos adicionales de CPU ganados. Por otro lado, la ejecución remota de un largo trabajo de cálculo con límites de cómputo puede convertirse en un problema. En general, los trabajos que implican gran número de pequeños cálculos, en particular aquellos que interactúan en gran medida con otros, pueden ser la causa de algunos problemas en los sistemas distribuidos con una comunicación lenta en términos relativos. Se dice que tales trabajos exhiben un **paralelismo de grano fino**. Por otro lado, los trabajos que implican grandes cálculos y bajas tasas de interacción, así como pocos datos, es decir, **paralelismo de grano grueso**, pueden ajustarse mejor a este modelo.

La tolerancia de fallas también tiene su precio. A veces, una buena confiabilidad se logra mejor con varios servidores que cooperen en forma cercana en una solicitud. Por ejemplo, si una solicitud llega a un servidor, éste podría enviar de manera inmediata una copia del mensaje a uno de sus colegas, de forma que si falla antes de terminar, el colega se pueda encargar de ella. Es natural que si concluye, debe informar al colega que ha terminado el trabajo, lo que representa otro mensaje. Así, tenemos al menos dos mensajes adicionales, que en el caso normal, cuestan tiempo y capacidad de la red y no producen una ganancia notable.

1.5.5. Escalabilidad

La mayor parte de los sistemas distribuidos están diseñados para trabajar con unos cuantos cientos de CPU. Es posible que los sistemas futuros tengan mayores órdenes de magnitud y las soluciones que funcionen bien para 200 máquinas fallen de manera total para 200 millones. Consideremos el caso siguiente. La PTT de Francia (la oficina de administración del correo, teléfono y telégrafo) está en proceso de instalar una terminal en cada casa y negocio en Francia. La terminal, conocida como **minitel**, permitirá el acceso en línea a una base de datos con todos los números telefónicos de Francia, con lo que eliminará la necesidad de imprimir y distribuir los costosos directorios telefónicos. Esto reducirá en gran medida la necesidad de operadoras de los servicios de información, que no hacen más que proporcionar teléfonos todo el día. Se ha calculado que el sistema se pagará a sí mismo en unos cuantos años. Si el sistema funciona en Francia, otros países adoptarán sin duda sistemas similares.

Una vez que las terminales estén en su lugar, la posibilidad de utilizarlas también para el correo electrónico (en especial junto con las impresoras) también estará presente. Puesto que los servicios postales pierden cantidad enorme de dinero en todos los países del mundo y los servicios telefónicos son muy lucrativos, existen grandes incentivos para el reemplazo del correo de papel por el electrónico.

Después vendrá el acceso interactivo a todo tipo de bases de datos y servicios, desde la banca electrónica hasta la reservación en aviones, trenes, hoteles, teatros y restaurantes, por sólo nombrar unos cuantos. Antes de que pase mucho tiempo, tendremos un sistema distribuido con decenas de millones de usuarios. La cuestión es si los métodos que se desarrollan en la actualidad podrán escalarse hacia tales grandes sistemas.

Aunque se sabe poco de los enormes sistemas distribuidos, es claro un principio guía: evitar los componentes, tablas y algoritmos centralizados (véase la figura 1-15). No sería una buena idea tener un solo servidor de correo para 50 millones de usuarios. Aunque tuviera la suficiente capacidad de CPU y almacenamiento, la capacidad de la red dentro y fuera de él, sí sería un problema. Además, el sistema no toleraría bien las fallas. Una falla en el suministro de la corriente eléctrica provocaría la caída del sistema. Por último, la mayoría del correo es local. Si un mensaje enviado por un usuario en Marsella a otro usuario a dos cuadras de él pasa a través de una máquina en París, ésta sería la ruta equivocada.

Concepto	Ejemplo
Componentes centralizados	Un solo servidor de correo para todos los usuarios
Tablas centralizadas	Un directorio telefónico en línea
Algoritmos centralizados	Realización de un ruteo con base en la información completa

Figura 1-15. Cuellos de botella potenciales que los diseñadores deben intentar evitar en los sistemas distribuidos de gran tamaño.

Las tablas centralizadas son casi tan malas como los componentes centralizados. ¿Cómo se podría tener un registro de los números telefónicos y direcciones de 50 millones de personas? Supongamos que cada registro de datos podría caber en 50 caracteres. Un solo disco de 2.5 gigabytes podría proporcionar el espacio adecuado de almacenamiento. Pero de nuevo, si sólo se tiene una base de datos, se saturarían todas las líneas de comunicación hacia dentro y afuera de ella. También sería vulnerable a las fallas (una partícula de polvo podría causar la ruptura de la cabeza y echar por tierra todo el servicio del directorio). Además, también en este caso, la valiosa capacidad de la red se desperdiciaría con largas colas para el procesamiento.

Por último, los algoritmos centralizados también son mala idea. En un sistema distribuido de gran tamaño, una enorme cantidad de mensajes debe tener una ruta a través de muchas líneas. Desde un punto de vista teórico, la vía óptima para hacer esto es recolectar la información completa relativa a la carga de todas las máquinas y todas las líneas, para después ejecutar un algoritmo de teoría de gráficas para calcular todas las rutas óptimas. Esta información se puede diseminar entonces en todo el sistema para mejorar en ruteo.

El problema es que la recolección y transporte de toda la información de entrada o salida sería una mala idea, por las razones ya analizadas. De hecho, hay que evitar cualquier algoritmo que opere mediante la recolección de información en todos los lugares, para enviarlos a una máquina para su procesamiento y después distribuir los resultados. Sólo se deben utilizar los algoritmos descentralizados. Éstos tienen las siguientes características, que los distingue de los algoritmos centralizados:

1. Ninguna máquina tiene la información completa acerca del estado del sistema.
2. Las máquinas toman decisiones sólo con base en la información local.
3. La falla de una máquina no arruina el algoritmo.
4. No existe una hipótesis implícita de la existencia de un reloj global.

Las primeras tres características provienen de lo que hemos señalado hasta el momento. La última es tal vez menos obvia, pero también es importante. Cualquier algoritmo que empiece con: "exactamente a las 12:00:00, todas las máquinas deben anotar el tamaño de su cola de salida" fracasará, puesto que es imposible sincronizar de manera precisa a todos los relojes. Los algoritmos deben tomar en cuenta la carencia de una sincronización precisa de los relojes. Mientras mayor sea el sistema, mayor será la incertidumbre. En una LAN, con un esfuerzo considerable se podría sincronizar los relojes dentro de un rango de unos cuantos milisegundos, pero hacer esto a nivel nacional es una locura. En el capítulo 3 analizaremos la sincronización de relojes en el caso distribuido.

1.6. RESUMEN

Los sistemas distribuidos constan de CPU autónomos que funcionan juntos para que todo el sistema parezca como una computadora. Tienen varios puntos favorables potenciales, como buena proporción precio/desempeño, se pueden ajustar bien a las aplicaciones distribuidas, pueden ser muy confiables y pueden aumentar su tamaño de manera gradual al aumentar la carga de trabajo. También tienen ciertas desventajas, como el hecho de tener un software más complejo, potenciales cuellos de botella en la comunicación y una seguridad débil. Sin embargo, existe un considerable interés mundial por su construcción e instalación.

Los modernos sistemas de cómputo tienen con frecuencia varios CPU. Éstos se pueden organizar como multiprocesadores (con memoria compartida) o como multicomputadoras (sin memoria compartida). Ambos tipos se pueden basar en un bus o en un conmutador. Los primeros tienden a ser fuertemente acoplados, mientras que los segundos tienden a ser débilmente acoplados.

El software para los sistemas con varios CPU se pueden dividir en tres clases. Los sistemas operativos de red permiten a los usuarios en estaciones de trabajo independientes la comunicación por medio de un sistema compartido de archivos, pero dejan que cada usuario domine su propia estación de trabajo. Los sistemas operativos distribuidos convierten toda la colección de hardware y software en un sistema integrado, muy parecido a un sistema tradicional de tiempo compartido. Los multiprocesadores con memoria compartida también ofrecen la imagen de único sistema, pero lo hacen mediante la vía de centralizar todo, por lo que en realidad, este caso es un sistema. Los multiprocesadores con memoria compartida no son sistemas distribuidos.

Los sistemas distribuidos deben diseñarse con cuidado, puesto que existen muchas trampas para los incautos. Un aspecto clave es la transparencia: ocultar la distribución a los usuarios e incluso a los programas de aplicación. Otro aspecto es la flexibilidad. Puesto que el campo se encuentra todavía en su infancia, el diseño se debe hacer con la idea de facilitar los cambios futuros. A este respecto, los micronúcleos son superiores a los núcleos monolíticos. Otros aspectos importantes son la confiabilidad, el desempeño y la escalabilidad.

PROBLEMAS

1. La razón precio/desempeño de las computadoras ha mejorado en algo así como 11 órdenes de magnitud desde que los primeros mainframe comerciales aparecieron a principios de la década de 1950. El texto muestra lo que significaría una ganancia similar en la industria del automóvil. Dé otro ejemplo de lo que significa una ganancia tan grande.

2. Mencione dos ventajas y dos desventajas de los sistemas distribuidos con respecto de los sistemas centralizados.

3. ¿Cuál es la diferencia entre un multiprocesador y una multicomputadora?

4. Los términos *sistema débilmente acoplado* y *sistema fuertemente acoplado* se utilizan con frecuencia para describir los sistemas de cómputo distribuidos. ¿Cuál es la diferencia entre ellos?

5. ¿Cuál es la diferencia entre una computadora MIMD y una computadora SIMD?

6. Un multiprocesador con base en un bus utiliza cachés monitores para conseguir una memoria coherente. ¿Funcionarán los semáforos en esta máquina?

7. Los conmutadores de cruceta permiten procesar a la vez gran número de solicitudes de memoria, proporcionando excelente desempeño. ¿Por qué se utilizan rara vez en la práctica?

8. Una multicomputadora con 256 CPU se organiza como una retícula de 16×16. ¿Cuál puede ser el mayor tiempo de retraso (correspondiente a los saltos) para un mensaje?

9. Considere ahora el caso de un hipercubo de 256 CPU. ¿Cuál sería ahora el máximo tiempo de retraso correspondiente a los saltos?

10. Un multiprocesador tiene 4 096 CPU de 50 MIPS conectados a la memoria por medio de una red omega. ¿Con qué rapidez deben permitir los conmutadores que una solicitud vaya a la memoria y regrese en un tiempo de instrucción?

11. ¿Qué significa imagen de único sistema?

12. ¿Cuál es la diferencia principal entre un sistema operativo distribuido y un sistema operativo de red?

13. ¿Cuáles son las tareas principales de un micronúcleo?

14. Mencione dos ventajas de un micronúcleo sobre un núcleo monolítico.

15. La transparencia con respecto a la concurrencia es un objetivo deseable en los sistemas distribuidos. ¿Tienen esta característica los sistemas centralizados en forma automática?

16. Explique con sus propias palabras el concepto de transparencia con respecto al paralelismo.

17. Un servidor experimental de archivos funciona 3/4 del tiempo y no funciona 1/4 del tiempo debido a ciertos errores. ¿Cuántas veces deberá duplicarse este servidor para obtener una disponibilidad de al menos 99%?

18. Suponga que debe compilar un programa fuente de gran tamaño, consistente en m archivos. La compilación tendrá lugar en un sistema con n procesadores, donde $n >> m$. Lo mejor que puede esperar es una mejora de magnitud m con respecto a la velocidad de un procesador. ¿Qué factores podrían hacer que la mejora sea menor que este máximo?

2

Comunicación en los sistemas distribuidos

La diferencia más importante entre un sistema distribuido y un sistema con un procesador es la comunicación entre procesos. En un sistema con un procesador, la mayor parte de la comunicación entre procesos supone de manera implícita la existencia de la memoria compartida. Un ejemplo típico es el problema de los productores y los consumidores, donde un proceso escribe en un buffer compartido y otro proceso lee de él. Incluso en la forma más básica de sincronización, el semáforo, hay que compartir una palabra (la propia variable del semáforo). En un sistema distribuido, no existe tal memoria compartida, por lo que toda la naturaleza de la comunicación entre procesos debe replantearse a partir de cero. En este capítulo analizaremos varios aspectos, ejemplos y problemas asociados con la comunicación entre procesos en los sistemas operativos distribuidos.

Comenzaremos con el análisis de las reglas a las que se deben apegar los procesos respecto a la comunicación, conocidas como protocolos. Para los sistemas distribuidos en un área amplia, estos protocolos toman con frecuencia la forma de varias capas, cada una con sus propios objetivos y reglas. Examinaremos dos conjuntos de capas, OSI y ATM. Después analizaremos con detalle el modelo cliente-servidor. Después se analizará la forma en que se intercambian los mensajes y las muchas opciones disponibles para los diseñadores del sistema.

Una opción particular, la llamada a un procedimiento remoto, es lo bastante importante como para disponer de su propia sección. Ésta es en realidad una mejor forma de empacar la transferencia de mensajes, de forma que se parezca más a la programación convencional y que, por lo tanto, sea más fácil de utilizar. Sin embargo, tiene sus propias peculiaridades y problemas, que también analizaremos.

Por último, concluiremos el capítulo mediante el estudio de la forma en que pueden comunicarse grupos de procesos, en vez de sólo dos procesos. Estudiaremos un ejemplo detallado de comunicación en grupo, ISIS.

2.1. PROTOCOLOS CON CAPAS

Debido a la ausencia de memoria compartida, toda la comunicación en los sistemas distribuidos se basa en la transferencia de mensajes. Cuando el proceso *A* quiere comunicarse con el proceso *B*, construye primero un mensaje en su propio espacio de direcciones. Entonces ejecuta una llamada al sistema para que el sistema operativo busque el mensaje y lo envíe a través de la red hacia *B*. Aunque esta idea básica suena muy sencilla, para evitar el caos, *A* y *B* deben coincidir en el significado de los bits que se envíen. Si *A* envía una nueva novela brillante escrita en francés, codificada en los caracteres EBCDIC de IBM y *B* espera el inventario de un supermercado escrito en inglés y codificado en ASCII, la comunicación no será óptima.

Se necesitan muchos puntos de acuerdo diferentes. ¿Cuántos voltios hay que utilizar para la señal correspondiente a un bit 0 y cuántos para un bit 1? ¿Cómo sabe el receptor cuál es el último bit del mensaje? ¿Cómo puede detectar si un mensaje ha sido dañado o perdido, y qué debe hacer si lo descubre? ¿Qué longitud tienen los números, cadenas y otros elementos de datos y cuál es la forma en que están representados? En resumen, se necesitan puntos de acuerdo en una amplia gama de niveles, desde los detalles de bajo nivel de transmisión de los bits hasta los de alto nivel acerca de la forma en que debe expresarse la información.

Para facilitar el trabajo con los distintos niveles y aspectos correspondientes a la comunicación, la organización internacional de estándares [International Standards Organization (ISO)] ha desarrollado un modelo de referencia que identifica en forma clara los distintos niveles, les da nombres estandarizados y señala cuál nivel debe realizar cada trabajo. Este modelo se llama el **modelo de referencia para interconexión de sistemas abiertos** (Day y Zimmerman, 1983), lo cual se abrevia por lo general como **ISO OSI** o a veces sólo como el **modelo OSI**. Aunque no pretendemos dar aquí una descripción completa de este modelo y todas sus implicaciones, será útil una breve introducción. Para más detalles, véase (Tanenbaum, 1988).

Para comenzar, el modelo OSI está diseñado para permitir la comunicación de los sistemas abiertos. Un **sistema abierto** es aquél preparado para comunicarse con cualquier otro sistema abierto mediante estándares que gobiernan el formato, contenido y significado de los mensajes enviados y recibidos. Estas reglas se formalizan en lo que se llama **protocolos**. En términos básicos, un protocolo es un acuerdo entre las partes de la forma en que debe desarrollarse la comunicación. Cuando se presenta una mujer a un hombre, ella puede optar por extender la mano. Él, a su vez, puede decidir si la estrecha o la besa según si, por ejemplo, ella es un abogado norteamericano en una junta de negocios o una princesa europea en un baile formal. La violación del protocolo hará que la comunicación sea más difícil, sino es que imposible.

En un nivel más tecnológico, muchas compañías tienen tarjetas de memoria para la PC de IBM. Cuando el CPU desea leer una palabra de la memoria, coloca la dirección y ciertas señales de control en el bus. Se espera que la tarjeta de memoria vea estas señales y responda mediante la colocación de la palabra solicitada en el bus dentro de cierto intervalo de tiempo. Si la tarjeta de memoria sigue el protocolo requerido para el bus, funcionará en forma correcta; si no, funcionará de forma incorrecta.

En forma análoga, para que un grupo de computadoras pueda comunicarse en una red, éstas deben coincidir en los protocolos por utilizar. El modelo OSI distingue entre dos tipos generales de protocolos. Con los protocolos **orientados hacia las conexiones**, antes de intercambiar los datos, el emisor y receptor establecen primero en forma explícita una conexión y es probable que negocien el protocolo por utilizar. Una vez hecho esto, deben terminar la conexión. El teléfono es un sistema de comunicación orientado hacia la conexión. En los protocolos **sin conexión**, no es necesaria una configuración previa. Simplemente, el emisor transmite el primer mensaje cuando está listo. Un ejemplo de comunicación sin conexión es el depósito de una carta en un buzón. Con las computadoras, ambos tipos de comunicación son comunes.

En el modelo OSI, la comunicación se divide hasta en siete niveles o capas, como se muestra en la figura 2-1. Cada capa se encarga de un aspecto específico de la comunicación. De esta forma, el problema se puede dividir en piezas manejables, cada una de las cuales se puede resolver en forma independiente de las demás. Cada capa proporciona una **interfaz** con la otra capa por encima de ella. La interfaz es un conjunto de operaciones que juntas definen el servicio que la capa está preparada para ofrecer a sus usuarios.

En el modelo OSI, cuando el proceso *A* de la máquina 1 desea comunicarse con el proceso *B* de la máquina 2, construye un mensaje y lo trasfiere a la capa de aplicación en su máquina. Esta capa podría ser un procedimiento de biblioteca, por ejemplo, pero también se puede implantar de alguna otra forma (como dentro del sistema operativo, en un circuito coprocesador externo, etc.). Entonces, el software de la capa de aplicación añade un **encabezado** al frente del mensaje y transfiere el mensaje resultante a través de la interfaz entre las capas 6 y 7 hacia la capa de presentación. A su vez, la capa de presentación añade su propio encabezado y transfiere el resultado hacia abajo a la capa de sesión, y así en lo sucesivo. Algunas capas no sólo añaden un encabezado al frente, sino también una parte al final. Al llegar a la parte inferior, la capa física transmite en realidad el mensaje, que ahora se ve como se muestra en la figura 2-2.

Cuando el mensaje llega a la máquina 2, se transfiere hacia arriba, y cada capa lo desmenuza y examina su propio encabezado. Por último, el mensaje llega al receptor, el proceso *B*, que puede responderle siguiendo la ruta inversa. El protocolo de la capa *n* utiliza la información en el encabezado de la capa *n*.

Como ejemplo de la importancia de los protocolos con capas, consideremos la comunicación entre dos compañías, Zippy Airlines y su proveedor, Mushy Meals. Cada mes, el jefe de servicio de pasajeros en Zippy le pide a su secretaria que haga contacto con la secretaria del gerente de ventas en Mushy, para ordenar 10 0000 cajas de pollo. Por lo general, las órdenes se solicitan por medio de la oficina postal. Sin embargo, al deteriorarse

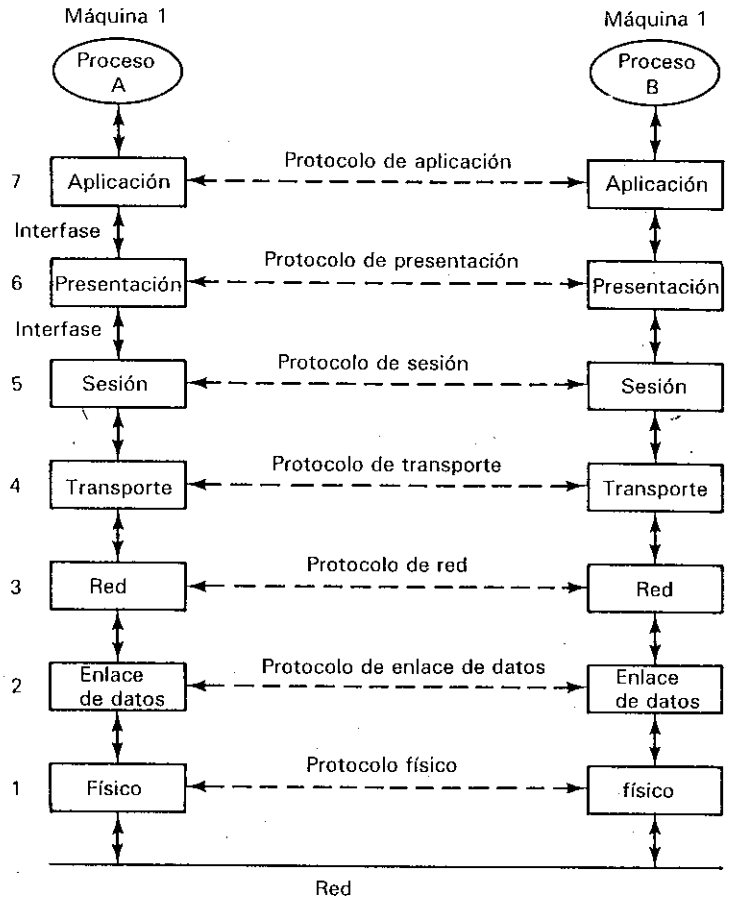


Figura 2-1. Capas, interfaces y protocolos en el modelo OSI.

el servicio postal, las dos secretarias deciden abandonarlo en cierto momento y comunicarse por fax. Lo pueden hacer sin molestar a sus jefes, puesto que su protocolo se refiere a la transmisión física de las órdenes y no a su contenido.

En forma análoga, el jefe del servicio de pasajeros puede decidir no pedir el pollo y pedir el nuevo especial de Mushy, un filete, sin que esa decisión afecte a las secretarias. Lo que debemos observar es que aquí tenemos dos capas, los jefes y las secretarias, y que cada capa tiene su propio protocolo (temas de discusión y tecnología) que se puede cambiar en forma independiente del otro. Esta independencia es precisamente el atractivo del protocolo con capas. Cada uno se puede modificar si se mejora la tecnología, sin que los otros se vean afectados.

En el modelo OSI, no existen dos capas, sino siete, como vimos en la figura 2-1. La colección de protocolos utilizados en un sistema particular se llama una **serie de protocolos** o **pila de protocolos**. En las siguientes secciones examinaremos en forma breve cada una

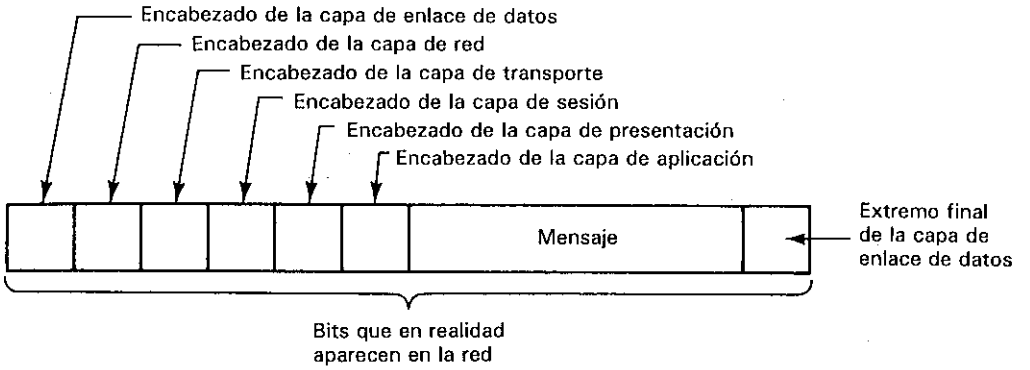


Figura 2-2. Un mensaje típico tal como aparece en la red.

de las capas, desde la parte inferior. Donde sea apropiado, señalaremos algunos de los protocolos utilizados en cada capa.

2.1.1. La capa física

La capa física se preocupa por la transmisión de los ceros y unos. El número de voltios a utilizar para 0 y 1, el número de bits por segundo que se pueden enviar, el hecho de que la transmisión se lleve a cabo en ambas direcciones en forma simultánea, son todos aspectos clave en la capa física. Además, el tamaño y forma del conector en la red (enchufe), así como el número de pins y el significado de cada uno son los temas de interés aquí.

El protocolo de la capa física se encarga de la estandarización de las interfaces eléctricas, mecánicas y de señalización, de forma que, cuando una máquina envíe un bit 0, sea en realidad recibido como un bit 0 y no como un bit 1. Se han desarrollado muchos estándares de la capa física (para medios distintos); por ejemplo, el estándar RS-232-C para las líneas de comunicación serial.

2.1.2. La capa de enlace de datos

La capa física sólo envía bits. Mientras no ocurran errores, todo está bien. Sin embargo, las redes reales de comunicación están sujetas a errores, por lo que es necesario cierto mecanismo para detectarlos y corregirlos. Este mecanismo es la tarea principal de la capa de enlace de datos. Lo que hace es agrupar los bits en unidades, que a veces se llaman **marcos**, y revisar que cada marco se reciba en forma correcta.

La capa de enlace de datos realiza su trabajo por medio de la colocación de un patrón especial de bits al inicio y al final de cada marco, para señalarlos, a la vez que calcula una **suma de verificación,** mediante la suma de todos los bytes del marco de cierta forma. La capa de enlace de datos añade la suma de verificación al marco. Al llegar el marco, el receptor vuelve a calcular la suma de verificación a partir de los datos y compara el resultado con la suma de verificación que sigue después del marco. Si coinciden, se considera que

el marco es correcto y se le acepta. En caso contrario, el receptor pide al emisor que los vuelva a transmitir. Los marcos tienen asignados números secuenciales (en el encabezado), de forma que se pueda saber con exactitud cuál es cuál.

En la figura 2-3 vemos un ejemplo (ligeramente patológico) de un proceso *A* que intenta enviar dos mensajes, 0 y 1, a *B*. En el instante 0, se envía el mensaje 0, pero cuando llega, en el instante 1, un ruido en la línea de transmisión provoca un daño, por lo que la suma de verificación es incorrecta. *B* observa esto y en el instante 2 pide una retransmisión utilizando un mensaje de control. Por desgracia, en el mismo instante, *A* envía el mensaje de datos 1. Cuando *A* recibe la solicitud de la retransmisión, vuelve a enviar 0. Sin embargo, cuando *B* obtiene el mensaje 1, en vez del mensaje solicitado 0, envía el mensaje de control 1 a *A* para quejarse de que deseaba un 0 y no un 1. Cuando *A* ve esto, encoge sus hombros, y envía el mensaje 0 por tercera vez.

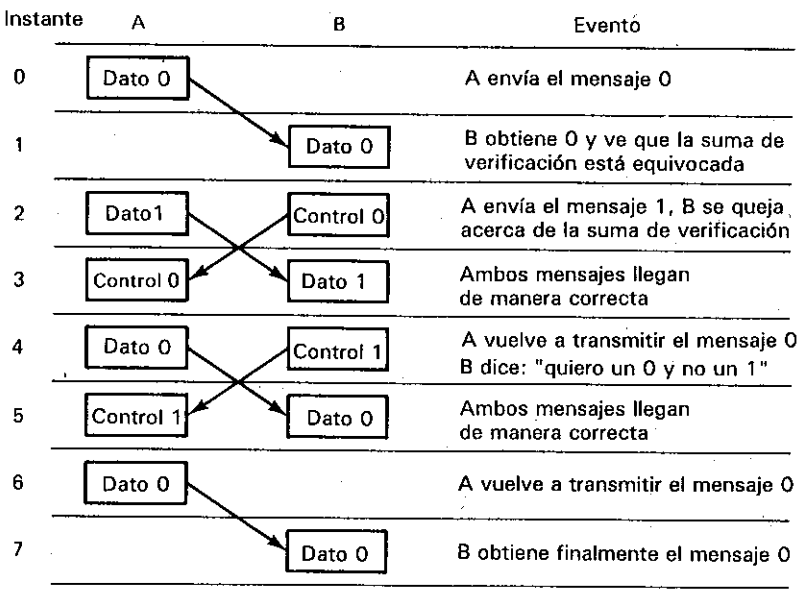


Figura 2-3. Discusión entre un receptor y un emisor en la capa de enlace de datos.

El punto aquí no es si el protocolo de la figura 2-3 es o no muy bueno (no lo es en realidad), sino más bien ilustrar que en cada capa existe una necesidad de discusión entre el emisor y el receptor. Los mensajes típicos son "por favor, retransmite el mensaje *n*", "ya lo retransmití", "no, no lo has hecho", "sí, sí lo hice", "muy bien, allá tú, pero envíalo otra vez", etc. Esta discusión tiene lugar en el campo del encabezado, donde se definen varias solicitudes y respuestas, y donde se pueden proporcionar varios parámetros (tales como el número de marco).

2.1.3. La capa de red

En una LAN, por lo general, no existe la necesidad de que el emisor localice al receptor. Sólo tiene que colocar el mensaje en la red y el receptor lo recibe. Sin embargo, una red de área amplia consta de un gran número de máquinas, cada una de ellas con cierto número de líneas hacia otras máquinas, un poco como un mapa a gran escala, que muestra las ciudades y caminos principales que los conectan. Para que un mensaje llegue del emisor al receptor, tiene que hacer un cierto número de saltos y, en cada uno de ellos, elegir una línea por utilizar. La cuestión de la elección de la mejor ruta se llama **ruteo** y es la tarea principal de la capa de red.

El problema se complica con el hecho de que la ruta más corta no siempre es la mejor. Lo que importa en realidad es la cantidad de retraso en una ruta dada, lo cual, a su vez, se relaciona con la cantidad de tráfico y el número de mensajes formados para la transmisión en las distintas líneas. Así, el retraso puede cambiar con el curso del tiempo. Algunos algoritmos de ruteo intentan adaptarse a cargas cambiantes, mientras que otros se conforman con tomar decisiones con base en promedios a largo plazo.

Dos protocolos en la capa de red tienen un uso amplio, uno orientado hacia las conexiones y otro sin conexión. El orientado hacia las conexiones se llama **X.25** y es favorecido por los operadores de las redes públicas, como las compañías telefónicas y las PTT europeas. En primer lugar, el usuario de X.25 envía una *solicitud de llamada* a un destino, el cual puede aceptar o rechazar la conexión propuesta. Si se acepta la conexión, quien hace la llamada obtiene un identificador de conexión para usarlo en las solicitudes posteriores. En muchos casos, la red escoge una ruta del emisor al receptor durante esta configuración y la utiliza para el tráfico posterior.

El protocolo sin conexión se denomina **IP** (Protocolo Internet) y es parte de la serie de protocolos DoD (Departamento de Defensa de los Estados Unidos). Un **paquete** IP (el término técnico para un mensaje en la capa de red) se puede enviar sin configuración alguna. Cada paquete IP tiene una ruta hacia su destino independiente de los demás paquetes. No se selecciona ruta interna alguna ni se recuerda ésta, como ocurre a menudo con X.25.

2.1.4. La capa de transporte

Los paquetes se pueden perder en el camino del emisor al receptor. Aunque ciertas aplicaciones pueden controlar su propia recuperación de los errores, otras prefieren una conexión confiable. La tarea de la capa de transporte es proporcionar este servicio. La idea es que la capa de sesión pueda enviar un mensaje a la capa de transporte, con la esperanza de que sea entregado sin pérdida alguna.

Al recibir un mensaje de la capa de sesión, la capa de transporte lo divide en pequeñas partes, de forma que cada una se ajuste a un paquete, asigna a cada uno un número secuencial y después los envía. La discusión en el encabezado de la capa de transporte se refiere a los paquetes que han sido enviados, a los paquetes recibidos, el número de paquetes que puede aceptar el receptor y temas similares.

Las conexiones confiables de transporte (que por definición están orientadas hacia la conexión) se pueden construir por arriba de X.25 o IP. En el primer caso, todos los paquetes llegarán en el orden correcto (si es que llegan), pero en el segundo caso es posible que un paquete tome una ruta diferente y llegue primero que el paquete enviado antes de él. El software de la capa de transporte se encarga de ponerlo todo en orden, para mantener la ilusión de que una conexión de transporte es como un gran tubo: sólo hay que colocar los mensajes en él y éstos llegan sin daño alguno, en el orden en el que fueron enviados.

El protocolo de transporte oficial ISO tiene cinco variantes, conocidas como **TP0** hasta **TP4**. Las diferencias se relacionan con el control de los errores y la capacidad de enviar varias conexiones de transporte mediante una conexión X.25. La elección de cuál utilizar depende de las propiedades de la capa de red subyacente.

El protocolo de transporte DoD se llama **TCP** (Transmission Control Protocol **[protocolo para el control de transmisiones)**]. Es similar a TP4. La combinación TCP/IP se utiliza con amplitud en las universidades y en la mayoría de los sistemas UNIX. La serie de protocolos DoD también soportan un protocolo de transporte sin conexión llamado **UDP** (Universal Datagrama Protocol **[protocolo datagrama universal)]**, que en esencia es igual a IP con ciertas adiciones menores. Los programas del usuario que no necesitan un protocolo orientado hacia las conexiones utilizan por lo general UDP.

2.1.5. La capa de sesión

La capa de sesión es en esencia una versión mejorada de la capa de transporte. Proporciona el control del diálogo, con el fin de mantener un registro de la parte que está hablando en cierto momento, y proporciona facilidades en la sincronización. Esto último es útil para permitir a los usuarios que inserten puntos de verificación en las transferencias de gran tamaño, de modo que si ocurre una falla, sólo sea necesario regresar al último punto de verificación, en vez de recorrer todo el camino hasta el principio. En la práctica, pocas aplicaciones están interesadas en la capa de sesión y rara vez se le soporta. Ni siquiera está presente en la serie de protocolo DoD.

2.1.6. La capa de presentación

A diferencia de las capas inferiores, preocupadas por la obtención de los bits del emisor al receptor en forma confiable y eficaz, la capa de presentación se preocupa por el significado de los bits. La mayoría de los mensajes no constan de una cadena aleatoria de bits, sino de información más estructurada, como nombres de personas, sus direcciones, cantidades de dinero, etc. En la capa de presentación es posible definir registros que contengan campos como éstos y que entonces el emisor notifique al receptor que un mensaje contiene un registro particular en cierto formato. Esto facilita la comunicación entre las máquinas con distintas representaciones internas.

2.1.7. La capa de aplicación

La capa de aplicación es en realidad una colección de varios protocolos para actividades comunes, como el correo electrónico, la transferencia de archivos y la conexión entre terminales remotas a las computadoras en una red. Los más conocidos de éstos son el protocolo de correo electrónico X.400 y el servidor de directorios X.500. Ni esta capa ni las dos capas en directo por debajo de ésta serán de interés para nosotros en este libro.

2.2. REDES CON MODO DE TRANSFERENCIA ASÍNCRONA

El mundo OSI bosquejado en la sección anterior fue desarrollado en la década de 1970 e implantado (en cierta medida) en la década de 1980. Los nuevos desarrollos en esta década están superando a OSI, lo cual es cierto en las capas inferiores, controladas por la tecnología. En esta sección revisaremos con brevedad estos avances en las redes, puesto que los futuros sistemas distribuidos seguramente se construirán sobre éstos, y es importante que los diseñadores de sistemas operativos estén conscientes de ellos. Para un tratamiento más completo del estado actual de la tecnología de redes, véase (Kleinrock, 1992; Partridge, 1993, 1994).

En el último cuarto de siglo, las computadoras han mejorado su rendimiento en *muchos* órdenes de magnitud. Las redes no lo han hecho. Cuando ARPANET, predecesor de Internet, se inauguró en 1969, utilizaba líneas de comunicación de 56 Kb por segundo entre los nodos. Éste era el estado de la comunicación en esa época. A finales de la década de 1970 y principios de la de 1980, muchas de estas líneas se reemplazaron con líneas T1, con una velocidad de 1.5 Mb por segundo. En cierto momento, la columna vertebral de la red evolucionó a una red T3, a 45 Mb por segundo, pero la mayor parte de las líneas en Internet siguen siendo T1 o más lentas.

Los nuevos desarrollos están cerca de lograr que 155 Mb por segundo sea el estándar en un extremo, mientras que los principales troncales tienen velocidad de 1 gigabit por segundo o mayor (Catlett, 1992; Cheung, 1992; y Lyles y Swinehart, 1992). Este rápido cambio tendrá un efecto enorme en los sistemas distribuidos, permitiendo que todos los tipos de aplicaciones antes inimaginables sean posibles, pero esto también representa nuevos retos. Describiremos esta nueva tecnología a continuación.

2.2.1. ¿Qué es el modo de transferencia asíncrona?

A fines de la década de 1980, las compañías telefónicas de todo el mundo por fin comenzaron a darse cuenta de que había más en las telecomunicaciones que transmitir la voz en canales analógicos de 4 KHz. Es cierto que las redes de datos, como X.25, existieron durante años, pero eran hijastros y con frecuencia tenían velocidad de 56 o 64 Kb por segundo. Sistemas como Internet fueron considerados como curiosidades académicas, similares a una vaca con dos cabezas en una exhibición de un circo. En la transmisión de la voz analógica era donde estaba la acción (y el dinero).

Cuando las compañías telefónicas decidieron construir redes para el siglo XXI, se enfrentaron a un dilema: el tráfico de las voces es suave y necesita un ancho de banda bajo, pero constante, mientras que el tráfico de datos es explosivo, no necesita por lo general, un ancho de banda (cuando no hay tráfico), pero a veces necesita gran cantidad de recursos durante periodos muy breves. Ni la conmutación de circuitos tradicional (utilizada en la red telefónica de conmutación pública) ni la conmutación de paquetes (utilizada en Internet) eran adecuadas para ambos tipos de tráfico.

Después de mucho estudio, se eligió una forma híbrida con bloques de tamaño fijo sobre circuitos virtuales, como un acuerdo que proporcionaba un rendimiento razonable para ambos tipos de tráfico. Este esquema, llamado **modo de trasferencia asíncrona (ATM, [Asynchronous Transfer Mode])** se ha convertido en un estándar internacional y es probable que juegue un papel principal en los sistemas distribuidos del futuro, tanto de área local como de área amplia. Para mayor información acerca de ATM, véase (Le Boudec, 1992; Minzer, 1989; y Newman, 1994).

El modelo ATM consiste en que un emisor establece primero una conexión (es decir, un circuito virtual) con el (o los) receptor(es). Durante el establecimiento de la conexión, se determina una ruta desde el emisor hasta el (los) receptor(es) y se guarda la información del ruteo en los conmutadores a lo largo del camino. Mediante esta conexión se pueden enviar los paquetes, pero el hardware los separa en pequeñas unidades de tamaño fijo llamadas **celdas**. Las celdas de un circuito virtual dado siguen todas la misma ruta guardada en los conmutadores. Cuando ya no se necesita la conexión, ésta se libera y se purga la información de ruteo de los conmutadores.

Este esquema tiene varias ventajas sobre la conmutación de paquetes y circuitos tradicional. La más importante es que ahora se puede utilizar una red para transportar una composición arbitraria de voces, datos, televisión, videocintas, radio y otras informaciones de manera eficiente, reemplazando lo que antes eran redes separadas (teléfono, X.25, televisión por cable, etc.). Los nuevos servicios, como las videoconferencias para las empresas, también las utilizarán. En todos los casos, lo que ve la red son celdas; no le interesa lo que haya en ellas. Esta integración representa un enorme ahorro en costos y en simplificación, que permitirán a cada hogar o empresa tener un cable (o fibra) de entrada para todas sus necesidades de comunicación e información. También permitirá tener nuevas aplicaciones, como los videos por solicitud, teleconferencias y el acceso a miles de bases de datos remotas.

La conmutación con celdas también se presta a la multitransmisión (una celda que va a varios destinos), una técnica necesaria para transmitir señales de televisión a miles de hogares al mismo tiempo. La conmutación convencional por medio de circuitos, utilizada en el sistema telefónico, no puede controlar esto. Los medios de transmisión, como la televisión por cable pueden hacerlo, pero no pueden controlar el tráfico puntual sin desperdiciar un ancho de banda (transmitiendo de manera eficaz cada mensaje). La ventaja de la conmutación con celdas es que puede controlar la transmisión puntual y la multitransmisión de manera eficaz.

Las celdas de tamaño fijo permiten una rápida conmutación, algo que es mucho más difícil de lograr con los actuales conmutadores de paquetes que guardan los mensajes y los envían más adelante. También eliminan el peligro de que un pequeño paquete se retrase debido a que un paquete enorme está ocupando la línea necesaria. Con la conmutación por celdas, después de transmitir cada una de ellas, se puede enviar una nueva, incluso una nueva que pertenezca a otro paquete.

ATM tiene su propio protocolo de jerarquías, como se muestra en la figura 2-4. La capa física tiene la misma funcionalidad que la capa 1 en el modelo OSI. La capa ATM trabaja con las celdas y el transporte de las mismas, incluyendo el ruteo, de modo que abarca la capa 2 y parte de la capa 3 de OSI. Sin embargo, a diferencia de la capa 2 de OSI, la capa ATM no recupera las celdas perdidas o dañadas. La capa de adaptación controla la separación de los paquetes en celdas y su ensamblaje en el otro extremo, lo que no aparece de manera explícita en el modelo OSI hasta después de la capa 4. El servicio que ofrece la capa de adaptación no es un servicio entre extremos perfectamente confiable, de modo que las conexiones de transporte deben implantarse en las capas superiores; por ejemplo, mediante el uso de las celdas ATM para transportar el tráfico TCP/IP.

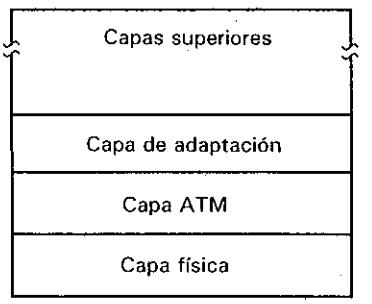


Figura 2-4. El modelo de referencia ATM.

En las siguientes secciones examinaremos las tres capas inferiores de la figura 2.4, desde la inferior hacia arriba.

2.2.2. La capa física ATM

Una tarjeta de adaptación ATM conectada a una computadora puede enviar un flujo de celdas a través de un cable o fibra. El flujo de transmisión debe ser continuo. Cuando no existen datos por enviar, se transmiten celdas vacías, lo que significa que, en la capa física, ATM es en realidad síncrono, no asíncrono. Sin embargo, dentro de un circuito virtual, es asíncrono.

Otra alternativa es que la tarjeta mencionada puede utilizar **SONET** (acrónimo de **red óptica síncrona**) en la capa física, colocando sus celdas en su porción correspondiente de los marcos SONET. La virtud de este método es su compatibilidad con el sistema interno de transmisión de AT&T y otras compañías de transmisión que utilizan SONET. En algunos

países de Europa se dispone de un sistema llamado **SDH (jerarquía digital síncrona)** que tiene un patrón cercano al de SONET.

En SONET, la unidad básica (análoga a un marco T1 de 193 bits) es un arreglo 9×90 de bytes llamado **marco**. De estos 810 bytes, 36 son un exceso, lo que deja 774 útiles. Un marco se transmite cada 125 μseg, para coincidir con la tasa de muestreo estándar del sistema telefónico, de 8 000 muestras por segundo, de modo que la tasa gruesa de transmisión de datos (incluyendo el exceso) es de 51.840 Mb por segundo y la tasa neta (excluyendo el exceso) es de 49.536 Mb por segundo.

Estos parámetros fueron elegidos después de cinco años de tortuosas negociaciones entre las compañías de Estados Unidos, Europa y Japón para controlar el flujo de datos T3 de Estados Unidos (44.736 Mb por segundo) y los estándares utilizados por los demás países. La industria de la computación no jugó un papel importante aquí (un arreglo 9×90 con 36 bytes de exceso no es precisamente algo que propondría un científico de la computación).

El canal básico de 51.840 Mb por segundo se llama **OC-1**. Es posible enviar un grupo de *n* marcos OC-1 como un grupo, que se designa como OC-*n*, cuando se utiliza para *n* canales OC-1 independientes y OC-*n*c (c de concatenado) cuando se utiliza para un canal de alta velocidad. Se han establecido estándares para OC-3, OC-12, OC-48 y OC-192. Los más importantes para ATM son OC-3c, a 155.520 Mb por segundo y OC-12c, a 622.080 Mb por segundo, puesto que es probable que las computadoras produzcan datos con esas tasas en el futuro cercano. Para la transmisión dentro del sistema telefónico, OC-12 y OC-48 son de uso más amplio en la actualidad.

Se dispone en la actualidad de adaptadores SONET OC-3c para computadoras y permitir que éstas produzcan como salida marcos SONET de manera directa. Se espera que pronto aparezcan OC-12c. Puesto que incluso OC-1 es excesivo para un teléfono, es poco probable que muchos audioteléfonos lleguen a utilizar ATM o SONET en directo (en vez de esto, utilizarán ISDN), pero para los videofonos, ATM y SONET son ideales.

2.2.3. La capa ATM

Cuando se desarrollaba ATM, también se desarrollaron dos facciones dentro del comité de estándares. Los europeos querían utilizar celdas de 32 bytes puesto que éstas tenían un retraso lo bastante pequeño como para que los supresores de eco no fueran necesarios en la mayoría de los países europeos. Los norteamericanos, que ya tenían supresores de eco, querían utilizar celdas de 64 bytes debido a su mayor eficiencia para el tráfico de los datos.

El resultado final fue una celda de 48 bytes que nadie quería en realidad. Es demasiado grande para la voz y demasiado pequeña para los datos. Para empeorar las cosas, se agregó un encabezado de 5 bytes, lo que proporcionó una celda de 53 bytes con un campo de datos de 48. Observe que una celda de 53 bytes no concuerda bien con el espacio de SONET de 774 bytes, de modo que las celdas ATM abarquen marcos SONET. Así, se necesitaron dos capas independientes de sincronización: una para detectar el inicio de un marco SONET, y otra para detectar el inicio de la primera celda ATM completa dentro del espacio SONET.

Sin embargo, existe un estándar para empacar las celdas ATM en marcos SONET y toda la capa se puede obtener en hardware.

La organización del encabezado de una celda de una computadora hasta el primer conmutador ATM aparece en la figura 2-5. Por desgracia, la organización de un encabezado de celda entre dos conmutadores ATM es diferente, pues el campo *GFC* es reemplazado por cuatro bits adicionales para el campo VPI. Desde el punto de vista de muchas personas, esto es una desgracia, pues introduce una distinción innecesaria entre las celdas computadora-conmutador y conmutador-conmutador, y por lo tanto en el hardware del adaptador. Ambos tipos de celdas tienen un espacio de 48 bytes después del encabezado.

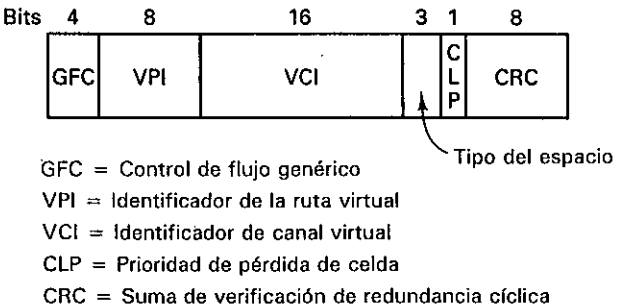


Figura 2-5. Organización del encabezado de una celda usuario-red.

El *GFC* podría utilizarse algún día para el control de flujo, si se llega a un acuerdo acerca de cómo hacerlo. Los campos *VPI* y *VCI* identifican juntos la ruta y el circuito virtual al que pertenece una celda. Las tablas de ruteo a lo largo del camino utilizan esta información para el ruteo. Estos campos se modifican en cada paso a lo largo de la ruta. La finalidad del campo *VPI* es la de agrupar una colección de circuitos virtuales para el mismo destino y permiten que un transportador cambie su ruta sin tener que examinar el campo *VCI*.

El campo *Tipo de espacio* distingue las celdas de los datos de las celdas de control y además identifica varios tipos de celdas de control. El campo *CLP* se puede utilizar para señalar ciertas celdas como menos importantes que otras, de modo que si ocurre un congestionamiento, las menos importantes serán descartadas. Por último, existe una suma de verificación de 1 byte sobre el encabezado (pero no sobre los datos).

2.2.4. La capa de adaptación ATM

A 155 Mb por segundo, una celda puede llegar cada 3 μseg. Pocas, si no es que ninguna CPU de la actualidad puede controlar un exceso de 300 000 interrupciones/segundo. Así, se necesita un mecanismo para permitir que una computadora envíe un paquete y que el hardware de ATM lo descomponga en celdas, transmita las celdas y después los vuelva a ensamblar en el otro extremo, generando una interrupción por paquete por celda, no por celda. Este ensamblaje/re-ensamblaje es tarea de la capa de adaptación. Se espera que la

mayor parte de las tarjetas de adaptación ejecuten la capa de adaptación en la tarjeta y proporcionen una interrupción por cada paquete que llegue, no por cada celda que llegue.

Por desgracia, aquí tampoco concuerdan los autores de estándares. En un principio, las capas de adaptación se definieron para cuatro clases de tráfico:

1. Tráfico con tasa constante de bits (para audio y video).
2. Tráfico con tasa variable de bits pero con retraso acotado.
3. Tráfico de datos orientado por conexiones.
4. Tráfico de datos sin conexión.

Rápidamente se descubrió que las clases 3 y 4 eran en esencia la misma, de modo que se fundieron en una nueva clase, 3/4. En ese momento, la industria de la computación despertó de su letargo y observó que ninguna de las capas de adaptación eran adecuadas para el tráfico de datos, de modo que se produjo AAL 5, para el tráfico entre computadoras (Suzuki, 1994). Su seudónimo, **SEAL (capa de adaptación sencilla y eficiente)** sugiere lo que pensaron sus diseñadores acerca de las otras tres capas AAL. (Para ser justos, debemos señalar que poner de acuerdo en un estándar a las personas de dos industrias con tradiciones muy distintas, telefonía y computadoras sería un logro no trivial.)

Nos centraremos en SEAL, debido a su sencillez. Sólo utiliza un bit en el encabezado ATM, uno de los bits en el campo *Tipo de espacio*. Por lo general, este bit es 0, pero cambia a 1 en la última celda de un paquete. La última celda contiene un final especial en los últimos 8 bytes. En la mayor parte de los casos, se cubrirán (con ceros) el espacio entre el final del paquete y el inicio del final especial. Con SEAL, el destino sólo ensambla las celdas que llegan por cada circuito virtual hasta que encuentra una con el bit del fin de paquete activado. Entonces extrae y procesa el final especial.

El final especial tiene cuatro campos. Los primeros dos tienen una longitud de 1 byte cada uno y no se utilizan. Después viene un campo de 2 bytes con la longitud del paquete y una suma de verificación de 4 bytes sobre el paquete, el espacio cubierto con ceros y el final especial.

2.2.5. Conmutación ATM

Las redes ATM están formadas por cables de cobre o fibras ópticas y conmutadores. La figura 2-6(a) ilustra una red con cuatro conmutadores. Las celdas que parten de cualquiera de las ocho computadoras unidas al sistema se pueden intercambiar con cualquiera de las demás computadoras recorriendo uno o más conmutadores. Cada uno de estos conmutadores tiene cuatro puertos, que se utilizan para entrada o salida.

En la figura 2-6(b) se ilustra el interior de un conmutador genérico. Tiene líneas de entrada y líneas de salida y una **malla de conmutadores** que los conecta. Puesto que una celda debe intercambiarse en 3 μseg (a OC-3) y que pueden llegar de una sola vez tantas celdas como líneas de entrada, la conmutación es esencial.

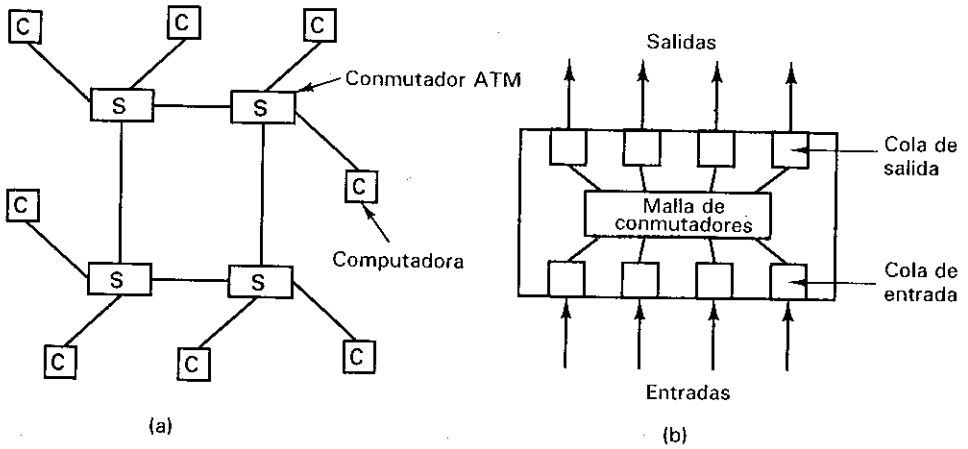


Figura 2-6. (a) Una red de conmutación ATM. (b) Interior de un conmutador.

Cuando llega una celda, se examinan sus campos VPI y VCI. Con base en esto y en la información guardada en el conmutador cuando se estableció el circuito virtual, la celda se envía al puerto de salida correcto. Aunque el estándar permite eliminar las celdas, se requiere que las celdas se entreguen en orden.

Surge un problema cuando dos celdas llegan al mismo tiempo en diferentes líneas de entrada y necesitan ir hacia el mismo puerto de salida. El estándar permite eliminar una de ellas, pero si su conmutador elimina más de una celda en 10^{12}, es poco probable que usted pueda vender muchos conmutadores. Un esquema alternativo es elegir una de ellas al azar y enviarla hacia adelante, deteniendo a la otra para enviarla posteriormente. En la siguiente ronda, se aplica de nuevo este algoritmo. Si dos puertos tienen flujos de celdas para el mismo destino, se construyen colas de entrada sustanciales, bloqueando a las demás celdas detrás de ellas y que desean ir hacia puertos de salida libres. Este problema se conoce como el **bloqueo de encabezado.**

Un diseño diferente de conmutadores copia la celda en una cola asociada con el buffer de salida y le permite esperar ahí, en vez de mantenerlo en el buffer de entrada. Este método elimina el bloqueo de encabezado y proporciona un mejor rendimiento. También es posible que un conmutador tenga una pila de buffers que puedan utilizarse para el almacenamiento de entrada y de salida. Otra posibilidad adicional es crear un buffer en el lado de la entrada, pero permitiendo conmutar la segunda o tercera celda en la fila, aunque la primera no se pueda.

Se han propuesto y probado muchos otros diseños de conmutadores. Éstos incluyen conmutadores con división del tiempo, que utilizan memoria compartida, buses o anillos, así como conmutadores con división del espacio, con una o más rutas entre cada entrada y cada salida. Algunos de estos conmutadores se analizan en (Ahmadi y Denzel, 1989; Anderson *et al.*, 1993; Gopal *et al.*, 1992; Pattavina, 1993; Rooholamini *et al.*, 1994; y Zegura, 1993).

2.2.6. Algunas implicaciones del ATM para sistemas distribuidos

La disponibilidad de redes ATM a 155 Mb por segundo, 622 Mb por segundo y potencialmente a 2.5 Gb por segundo, tiene ciertas implicaciones importantes para el diseño de los sistemas distribuidos. En gran parte, los efectos se deben principalmente al enorme ancho de banda disponible de manera súbita, más que a las propiedades específicas de las redes ATM. Los efectos son más evidentes en los sistemas distribuidos de área amplia.

Para comenzar, consideremos el envío de un archivo de un Mbit a través de Estados Unidos y que espera el reconocimiento de que arribó de manera correcta. La velocidad de la luz en un alambre de cobre o una fibra óptica es cercana a 2/3 de la velocidad de la luz en el vacío, por lo que tarda cerca de 15 milisegundos en recorrer Estados Unidos. A 64 Kb por segundo, los bits tardan 15.6 segundos en su recorrido, de modo que el retraso adicional de 30 milisegundos por el viaje redondo no representa mucho más. A 622 Mb por segundo, el recorrido del archivo tarda 1/622 de segundo, aproximadamente 1.6 milisegundos. En el mejor de los casos, la respuesta puede regresar después de 31.6 milisegundos, tiempo durante el cual la línea estuvo inactiva durante 30 milisegundos, o 95% del total. Al mejorar las velocidades, el tiempo de respuesta tiende asintóticamente a 30 milisegundos y la fracción del ancho de banda disponible en el circuito virtual que puede ser utilizado tiende a 0. Para mensajes menores de 1 Mb por segundo, comunes en los sistemas distribuidos, esto es aún peor. La conclusión es que para los sistemas distribuidos de área amplia y alta velocidad, se necesitarán nuevos protocolos y arquitecturas de sistema que enfrenten la latencia en muchas aplicaciones, en particular en las interactivas.

Otro problema es el control del flujo. Supongamos que tenemos un archivo en realidad extenso, como una videocinta de 10 GB. El emisor comienza la transmisión a 622 Mb por segundo y los datos comienzan a fluir hacia el receptor. Es probable que el receptor no tenga un buffer de 10 GB a la mano, por lo que envía una celda de regreso que diga ALTO. Cuando la celda ALTO haya regresado al emisor, 30 milisegundos más tarde, casi 20 Mbits de datos están en camino. Si la mayor parte de ellos se pierde debido a un espacio inadecuado en el buffer, tendrá que transmitirse de nuevo. Con un protocolo de ventana tradicional regresamos a la situación anterior, es decir, si el emisor sólo puede enviar 1 Mbit y tiene que esperar un reconocimiento, el circuito virtual está inactivo 95% del tiempo. Otra alternativa es colocar gran capacidad de almacenamiento en los conmutadores y en las tarjetas de los adaptadores, pero con un costo mayor. Otra posibilidad más es el control de la tasa, donde el emisor y el receptor se ponen de acuerdo de antemano en la cantidad de bits/segundo que puede transmitir el emisor. El control de flujo y el control de congestionamientos en las redes ATM se analizan en (Eckberg, 1992; Hong y Suda, 1991; y Trajkovic y Golestani, 1992). En (Nikolaidis y Onvural, 1992) se proporciona una bibliografía con más de 250 referencias relativas al rendimiento en redes ATM.

Un método diferente para trabajar con la (por el momento, enorme) latencia de 30 milisegundos consiste en enviar algunos bits, detener el proceso de envío y ejecutar algo más mientras se espera la respuesta. El problema con esta estrategia es que las computadoras son cada vez más baratas, de modo que para muchas aplicaciones, cada proceso tiene su

propia computadora, por lo que no hay nada más que ejecutar. El desperdicio del tiempo de CPU no es importante, pues es barato, pero es claro que el paso de 64 Kb por segundo a 622 Mb por segundo no ha traído una ganancia de 10 000 % en el rendimiento, aún en aplicaciones limitadas por la comunicación.

El efecto del retraso transcontinental puede exhibirse de varias maneras. Por ejemplo, si cierto programa de aplicación en Nueva York realiza 20 solicitudes secuenciales a un servidor en California para obtener una respuesta, el retraso de 600 milisegundos será observado por el usuario, ya que a las personas les molestan los retrasos superiores a 200 milisegundos.

Otra alternativa consiste en mover el propio cálculo hacia la máquina en California y permitir que cada tecla oprimida por el usuario represente una celda independiente que recorra el país y regrese para desplegarse en la pantalla. Al hacer esto, se sumarían 60 milisegundos a cada golpe de tecla, que nadie notaría. Sin embargo, este razonamiento nos conduce rápidamente a abandonar la idea de un sistema distribuido y colocar todo el cómputo en un lugar, con usuarios remotos. De hecho, hemos construido un enorme sistema centralizado de tiempo compartido, donde sólo los usuarios están distribuidos.

Una observación relacionada con las propiedades específicas de ATM es el hecho de que los conmutadores pueden eliminar celdas al verse congestionados. La eliminación incluso de una celda probablemente significaría la espera de cierto tiempo para que todo el paquete sea retransmitido. Para los servicios que necesitan una tasa uniforme, como cuando se ejecuta cierta música, esto podría representar un problema. (Por desgracia, el oído es mucho más sensible que el ojo a una entrega irregular.)

Como consecuencia de éstos y otros problemas, aunque las redes de alta velocidad, en general, y ATM en lo particular, introducen nuevas oportunidades, no será sencillo aprovecharlas. Se necesitará una buena cantidad de investigación antes de saber cómo trabajar con ellas de manera eficiente.

2.3. EL MODELO CLIENTE-SERVIDOR

Aunque las redes ATM serán importantes en el futuro, por el momento son demasiado caras para la mayor parte de las aplicaciones, por lo que regresaremos a las redes más convencionales. A primera vista, los protocolos con capas a lo largo de las líneas OSI se ven como una buena forma de organizar un sistema distribuido. En efecto, un emisor establece una conexión (un entubamiento de bits) con el receptor y entonces bombea los bits, que llegan sin error, en orden, al receptor. ¿Qué podría estar mal en esto?

Mucho. Para comenzar, observemos la figura 2-2. La existencia de todos esos encabezados genera un costo excesivo. Cada vez que se envía un mensaje, se debe procesar cerca de media docena de capas, cada una de las cuales genera y añade un encabezado en el camino hacia abajo o elimina y examina el encabezado en el camino hacia arriba. Todo este trabajo toma tiempo. En las redes de área amplia, donde el número de bits/segundo que se pueden enviar es por lo general bajo (a menudo tan poco como 64K bits/segundo), este costo excesivo no es serio. El factor limitante es la capacidad de las líneas, e incluso

con todo el manejo de los encabezados, los CPU son muy rápidos como para mantener las líneas en ejecución a toda velocidad. Así, un sistema distribuido de área amplia podría utilizar el protocolo OSI o el protocolo TCP/IP sin pérdida en el rendimiento (ya de por sí magro). Con ATM, podrían surgir problemas más serios.

Sin embargo, para un sistema distribuido basado en una LAN, el costo excesivo del protocolo es con frecuencia sustancial. Así, es tal el tiempo de CPU desperdiciado al ejecutar los protocolos, que el resultado efectivo a través de la LAN es con frecuencia una pequeña fracción de lo que la LAN puede hacer. Como consecuencia, la mayoría de los sistemas distribuidos basados en LAN no utilizan de modo alguno los protocolos con capas; o bien, si lo hacen, sólo utilizan un subconjunto de toda una pila de protocolos.

Además, el modelo OSI sólo se enfoca hacia un pequeño aspecto del problema: llevar los bits del emisor hacia el receptor (y en los niveles superiores, su significado). No dice nada acerca de la forma de estructurar al sistema distribuido. Se necesita algo adicional.

2.3.1. Clientes y servidores

Ese algo adicional es el modelo cliente-servidor que se presentó en el capítulo anterior, con la idea de estructurar el sistema operativo como un grupo de procesos en cooperación, llamados **servidores**, que ofrezcan servicios a los usuarios, llamados **clientes**. Las máquinas de los clientes y servidores ejecutan por lo general el mismo micronúcleo y ambos se ejecutan como procesos del usuario, como ya hemos visto. Una máquina puede ejecutar un proceso o varios clientes, varios servidores o combinaciones de ambos.

Para evitar un gasto excesivo en los protocolos orientados hacia la conexión como OSI o TCP/IP, lo usual es que el modelo cliente-servidor se base en un **protocolo solicitud/respuesta** sencillo y sin conexión. El cliente envía un mensaje de solicitud al servidor para

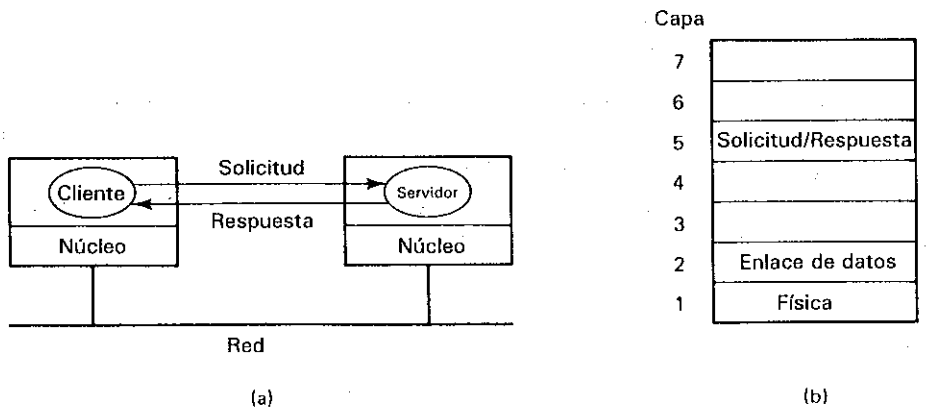


Figura 2-7. El modelo cliente-servidor. Aunque en realidad la trasferencia de mensajes la realizan los núcleos, este dibujo simplificado se utilizará en caso de que no surjan ambigüedades.

pedir cierto servicio (por ejemplo, la lectura de un bloque de cierto archivo). El servidor hace el trabajo y regresa los datos solicitados o un código de error para indicar la razón por la cual un trabajo no se pudo llevar a cabo, como se muestra en la figura 2-7(a).

La principal ventaja de la figura 2-7(a) es su sencillez. El cliente envía un mensaje y obtiene una respuesta. No se tiene que establecer una conexión sino hasta que ésta se utilice. El mensaje de respuesta sirve como reconocimiento de la solicitud.

Después de la sencillez viene otra ventaja: la eficiencia. La pila del protocolo es más corta y por tanto más eficiente. Si todas las máquinas fuesen idénticas, sólo se necesitarían tres niveles de protocolos, como se muestra en la figura 2-7(b). Las capas física y de enlace de datos se encargan de llevar los paquetes del cliente al servidor y viceversa. Esto siempre lo maneja el hardware; por ejemplo, un circuito Ethernet o un anillo de elementos. No se necesita un ruteo y tampoco se establecen conexiones, por lo que no se utilizan las capas 3 y 4. La capa 5 es el protocolo solicitud/respuesta. Define el conjunto de solicitudes válidas y el conjunto de respuestas válidas a estas solicitudes. No existe administración de la sesión, puesto que éstas no existen. Tampoco se utilizan las capas superiores.

Debido a esta estructura tan sencilla, se pueden reducir los servicios de comunicación que presta el (micro)núcleo; por ejemplo, reducirse a dos llamadas al sistema, una para el envío de mensajes y otra para la recepción. Estas llamadas al sistema se pueden pedir a través de procedimientos de biblioteca, como *send(dest,&mptr)* y *receive(addr, &mptr)*. El primero envía el mensaje al que apunta *mptr* a un proceso que se identifica como *dest* y provoca que quien hace la llamada se bloquee hasta que se envíe el mensaje. El segundo hace que quien hizo la llamada se bloquee hasta que reciba un mensaje. Cuando llega un mensaje, éste se copia en el buffer al que apunta *mptr* y quien hizo la llamada se desbloquea. El parámetro *addr* determina la dirección a la cual escucha el receptor. Existen muchas variantes de estos dos procedimientos y sus parámetros. Los analizaremos en una sección posterior de este capítulo.

2.3.2. Un ejemplo cliente-servidor

Para tener una idea mejor del funcionamiento de los clientes y servidores, en esta sección presentaremos un esquema de un cliente y un servidor de archivos en C. Tanto el cliente como el servidor deben compartir algunas definiciones, de modo que reunimos éstas en un archivo de nombre *header.h*, el cual se muestra en la figura 2-8. Tanto el cliente como el servidor lo incluyen mediante el uso del enunciado

```
#include <header.h>
```

Este enunciado tiene el efecto de provocar que un preprocesador inserte de manera literal todo el contenido de *header.h* en el programa fuente justo antes de que el compilador comience la compilación.

```
/* Definiciones necesarias para los clientes y servidores. */
#define MAX_PATH        255         /* longitud máxima del nombre de un archivo */
#define BUF_SIZE       1024         /* cantidad de datos que se pueden
                                       transferir de una sola vez */
#define FILE_SERVER     243         /* dirección en la red del servidor de archivos *
/
Definiciones de las operaciones permitidas.
#define CREATE            1         /* crea un nuevo archivo */
#define READ              2         /* lee una parte de un archivo y la regresa */
#define WRITE             3         /* escribe una parte de un archivo */
#define DELETE            4         /* elimina un archivo existente */
/* Códigos de error. */
#define OK                0         /* operación desarrollada en forma correcta */
#define E_BAD_OPCODE     -1         /* solicitud de una operación desconocida */
#define E_BAD_PARAM      -2         /* error en un parámetro */
#define E_IO             -3         /* error en disco u otro error en E/S */

/* Definición del formato del mensaje. */
struct message {
   long source;                     /* identidad del emisor */
   long dest;                       /* identidad del receptor */
   long opcode;                     /* operación: CREATE, READ, etcétera */
   long count;                      /* número de bytes por transferir */
   long offset;                     /* lugar del archivo donde comienza la lectura o
                                       la escritura */
   long extra 1;                    /* campo adicional */
   long extra 2;                    /* campo adicional */
   long result;                     /* resultado de la operación de la que se informa */
   char name [MAX_PATH];            /* nombre del archivo en el cual se opera */
   char data [BUF_SIZE];            /* datos por leer o escribir */
```

Figura 2-8. El archivo *header.h* que utilizan el cliente y el servidor.

Analizaremos primero *header.h*. Comienza con la definición de dos constantes, *MAX_PATH y BUF_SIZE*, las cuales determinan el tamaño de dos arreglos necesarios en el mensaje. La primera indica el número de caracteres que puede tener un nombre de archivo (es decir, el nombre de una ruta de acceso como */usr/ast/books/opsys/chapter1.t*). La segunda fija la cantidad de datos que se pueden leer o escribir en una operación, al establecer el tamaño del buffer. La siguiente constante, *FILE_SERVER*, proporciona la dirección en la red del servidor de archivos, de forma que los clientes le puedan enviar mensajes.

El segundo grupo de constantes define los números de operación, necesarios para garantizar que el cliente y el servidor queden de acuerdo en el código que representará un READ, el código que representará un WRITE, etcétera. Aquí sólo hemos mostrado cuatro, pero lo usual en un sistema real es que sean más.

Cada respuesta contiene un código de resultado. Si la operación tiene éxito, es frecuente que este código contenga información útil (como el número de bytes que se leen en realidad). Si no existe un valor por regresar (como cuando se crea un archivo), se utiliza el valor *OK*.

Si por alguna razón la operación fracasa, el código de resultado indica dicha razón, mediante códigos tales como *E_BAD_OPCODE, E_BAD_PARAM*, etcétera.

Por último, llegamos a la parte más importante de *header.h*, la definición del propio mensaje. En nuestro ejemplo, es una estructura de 10 campos. Todas las solicitudes del cliente al servidor utilizan este formato, al igual que lo hacen las respuestas. En un sistema real, tal vez no se tenga un formato fijo de mensaje (puesto que no se necesitan todos los campos en todos los casos), pero esto hace más sencilla la explicación. Los campos *source* y *dest* identifican al emisor y al receptor, respectivamente. El campo *opcode* es una de las operaciones definidas antes; es decir, *CREATE, READ, WRITE* o *DELETE*. Los campos *count* y *offset* se utilizan como parámetros y otros dos campos, *extra1* y *extra2* se definen con el fin de tener un espacio para más parámetros en caso de que el servidor tenga un desarrollo posterior. El campo *result* no se utiliza para las solicitudes del cliente al servidor, sino que conserva el valor resultado en las respuestas del servidor al cliente. Por último, tenemos dos arreglos. El primero, *name*, contiene el nombre del archivo al que se tiene acceso. El segundo, *data*, contiene los datos que se envían de regreso como respuesta a un READ o los datos que se envían al servidor en un WRITE.

Analicemos el código, que se muestra en la figura 2-9. En (a) tenemos el servidor; en (b) tenemos el cliente. El servidor es directo. El ciclo principal comienza con una llamada a *receive* para obtener un mensaje de solicitud. El primer parámetro identifica a quien hizo la llamada, mediante su dirección; el segundo apunta a un buffer de mensajes donde se puede guardar el mensaje que llegue. El procedimiento de biblioteca *receive* hace un señalamiento al núcleo para suspender al servidor hasta que llegue un mensaje. Cuando llega uno, el servidor continúa y se encarga del tipo opcode. Para cada opcode se llama a un procedimiento distinto. Se dan como parámetros el mensaje de entrada y un buffer para el mensaje de salida. El procedimiento examina el mensaje de entrada *m1* y construye la respuesta en *m2*. También regresa el valor de una función en el campo *result*. Después de terminar *send*, el servidor regresa a la parte superior del ciclo para ejecutar *receive* y esperar a que llegue el siguiente mensaje.

En la figura 2-9(b) vemos un procedimiento para copiar un archivo mediante el servidor. Su cuerpo consta de un ciclo que lee un bloque del archivo fuente y lo escribe en el archivo destino. El ciclo se repite hasta copiar todo el archivo fuente, lo cual se indica mediante un código de retorno de la lectura nulo o negativo.

La primera parte del ciclo se preocupa por construir un mensaje para la operación READ y enviarla al servidor. Después de recibir la respuesta, se entra a la segunda parte del ciclo, la cual toma los datos recién recibidos y los envía de regreso al servidor en la forma de un WRITE al archivo de destino. Los programas de la figura 2-9 son sólo un bosquejo del código. Se han omitido muchos detalles. Por ejemplo, no se muestran los procedimientos *do_xxx* (los que en realidad desarrollan el trabajo) y no se lleva a cabo una verificación de los errores. Aun así, debe quedar clara la idea general de la interacción del cliente y el servidor. En las secciones siguientes analizaremos con mayor detalle algunos de los aspectos relacionados con los clientes y los servidores.

```
# include <header .h>
void main(void)
{
   struct message m1, m2                    /* mensajes de entrada y salida */
   int r;                                   /* código del resultado */

   while (1) {                              /* el servidor ejecuta un ciclo infinito */
         receive (FILE_SERVER, &m1)         /* se bloquea en espera de un mensaje */
         switch (m1.opcode) {               /* se encarga del tipo de solicitud */
                  case CREATE:              r = do_crear (&m1, &m2);        break
                  case READ:                r = do_leer (&m1, &m2);         break
                  case WRITE:               r = do_escribir (&m1, &m2);     break
                  case DELETE:              r = do_borrar (&m1, &m2);       break
                  default:                  r = E_BAD_OPCODE;
         }
         m2.result = r;                     /* regresa el resultado al cliente */
         send(m1.source, &m2);              /* envía la respuesta */
   }
}
                                            (a)
#include <header .h>
int copy (char *scr, char *dst)             /* procedimiento para el copiado de un archivo
{                                           mediante el servidor */
   struct message m1;                       /* buffer de mensajes */
   long position;                           /* posición actual del archivo */
   long client = 110;                       /* dirección del cliente */

   initialize ( );                          /* se prepara para la ejecución */
   position = 0;
   do {
         /* Obtiene un bloque de datos del archivo fuente. */
         m1. opcode = READ;                 /* la operación es una lectura */
         m1. offset = position;             /* posición actual en el archivo
         m1. count = BUF_SIZE;              /* número de bytes por leer
         strcpy (&m1. name, src);           /* copia al mensaje el nombre del archivo por leer */
         send (FILE_SERVER, &m1);           /* envía el mensaje al servidor de archivos */
         receive (client, &m1);             /* se bloquea en espera de la respuesta */

         /* Escribe los datos recién recibidos en el archivo destino. */
         m1. opcode = WRITE                 /* la operación es una escritura */
         m1. offset = position;             /* posición actual en el archivo */
         m1. count = m1.result;             /* número de bytes por escribir */
         strcpy (&m1.name, dst);            /* copia al mensaje el nombre del archivo en el
         send(FILE_SERVER, &m1);            /* que se va a escribir */
         receive (client, &m1)              /* envía el mensaje al servidor de archivos */
         position += m1. result;            /* se bloquea en espera de la respuesta */
   } while (m1.result > 0);                 /* m1.result es el número de bytes escritos */
   return(m1. result >= 0 ? OK:             /* continuar el ciclo hasta terminar */
   m1. result);                             /* regresa OK o el código del error */

}                                           (b)
```

Figura 2-9. (a) Ejemplo de servidor. (b) Un procedimiento cliente, el cual utiliza dicho servidor para copiar un archivo.

2.3.3. Direccionamiento

Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste. En el ejemplo de la sección anterior, la dirección del servidor sólo se coloca dentro de *header.h* como constante. Aunque esta estrategia podría funcionar en un sistema en particular sencillo, por lo general se necesita una forma más compleja de direccionamiento. En esta sección describiremos algunos de los aspectos relacionados con este concepto.

En nuestro ejemplo, el servidor de archivos tiene asignada una dirección numérica (243), pero en realidad no hemos determinado lo que esto significa. En particular, ¿se refiere a una máquina específica, o a un proceso específico? Si se refiere a una máquina determinada, el núcleo emisor puede extraerla de la estructura de mensajes y utilizarla como dirección física para enviar el paquete al servidor. Todo lo que tiene que hacer el núcleo emisor es construir un marco indicando el 243 como la dirección de enlace de los datos y colocar el marco en la LAN. La tarjeta de interfaz del servidor verá el marco, reconocerá el 243 como propia dirección y la aceptará.

Si sólo existe un proceso en ejecución en la máquina destino, el núcleo sabrá qué hacer con el mensaje recibido (dárselo al único proceso en ejecución). Sin embargo, ¿qué ocurre si existen varios procesos en ejecución en la máquina destino? ¿Cuál de ellos obtiene el mensaje? El núcleo no tiene forma de decidir. En consecuencia, un esquema que utilice las direcciones en la red para la identificación de los procesos indica que sólo se puede ejecutar un proceso en cada máquina. Aunque esta limitación no es fatal, a veces es una seria restricción.

Otro tipo de sistema de direccionamiento envía mensajes a los procesos en vez de a las máquinas. Aunque este método elimina toda ambigüedad acerca de quién es el verdadero receptor, presenta el problema de cómo identificar los procesos. Un esquema común consiste en utilizar nombres con dos partes, para especificar tanto la máquina como el proceso. Así, 243.4, 4@243 o algo similar designa el proceso 4 de la máquina 243. El núcleo utiliza el número de máquina para que el mensaje sea entregado de manera correcta a la máquina adecuada, a la vez que utiliza el número de proceso en esa máquina para determinar a cuál proceso va dirigido el mensaje. Una característica importante de este método es que cada máquina puede numerar sus procesos a partir de 0. No se necesitan una coordinación global, puesto que nunca existe confusión entre el proceso 0 de la máquina 243 y el proceso 0 de la máquina 199. El primero es 243.0 y el segundo es 199.0. Este esquema se ilustra en la figura 2-10(a).

Una ligera variación de este esquema de direccionamiento utiliza *machine.local-id* en vez de *machine.process*. El campo *local-id* es por lo general, un entero aleatorio de 16 o 32 bits (o el siguiente de una serie). Se inicia un proceso, por lo general un servidor, mediante una llamada al sistema para indicarle al núcleo que desea escuchar a *local-id*. Más tarde, cuando se envía un mensaje dirigido a *machine.local-id*, el núcleo sabe a cuál proceso debe dar el mensaje. Por ejemplo, la mayoría de la comunicación en el UNIX de Berkeley utiliza este método, con direcciones Internet de 32 bits, las cuales se utilizan para especificar máquinas, y números de 16 bits para los campos *local-id*.

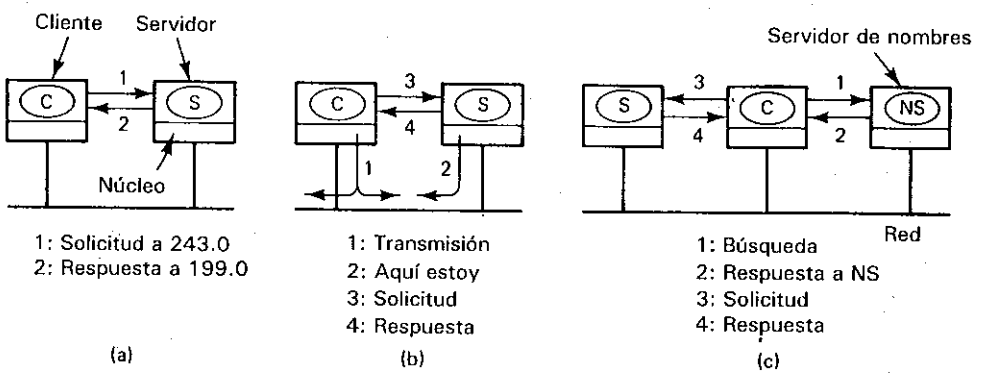


Figura 2-10. (a) Direccionamiento *machine.process*. (b) Direccionamiento de procesos con transmisión. (c) Búsqueda de direccionamiento por medio de un servidor de nombres.

Sin embargo, el direccionamiento *machine.process* está lejos de ser el ideal. En especial porque no es transparente, puesto que es claro que el usuario debe conocer la posición del servidor, y la transparencia es uno de los principales objetivos de la construcción de un sistema distribuido. Para ver la importancia de esto, supongamos que el servidor de archivos se ejecuta por lo general en la máquina 243, pero que un día ésta se descompone. La máquina 176 está disponible, pero los programas compilados anteriormente con *header.h* tienen integrado el número 243, por lo que no funcionarán si el servidor no está disponible. Es claro que no es recomendable esta situación.

Otro método consiste en asignarle a cada proceso una dirección que no contenga un número de máquina. La forma de lograr esto es mediante un asignador centralizado de direcciones a los procesos que mantenga tan sólo un contador. Al recibir una solicitud de dirección, el asignador regresa el valor actual del contador y lo incrementa en uno. La desventaja de este esquema es que los componentes centralizados no se pueden extender a los grandes sistemas, por lo cual hay que evitarlo.

Existe otro método más para la asignación de identificadores a los procesos, el cual consiste en dejar que cada proceso elija el propio identificador de un gran espacio de direcciones dispersas, como el espacio de enteros binarios de 64 bits. La probabilidad de que dos procesos elijan el mismo número es muy pequeña y el método puede utilizarse en sistemas más extensos. Sin embargo, aquí también existe un problema: ¿Cómo sabe el núcleo emisor a cuál máquina enviar el mensaje? En una LAN que soporte transmisiones, el emisor puede transmitir un **paquete especial de localización** con la dirección del proceso destino. Puesto que es un paquete de transmisión, será recibido por todas las máquinas de la red. Todos los núcleos verifican si la dirección es la suya y, en caso que lo sea, regresa un mensaje **aquí estoy** con su dirección en la red (número de máquina). El núcleo emisor utiliza entonces esa dirección y la captura, para evitar el envío de otra transmisión la próxima vez que necesite al servidor. Este método se muestra en la figura 2-10(b).

Aunque este esquema es transparente, incluso con ocultamiento, la transmisión provoca una carga adicional en el sistema. Esta carga se evita mediante una máquina adicional para la asociación a alto nivel (es decir, en ASCII) de los nombres de servicios con las direcciones de las máquinas, como se muestra en la figura 2-10(c). Al utilizar este sistema, se hace referencia a los procesos del tipo de los servidores mediante cadenas en ASCII, las cuales se introducen en los programas y no los números en binario de las máquinas o los procesos. Cada vez que se ejecute un cliente, en su primer intento por utilizar un servidor, el cliente envía un mensaje de solicitud a un servidor especial de asociaciones, el cual se conoce a menudo como **servidor de nombres**, para pedir el número de la máquina donde se localiza en ese momento el servidor. Una vez obtenida la dirección, se puede enviar la solicitud de manera directa. Como en el caso anterior, las direcciones se pueden ocultar.

En resumen, tenemos los métodos siguientes para el direccionamiento de los procesos:

1. Integrar *machine.number* al código del cliente.
2. Dejar que los procesos elijan direcciones al azar; se localizan mediante transmisiones.
3. Colocar los nombres en ASCII de los servidores en los clientes; buscarlos al tiempo de la ejecución.

Cada uno de estos métodos tiene problemas. El primero no es transparente; el segundo genera carga adicional en el sistema y el tercero necesita un componente centralizado, el servidor de nombres. Por supuesto, el servidor de nombres puede duplicarse, pero esto presenta problemas asociados con el mantenimiento de la consistencia.

Un método por completo distinto utiliza un hardware especial. Se deja que los procesos elijan su dirección en forma aleatoria. Sin embargo, en vez de localizarlos mediante transmisiones a toda la red, los circuitos de interfaz de la red se diseñan de modo que permitan a los procesos guardar direcciones de procesos en ellos. Entonces, los marcos usarían direcciones de procesos en vez de direcciones de máquinas. Al recibir cada marco, el circuito de interfaz de la red sólo tendría que examinar el marco para ver si el proceso destino se encuentra en su máquina. En caso afirmativo, aceptaría el marco; en caso negativo, no se aceptaría.

2.3.4. Primitivas con bloqueo *vs.* sin bloqueo

Las primitivas de trasferencia de mensajes descritas hasta el momento reciben el nombre de **primitivas con bloqueo** (a veces llamadas **primitivas síncronas**). Cuando un proceso llama a *send*, especifica un destino y un buffer dónde enviar ese destino. Mientras se envía el mensaje, el proceso emisor se bloquea (es decir, se suspende). La instrucción que sigue a la llamada a *send* no se ejecuta sino hasta que el mensaje se envía en su totalidad, como se muestra en la figura 2-11(a). De manera análoga, una llamada a *receive* no regresa el control sino hasta que en realidad se reciba un mensaje y éste se coloque en el buffer de mensajes adonde apunta el parámetro. En *receive*, el proceso se suspende hasta que llega

un mensaje, incluso aunque tarde varias horas. En ciertos sistemas, el receptor puede especificar de quiénes quiere recibir mensajes, en cuyo caso permanece bloqueado hasta que llegue un mensaje del emisor especificado.

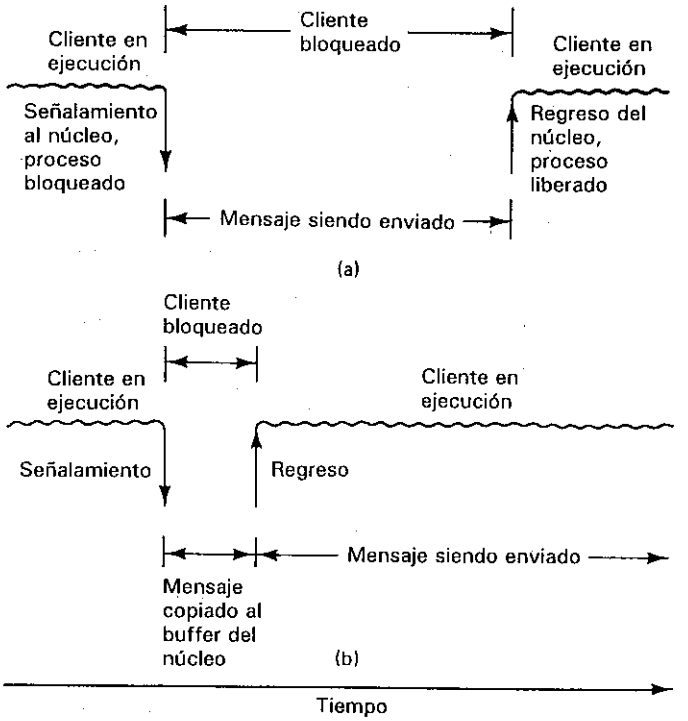


Figura 2-11. (a) Una primitiva send con bloqueo. (b) Una primitiva send sin bloqueo.

Una alternativa a las primitivas con bloqueo son las **primitivas sin bloqueo** (a veces llamadas **primitivas asíncronas**). Si *send* no tiene bloqueo, regresa de inmediato el control a quien hizo la llamada, antes de enviar el mensaje. La ventaja de este esquema es que el proceso emisor puede continuar su cómputo en forma paralela con la transmisión del mensaje, en vez de tener inactivo al CPU (suponiendo que ningún otro proceso sea ejecutable). La elección entre las primitivas con o sin bloqueo la hacen por lo general los diseñadores del sistema (es decir, se dispone de una primitiva o de la otra), aunque en algunos cuantos sistemas se dispone de ambas y los usuarios pueden elegir su favorita.

Sin embargo, la ventaja de desempeño que ofrecen las primitivas sin bloqueo se ve afectada por una seria desventaja: el emisor no puede modificar el buffer de mensajes sino hasta que el mensaje haya sido enviado. Las consecuencias de que el proceso escriba sobre el mensaje durante la transmisión son demasiado graves. Peor aún, el proceso emisor no tiene idea de cuándo termine la transmisión, por lo que no sabe cuándo será seguro volver a utilizar el buffer. Es difícil que evite utilizarlo por siempre.

Existen dos formas para salir del problema. La primera solución es que el núcleo copie el mensaje a un buffer interno del núcleo y que entonces permita al proceso que continúe, como se muestra en la figura 2-11(b). Desde el punto de vista del emisor, este esquema es el mismo que el de una llamada con bloqueo: tan pronto como recupera el control, es libre de volver a utilizar el buffer. Por supuesto, el mensaje no ha sido enviado todavía, pero el emisor no se preocupa por este hecho. La desventaja del método es que cada mensaje de salida debe ser copiado desde el espacio del usuario al espacio del núcleo. Con muchas interfaces de red, de todas formas el mensaje deberá copiarse posteriormente a un buffer de transmisión en hardware, de modo que, en esencia, la primera copia se desperdicia. La copia adicional puede reducir el desempeño del sistema en forma considerable.

La segunda solución es interrumpir al emisor cuando se envíe el mensaje, para informarle que el buffer está de nuevo disponible. No se requiere de una copia, lo que ahorra tiempo, pero las interrupciones a nivel usuario hacen que la programación sea truculenta, difícil y sujeta a condiciones de competencia, lo que la hace irreproducible. La mayoría de los expertos coinciden en que, a pesar de que este método es muy eficiente y permite un máximo paralelismo, las desventajas tienen mayor peso sobre las ventajas: es difícil escribir de forma correcta los programas que se basan en interrupciones y es casi imposible depurarlos cuando están incorrectos.

En ciertas ocasiones, la interrupción se puede disfrazar, al iniciar un nuevo hilo de control (analizados en el capítulo 4), dentro del espacio de direcciones del emisor. Aunque esto es un poco más limpio que una interrupción en bruto, es mucho más complicado que la comunicación síncrona. Si sólo se dispone de un hilo de control, las opciones son:

1. *Send* con bloqueo (CPU inactivo durante la transmisión de los mensajes).
2. *Send* sin bloqueo, con copia (se desperdicia el tiempo del CPU para la copia adicional).
3. *Send* sin bloqueo, con interrupción (dificulta la programación).

En condiciones normales, la primera opción es la mejor. No maximiza el paralelismo, pero es fácil de comprender e implantar. No requiere el manejo de buffers en el núcleo. Además, como se puede ver al comparar la figura 2-11(a) con la figura 2-11(b), el mensaje saldrá más rápido si no se necesita una copia. Por otro lado, si son esenciales el procesamiento y la transmisión traslapados para alguna aplicación, la mejor opción es el *send* sin bloqueo con copia.

Por cierto, queremos señalar que algunos autores utilizan un criterio distinto para distinguir las primitivas síncronas de las asíncronas (Andrews, 1991). Desde nuestro punto de vista, la diferencia esencial entre una primitiva síncrona y una asíncrona es si el emisor puede volver a utilizar el buffer de mensajes en forma inmediata después de recuperar el control sin miedo de revolver el *send*. El momento en que el mensaje llega al receptor es irrelevante.

Desde otro punto de vista, una primitiva síncrona es aquella donde el emisor se bloquea hasta que el receptor ha aceptado el mensaje y el reconocimiento regresa al emisor. Todo lo demás es asíncrono en este esquema. Existe completo acuerdo en que si el emisor recupera el control antes de que se copie o envíe el mensaje, la primitiva es asíncrona. De manera análoga, todos coinciden en que si el emisor se bloquea hasta que el receptor reconoce el mensaje, tenemos una primitiva síncrona.

El desacuerdo se presenta en los casos intermedios (cuando el mensaje se copia, o se copia y envía, pero sin reconocimiento). Los diseñadores de sistemas operativos tienden a preferir nuestro punto de vista, puesto que su interés está en el manejo de buffers y la transmisión de mensajes. Los diseñadores de los lenguajes de programación tienden a preferir la otra definición, puesto que esto es lo que importa a nivel del lenguaje.

Así como *send* puede tener o no bloqueo, también *receive*. Un *receive* sin bloqueo sólo le indica al núcleo la localización del buffer y regresa el control de manera casi inmediata. De nuevo, ¿cómo sabe el que hizo la llamada cuándo se llevó a cabo la operación? Una forma es proporcionar una primitiva explícita *wait* que permita al receptor bloquearse cuando lo desee. Otra alternativa (o un agregado a *wait*) es que los diseñadores proporcionen una primitiva *test* que permita hacer un muestreo del núcleo para verificar su estado. Una variante de esta idea es una primitiva *conditional_receive*, que puede obtener un mensaje o señalar una falla, pero en cualquier caso regresa el control de manera inmediata, o después de un cierto intervalo de tiempo. Por último, también en este caso, se pueden utilizar las interrupciones para señalar el fin del proceso. En general, se prefiere la versión de *receive* con bloqueo, que es más sencilla.

Si se dispone de varios hilos de control dentro de un mismo espacio de direcciones, la llegada de un mensaje puede provocar la creación espontánea de un hilo. Regresaremos a este aspecto después de revisar los hilos en el capítulo 4.

Un tema muy relacionado con las llamadas con/sin bloqueo es el de los tiempos de espera. En un sistema donde *send* llama a un bloqueo y no existe respuesta, el emisor se bloqueará por siempre. Para evitar esta situación en ciertos sistemas, quien hace la llamada puede especificar un intervalo para esperar una respuesta. Si nada llega en ese tiempo, la llamada *send* termina con un estado de error.

2.3.5. Primitivas almacenadas en buffer *vs.* no almacenadas

Así como los diseñadores de sistemas pueden elegir entre las primitivas con o sin bloqueo, también pueden elegir entre las primitivas almacenadas en buffer o no almacenadas. Las primitivas descritas hasta ahora son en esencia **primitivas no almacenadas**. Esto significa que una dirección se refiere a un proceso específico, como en la figura 2-9. Una llamada *receive(addr,&m)* indica al núcleo de la máquina donde se ejecuta ésta que el proceso que llamó escucha a la dirección *addr* y que está preparada para recibir un mensaje enviado a esa dirección. Se dispone de un buffer de mensajes, al que apunta *m*, con el fin de capturar el mensaje por llegar. Cuando el mensaje llega, el núcleo receptor lo copia al buffer y elimina el bloqueo del proceso receptor. El uso de una dirección para hacer referencia a un proceso específico se muestra en la figura 2-12(a).

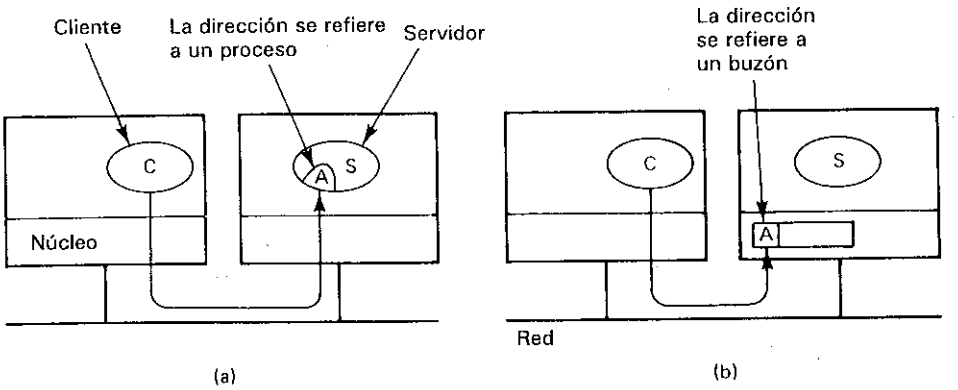


Figura 2-12. (a) Trasferencia de mensajes sin almacenamiento en buffer. (b) Transferencia de mensajes con buffers.

Este esquema funciona bien, mientras el servidor llame a *receive* antes de que el cliente llame a *send*. La llamada a *receive* es el mecanismo que indica al núcleo del servidor la dirección que utiliza el servidor y la posición dónde colocar el mensaje que está por llegar. El problema surge cuando *send* se lleva a cabo antes de *receive*. ¿Cómo sabe el núcleo del servidor cuál de sus procesos (si existe alguno) utiliza la dirección en el mensaje recién llegado? ¿Cómo sabe dónde copiar el mensaje? La respuesta es sencilla: no lo sabe.

Una estrategia de implantación consiste sólo en descartar el mensaje, dejar que el cliente espere y confiar en que el servidor llame a *receive* antes de que el cliente vuelva a transmitir. Este método se puede implantar con facilidad, pero con algo de mala suerte, el cliente (o más probable, el núcleo del cliente) deberá intentar varias veces antes de tener éxito. Peor aún, si fracasa un número suficiente de intentos consecutivos, el núcleo del cliente se podría dar por vencido y concluir erróneamente que el servidor se ha descompuesto o que la dirección no es válida.

De manera similar, supongamos que dos o más clientes utilizan el servidor de la figura 2-9(a). Después que el servidor ha aceptado un mensaje de uno de ellos, deja de escuchar a su dirección hasta que termina su trabajo y regresa al principio del ciclo para volver a llamar a *receive*. Si tarda un poco en hacer el trabajo, los demás clientes podrían realizar varios intentos de envíos y alguno de ellos se podría rendir, según los valores de su cronómetro de retransmisión y su impaciencia.

El segundo método para enfrentar este problema es que el núcleo receptor mantenga pendientes los mensajes por un instante, sólo para prevenir que un *receive* adecuado se realice en poco tiempo. Siempre que llegue un mensaje "no deseado", se inicia un cronómetro. Si el tiempo expira antes de que ocurra un *receive* apropiado, el mensaje se descarta.

Aunque este método reduce la probabilidad de que un mensaje se pierda, presenta el problema de almacenar y manejar los mensajes que van llegando en forma prematura. Se necesitan los buffers y tienen que ser asignados, liberados y en general, manejados. Una forma conceptual sencilla de enfrentar este manejo de los buffers es definir una nueva estructura de datos llamada **buzón**. Un proceso interesado en recibir mensajes le indica al núcleo que cree un buzón para él y especifica una dirección en la cual busca los paquetes de la red. Así, todos los mensajes que lleguen con esa dirección se colocan en el buzón. La llamada a *receive* elimina ahora un mensaje del buzón o se bloquea (si se utilizan primitivas con bloqueo) si no hay un mensaje presente. De esta manera, el núcleo sabe qué hacer con los mensajes que lleguen y tiene un lugar para colocarlos. Esta técnica se conoce a menudo como **primitiva con almacenamiento en buffers**, y se ilustra en la figura 2-12(b).

En primera instancia, parece que los buzones eliminan las condiciones de competencia provocadas por el hecho de que los mensajes se descarten y los clientes se den por vencidos. Sin embargo, los buzones son finitos y pueden ocuparse en su totalidad. Cuando llega un mensaje a un buzón por completo ocupado, el núcleo se enfrenta de nuevo a la elección de mantener el mensaje pendiente por un momento, en espera de que al menos un mensaje sea extraído del buzón a tiempo, o bien descartar dicho mensaje. Éstas son precisamente las mismas opciones que teníamos en el caso sin almacenamiento en buffers. Aunque también hemos reducido la probabilidad de problemas, no lo hemos eliminado e incluso no hemos podido cambiar su naturaleza.

En ciertos sistemas, se dispone de otra opción: no dejar que un proceso envíe un mensaje si no existe espacio para su almacenamiento en el destino. Para que este esquema funcione, el emisor debe bloquearse hasta que obtenga de regreso un reconocimiento, el cual indica que el mensaje ha sido recibido. Si el buzón está por completo ocupado, el emisor puede hacer un respaldo y suspenderse de manera retroactiva, como si el planificador hubiera decidido suspenderlo justo *antes* de que intentara enviar el mensaje. Cuando haya un espacio disponible en el buzón, se hará que el emisor intente de nuevo.

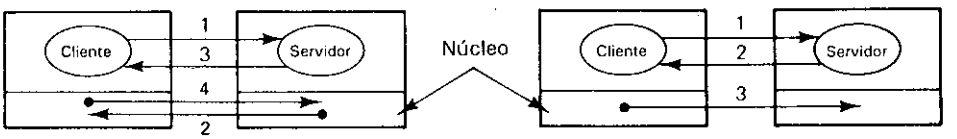
2.3.6. Primitivas confiables *vs.* no confiables

Hasta aquí, hemos supuesto de manera implícita que, cuando un cliente envía un mensaje, el servidor lo recibirá. Como es usual, la realidad es más compleja que nuestro modelo abstracto. Los mensajes se pueden perder, lo cual afecta la semántica del modelo de transferencia de mensajes. Supongamos que se utilizan las primitivas por bloqueo. Cuando un cliente envía un mensaje, se le suspende hasta que el mensaje ha sido enviado. Sin embargo, cuando vuelve a iniciar, no existe garantía alguna de que el mensaje haya sido entregado. El mensaje podría haberse perdido.

Existen tres distintos enfoques de este problema. El primero consiste en volver a definir la semántica de *send* para hacerla no confiable. El sistema no da garantía alguna acerca de la entrega de los mensajes. La implantación de una comunicación confiable se deja por completo en manos de los usuarios. La oficina de correos funciona de esta manera. Cuando usted

deposita una carta en un buzón, la oficina de correos hace lo mejor (más o menos) por entregarla pero no promete nada.

El segundo método exige que el núcleo de la máquina receptora envíe un reconocimiento al núcleo de la máquina emisora. Sólo cuando se reciba este reconocimiento, el núcleo emisor liberará al proceso usuario (cliente). El reconocimiento va de un núcleo al otro; ni el cliente ni el servidor ven alguna vez un reconocimiento. De la misma forma que la solicitud de un cliente a un servidor es reconocida por el núcleo del servidor, la respuesta del servidor de regreso al cliente es reconocida por el núcleo del cliente. Así, una solicitud de respuesta consta de cuatro mensajes, como se muestra en la figura 2-13(a).



1. Solicitud (del cliente al servidor)
2. Reconocimiento (de núcleo a núcleo)
3. Respuesta (del servidor al cliente)
4. Reconocimiento (de núcleo a núcleo)

1. Solicitud (del cliente al servidor)
2. Respuesta (del servidor al cliente)
3. Reconocimiento (de núcleo a núcleo)

Figura 2-13. (a) Mensajes reconocidos en forma individual. (b) La respuesta se utiliza como reconocimiento de la solicitud. Observe que los reconocimientos se manejan totalmente dentro de los núcleos.

El tercer método aprovecha el hecho de que la comunicación cliente-servidor se estructura como solicitud del cliente al servidor, seguida de una respuesta del servidor al cliente. En este método, el cliente se bloquea después de enviar un mensaje. El núcleo del servidor no envía de regreso un reconocimiento sino que la misma respuesta funciona como tal. Así, el emisor permanece bloqueado hasta que regresa la respuesta. Si tarda demasiado, el núcleo emisor puede enviar de nuevo la solicitud para protegerse contra la posibilidad de pérdida del mensaje. Este método se muestra en la figura 2-13(b).

Aunque la respuesta funciona como un reconocimiento de la solicitud, no existe un reconocimiento por la respuesta. El hecho de que esta omisión sea seria o no depende de la naturaleza de la solicitud. Por ejemplo, si el cliente pide al servidor que lea un bloque de un archivo y la respuesta se pierde, todo lo que tiene que hacer el cliente es repetir la solicitud y el servidor enviará de nuevo el bloque. No existe ningún daño y se pierde poco tiempo.

Por otro lado, si la solicitud requiere de un cómputo extenso por parte del servidor, estaría mal descartar la respuesta antes de que el servidor estuviera seguro de que el cliente haya recibido la respuesta. Por esta razón, algunas veces se utiliza un reconocimiento del núcleo del cliente al núcleo del servidor. Hasta no recibir este paquete, el *send* del servidor no termina y permanece bloqueado (si se utilizan primitivas con bloqueo). En todo caso, si se pierde la respuesta y la solicitud se transmite de nuevo, el núcleo del servidor puede ver si la solicitud es una anterior y sólo envía la respuesta de nuevo sin despertar al servidor. Así, en ciertos sistemas, la respuesta se reconoce y en otros, no [véase la figura 2-13(b)].

La mediación entre la figura 2-13(a) y la figura 2-13(b) que funciona a menudo es como sigue. Al llegar una solicitud al núcleo del servidor, se inicia un cronómetro. Si el servidor envía la respuesta muy rápido (antes que termine el tiempo del cronómetro), ésta funciona como el reconocimiento. Si el tiempo del cronómetro se termina, se envía un reconocimiento separado. Así, en la mayoría de los casos, sólo se necesitan dos mensajes, pero si se lleva a cabo una solicitud complicada, se utiliza un tercero.

2.3.7. Implantación del modelo cliente-servidor

En las secciones anteriores analizamos cuatro aspectos del diseño: direccionamiento, bloqueo, almacenamiento en buffers y confiabilidad, cada uno con distintas opciones. Las principales alternativas se resumen en la figura 2-14. Para cada elemento enumeramos tres posibilidades. Una aritmética simple muestra que existen $3^4 = 81$ combinaciones. No todas ellas son igual de buenas. Sin embargo, justo en esta área (trasferencia de mensajes), los diseñadores de sistemas tienen amplio margen en la elección de un conjunto (o varios conjuntos) de primitivas de comunicación.

Elemento	Opción 1	Opción 2	Opción 3
Direccionamiento	Número de máquina	Direcciones ralas de procesos	Búsqueda de nombres en ASCII por medio del servidor
Bloqueo	Primitivas con bloqueo	Sin bloqueo, con copia al núcleo	Sin bloqueo, con interrupciones
Almacenamiento en buffers	No usar el almacenamiento en buffers, descartar los mensajes inesperados	Sin almacenamiento en buffers, mantenimiento temporal de los mensajes inesperados	Buzones
Confiabilidad	No confiable	Solicitud-Reconocimiento-Respuesta-Reconocimiento	Solicitud-Respuesta-Reconocimiento

Figura 2-14. Cuatro aspectos de diseño para las primitivas de comunicación y algunas de las principales opciones disponibles.

Mientras que los detalles de implantación de la transferencia de mensajes dependen en cierta medida de las opciones elegidas, es posible hacer algunos comentarios generales acerca de la implantación, protocolos y software. Para comenzar, virtualmente todas las redes tienen un tamaño máximo de paquete, por lo general, de a lo más unos cuantos miles de bytes. Los mensajes que exceden esta cantidad deben dividirse en varios paquetes y enviarse de manera independiente. Algunos de estos paquetes se pierden o mezclan e incluso pueden llegar en orden equivocado. Para enfrentarse a este problema, por lo general basta asignar un número de mensaje a cada uno de ellos y colocarlo dentro de cada paquete perteneciente al mismo, junto con un número secuencial que dé el orden de los paquetes.

Sin embargo, todavía hay que resolver un aspecto: el uso de los reconocimientos. Una estrategia es el reconocimiento de cada paquete individual. Otra es agradecer sólo los mensajes completos. La primera tiene la ventaja de que si se pierde un paquete, sólo hay que retransmitirlo, pero tiene la desventaja de que se necesitan más paquetes en la red. La segunda tiene la ventaja de menos paquetes, pero la desventaja de una recuperación más compleja cuando se pierde uno de ellos (puesto que una espera por parte del cliente requiere de la retransmisión de todo el mensaje). La elección requiere en gran medida de la tasa de pérdidas de la red que se utilice.

Otro aspecto interesante es el protocolo subyacente utilizado en la comunicación cliente-servidor. La figura 2-15 muestra seis tipos de paquetes que se utilizan comúnmente para la implantación de los protocolos cliente-servidor. El primero es el paquete REQ, utilizado para enviar un mensaje de solicitud de un cliente a un servidor. (Para hacer más sencilla esta exposición, el resto de esta sección supondremos que cada mensaje cabe dentro de un paquete.) El siguiente es el paquete REP, que regresa los resultados del servidor al cliente. Después viene el paquete ACK, el cual se utiliza en protocolos confiables para confirmar la recepción correcta de un paquete anterior.

Código	Tipo de paquete	De	A	Descripción
REQ	Solicitud	Cliente	Servidor	El cliente desea servicio
REP	Respuesta	Servidor	Cliente	Respuesta del servidor al cliente
ACK	Reconocimiento	Cual-quiera	Algún otro	El paquete anterior que ha llegado
AYA	¿Estás vivo?	Cliente	Servidor	Verifica si el servidor se ha descompuesto
IAA	Estoy vivo	Servidor	Cliente	El servidor no se ha descompuesto
TA	Intenta de nuevo	Servidor	Cliente	El servidor no tiene espacio
AU	Dirección desconocida	Servidor	Cliente	Ningún proceso utiliza esta dirección

Figura 2-15. Tipos de paquetes utilizados en los protocolos cliente-servidor.

Los siguientes cuatro tipos de paquetes no son esenciales, pero son con frecuencia útiles. Consideremos la situación en la que una solicitud se envía con éxito del cliente al servidor y se recibe el reconocimiento. En este instante, el núcleo del cliente sabe que el servidor trabaja en la solicitud. Pero ¿qué ocurre si no regresa una respuesta dentro de un tiempo razonable? ¿Será que dicha solicitud es muy complicada? ¿Se habrá descompuesto el servidor? Para poder distinguir entre estos dos casos, el paquete AYA se utiliza en ciertas ocasiones, de modo que el cliente pueda preguntar al servidor qué es lo que ocurre. Si la respuesta es IAA, el núcleo del cliente sabe que todo está bien y sólo tiene que seguir esperando. Por supuesto, es mucho mejor un paquete REP. Si el AYA no genera respuesta alguna, entonces el núcleo del cliente espera un corto intervalo y lo intenta de nuevo. Si este procedimiento falla más

de un cierto número determinado de veces, el núcleo del cliente se dará por lo general por vencido e informará de una falla al usuario. Los paquetes AYA y IAA también se pueden utilizar incluso en un protocolo en el que los paquetes REQ no sean reconocidos. Ellos permiten al cliente verificar el estado del servidor.

Por último, llegamos a los dos últimos tipos de paquetes, que son útiles en caso de que un paquete REQ no sea aceptado. Existen dos razones por las que podría ocurrir esto y es importante que el núcleo del cliente pueda distinguirlas. Una razón es que el buzón al que se envió la solicitud esté por completo ocupado. Al enviar de regreso este paquete al núcleo del cliente, el núcleo del servidor puede indicar que la dirección es válida y que la solicitud debe repetirse más tarde. La otra razón es que la dirección no pertenezca a ningún proceso o buzón. Su repetición posterior no ayudará en nada.

Esta situación también puede aparecer cuando no se utilice el almacenamiento en buffers y el servidor no esté bloqueado por el momento en una llamada *receive*. Puesto que el hecho de que el núcleo del servidor olvide incluso la existencia de la dirección entre las llamadas a *receive* puede conducir a ciertos problemas, en ciertos sistemas, un servidor puede hacer una llamada cuya única función sea registrar una cierta dirección en el núcleo. De esa forma, al menos el núcleo puede decir la diferencia entre una dirección a la que nadie escucha y una que sólo está equivocada. Entonces puede enviar TA en primer caso y AU en el segundo.

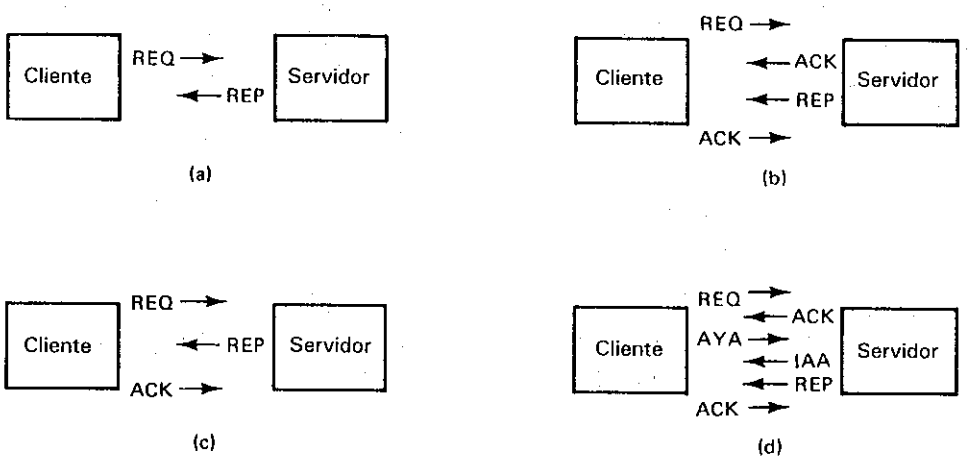


Figura 2-16. Algunos ejemplos de intercambio de paquetes para la comunicación cliente-servidor.

Son posibles muchas secuencias de paquetes. Las más comunes se muestran en la figura 2-16. En la figura 2-16(a), tenemos la solicitud/respuesta directa, sin reconocimientos. En la figura 2-16(b), tenemos un protocolo en el que cada mensaje se reconoce en forma individual. En la figura 2-16(c), vemos a la respuesta actuar como el reconocimiento, lo cual hace que la secuencia tenga de nuevo tres paquetes. Por último, en la figura 2-16(d) vemos un cliente nervioso que verifica si el servidor sigue ahí.

2.4. LLAMADA A UN PROCEDIMIENTO REMOTO (RPC)

Aunque el modelo cliente-servidor es una forma conveniente de estructurar un sistema operativo distribuido, adolece de una enfermedad incurable: el paradigma básico en torno al cual se construye la comunicación es la entrada/salida. Los procedimientos *send* y *receive* están reservados para la realización de E/S. Puesto que la E/S no es uno de los conceptos fundamentales de los sistemas centralizados, el hecho de que sean la base del cómputo distribuido es visto por las personas que laboran en este campo como un grave error. Su objetivo es lograr que el cómputo distribuido se vea como el cómputo centralizado. La construcción de todo en torno de la E/S no es la forma de lograrlo.

Este problema se conoce desde hace tiempo, pero se había hecho poco con respecto de él, hasta que un artículo de Birrell y Nelson (1984) presentó una forma por completo distinta de abordar el problema. Aunque la idea es vitalizante y sencilla (una vez que se ha pensado en ella), las implicaciones suelen ser sutiles. En esta sección examinaremos el concepto, su implantación, sus aspectos fuertes y sus debilidades.

Dicho de forma breve, lo que sugirieron Birrell y Nelson fue permitir a los programas que llamasen a procedimientos localizados en otras máquinas. Cuando un proceso en la máquina *A* llama a un procedimiento en la máquina *B*, el proceso que realiza la llamada a *A* se suspende y la ejecución del procedimiento se realiza en *B*. La información se puede transportar de un lado a otro mediante los parámetros y puede regresar en el resultado del procedimiento. El programador no se preocupa de una transferencia de mensajes o de la E/S. Este método se conoce como **llamada a un procedimiento remoto** o **RPC** (Remote Procedure Call).

Aunque la idea central parece sencilla, existen algunos problemas sutiles. Para comenzar, puesto que el procedimiento que realiza la llamada y el que la recibe se ejecutan en máquinas diferentes, utilizan espacios de direcciones distintos, lo que provoca algunas complicaciones. También se deben trasferir los parámetros y los resultados, lo que se puede complicar, en especial si las máquinas no son idénticas. Por último, se pueden descomponer ambas máquinas y cada una de las posibles fallas puede ser la causa de diversos problemas. Aún así, todos estos problemas se pueden enfrentar y RPC es una técnica de uso amplio que subyace en muchos sistemas operativos distribuidos.

2.4.1. Operación básica de RPC

Para comprender el funcionamiento de RPC, es importante entender en primer lugar el funcionamiento de una llamada convencional a un procedimiento (es decir, en una sola máquina). Consideremos una llamada como

```
count = read(fd, buf, nbytes);
```

en donde *fd* es un entero, *buf* es un arreglo de caracteres y *nbytes* es otro entero. Si la llamada se hace desde el programa principal, la pila se verá como en la figura 2-17(a) antes de la llamada. Para hacer ésta, quien la hace coloca los parámetros en la pila, en orden, el último

en el primer lugar, como se muestra en la figura 2-17(b). (La razón de que los compiladores de C introduzcan los datos en orden inverso tiene que ver con *printf*; al hacerlo así, *printf* siempre localiza su primer parámetro, la cadena de formato.) Después de que *read* termina su ejecución, coloca el valor de regreso en un registro, elimina la dirección y transfiere de nuevo el control a quien hizo la llamada. Este último elimina entonces los parámetros de la pila y regresa a su estado original, como se ve en la figura 2-17(c).

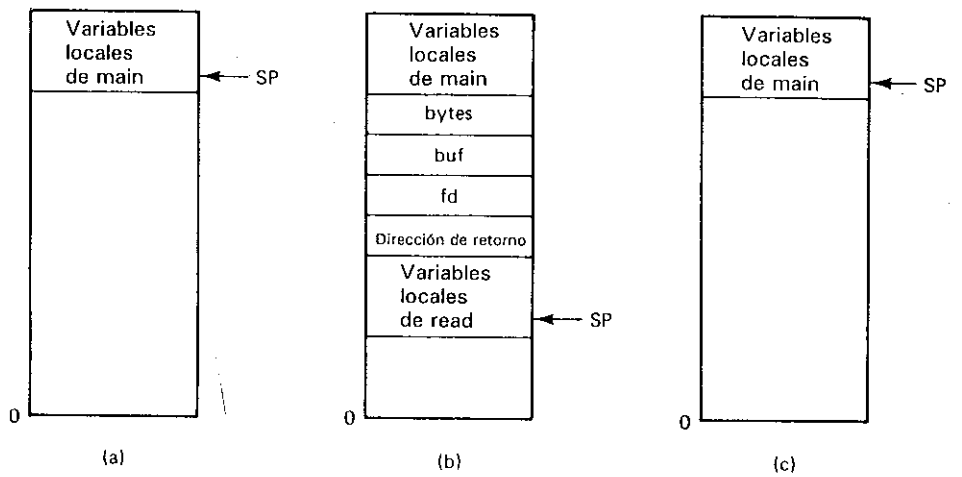


Figura 2-17. (a) La pila antes de la llamada a *read*. (b) La pila mientras el procedimiento llamado está activo. (c) La pila después del regreso a quien hizo la llamada.

Es importante hacer aquí algunas observaciones. La primera es que, en C, los parámetros pueden **llamarse por valor** o **por referencia**. Un parámetro por valor, como *fd* o *nbytes*, solo se copia a la pila como se muestra en la figura 2-17(b). Para el procedimiento que recibe la llamada, un parámetro por valor es tan sólo una variable local ya iniciada. El procedimiento llamado podría modificarla, pero esto no afecta el valor de la variable original en el procedimiento que hizo la llamada.

Un parámetro por referencia en C es un apuntador a una variable (es decir, la dirección de la variable), en lugar del valor de la variable. En la llamada a *read*, el segundo parámetro es un parámetro por referencia, puesto que en C los arreglos siempre se trasfieren por referencia. Lo que se introduce en realidad a la pila es la dirección del arreglo de caracteres. Si el procedimiento llamado utiliza este parámetro para guardar algo en el arreglo de caracteres, esto *sí* modifica el arreglo en el procedimiento que hizo la llamada. La diferencia entre los parámetros llamados por valor o por referencia es importante para RPC, como veremos más adelante.

Existe otro mecanismo para el paso de parámetros, aunque no se utiliza en C. Es la **llamada con copia/restauración**. En este caso, quien recibe la llamada copia la variable

en la pila, como en la llamada por valor, y entonces la copia de regreso después a la llamada, escribiendo sobre el valor original. En la mayoría de los casos, esto tiene el mismo efecto que la llamada por referencia, pero en otros, como cuando el mismo parámetro está presente varias veces en la lista de parámetros, la semántica es distinta.

La decisión relativa al mecanismo que debe utilizarse para el paso de parámetros la toman, por lo general, los diseñadores del sistema y es una propiedad fija del lenguaje. A veces depende del tipo de datos por transferir. Por ejemplo, en C, los enteros y otros tipos escalares se trasfieren siempre por valor, mientras que los arreglos siempre se trasfieren por referencia, como ya hemos visto. Por el contrario, los programadores en Pascal pueden elegir el mecanismo que deseen para cada parámetro. El valor predefinido es por valor, pero los programadores pueden obligar a que el paso sea por referencia, al insertar la palabra reservada **var** antes de los parámetros específicos. Algunos compiladores de Ada® utilizan la copia/restauración para los parámetros **in out**, pero otros utilizan la llamada por referencia. La definición del lenguaje permite cualquiera de las opciones, lo que hace que la semántica parezca un poco confusa.

La idea detrás de RPC es que una llamada a un procedimiento remoto se parezca lo más posible a una llamada local. En otras palabras, queremos que la RPC sea transparente; el procedimiento que hace la llamada no debe ser consciente de que el procedimiento llamado se ejecuta en una máquina distinta, o viceversa. Supongamos que un programa necesita leer ciertos datos de un archivo. El programador coloca una llamada a *read* en el código con el fin de obtener los datos. En un sistema tradicional (con un procesador), el ligador extrae la rutina *read* de la biblioteca y lo inserta en el programa objeto. Es un procedimiento corto, por lo general escrito en lenguaje ensamblador, que coloca los parámetros en registros y después hace una llamada al sistema READ mediante un señalamiento al núcleo. En esencia, el procedimiento *read* es cierto tipo de interfaz entre el código del usuario y el sistema operativo.

Aunque *read* realiza un señalamiento al núcleo, se le llama de la manera usual, al colocar los parámetros en la pila, como se muestra en la figura 2-17. Así, el programador no tiene que saber que en *read* hay gato encerrado.

RPC logra su transparencia de manera análoga. Si *read* es en realidad un procedimiento remoto (por ejemplo, uno que se ejecuta en la máquina del servidor de archivos), se coloca en la biblioteca una versión distinta de *read*, llamada **resguardo del cliente**. Como el original, también se le llama mediante la secuencia de llamada de la figura 2-17. También como el original, éste hace un señalamiento al núcleo. Pero a diferencia del original, no coloca los parámetros en registros y le pide al núcleo que le proporcione datos, sino que empaca los parámetros en un mensaje y le pide al núcleo que envíe el mensaje al servidor, como se muestra en la figura 2-18. Después de la llamada a *send*, el resguardo del cliente llama a *receive* y se bloquea hasta que regrese la respuesta.

Cuando el mensaje llega al servidor, el núcleo lo trasfiere a un **resguardo del servidor** conectado con el servidor real. Por lo general, el resguardo del servidor ya habrá llamado a *receive* y estará bloqueado en espera de que le lleguen mensajes. El resguardo del servidor desempaca los parámetros del mensaje y después llama al procedimiento del servidor de

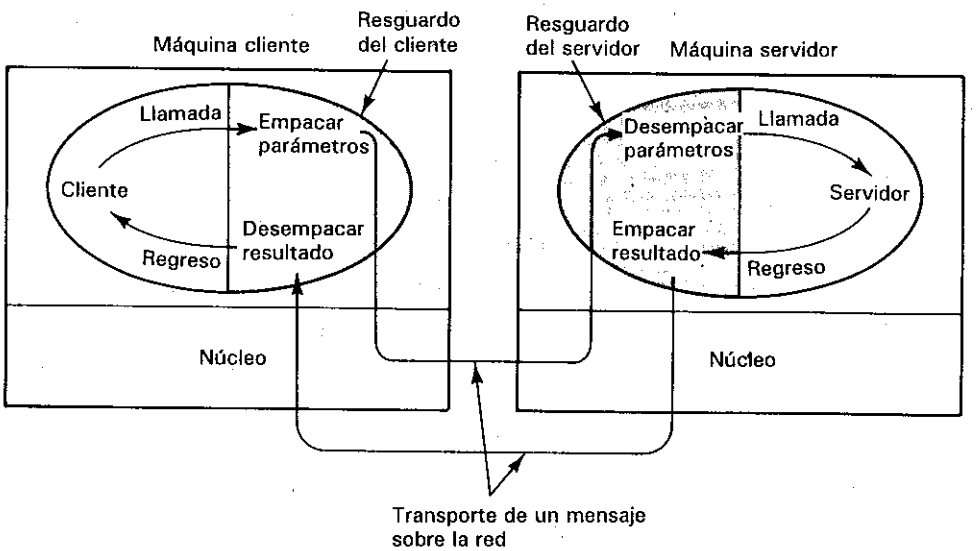


Figura 2-18. Llamadas y mensajes en una RPC. Cada elipse representa un solo proceso, en donde la porción sombreada es el resguardo.

manera usual (es decir, como en la figura 2-17). Desde el punto de vista del servidor, es como si tuviera una llamada directa del cliente: los parámetros y dirección de retorno están todos en la pila donde pertenecen y nada parece fuera de lo común. El servidor lleva a cabo su trabajo y después regresa el resultado a quien hizo la llamada, de forma usual. Por ejemplo, en el caso de *read*, el servidor envía datos al buffer al que apunta el segundo parámetro. El buffer puede ser uno interno del resguardo del cliente.

Cuando el resguardo del servidor recupera el control después de terminar con la llamada, empaca el resultado (el buffer) en un mensaje y llama a *send* para regresarlo al cliente. Entonces regresa a la parte superior del propio ciclo para llamar a *receive*, y espera el siguiente mensaje.

Cuando el mensaje regresa a la máquina cliente, el núcleo ve que está dirigido al proceso cliente (a la parte de resguardo de este proceso, pero eso no lo sabe el núcleo). El mensaje se copia al buffer en espera y el proceso cliente elimina su bloqueo. El resguardo del cliente examina el mensaje, desempaca el resultado, lo copia a quien hizo la llamada y regresa de la manera usual. Cuando el proceso que realizó la llamada obtiene el control después de llamar a *read*, todo lo que sabe es que ahora dispone de los datos. No tiene la menor idea de que el trabajo se realizó de manera remota y no en el núcleo local.

Esta placentera ignorancia por parte del cliente es la belleza de todo el esquema. En lo que respecta al cliente, se tiene acceso a los servicios remotos mediante llamadas ordinarias a procedimientos (es decir, locales) y no mediante las llamadas *send* y *receive* de la figura 2-9. Todo el detalle de la transferencia de mensajes se oculta en dos procedimientos de biblioteca, al igual que los detalles de la realización de interrupciones a las llamadas al sistema se ocultan en las bibliotecas tradicionales.

En resumen, una llamada a un procedimiento remoto se realiza mediante los siguientes pasos:

1. El procedimiento cliente llama al resguardo del cliente de la manera usual.
2. El resguardo del cliente construye un mensaje y hace un señalamiento al núcleo.
3. El núcleo envía el mensaje al núcleo remoto.
4. El núcleo remoto proporciona el mensaje al resguardo del servidor.
5. El resguardo del servidor desempaca los parámetros y llama al servidor.
6. El servidor realiza el trabajo y regresa el resultado al resguardo.
7. El resguardo del servidor empaca el resultado en un mensaje y hace un señalamiento al núcleo.
8. El núcleo remoto envía el mensaje al núcleo del cliente.
9. El núcleo del cliente da el mensaje al resguardo del cliente.
10. El resguardo desempaca el resultado y regresa al cliente.

El efecto neto de todos estos pasos es convertir la llamada local del procedimiento cliente al resguardo del cliente, en una llamada local al procedimiento servidor sin que el cliente o el servidor se den cuenta de los pasos intermedios.

2.4.2. Transferencia de parámetros

La función del resguardo del cliente es tomar sus parámetros, empacarlos en un mensaje y enviarlos al resguardo del servidor. Aunque esto parece directo, no es tan sencillo como aparenta. En esta sección analizaremos algunos de los aspectos relacionados con el paso de parámetros en los sistemas RPC. El empacamiento de parámetros en un mensaje se llama **ordenamiento de parámetros**.

El ejemplo más sencillo es considerar un procedimiento remoto, *sum(i,j)*, que toma dos parámetros enteros y regresa su suma aritmética. (En la práctica, uno no haría este procedimiento de manera remota debido a su costo excesivo, pero nos servirá como ejemplo.) La llamada a *sum*, con los parámetros 4 y 7, aparece en la parte izquierda del proceso cliente de la figura 2-19. El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje de la forma que se indica. También coloca en el mensaje el nombre o número del procedimiento por llamar, puesto que el servidor podría soportar varias llamadas y se le tiene que indicar cuál de ellas se necesita.

Cuando el mensaje llega al servidor, el resguardo examina éste para ver cuál procedimiento necesita y entonces lleva a cabo la llamada apropiada. Si el servidor también soporta los procedimientos remotos *difference, product* y *quotient*, el resguardo del servidor podría tener dentro un enunciado de conmutación para seleccionar el procedimiento por llamar, según el primer campo del mensaje. La llamada real del resguardo al servidor se parece mucho a la llamada original del cliente, excepto que los parámetros son variables iniciadas a partir del mensaje recibido, en vez de ser constantes.

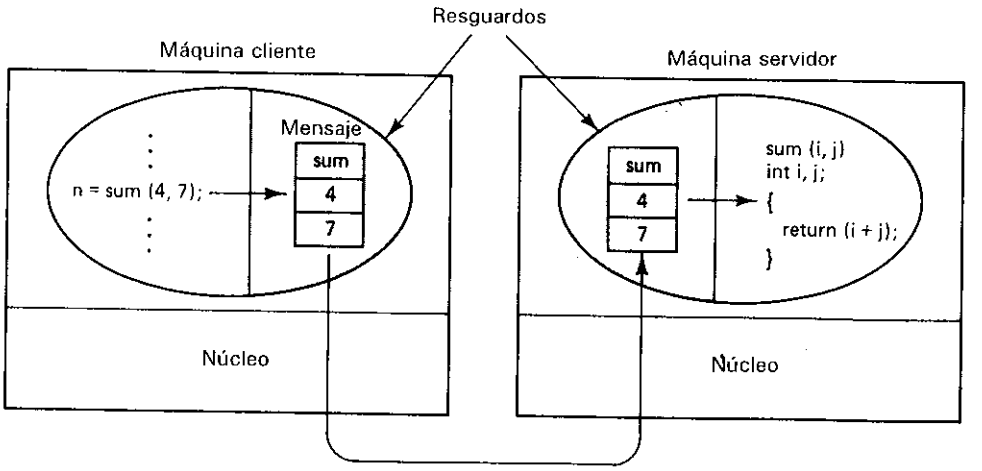


Figura 2-19. Cálculo remoto de *sum* (4,7).

Cuando el servidor termina su labor, su resguardo recupera el control. Toma el resultado proporcionado por el servidor y lo empaca en un mensaje. Este mensaje se envía de regreso al resguardo del cliente, que lo desempaca y regresa el valor al procedimiento cliente (el cual no se muestra en la figura).

Si las máquinas cliente y servidor son idénticas y todos los parámetros y resultados son de tipo escalar, como enteros, caracteres o booleanos, este modelo funciona bien. Sin embargo, en un sistema distribuido de gran tamaño, es común tener distintos tipos de máquinas. Cada una tiene a menudo su propia representación de los números, caracteres y otros elementos. Por ejemplo, las mainframes de IBM utilizan el código de caracteres EBCDIC, mientras que las computadoras personales de IBM utilizan ASCII. En consecuencia, no es posible pasar un parámetro carácter de una PC de IBM cliente a una mainframe IBM servidor mediante el sencillo esquema de la figura 2-19; el servidor interpretará el carácter de manera incorrecta.

Aparecen otros problemas similares con la representación de enteros (complemento a 1 o complemento a 2) y de manera particular, en los números de punto flotante. Además, existe un problema todavía más irritante, puesto que ciertas máquinas, como la Intel 486, numeran sus bytes de derecha a izquierda, mientras que otras, como la Sun SPARC, los numeran en el orden contrario. El formato de Intel se llama **little endian** (partidarios del extremo menor) y el de SPARC **big endian** (partidarios del extremo mayor), en alusión a los políticos en *Los viajes de Gulliver* que se declararon la guerra por una discusión en torno a por dónde debería romperse el extremo del huevo (Cohen, 1981). Como ejemplo, consideremos un servidor con dos parámetros, un entero y una cadena de 4 caracteres. Cada parámetro necesita una palabra de 32 bits. La figura 2-20(a) muestra la apariencia de la parte del parámetro construida por un resguardo del cliente en una Intel 486. La primer palabra contiene el parámetro entero, 5 en este caso y el segundo contiene la cadena "JILL".

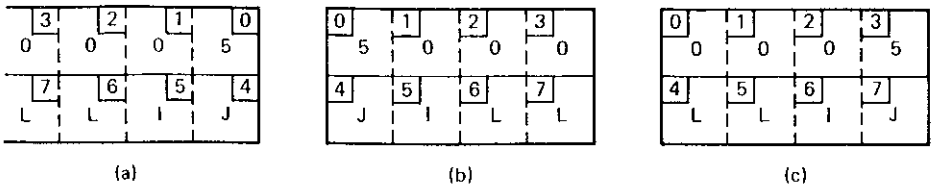


Figura 2-20. (a) El mensaje original en la 486. (b) El mensaje después de ser recibido en la SPARC. (c) El mensaje después de ser invertido. Los números pequeños en las cajas indican la dirección de cada byte.

Puesto que los mensajes se transfieren en un byte a la vez (de hecho, un bit a la vez) en la red, el primer byte que se envía es el primero en llegar. En la figura 2-20(b) mostramos cómo luciría el mensaje de la figura 2-20(a) si fuese recibido por una SPARC, la cual numera sus bytes de forma que el byte 0 esté a la izquierda (byte de mayor orden) en vez de a la derecha (byte de menor orden) como lo hacen todos los circuitos Intel. Cuando el resguardo del cliente lee los parámetros en las direcciones 0 y 4, respectivamente, encontrará un entero igual a 83 886 080 (5×2^{24}) y una cadena "JILL".

Un método evidente, pero que por desgracia es incorrecto, es invertir los bytes de cada palabra después de su recepción, lo que produce la figura 2-20(c). Ahora el entero es 5, pero la cadena es "LLIJ". El problema aquí es que los enteros se invierten con el otro orden de los bytes, pero no las cadenas. Si no se dispone de información adicional acerca de lo que es un entero o una cadena, no existe forma alguna de reparar el daño.

Por fortuna, se dispone de esta información en forma implícita. Recordemos que los elementos del mensaje corresponden al identificador del procedimiento y los parámetros. Tanto el cliente como el servidor conocen los tipos de los parámetros. Así, un mensaje que corresponda a un procedimiento remoto con n parámetros tendrá $n + 1$ campos, uno para identificar al procedimiento y uno para cada uno de los n parámetros. Si se llega a un acuerdo en la representación de cada uno de los tipos básicos de datos, es posible, dada una lista de parámetros y un mensaje, deducir los bytes que pertenecen a cada parámetro, con lo que se resuelve el problema.

Como ejemplo, consideremos el procedimiento de la figura 2-21(a). Tiene tres parámetros: un carácter, un número de punto flotante y un arreglo de cinco enteros. Podríamos decidir que la transmisión de un carácter se realice en el byte del extremo derecho de una palabra (lo cual deja los siguientes 3 bytes vacíos), un flotante como una palabra completa y un arreglo como grupo de palabras igual a la longitud del arreglo, precedida por una palabra con su longitud, como se muestra en la figura 2-21(b). Con esta reglas, el resguardo del cliente de *foobar* sabe que debe utilizar el formato de la figura 2-21(b) y el resguardo del servidor sabe que los mensajes que reciba de *foobar* tendrán el formato de la figura 2-21(b). Al tener la información de los parámetros se pueden llevar a cabo todas las conversiones necesarias.

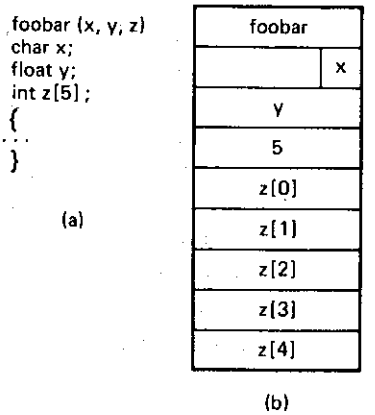


Figura 2-21. (a) Un procedimiento. (b) El mensaje correspondiente.

Incluso con la información adicional, quedan pendientes ciertos puntos. En particular, ¿cómo se debe representar la información en los mensajes? Una forma es diseñar un estándar de red o **forma canónica** para los enteros, caracteres, booleanos, números de punto flotante, etc., y pedir a todos los emisores que conviertan sus representaciones internas a esta forma durante el ordenamiento. Por ejemplo, supongamos que se decide utilizar complemento a 2 para los enteros, ASCII para los caracteres, 0 (falso) y 1 (cierto) para los booleanos, y el formato IEEE para los números de punto flotante; todo guardado como little endian. Para cualquier lista de enteros, caracteres, booleanos y números de punto flotante, el patrón preciso necesario queda por completo determinado hasta el último bit. Como resultado, el resguardo del servidor no tiene que preocuparse más por el orden que tiene el cliente, puesto que el orden de los bits en el mensaje ya está determinado, en forma independiente del hardware del cliente.

El problema con este método es que a veces es ineficiente. Supongamos que un cliente big endian se comunica con un servidor big endian. De acuerdo con las reglas, el cliente debe convertir todo a little endian en el mensaje y el servidor debe volver a convertir todo cuando el mensaje llegue. Aunque esto no contiene ambigüedades, requiere de dos conversiones que en realidad no son necesarias. Esta observación da lugar a un segundo método: el cliente utiliza su propio formato original e indica en el primer byte del mensaje su formato. Así, un cliente little endian construye un mensaje little endian y un cliente big endian construye un mensaje big endian. Tan pronto llega el mensaje, el resguardo del servidor examina el primer byte para ver quién es el cliente. Si es igual al servidor, no se necesita la conversión. En caso contrario, el resguardo del servidor hace toda la conversión. Aunque sólo hemos analizado la conversión entre endians, se puede realizar de la misma manera la conversión entre complemento a 1 y complemento a 2, EBCDIC a ASCII, etc. El truco consiste en conocer la organización del mensaje y la identidad del cliente. Una vez que esto se sepa, el resto es sencillo (siempre que todos puedan realizar las conversiones de los formatos de los demás).

Llegamos ahora a la cuestión del origen de los procedimientos de resguardo. En muchos de los sistemas basados en RPC, éstos se generan de forma automática. Como hemos visto, dada una especificación del procedimiento servidor y las reglas de codificación, el formato del mensaje queda determinado de manera única. Así, es posible tener un compilador que lea las especificaciones del servidor y genere un resguardo del cliente que empaque sus parámetros en el formato oficial de los mensajes. En forma similar, el compilador también puede generar un resguardo del servidor que los desempaque y que llame al servidor. El hecho de disponer de ambos procedimientos de resguardo a partir de una especificación formal del servidor no sólo facilita la vida de los programadores, sino que reduce la probabilidad de error y hace que el sistema sea transparente con respecto a las diferencias en la representación interna de los elementos de los datos.

Al fin, llegamos al último y más difícil de los problemas: ¿cómo se trasfieren los apuntadores? La respuesta es: sólo con el máximo de dificultad, si es que se puede. Recordemos que un apuntador sólo tiene sentido dentro del espacio de direcciones del proceso en el que se utiliza. De regreso al ejemplo *read* ya analizado, si el segundo parámetro (la dirección del buffer) es 1000 en el cliente, uno no puede transferir el número 1000 al servidor y esperar que funcione. La dirección 1000 del servidor podría estar a la mitad del texto del programa.

Una solución consiste en olvidarse en general de los apuntadores y los parámetros por referencia. Sin embargo, son tan importantes, que esta solución es poco recomendable. De hecho, ni siquiera es necesaria. En el ejemplo *read*, el resguardo del cliente sabe que el segundo parámetro apunta a un arreglo de caracteres. Supongamos por el momento que también conoce el tamaño del arreglo. Surge entonces una estrategia: copiar el arreglo en el mensaje y enviarlo al servidor. El resguardo del servidor puede entonces llamar a éste con un apuntador a este arreglo, aunque este apuntador tenga un valor numérico distinto al del segundo parámetro de *read*. Los cambios que realice el servidor mediante el apuntador (por ejemplo, el almacenamiento de datos en el arreglo) afectarán de manera directa al buffer de mensajes dentro del resguardo del servidor. Cuando concluye su labor el servidor, el mensaje original se puede enviar de regreso al resguardo del cliente, que a su vez lo copia de nuevo al cliente. De hecho, la llamada por referencia se sustituye por la copia/restauración. Aunque estos métodos no siempre son idénticos, con frecuencia son lo bastante buenos.

Una optimización duplica la eficiencia de este mecanismo. Si el resguardo sabe que el buffer es un parámetro de entrada o un parámetro de salida hacia el servidor, puede eliminar una de las copias. Si el arreglo es una entrada al servidor (por ejemplo, en una llamada a *write*), no necesita volverse a copiar. Si es una salida, no necesita enviarse de regreso. La forma para indicarlo está en la especificación formal del procedimiento servidor. Así, a cada procedimiento remoto se le asocia una especificación formal del procedimiento, escrita en cierto lenguaje de especificación, para indicar la naturaleza de los parámetros, los que son de entrada o de salida (o ambos) y sus tamaños (máximos). A partir de esta especificación formal se generan los resguardos, mediante un compilador especial de resguardos.

Como comentario final, es importante observar que aunque ahora podemos manejar los apuntadores a arreglos y estructuras sencillas, no podemos todavía manejar el caso más general de un apuntador a una estructura de datos arbitraria, como lo es una gráfica compleja. Ciertos sistemas intentan resolver este caso mediante la transferencia real del apuntador al resguardo del servidor y la generación de un código especial para el uso de apuntadores en el procedimiento servidor.

Lo usual es que un apuntador sea seguido (referenciado de manera inversa) mediante su colocación en un registro y realizando un direccionamiento indirecto a través del registro. Si se utiliza esta técnica particular, la referencia inversa se realiza enviando un mensaje de regreso al resguardo del cliente para pedirle que lo busque; después se envía el elemento al que apunta (lectura) o se guarda un valor en la dirección a la que apunta (escritura). Aunque este método funciona, es muy ineficiente. Imaginemos que el servidor de archivos almacena los bytes en el buffer mediante el envío de cada uno en un mensaje independiente. Aun así, es mejor que nada y ciertos sistemas lo utilizan.

2.4.3. Conexión dinámica

Un tema que hemos dejado de lado hasta ahora es la forma en que el cliente localiza al servidor. Un método consiste en integrar dentro del código del cliente la dirección (en la red) del servidor. El problema de este método es que es muy rígido. Si el servidor se desplaza, se duplica o si cambia la interfaz, habría que localizar y volver a compilar numerosos programas. Para evitar todos esos problemas, ciertos sistemas distribuidos utilizan lo que se conoce como **conexión dinámica** para que concuerden los clientes y los servidores. En esta sección describiremos las ideas detrás de la conexión dinámica.

El punto de partida de la conexión dinámica es la especificación formal del servidor. Como ejemplo, consideremos el servidor de la figura 2-9(a), especificado en la figura 2-22. La especificación indica el nombre del servidor (*file_server*), el número de versión (3.1) y una lista de los procedimientos que proporciona (*read, write, create* y *delete*).

Se tienen los tipos de los parámetros para cada procedimiento. Cada parámetro queda determinado como de tipo *in, out,* o *in-out*. La dirección es relativa al servidor. Un parámetro tipo *in*, como el nombre del archivo, *name*, se envía del cliente al servidor. Éste se utiliza para indicar al servidor el archivo que debe leer, escribir, crear o eliminar. En forma análoga, *bytes* indica al servidor el número de bytes por transferir y *position* indica el sitio del archivo donde comenzar la lectura o la escritura. Un parámetro tipo *out*, como *buf* en *read*, se envía del servidor al cliente. *Buf* es el lugar donde el servidor de archivos coloca los datos solicitados por el cliente. Un parámetro *in-out*, de los cuales no existen en nuestro ejemplo, es aquél que se envía del cliente al servidor, donde se le modifica y que después se envía de regreso al cliente (copia/restauración). La copia/restauración se utiliza por lo general para los parámetros tipo apuntador, en los casos en que el servidor lee y modifica la estructura de datos a la que apuntan. Las direcciones son cruciales, por lo que el resguardo del cliente

```
#include <header.h>

specification of file_server, version 3.1:

   long read(in char name[MAX_PATH], out char buf[BUF_SIZE],
             in long bytes, in long position);

   long write(in char name[MAX_PATH], in char buf[BUF_SIZE],
             in long bytes, in long position);

   int create(in char[MAX_PATH], in int mode);

   int delete(in char[MAX_PATH]);

end;
```

Figura 2-22. Una especificación del servidor sin estado de la figura 2-9.

conoce los parámetros que debe enviar al servidor y el resguardo del servidor sabe cuáles debe enviar de regreso.

Como hemos señalado, este ejemplo particular es un servidor sin estado. Para un servidor del tipo de UNIX,se tendrían otros procedimientos, *open* y *close*, y distintos parámetros de *read* y *write*. El concepto de RPC es en sí mismo neutral y permite a los diseñadores de sistemas construir cualquier tipo deseado de servidores.

El principal uso de la especificación formal de la figura 2-22 es como una entrada del generador de resguardos, el cual produce tanto el resguardo del cliente como el del servidor. Ambos se colocan entonces en las bibliotecas respectivas. Cuando un programa usuario (cliente) llama a cualquiera de los procedimientos definidos mediante esa especificación, el correspondiente procedimiento de resguardo del cliente se liga con su binario. En forma análoga, si se compila un servidor, los resguardos del servidor se le ligan también.

Cuando el servidor inicia su ejecución, la llamada a *initialize* que se encuentra fuera del ciclo principal [véase la figura 2-9(a)] **exporta** la interfaz del servidor. Esto quiere decir que el servidor envía un mensaje a un programa llamado **conector** para darle a conocer su existencia. Este proceso se conoce como **registro** del servidor. Para registrarse, el servidor proporciona al conector su nombre, número de versión, un identificador, que por lo general tiene una longitud de 32 bits y un **asa** que se utiliza para localizarlo. El asa depende del sistema y puede ser una dirección Ethernet, una dirección IP, una dirección X.500, un identificador ralo de procesos o alguna otra cosa. Además, se puede proporcionar otra información, como por ejemplo, la relativa a la autenticación. Un servidor también puede cancelar su registro con el conector, si ya no está preparado para prestar algún servicio. La interfaz del conector se muestra en la figura 2-23.

Llamada	Entrada	Salida
Registro	Nombre, versión, asa, identificación única	
De registro	Nombre, versión, identificación única	
Búsqueda	Nombre, versión	Asa, identificador único

Figura 2-23. La interfaz del conector.

Con estas bases, consideremos ahora la forma en que el cliente localiza al servidor. Cuando el cliente llama a alguno de los procedimientos remotos por primera vez (digamos a *read*), el resguardo del cliente ve que todavía no está conectado a un servidor, por lo que envía un mensaje al conector, donde solicita la **importación** de la versión 3.1 de la interfaz *file_server*. El conector verifica si uno o más servidores ya han exportado una interfaz con ese nombre y número de versión. Si ninguno de los servidores que se ejecuten en ese momento están dispuestos a soportar esa interfaz, la llamada a *read* fracasa. Al incluir el número de versión en el proceso concordante, el conector puede garantizar que los clientes que utilicen interfaces obsoletas no podrán localizar al servidor, en vez de ubicarlo y obtener resultados impredecibles debido a parámetros incorrectos.

Por otro lado, si existe un servidor adecuado, el conector proporciona su asa e identificador único al resguardo del cliente; éste utiliza el asa como la dirección a la cual enviar el mensaje solicitado. El mensaje contiene los parámetros y el identificador único, que son utilizados por el núcleo del servidor para dirigir el mensaje recibido al servidor correcto, en caso de que se ejecuten varios servidores en la misma máquina.

Este método de exportación e importación de interfaces es altamente flexible. Por ejemplo, puede manejar varios servidores que soporten la misma interfaz. El conector puede difundir los clientes de manera aleatoria entre los servidores, para que la carga sea más justa, si así lo desea. También puede hacer un muestreo periódico de los servidores y cancelar el registro del servidor que no responda, con el fin de lograr un cierto grado de tolerancia de fallas. Además, también puede ayudar en la autenticación. Por ejemplo, un servidor podría especificar su deseo de ser utilizado por una lista determinada de usuarios, en cuyo caso el conector rechazaría a los usuarios que no se encontrasen en la misma. El conector también puede verificar que tanto el cliente como el servidor utilicen la misma versión de interfaz.

Sin embargo, esta forma de conexión dinámica también tiene sus desventajas. Existe un costo adicional en el tiempo generado por la exportación e importación de las interfaces. Puesto que muchos procesos cliente tienen vida corta y cada proceso tiene que comenzar de nuevo, el efecto puede ser significativo. Además, en un sistema distribuido de gran tamaño, el conector se puede convertir en un cuello de botella, por lo que se necesitarían varios conectores. En consecuencia, sin importar el hecho de que una interfaz se registre o cancele su registro, se necesitará de gran número de mensajes para mantener sincronizados y actualizados todos los conectores, lo cual crea un nuevo gasto adicional.

2.4.4. Semántica de RPC en presencia de fallas

El objetivo de RPC es ocultar la comunicación, al hacer que las llamadas a procedimientos remotos se parezcan a las locales. Con unas cuantas excepciones, como la incapacidad para manejar variables globales y las diferencias sutiles entre el uso de copia/restauración para los parámetros tipo apuntador en vez de la llamada por referencia, hemos mejorado mucho. En realidad, si tanto el cliente como el servidor funcionan de manera perfecta, RPC hace su trabajo en forma excelente. El problema se presenta cuando aparecen los errores. Es entonces cuando las diferencias entre las llamadas locales y remotas no son tan fáciles de encubrir. En esta sección examinaremos algunos de los posibles errores y lo que se puede hacer con ellos.

Para estructurar nuestro análisis, distinguiremos entre cinco clases distintas de fallas que pueden ocurrir en los sistemas RPC:

1. El cliente no puede localizar al servidor.
2. Se pierde el mensaje de solicitud del cliente al servidor.
3. Se pierde el mensaje de respuesta del servidor al cliente.
4. El servidor falla antes de recibir una solicitud.
5. El cliente falla después de enviar una solicitud.

Cada una de estas categorías tiene distintos problemas y necesita distintas soluciones.

El cliente no puede localizar al servidor

Para comenzar, puede ocurrir que el cliente no pueda localizar un servidor adecuado. Por ejemplo, el servidor podría estar inactivo. En forma alternativa, supongamos que el cliente se compila mediante una versión particular del resguardo del cliente y que el binario no se utiliza durante un período considerable. Mientras tanto, el servidor evoluciona y se instala una nueva versión de la interfaz, con lo que se generan y empiezan a utilizar nuevos resguardos. Cuando el cliente se ejecuta, el conector no podrá hacer que concuerde con un servidor e informará de una falla. Aunque este mecanismo se utiliza para proteger al cliente de la comunicación accidental con un servidor que podría no coincidir en términos de los parámetros requeridos o de la operación solicitada, permanece el problema de enfrentar esta falla.

Con el servidor de la figura 2-9(a), cada uno de los procedimientos regresa un valor, donde el código −1 se utiliza por lo general para indicar una falla. Para tales procedimientos, el regreso del valor −1 indicará con claridad a quien realizó la llamada que algo está mal. En UNIX, se asigna una variable global *errno* un valor que indica el tipo de error. En tal sistema, es fácil añadir este nuevo tipo de error "No se pudo localizar al servidor".

El problema es que esta solución no es lo bastante general. Consideremos el procedimiento *sum* de la figura 2-19. En este caso, –1 es un valor válido de retorno; por ejemplo, el resultado de la suma de 7 y –8. Se necesita otro mecanismo para informar de los errores.

Un posible candidato es que el error provoque una **excepción**. En ciertos lenguajes (por ejemplo, Ada), los programadores pueden escribir procedimientos especiales que sean llamados en errores específicos, como la división entre cero. En C, se pueden utilizar los controladores de señales con este fin. En otras palabras, podríamos definir un nuevo tipo de señal *SIGNOSERVER* y permitir su manejo igual al de las demás excepciones y señales.

Este método también tiene sus desventajas. Para comenzar, no todos los lenguajes tienen excepciones o señales. Uno de los que no tienen es Pascal. Otro punto es que el hecho de escribir una excepción o un controlador de señales destruye la transparencia que deseábamos lograr. Suponga que usted es un programador y que su jefe le dice que escriba el procedimiento *sum*. Usted sonríe y responde que lo escribirá, lo probará y documentará en cinco minutos. Entonces ella menciona que también debe escribir un controlador de excepciones, en caso de que el procedimiento no se encuentre el día de hoy. En este punto, es difícil mantener la seguridad de que los procedimientos remotos no son tan diferentes de los locales, puesto que la escritura de un controlador de excepciones para "no se pudo localizar al servidor" sería una solicitud poco usual en un sistema con un procesador.

Pérdida de mensajes de solicitud

El segundo elemento de la lista trata de los mensajes de solicitud perdidos. Éste es más fácil de tratar: lo único que hay que hacer es que el núcleo inicie un cronómetro al enviar la solicitud. Si el tiempo se termina antes de que regrese una respuesta o reconocimiento, el núcleo vuelve a enviar el mensaje. Si el mensaje en realidad se perdió, el servidor no notará la diferencia entre la retransmisión y el original y todo funcionará bien. A menos, por supuesto, que se pierdan tantos mensajes que el núcleo se dé por vencido y concluya erróneamente que el servidor está inactivo, en cuyo caso regresamos al caso "no se pudo localizar al servidor".

Pérdida de mensajes de respuesta

La pérdida de las respuestas es más difícil de enfrentar. La solución obvia es basarse de nuevo en un cronómetro. Si no llega una respuesta en un periodo razonable, sólo hay que enviar de nuevo la solicitud. El problema con esta solución es que el núcleo del cliente no estará seguro de la razón por la que no hubo respuesta. ¿Se perdió la solicitud? ¿Se perdió la respuesta? ¿Ocurre tan sólo que el servidor es lento? Esto puede ser una diferencia.

En particular, ciertas operaciones se pueden repetir con seguridad tantas veces como sea necesario sin que ocurra algún daño. Una solicitud de los primeros 1024 bytes de un archivo no tiene efectos colaterales y se puede ejecutar tantas veces como sea necesario sin causar daño alguno. Una solicitud con esta propiedad es **idempotente**.

Consideremos ahora una solicitud a un servidor bancario para trasferir un millón de dólares de una cuenta a otra. Si la solicitud llega y se lleva a cabo, pero se pierde la respuesta, el cliente no sabrá esto y volverá a transmitir el mensaje. El servidor del banco interpretará esta solicitud como nueva y también la llevará a cabo. Se trasferirán dos millones de dólares. Imaginemos lo que ocurriría si la respuesta se pierde diez veces. La trasferencia de dinero no es idempotente.

Una forma de resolver este problema es intentar estructurar de alguna manera todas las solicitudes de modo que sean idempotentes. Sin embargo, en la práctica, muchas solicitudes (como la trasferencia de dinero) tienen naturaleza no idempotente, por lo que se necesita algo más. En otro método, el núcleo del cliente asigna a cada solicitud un número secuencial. Si el núcleo del servidor mantiene un registro del número secuencial de recepción más reciente de cada uno de los núcleos de clientes que lo utilicen, el núcleo del servidor podrá notar la diferencia entre una solicitud original y una retransmisión y puede rechazar la realización de cualquier solicitud por segunda vez. Una protección adicional consiste en tener un bit en el encabezado del mensaje para distinguir las solicitudes adicionales de las retransmisiones (la idea es que siempre es seguro llevar a cabo una solicitud original; las retransmisiones requieren de más cuidado).

Fallas del servidor

La siguiente falla de la lista es una del servidor. También se relaciona con la idempotencia; por desgracia, no se puede resolver mediante números secuenciales. La secuencia normal de eventos en un servidor se muestra en la figura 2-24(a). Llega una solicitud, se lleva a cabo y se envía una respuesta. Consideremos ahora la figura 2-24(b). Llega una solicitud y se lleva a cabo, como antes, pero el servidor falla antes de que pueda enviar la respuesta. Por último, veamos la figura 2-24(c). De nuevo, llega una solicitud, pero esta vez el servidor falla antes de que la pueda llevar a cabo.

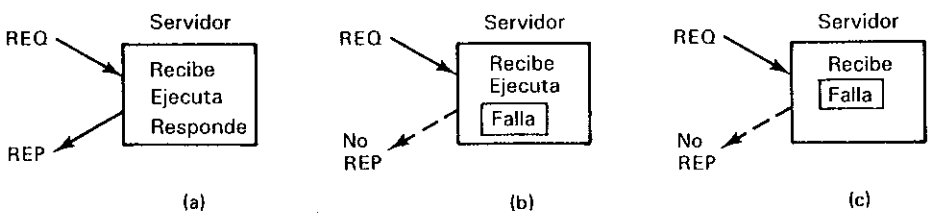


Figura 2-24. (a) Caso normal. (b) Falla después de responder. (c) Falla antes de responder.

La parte desagradable de la figura 2-24 es que el tratamiento correcto difiere en los casos (b) y (c). En (b), el sistema tiene que informar de la falla de regreso al cliente (por ejemplo, hacer una excepción), mientras que en (c) sólo tiene que transmitir de nuevo la solicitud. El problema es que el núcleo del cliente no puede decidir cuál es cuál. Todo lo que sabe es que el tiempo se ha terminado.

Existen tres escuelas en torno a lo que se debe hacer en este caso. Una filosofía consiste en esperar hasta que el servidor vuelva a arrancar (o se reconecte a un nuevo servidor) e intente realizar de nuevo la operación. La idea es mantener el intento hasta recibir una respuesta, para dársela entonces al cliente. Esta técnica se llama **semántica al menos una vez** y garantiza que la RPC se ha realizado al menos una vez, pero es posible que se realice más veces.

La segunda filosofía se da por vencida en forma inmediata e informa de la falla. Esta forma se llama **semántica a lo más una vez** y garantiza que la RPC se realiza a lo más una vez, pero es posible que no se realice ni una sola.

La tercera filosofía consiste en no garantizar nada. Cuando un servidor falla, el cliente no obtiene ayuda o alguna promesa. La RPC se realiza en cualquier lugar, un número de veces que va desde 0 hasta un número muy grande. La principal virtud de este esquema es que es fácil de implantar.

Ninguna de estas opciones es más atractiva que las otras. Lo que uno desearía es una **semántica de exactamente una**, pero es fácil de comprender que no existe forma de garantizar esto en general. Imaginemos que la operación remota consiste en imprimir cierto texto y se lleva a cabo al cargar el buffer de la impresora, para entonces activar un bit en cierto registro de control para iniciar la impresora. La falla puede ocurrir un microsegundo antes de activar el bit o un microsegundo después. El procedimiento de recuperación depende por completo del momento en que se lleve a cabo la falla, pero el cliente no tiene forma de descubrir ese instante.

En resumen, la posibilidad de fallas del servidor cambia de manera radical la naturaleza de la RPC y distingue de manera clara los sistemas con un procesador de los sistemas distribuidos. En el primer caso, la falla de un servidor implica también una falla del cliente, por lo que la recuperación no es ni posible ni necesaria. En el segundo caso, es posible y necesario realizar cierta acción.

Fallas del cliente

El último punto de la lista de fallas es el que se refiere al cliente. ¿Qué ocurre si un cliente envía una solicitud a un servidor para que se realice cierto trabajo y falla antes de que el servidor responda? En este momento, está activa una labor de cómputo y ningún padre espera el resultado. Esta labor de cómputo no deseado se llama **huérfano**.

Los huérfanos provocan varios problemas. Como mínimo, desperdician los ciclos del CPU. Bloquean archivos o capturan recursos valiosos. Por último, si el cliente vuelve a arrancar y realiza de nuevo la RPC, pero la respuesta del huérfano regresa de inmediato, puede surgir una confusión.

¿Qué se puede hacer con los huérfanos? Nelson (1981) propuso cuatro soluciones. En la solución 1, antes de que el resguardo del cliente envíe un mensaje RPC, crea una entrada en un registro que indica lo que va a hacer. El registro se mantiene en el disco o algún otro medio que sobreviva a las fallas. Después de volver a arrancar, se verifica

el contenido del registro y el huérfano se elimina en forma explícita. Esta solución se llama **exterminación**.

La desventaja de este esquema es el enorme gasto de escritura en el registro del disco para cada RPC. Además, no funcionaría, puesto que los propios huérfanos pueden realizar RPC, con lo que crearían **huérfanos de huérfanos** o descendientes posteriores imposibles de localizar. Por último, la red se puede dividir mediante particiones, debido a la falla de una compuerta, lo que haría imposible su eliminación, aun cuando se pudiera localizar. En resumen, éste no sería un método promisorio.

En la solución 2, llamada **reencarnación**, se pueden resolver todos estos problemas sin necesidad de escribir registros en disco. La forma en que funciona consiste en dividir el tiempo en épocas numeradas de manera secuencial. Cuando un cliente arranca de nuevo, envía un mensaje a todas las máquinas que declaran el inicio de una nueva época. Al recibir dicha transmisión, se eliminan todos los cómputos remotos. Por supuesto, si se divide la red mediante particiones, podrían sobrevivir ciertos huérfanos. Sin embargo, cuando vuelven a reportarse, sus respuestas contendrán un número de época obsoleto, lo que facilitaría su detección.

La solución 3 es una variante de esta idea, pero menos draconiana. Se le llama **reencarnación sutil**. Cuando llega un mensaje de cierta época, cada máquina verifica si tiene cómputos remotos y, en caso afirmativo, intenta localizar a su poseedor. Sólo en caso de que no pueda encontrar al poseedor se elimina el cómputo.

Por último, tenemos la solución 4, **expiración**, en la cual a cada RPC se le asigna una cantidad estándar de tiempo T para que realice su trabajo. Si no lo puede terminar, debe pedir en forma explícita otro quantum, lo que es una inconveniencia. Por otro lado, si después de la falla el servidor espera un tiempo T antes de volver a arrancar, es seguro que todos los huérfanos hayan desaparecido. El problema a resolver aquí es elegir un valor razonable de T, en vista de que pueden existir RPC con una amplia variedad de requisitos diversos.

En la práctica, ninguno de estos métodos es recomendable. Peor aún, la eliminación de un huérfano puede tener consecuencias imprevisibles. Por ejemplo, supongamos que un huérfano ha bloqueado uno o más archivos o registros de una base de datos. Si el huérfano se elimina de manera súbita, estos bloqueos pueden permanecer para siempre. Además, un huérfano puede crear varias entradas en colas remotas para iniciar otros procesos en un tiempo futuro, por lo que su eliminación no borraría todas sus huellas. La eliminación de huérfanos se analiza con más detalle en Panzieri y Shrivastava (1988).

2.4.5. Aspectos de la implantación

El éxito o fracaso de un sistema distribuido descansa a menudo en su desempeño. Éste, a su vez, depende de manera crítica de la velocidad de la comunicación. Con frecuencia, la velocidad aumenta o disminuye con su implantación y no tanto por sus principios abstractos. En esta sección analizaremos ciertos aspectos de la implantación de los sistemas RPC, con énfasis especial en el desempeño y en los puntos donde se gasta el tiempo.

Protocolos RPC

El primer aspecto es la elección del protocolo RPC. En teoría, cualquier protocolo antiguo puede funcionar si obtiene los bits del núcleo del cliente y los lleva al núcleo del servidor; pero en la práctica, hay que tomar varias decisiones importantes en este punto, decisiones que pueden tener fuerte impacto en el desempeño. La primera decisión está entre un protocolo orientado a la conexión o un protocolo sin conexión. En el primer caso, al momento en que el cliente se conecta con el servidor, se establece una conexión entre ellos. Todo el tráfico, en ambas direcciones, utiliza esta conexión.

La ventaja de contar con una conexión es que la comunicación es más fácil. Cuando un núcleo envía un mensaje, no tiene que preocuparse por su pérdida, ni tampoco por los reconocimientos. Todo ello se maneja en nivel inferior, mediante el software que soporta la conexión. Cuando se opera una red de área amplia, esta ventaja es con frecuencia muy irresistible.

La desventaja, en particular en una LAN, es la pérdida en el desempeño. Todo ese software adicional estorba en el camino. Además, la ventaja principal (no perder los paquetes) difícilmente se necesita en una LAN, puesto que las LAN son confiables en este sentido. En consecuencia, la mayoría de los sistemas distribuidos que pretenden utilizarse en un edificio o campus utilizan los protocolos sin conexión.

La segunda opción principal está en utilizar un protocolo estándar de propósito general o alguno diseñado de forma específica para RPC. Puesto que no existen estándares en esta área, el uso de un protocolo RPC adaptado quiere decir por lo general que cada quien diseñe el suyo (o pedir prestado el de un amigo). Los diseñadores de sistemas se dividen casi a la mitad en este aspecto.

Algunos sistemas distribuidos utilizan IP (o UDP, integrado a IP) como el protocolo básico. Esta opción tiene ciertos puntos a su favor:

1. El protocolo ya ha sido diseñado, lo que ahorra un trabajo considerable.
2. Se dispone de muchas implantaciones, lo cual, de nuevo, ahorra trabajo.
3. Estos paquetes se pueden enviar y recibir en casi todos los sistemas UNIX.
4. Los paquetes IP y UDP son soportados por muchas de las redes existentes.

En resumen, IP y UDP son fáciles de utilizar y se ajustan bien a los sistemas UNIX existentes y a redes como Internet. Esto hace que sea directo escribir clientes y servidores que se ejecuten en sistemas UNIX, lo cual por cierto ayuda a que el código se pueda ejecutar y se pruebe pronto.

Como es usual, el lado malo es el desempeño. IP no se diseñó como un protocolo para un usuario final. Se diseñó como base para poder establecer conexiones confiables TCP entre las redes recalcitrantes. Por ejemplo, puede trabajar con compuertas que fragmentan paquetes en pedazos pequeños, de forma que puedan pasar a través de las redes con un tamaño pequeño máximo de paquete. Aunque esta característica nunca se necesita en un sistema distribuido

basado en una LAN, los campos del encabezado del paquete IP relacionados con la fragmentación deben ser llenados por el emisor y verificados por el receptor para que sean paquetes válidos de IP. Los paquetes IP tienen en total 13 campos de encabezado, de los cuales tres son útiles: las direcciones fuente y destino y la longitud del paquete. Los otros 10 son añadidos y en el caso de uno de ellos, la suma de verificación, su cálculo consume tiempo. Para empeorar las cosas, UDP tiene otra suma de verificación, que también abarca los datos.

La alternativa es utilizar un protocolo especializado para RPC que, a diferencia de IP, no intente trabajar con paquetes que han estado brincando a través de la red durante unos cuantos minutos y que después surgen de la nada en un momento inconveniente. Por supuesto, el protocolo debe ser inventado, implantado, probado e insertado en los sistemas existentes, por lo que es un trabajo adicional considerable. Además, el resto del mundo no tiende a brincar de gusto ante el nacimiento de un nuevo protocolo. A largo plazo, el camino a seguir es el desarrollo y amplia aceptación de un protocolo RPC de alto desempeño, pero aún no estamos en ese punto.

Un último aspecto relacionado con los protocolos es la longitud del paquete y el mensaje. La realización de una RPC tiene un costo excesivo fijo de gran magnitud, independiente de la cantidad de datos enviados. Así, la lectura de un archivo de 64K en una RPC de 64K es más eficiente que la lectura en 64 RPC de 1K. Por lo tanto, es importante que el protocolo y la red permitan las transmisiones largas. Algunos sistemas RPC se limitan a tamaños pequeños (por ejemplo, el límite de Sun Microsystems es 8K). Además, muchas redes no pueden manejar grandes paquetes (el límite de Ethernet es de 1536 bytes), por lo que una RPC se divide en varios paquetes, lo cual causa un costo adicional.

Reconocimientos

En caso que las RPC de gran tamaño tengan que dividirse en muchos paquetes pequeños de la manera descrita, surge una nueva cuestión: ¿deben reconocerse los paquetes de manera individual o no? Supongamos, a manera de ejemplo, que un cliente desea escribir un bloque de datos de 4K en un servidor de archivos, pero que el sistema no puede manejar paquetes mayores de 1K. Una estrategia, conocida como el **protocolo detenerse y esperar**, establece que el cliente envíe el paquete 0 con el primer 1K, para después esperar un reconocimiento del servidor, como se muestra en la figura 2-25(b). Entonces, el cliente envía el segundo 1K, espera otro reconocimiento, etcétera.

Otra alternativa, llamada a menudo **protocolo de chorro**, establece que el cliente envíe todos los paquetes tan pronto como pueda. Con este método, el servidor reconoce todo el mensaje al recibir *todos* los paquetes, no uno por uno. El protocolo de chorro se ilustra en la figura 2-25(c).

Estos protocolos tienen ciertas propiedades distintas. Con el protocolo detenerse y esperar, si se daña o pierde un paquete, el cliente no puede recibir a tiempo un reconocimiento, por lo que vuelve a transmitir el paquete defectuoso. Con el protocolo de chorro, el servidor se enfrenta a una decisión cuando, por ejemplo, se pierde el paquete 1, pero el paquete 2 llega de manera correcta. Puede abandonar todo, no hacer nada y esperar a que

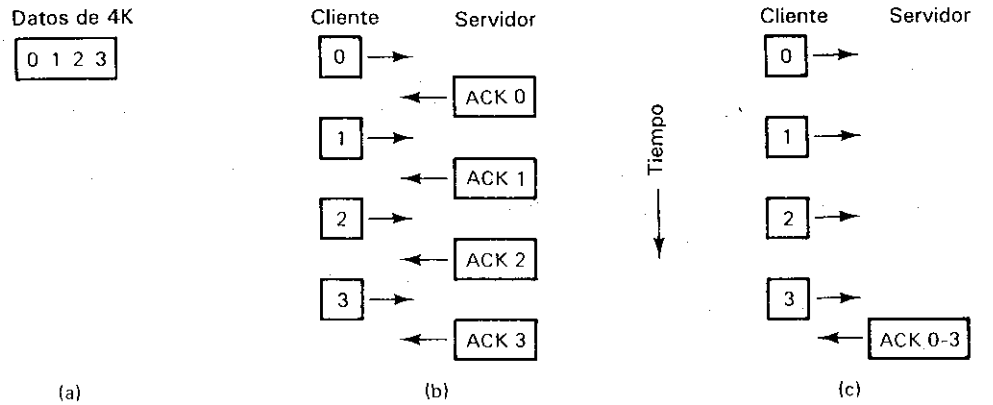


Figura 2-25. (a) Un mensaje de 4K. (b) Un protocolo detenerse y esperar. (c) Un protocolo de chorro.

el cliente haga un receso y vuelva a transmitir todo el mensaje. O bien, puede guardar el paquete 2 en un buffer (junto con el 0), con la esperanza que el 3 llegue en forma correcta, para después pedir al cliente que le envíe el paquete 1. Esta técnica se llama **repetición selectiva.**

Tanto el método detenerse y esperar como el de abandono total al aparecer un error son fáciles de implantar. La repetición selectiva necesita más administración, pero utiliza menos ancho de banda en la red. En las LAN muy confiables, los paquetes perdidos son tan raros que el uso de la repetición selectiva causa más problemas que beneficios, pero en las redes de área amplia con frecuencia es buena idea.

Sin embargo, fuera del control de los errores, existe otra consideración que en realidad es más importante: el **control del flujo**. Muchos circuitos de interfaz de una red pueden enviar paquetes consecutivos con un espacio muy pequeño entre ellos, pero no siempre pueden recibir un número ilimitado de paquetes adyacentes debido a la capacidad finita del circuito. Con ciertos diseños, un circuito ni siquiera puede aceptar dos paquetes adyacentes, puesto que después de recibir el primero, dicho circuito se desactiva por un momento, durante la interrupción correspondiente, por lo que pierde el principio del segundo. Cuando un paquete llega a un receptor que no lo puede aceptar, ocurre un **error de sobre-ejecución** y el paquete se pierde. En la práctica, los errores de sobre-ejecución son un problema mucho más serio que la pérdida de paquetes debido al ruido u otras formas de daño.

Los dos métodos de la figura 2-25 son un poco distintos en cuanto a los errores de sobre-ejecución. Con el protocolo detenerse y esperar, estos errores son imposibles, puesto que el segundo paquete no se envía sino hasta que el receptor indica de manera explícita que está listo para recibirlo. (Por supuesto, estos errores pueden aparecer en el caso de varios emisores.)

Con el protocolo del chorro, puede ocurrir un error de sobre-ejecución, lo que es una lástima, puesto que es claro que este protocolo es más eficiente que el protocolo detenerse y esperar. Sin embargo, existen formas de enfrentar la sobre-ejecución. Si, por un lado, el

problema es provocado por el circuito que se desactiva de manera temporal mientras procesa una interrupción, entonces un emisor inteligente puede insertar un retraso entre los paquetes para darle tiempo al receptor para generar la interrupción correspondiente al paquete y volver a estar listo. Si el retraso necesario es corto, el emisor puede hacer un ciclo (espera ocupada); si es largo, puede establecer una interrupción con un cronómetro y hacer algo mientras espera. Si es algo intermedio (unos cuantos cientos de microsegundos), lo cual ocurre con frecuencia, tal vez la mejor solución sea la espera ocupada y sólo aceptar el tiempo desperdiciado como un mal necesario.

Si, por otro lado, la sobre-ejecución se debe a la capacidad finita del buffer en el circuito de la red, entonces el emisor puede enviar *n* paquetes y después un espacio considerable (o bien, definir el protocolo para que pida un reconocimiento después de cada *n* paquetes).

Debe quedar claro que la minimización de reconocimientos de los paquetes y la obtención de un buen desempeño dependen de las propiedades de sincronización del circuito de la red, de modo que el protocolo tenga que ajustarse al hardware utilizado. Un protocolo RPC diseñado para su adaptación puede tomar en cuenta más fácilmente aspectos tales como el control del flujo con respecto de los protocolos de propósito general, razón por la cual los protocolos especializados RPC tienen un mejor desempeño que los sistemas basados en IP o UDP, por un margen amplio.

Antes de dejar el tema de los reconocimientos, existe otro punto delicado que es importante hacer notar. En la figura 2-16(c), el protocolo consta de una solicitud, una respuesta y un reconocimiento. Este último se necesita para indicarle al servidor que puede descartar la respuesta como si hubiese llegado con seguridad. Supongamos ahora que se pierde el reconocimiento en el tránsito (es poco probable, pero no imposible). El servidor no descartará la respuesta. Peor aún, en lo que respecta al cliente, el protocolo ha terminado. No funcionan ya los cronómetros ni se esperan paquetes.

Podríamos cambiar este protocolo para que los propios reconocimientos se reconozcan a sí mismos, pero esto añade complejidad y costo adicional con poca ganancia potencial. En la práctica, el servidor inicia un cronómetro al enviar la respuesta y descartar ésta cuando llega el reconocimiento o se termina el tiempo. Además, se interpreta una nueva solicitud del cliente como signo de que llegó la respuesta; de otro modo, el cliente no proporcionaría la solicitud siguiente.

Ruta crítica

Puesto que el código RPC es de importancia crucial para el desempeño del sistema, analizaremos más de cerca lo que ocurre en realidad cuando un cliente lleva a cabo una RPC con un servidor remoto. La serie de instrucciones que se ejecutan con cada RPC se llama la **ruta crítica** y se muestra en la figura 2-26. Inicia cuando el cliente llama al resguardo del cliente, sigue con el señalamiento al núcleo, la transmisión del mensaje, la interrupción del lado del servidor, el resguardo del servidor y por último llega al servidor, el cual realiza el trabajo y envía la respuesta de regreso a lo largo de la ruta inversa.

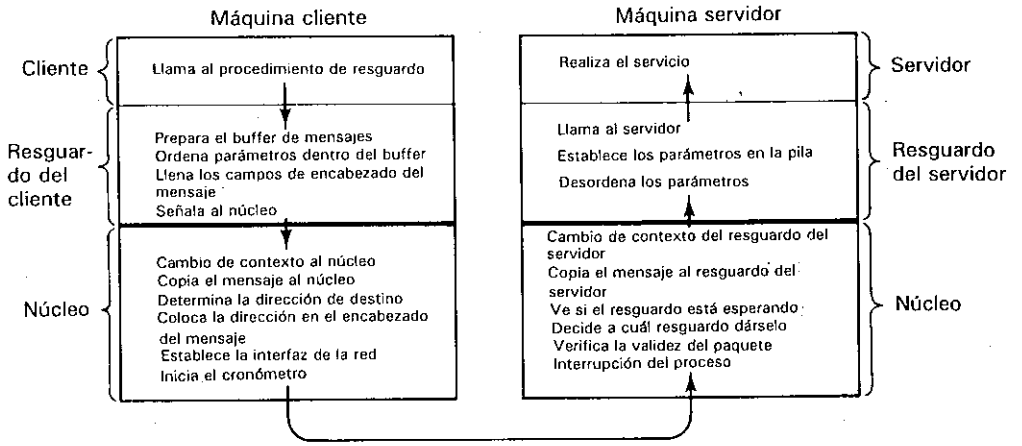


Figura 2-26. Ruta crítica del cliente al servidor.

Examinemos estos pasos con un poco más de cuidado. Después de la llamada al resguardo del cliente, su primer trabajo es conseguir un buffer en el cual concentrar el mensaje de salida. En ciertos sistemas, el resguardo del cliente tiene un buffer fijo que se va llenando a partir de cero en cada llamada. En otros sistemas, se dispone de un fondo de buffers parcialmente utilizados y se obtiene uno adecuado para el servidor necesario. Este método es en particular adecuado cuando el formato subyacente del paquete tiene gran número de campos por llenar, pero que no cambian entre las llamadas.

A continuación, se convierten los parámetros al formato apropiado y se insertan en el buffer de mensajes, junto con el resto de los campos del encabezado, en caso de que existan. En este punto, el mensaje está listo para su transmisión, de modo que se hace un señalamiento al núcleo.

Cuando obtiene el control, el núcleo cambia de contexto, guarda los registros del CPU y el mapa de la memoria y configura un nuevo mapa para utilizarlo durante la ejecución en modo núcleo. Puesto que los contextos del usuario y el núcleo son ajenos por lo general, el núcleo debe copiar ahora en forma explícita el mensaje en su espacio de direcciones de modo que pueda tener acceso a él, llenar la dirección de destino (y los otros posibles campos del encabezado) y copiarlo a la interfaz de la red. En este momento, termina la ruta crítica del cliente, puesto que todo el trabajo que se realice de aquí en adelante no afecta en modo alguno el tiempo total de RPC: nada de lo que pueda hacer el núcleo afectará el tiempo que tarde en llegar el paquete al servidor. Después de iniciar el cronómetro de retransmisión, el núcleo puede entrar en un ciclo de espera ocupada para esperar la respuesta o llamar al planificador para buscar otro proceso por ejecutar. En el primer caso, aumenta la velocidad de procesamiento de la respuesta, pero en términos netos implica que no se puede llevar a cabo una multiprogramación.

Del lado del servidor, llegarán los bits y el hardware receptor los coloca en un buffer contenido en una tarjeta o en la memoria. Cuando todos lleguen, el receptor generará una interrupción. Entonces, el controlador de interrupciones examina el paquete, para ver si es válido y determinar el resguardo al cual va dirigido. Si ningún resguardo lo espera, el controlador puede almacenarlo en un buffer o descartarlo. Si un resguardo lo esperaba, el mensaje se copia en éste. Por último, se hace un cambio de contexto, para restaurar los registros y el mapa de memoria con los valores que tenían al momento en que el resguardo llamó a *receive*.

El servidor puede entonces reiniciarse. Reordena los parámetros y configura un ambiente en el que se puede llamar al servidor. Cuando todo está listo, se lleva a cabo la llamada. Después de la ejecución del servidor, la ruta de regreso al cliente es similar a la ruta hacia adelante, pero en el otro sentido.

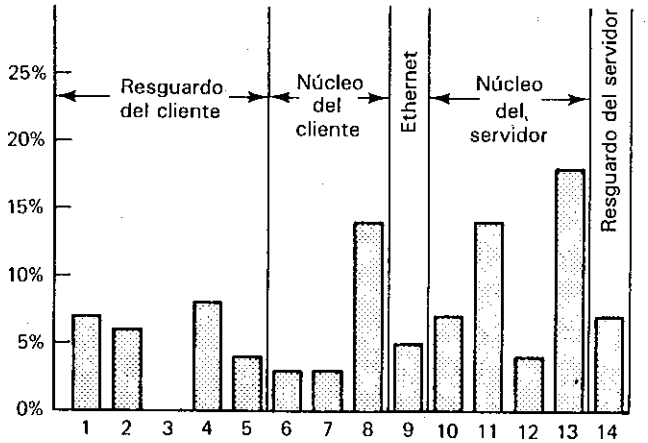
Una cuestión en la que están en particular interesados los implantadores es: ¿en qué parte de la ruta crítica se ocupa la mayor parte del tiempo? Una vez determinada, se puede comenzar a acelerar dicha parte. Schroeder y Burrows (1990) nos han abierto las perspectivas mediante un análisis detallado de la ruta crítica de la RPC en la estación de trabajo con multiprocesador DEC Firefly. Los resultados de su trabajo se muestran como histogramas en la figura 2-27, con 14 barras, las cuales corresponden a uno de los pasos del cliente al servidor (la ruta inversa no se muestra, pero es análoga). La figura 2-27(a) da los resultados de una RPC nula (sin datos) y la figura 2-27(b) da los resultados para un parámetro tipo arreglo de 1 440 bytes. Aunque el costo fijo es el mismo en ambos casos, se necesita mucho más tiempo para el ordenamiento de los parámetros y el desplazamiento de los datos de un lado a otro en el segundo caso.

En el caso de una RPC nula, los costos dominantes son el cambio de contexto al resguardo del servidor al llegar un paquete, la rutina de servicio de interrupciones y el movimiento del paquete a la interfaz de la red para su transmisión. Para la RPC de 1440 bytes, la imagen cambia de manera considerable, pues ahora el tiempo de transmisión en Ethernet es el principal componente, seguido muy de cerca por el tiempo que tarda el desplazamiento del paquete hacia adentro y afuera de la interfaz.

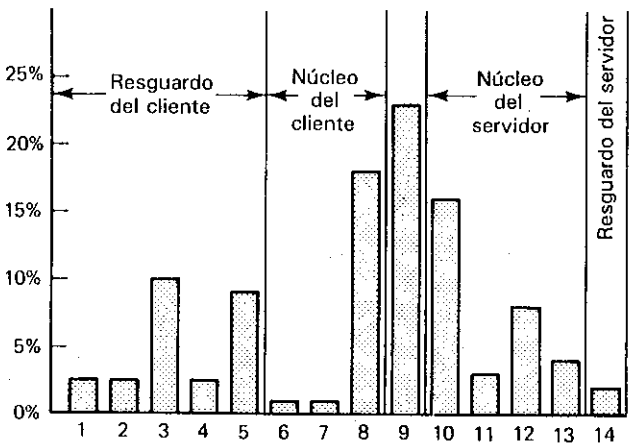
Aunque la figura 2-27 proporciona buena idea de la distribución del tiempo, hay que hacer algunas advertencias necesarias para la interpretación de los datos. En primer lugar, la Firefly es un multiprocesador con cinco CPU VAX. Si se mide de nuevo con un CPU, el tiempo de RPC se duplica, lo cual indica que en este caso se realiza un procesamiento paralelo esencial, algo que no sería cierto en la mayoría de las demás máquinas.

En segundo lugar, la Firefly utiliza UDP y su sistema operativo maneja un fondo de buffers UDP, utilizados por los resguardos de los clientes para no tener que llenar todo el encabezado UDP cada vez.

En tercer lugar, el núcleo y el usuario comparten el mismo espacio de direcciones, lo que elimina la necesidad de cambios de contexto y de copiado entre los espacios del usuario y del núcleo, lo cual ahorra mucho tiempo. Los bits de protección en la tabla de páginas evitan que el usuario lea o escriba en partes del núcleo distintas de los buffers compartidos y otras partes a las que tiene acceso el usuario. Este diseño utiliza de manera inteligente las



(a)



(b)

1. Llama al resguardo
2. Obtiene el buffer de mensajes
3. Ordena los parámetros
4. Llena los encabezados
5. Calcula la suma de verificación UDP
6. Señala al núcleo
7. Forma una fila de paquetes para la transmisión
8. Desplaza el paquete al controlador sobre el QBus
9. Tiempo de transmisión en Ethernet
10. Obtiene el paquete del controlador
11. Interrumpe la rutina de servicio
12. Calcula la suma de verificación UDP
13. Cambio de contexto al espacio del usuario
14. Código del resguardo del servidor

(c)

Figura 2-27. Descomposición de la ruta crítica de RPC. (a) Para una RPC nula. (b) Para una RPC con un parámetro dado por un arreglo de 1 440 bytes. (c) Los 14 pasos de la RPC desde el cliente hasta el servidor.

características particulares de la arquitectura de la VAX, las cuales facilitan el uso compartido del espacio del usuario y del núcleo, pero no es aplicable a todas las computadoras.

En cuarto y último lugar, todo el sistema RPC fue codificado con cuidado en lenguaje ensamblador y optimizado en forma manual. Tal vez este último punto sea la razón de que los distintos componentes de la figura 2-27 sean tan uniformes. No existe duda alguna de que al realizar las primeras mediciones, tenían mayor desviación, lo que hizo que los autores atacaran las partes que consumían más tiempo hasta que ya no provocaran tantos problemas.

Con base en su experiencia, Schroeder y Burrows dan ciertos consejos a los futuros diseñadores. Para comenzar, recomiendan evitar el uso de hardware extraño (sólo uno de los cinco procesadores de Firefly tiene acceso a Ethernet, por lo que los paquetes se deben copiar a dicho procesador antes de ser enviados y su envío hacia él es poco agradable). Tampoco están contentos con haber basado su sistema en UDP. El costo en tiempo, en particular lo referente a la suma de verificación, fue excesivo. En retrospectiva, ellos piensan que hubiese sido mejor un simple protocolo RPC adaptado. Por último, el uso de la espera ocupada en vez de dejar dormir al resguardo del servidor hubiera reducido de manera sustancial la depresión de tiempo mayor de la figura 2-27(a).

Copiado

Un aspecto que domina con frecuencia los tiempos de ejecución de RPC es el copiado. En la máquina Firefly, este efecto no es aparente, puesto que los buffers se asocian con los espacios de direcciones del usuario y del núcleo, pero en la mayoría de los demás sistemas, estos espacios son ajenos. El número de veces que se debe copiar un mensaje varía de uno a ocho, según el hardware, software y tipo de llamada. En el mejor de los casos, el circuito de la red puede utilizar el acceso directo a la memoria (DMA) para transferir el mensaje del espacio de direcciones del resguardo del cliente a la red (copia 1), depositarlo en la memoria del núcleo del servidor en tiempo real (es decir, la interrupción correspondiente al paquete ocurre unos cuantos microsegundos después de que el último bit haya sido transferido por el DMA desde la memoria del resguardo del cliente). Entonces, el núcleo inspecciona el paquete y asocia la página que la contiene en el espacio de direcciones del servidor. Si no se puede llevar a cabo este tipo de asociación, el núcleo copia el paquete al resguardo del servidor (copia 2).

En el peor de los casos, el núcleo copia el mensaje del resguardo del cliente en un buffer del núcleo para su posterior transmisión, ya sea porque no es conveniente transmitirlos en forma directa del espacio del usuario o porque la red se encuentra ocupada por el momento (copia 1). Más tarde, el núcleo copia el mensaje, en software, a un buffer de hardware en la tarjeta de interfaz de la red (copia 2). En este momento, se inicia el hardware, lo que hace que el paquete se desplace a través de la red hacia la tarjeta de interfaz de la máquina destino (copia 3). Cuando la interrupción correspondiente al paquete recién arribado aparece en la máquina del servidor, el núcleo lo copia a un buffer del núcleo, tal vez porque no puede decidir dónde colocarlo hasta no examinarlo, lo cual no es posible sino hasta que lo extraiga del buffer en hardware (copia 4). Por último, el mensaje debe ser copiado al resguardo del

servidor (copia 5). Además, si la llamada transfiere como parámetro un arreglo de gran tamaño, dicho arreglo debe copiarse en la pila del cliente para la llamada del resguardo, de la pila al buffer de mensajes durante el ordenamiento dentro del resguardo del cliente y del mensaje recibido en el resguardo del servidor a la pila del servidor que antecede a la llamada al servidor; es decir, tres copias más, para un total de ocho.

Supongamos que el tiempo para copiar una palabra de 32 bits es de 500 nanosegundos en promedio; entonces, con ocho copias, cada palabra necesita 4 microsegundos, lo que da una tasa máxima de transmisión de los datos de 1Mbyte/segundo, sin importar la velocidad de la red. En la práctica, se considera bueno lograr 1/10 de esta tasa.

Una característica del hardware que es de gran ayuda para eliminar el copiado innecesario es la **dispersión-asociación**. Un circuito de la red que realice la dispersión-asociación se puede configurar de tal manera que organice un paquete mediante la concatenación de dos o más buffers de memoria. La ventaja de este método es que el núcleo puede construir el encabezado del paquete en el espacio del núcleo, lo cual deja los datos del usuario en el resguardo del cliente y el hardware los "empuja" en forma conjunta al salir el paquete. El hecho de poder asociar un paquete a partir de varias fuentes elimina el copiado. En forma similar, si se puede dispersar el encabezado y cuerpo de un paquete recibido en distintos buffers, esto también es de gran ayuda en el otro extremo.

En general, es más fácil eliminar el copiado en el lado emisor que en el receptor. Con un hardware cooperativo, se pueden colocar dentro de la red un encabezado de paquete reutilizable (dentro del núcleo) y un buffer de datos (en el espacio del usuario) sin que haya un copiado interno del lado emisor. Sin embargo, cuando esto llega al lado receptor, incluso un circuito de red muy inteligente no podría determinar el servidor al cual asignarlo, por lo que lo mejor que puede hacer el hardware es vaciarlo a un buffer del núcleo y dejar que éste determine qué debe hacerse con él.

En los sistemas operativos con memoria virtual, se puede utilizar un truco para evitar el copiado al resguardo. Si el buffer del paquete en el núcleo ocupa toda una página, a partir de una frontera de página y el buffer receptor del resguardo del servidor también es toda una página e inicia en una frontera de página, el núcleo puede modificar el mapa de la memoria para asociar el buffer con el paquete en el espacio de direcciones del servidor y enviar de manera simultánea el buffer del resguardo del servidor al núcleo. Cuando el resguardo del servidor inicia su ejecución, su buffer contendrá el paquete, lo cual se habrá logrado sin copiado.

Es importante analizar si todo este trabajo vale la pena. Supongamos de nuevo que el copiado de una palabra de 32 bits tarda 500 nanosegundos y que el copiado de un paquete de 1K tarda 128 microsegundos. Si el mapa de la memoria se puede actualizar en menos tiempo, entonces la asociación del buffer al espacio de direcciones será más rápida que el copiado; si eso no se puede lograr, ocurrirá el caso contrario. Este método también requiere de un control cuidadoso de los buffers; hay que asegurarse de que todos éstos estén alineados en forma adecuada con respecto de las fronteras de página. Si un buffer inicia en una frontera, el proceso usuario puede ver todo el paquete, incluidos los encabezados de bajo nivel, algo que la mayoría de los sistemas intenta ocultar en aras de la portabilidad.

En forma alternativa, si los buffers están alineados de forma que el encabezado se encuentre al final de una página y los datos estén al principio de la siguiente, los datos se pueden asociar con el espacio de direcciones sin el encabezado. Este método es más claro y conciso, pero tiene un costo de dos páginas por buffer: una casi vacía excepto por unos cuantos bytes para el encabezado al final y otra para los datos.

Por último, muchos paquetes tienen unos cuantos cientos de bytes, por lo que no hay duda que la asociación al espacio de direcciones vencerá al copiado. Aún así, es una idea interesante que se debe tomar en cuenta.

Manejo del cronómetro

Todos los protocolos constan de un intercambio de mensajes a través de cierto medio de comunicación. En la gran mayoría de los sistemas, los mensajes se pueden perder de manera ocasional, ya sea debido al ruido o a la sobre-ejecución del receptor. En consecuencia, la mayoría de los protocolos inicia un cronómetro cada vez que se envía un mensaje y se espera una respuesta (réplica o reconocimiento). Si ésta no llega en el tiempo esperado, el contador se detiene y se vuelve a transmitir el mensaje original. Este proceso se repite hasta que el emisor se cansa y se da por vencido.

La cantidad de tiempo de máquina que se dedica al manejo de los cronómetros no debe subestimarse. El establecimiento de un cronómetro requiere la construcción de una estructura de datos que especifique el momento en que el cronómetro debe detenerse y la acción a realizar en caso de que eso suceda. La estructura de datos se inserta entonces en una lista de cronómetros pendientes. Por lo general, esta lista se ordena según el tiempo, con el tiempo de expiración más pequeño al principio de la lista y el más grande al final, como se muestra en la figura 2-28.

Si llega un reconocimiento o réplica antes de que termine el tiempo, hay que localizar la entrada correspondiente y eliminarla de la lista. En la práctica, muy pocos cronómetros ocupan todo su tiempo, por lo que la mayoría del trabajo de introducir y eliminar un cronómetro de la lista es un esfuerzo desperdiciado. Además, no es necesario que los cronómetros sean muy precisos. Por lo general, el valor del tiempo de expiración es, en primera instancia, una aproximación burda ("me parece que unos cuantos segundos están bien"). Además, una elección inadecuada de dicho valor no afecta lo correcto del protocolo, sino sólo su desempeño. Un valor muy pequeño hará que los cronómetros expiren con mucha frecuencia y que se realicen retransmisiones innecesarias. Un valor muy grande provocará un largo retraso innecesario en caso que el paquete en realidad se haya perdido.

La combinación de estos factores sugiere que podría ser eficiente una manera distinta de manejar los cronómetros. La mayoría de los sistemas mantienen una tabla de procesos, donde una entrada contiene toda la información correspondiente a cada proceso del sistema. Mientras se lleva a cabo una RPC, el núcleo tiene un apuntador a la entrada de la tabla correspondiente al proceso activo en una variable local. En vez de guardar los tiempos de expiración en una lista ligada ordenada, cada entrada de la tabla de procesos tiene un campo para su tiempo de expiración, si es que éste existe, como se muestra en la figura 2-28(b).

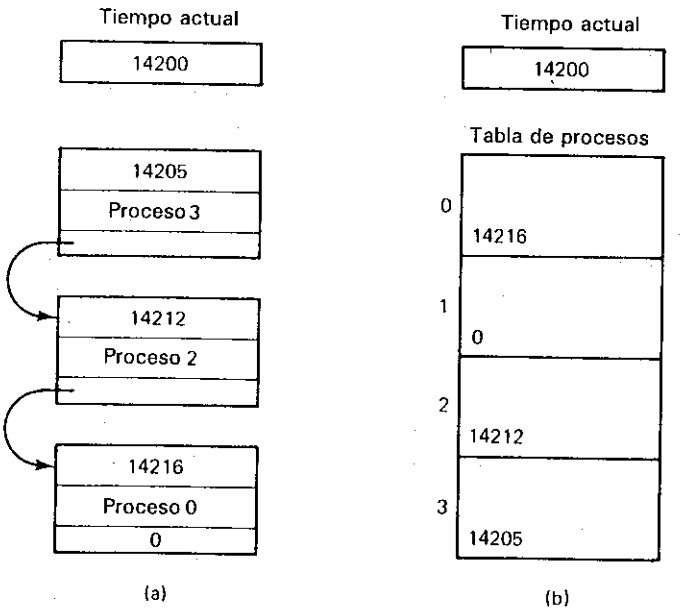


Figura 2-28. (a) Tiempos de espera en una lista ordenada. (b) Tiempos de espera en una tabla de procesos.

La activación de un cronómetro para una RPC consta ahora de la suma de la longitud de su tiempo de expiración al tiempo actual y su almacenamiento en la tabla de procesos. La desactivación de un cronómetro es entonces el almacenamiento del valor 0 en el campo de éste. Así, las acciones de activación y desactivación de los cronómetros se reducen ahora a unas cuantas instrucciones de máquina.

Para que este método funcione, el núcleo debe revisar en forma periódica (digamos, una vez cada segundo) toda la tabla de procesos, para comparar el valor de cada cronómetro con el tiempo actual. Cualquier valor distinto de cero que sea menor o igual que el tiempo actual corresponde a un cronómetro donde el tiempo ha expirado, el cual se procesa y vuelve a activar. Por ejemplo, para un sistema que envíe 100 paquetes/segundo, el trabajo de rastreo de la tabla de procesos una vez por cada segundo es sólo una fracción del trabajo de búsqueda y actualización de una lista ligada 200 veces por segundo. Los algoritmos que operan mediante una transferencia secuencial periódica por medio de una tabla como ésta se llaman **algoritmos de barrido.**

2.4.6. Áreas de problemas

La llamada a procedimientos remotos mediante el modelo cliente-servidor se utiliza ampliamente como base de los sistemas operativos distribuidos. Es una abstracción sencilla, que hace más manejable el uso de la complejidad inherente en un sistema distribuido de lo

que sería con la trasferencia de mensajes pura. Sin embargo, existen algunas áreas problemáticas por resolver. En esta sección analizaremos algunas de ellas.

Lo ideal es que la RPC sea transparente. Es decir, el programador no debe saber si los procedimientos de biblioteca son locales o remotos. También debe escribir procedimientos sin importar si serán ejecutados en forma local o remota. Más estrictamente, la introducción de RPC en un sistema que se ejecutaba antes en un CPU no debe ir acompañada de una serie de reglas que prohíban construcciones ya válidas o que exijan construcciones que antes eran opcionales. Con este criterio tan estrecho, pocos, si no es que ninguno de los sistemas distribuidos actuales, pueden considerarse transparentes. Así, el cáliz sagrado de la transparencia permanece como un tema de investigación en el futuro cercano.

Como ejemplo de esto, consideremos el problema de las variables globales. En los sistemas con un CPU, éstas eran válidas, incluso para los procedimientos de biblioteca. Por ejemplo, en UNIX existe una variable global *errno*. Después de una llamada incorrecta al sistema, *errno* contiene un código que indica lo que estuvo mal. La existencia de *errno* es información pública, ya que el estándar oficial de UNIX, POSIX, exige que sea visible en uno de los archivos de encabezado imperativos, *errno.h*. Así, no se permite que una implantación lo oculte de los programadores.

Supongamos ahora que un programador escribe dos procedimientos que tienen acceso directo a *errno*. Uno de ellos se ejecuta en forma local; el otro en forma remota. Puesto que el compilador no conoce (y no puede conocer) la posición de las variables y procedimientos, no importa la localización de *errno*, uno de los procedimientos tendrá un acceso incorrecto. El problema es que no se puede implantar el permiso para el acceso irrestricto de los procedimientos locales a las variables globales remotas y viceversa, a la vez que la prohibición de este acceso viola el principio de transparencia (esos programas no deben actuar de manera distinta debido a RPC).

Un segundo problema son los lenguajes débilmente tipificados, como C. En un lenguaje fuertemente tipificado, como Pascal, el compilador y con él el procedimiento de resguardo, conocen todo lo relativo a todos los parámetros. Este conocimiento permite al resguardo ordenar los parámetros sin dificultad. Sin embargo, en el caso de C, es perfectamente válido escribir un procedimiento que calcule el producto interior de dos vectores (arreglos), sin especificar el tamaño de cada uno. Cada uno de ellos podría terminar con un valor especial conocido sólo por el procedimiento que hace la llamada y el que lo recibe. En esas circunstancias, es esencialmente imposible que el resguardo del cliente ordene los parámetros: no tiene forma de determinar su tamaño.

La solución usual es obligar al programador a definir el tamaño máximo cuando escriba la definición formal del servidor, pero ¿y si el programador quisiera que su procedimiento funcionara con cualquier tamaño? Podría imponer cierto límite a la especificación, digamos, un millón, pero eso quiere decir que debería transferir un millón de elementos, aunque el tamaño real del arreglo fuese de 100. Además, la llamada fallaría si el arreglo fuese de 1 000 001 elementos o si la memoria total sólo pudiese contener 200 000 elementos.

Un problema similar aparece al transferir como parámetro un apuntador a una gráfica compleja. En un sistema con una CPU, esto funciona bien, pero con RPC, el resguardo del cliente no tiene forma de encontrar toda la gráfica.

También puede ocurrir otro problema, puesto que no siempre es posible deducir los tipos de los parámetros, ni siquiera a partir de una especificación formal del propio código. Un ejemplo de esto es *printf*, que puede tener un número arbitrario de parámetros (al menos uno) que pueden ser una mezcla arbitraria de enteros, cortos, largos, caracteres, cadenas, números con punto flotante de cualquier longitud y otros tipos. El intento por llamar a *printf* como un procedimiento remoto sería prácticamente imposible, puesto que C permite muchas cosas. Sin embargo, una regla que indique que RPC se puede utilizar siempre y cuando no se programe en C violaría la transparencia.

Los problemas anteriores se refieren a la transparencia, pero existe otra clase de dificultades que son todavía más fundamentales. Consideremos la implantación del comando de UNIX

```
sort <f2 >f2
```

Puesto que *sort* sabe que está leyendo de la entrada estándar y escribiendo en la salida estándar, puede actuar como un cliente de ambas y llevar a cabo RPC con el servidor del cliente para leer *f1*, así como hacer RPC con el servidor de archivos para escribir *f2*. En forma similar, en el comando

```
grep rat <f3 >f4
```

el programa *grep* actúa como un cliente para leer el archivo *f3* y extraer de él todas las líneas que contengan la línea "rat" y escribirlas en *f4*.

Consideremos ahora el entubamiento en UNIX

```
grep rat < f5 | sort >f6
```

Como hemos visto, tanto *grep* como *sort* actúan como clientes de la entrada y la salida estándares. Este comportamiento se puede compilar en el código para que ambos ejemplos funcionen. ¿Pero qué hacer si interactúan? ¿Actúa *grep* como cliente para escribir al servidor *sort*, o bien *sort* actúa como el cliente para leer del servidor *grep*? De cualquier forma, alguno de ellos debe actuar como el servidor (es decir, pasivo), pero como hemos visto, ambos se programan como clientes (activos). La dificultad aquí es que el modelo cliente-servidor no es adecuado.

En general, existe un problema con todos los entubamientos de la forma

```
p1  <f1 | p2 | p3 > f2
```

Un método ya visto para evitar esta interfaz cliente-cliente es que todo el entubamiento sea **manejado por la lectura**, como se muestra en la figura 2-29(b). El programa *p1* actúa como el cliente (activo) y hace una solicitud de lectura al servidor de archivos para obtener *f1*. El programa *p2*, que también actúa como cliente, hace una solicitud de lectura a *p1* y el programa *p3* hace una solicitud de lectura a *p2*. Hasta aquí, todo va bien. El problema es que el servidor de archivos no actúa como un cliente y hace solicitudes de lectura a *p3* para recolectar la salida final. Así, un entubamiento manejado por la lectura no funciona.

En la figura 2-29(c) vemos el método controlado por la escritura. Tiene un problema similar al anterior. Aquí, *p1* actúa como un cliente, escribiendo en *p2*, el cual también actúa como un cliente, escribiendo a *p3*, que también actúa como cliente, escribiendo en el servidor

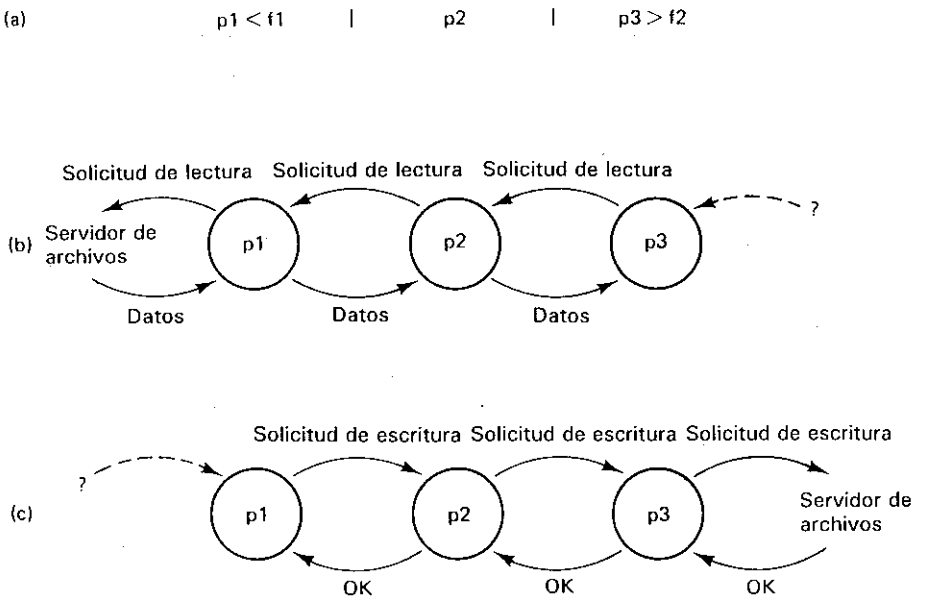


Figura 2-29. (a) Un entubamiento. (b) El método determinado por la lectura. (c) El método determinado por la escritura.

de archivos. Sólo que entonces no existe un cliente que haga llamadas a *p1* y pida que acepte el archivo de entrada.

Aunque se pueden encontrar soluciones adecuadas, debe quedar claro que el modelo cliente-servidor inherente a RPC no se ajusta a este tipo de patrón de comunicación. Por otro lado, una posible solución adecuada es implantar los entubamientos como servidores duales, que respondan a solicitudes de la izquierda y lean solicitudes de la derecha. Otra alternativa es que los entubamientos se pueden implantar con archivos temporales que siempre se leen o escriben en el servidor de archivos. Sin embargo, esto genera un costo adicional.

Un problema similar aparece cuando el shell desea obtener una entrada del usuario. Por lo general, envía solicitudes de lectura al servidor de la terminal, la cual reúne las combinaciones de teclas y espera hasta que shell las solicite. ¿Qué ocurre si el usuario oprime la tecla de interrupción (DEL, CTRL-C, break, etc.)? Si el servidor de la terminal sólo coloca de manera pasiva el carácter de interrupción en el buffer y espera a que shell lo solicite, será imposible que el usuario detenga el programa que se ejecute en ese momento. Por otro lado, ¿cómo podría actuar el servidor de la terminal como un cliente y hacer una RPC al shell, que no se espera actúe como un servidor? Es claro que esta inversión de papeles provoca ciertos problemas, al igual que lo hace la ambigüedad de papeles en el entubamiento. De hecho, cada vez que se deba enviar un mensaje inesperado existe un problema potencial. Aunque el modelo cliente-servidor se ajusta a muchos casos, no es perfecto.

2.5. COMUNICACIÓN EN GRUPO

Una hipótesis subyacente e intrínseca de RPC es que la comunicación sólo es entre *dos* partes, el cliente y el servidor. A veces, existen ciertas circunstancias en las que la comunicación es entre varios procesos y no solamente dos. Por ejemplo, consideremos un grupo de servidores de archivos que cooperan para ofrecer un servicio de archivos tolerante de fallas. En tal sistema, sería recomendable que un cliente envíe el mensaje a todos los servidores, para garantizar que la solicitud se lleve a cabo, aún en el caso en que uno de ellos falle. RPC no puede controlar la comunicación de un servidor con muchos receptores, a menos que realice RPC con cada uno en forma individual. En esta sección analizaremos las alternativas de comunicación en las que un mensaje se puede enviar a varios receptores en una operación.

2.5.1. Introducción a la comunicación en grupo

Un grupo es una colección de procesos que actúan juntos en cierto sistema o alguna forma determinada por el usuario. La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros de éste lo reciben. Es una forma de comunicación **uno-muchos** (un emisor, muchos receptores) y contrasta con la **comunicación puntual** de la figura 2-30.

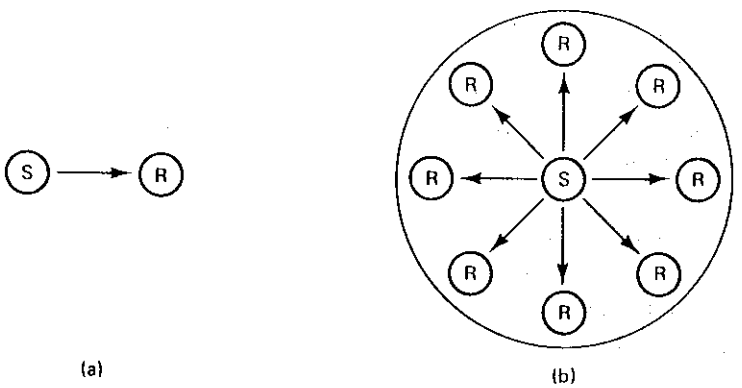


Figura 2-30. (a) La comunicación puntual se establece de un emisor a un receptor. (b) La comunicación uno-muchos se establece entre un emisor y varios receptores.

Los grupos son dinámicos. Se pueden crear nuevos grupos y destruir grupos anteriores. Un proceso se puede unir a un grupo o dejar otro. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se necesitan mecanismos para el manejo de grupos y la membresía de los mismos.

Los grupos son algo parecido a las organizaciones sociales. Una persona puede ser miembro de un club de lectores, un club de tenis y una organización ambientalista. En un día particular, él podría recibir correo (mensajes) que le avisen de un nuevo libro para hornear pasteles de cumpleaños, el torneo anual del día de las madres y el inicio de una campaña para salvar a las marmotas del sur. En cualquier momento, él es libre de dejar todos o alguno de estos grupos y unirse a otros.

Aunque en este libro sólo estudiaremos los grupos (de procesos) del sistema operativo, es importante mencionar que es común encontrar otros grupos en los sistemas de cómputo. Por ejemplo, en la red de cómputo USENET, existen cientos de grupos de noticias, cada uno en torno de un tema específico. Cuando una persona envía un mensaje a un grupo particular de noticias, todos los miembros del grupo lo reciben, aunque existan decenas de cientos de ellos. Estos grupos de nivel superior tienen reglas más vagas acerca de quién es un miembro, la semántica exacta de la entrega de mensajes, etc., en relación con los grupos del sistema operativo. En la mayoría de los casos, esta vaguedad no es un problema.

La finalidad de presentar los grupos es permitir a los procesos que trabajen con colecciones de procesos como una abstracción. Así, un proceso puede enviar un mensaje a un grupo de servidores sin tener que conocer su número o su localización, que puede cambiar de una llamada a la siguiente.

La implantación de la comunicación en grupo depende en gran medida del hardware. En ciertas redes, es posible crear una dirección especial de red (por ejemplo, indicada al hacer que uno de los bits de orden superior tome el valor 1) a la que pueden escuchar varias máquinas. Cuando se envía un mensaje a una de estas direcciones, se entrega de manera automática a todas las máquinas que escuchan a esa dirección. Esta técnica se llama **multitransmisión**. La implantación de los grupos mediante multitransmisión es directa: sólo hay que asignar a cada grupo una dirección de multitransmisión distinta.

Las redes que no tienen multitransmisión a menudo siguen teniendo **transmisión simple**, lo que significa que los paquetes que contienen cierta dirección (por ejemplo, 0) se entregan a todas las máquinas. La transmisión simple también se puede utilizar para implantar los grupos, pero es menos eficiente. Cada máquina recibe su transmisión simple, por lo que su software debe verificar si el paquete va dirigido a ella. Si no, el paquete se descarta, pero se pierde cierto tiempo durante el procesamiento de la interrupción. Sin embargo, un paquete llega a todos los miembros del grupo.

Por último, si no se dispone de la multitransmisión o la transmisión simple, se puede implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo. Para un grupo con n miembros, se necesitan n paquetes, en vez de un paquete en el caso de la multitransmisión o la transmisión simple. Aunque es menos eficiente, esta implantación también funciona, en particular con grupos pequeños. El envío de un mensaje de un emisor a un receptor se llama a veces **unitransmisión**, para distinguirla de los otros tipos de transmisión.

2.5.2. Aspectos del diseño

La comunicación en grupo tiene posibilidades de diseño similares a la transferencia regular de mensajes, como el almacenamiento en buffers *vs.* el no almacenamiento, bloqueo *vs.* no bloqueo, etc. Sin embargo, también existen un gran número de opciones adicionales por realizar, ya que el envío a un grupo es distinto de manera inherente del envío a un proceso. Además, los grupos se pueden organizar internamente de varias formas. También se pueden direccionar de formas novedosas que no son importantes en la comunicación puntual. En esta sección analizaremos algunos de los aspectos más importantes del diseño y señalaremos las distintas alternativas.

Grupos cerrados *vs.* grupos abiertos

Los sistemas que soportan la comunicación en grupo se pueden dividir en dos categorías, según quién pueda enviar a quién. Algunos sistemas soportan los **grupos cerrados**, donde sólo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a miembros del grupo en lo individual. En contraste, otros sistemas soportan los **grupos abiertos**, que no tienen esta propiedad. Si se utilizan los grupos abiertos, cualquier proceso del sistema puede enviar a cualquier grupo. La diferencia entre los grupos cerrados y abiertos se muestra en la figura 2-31.

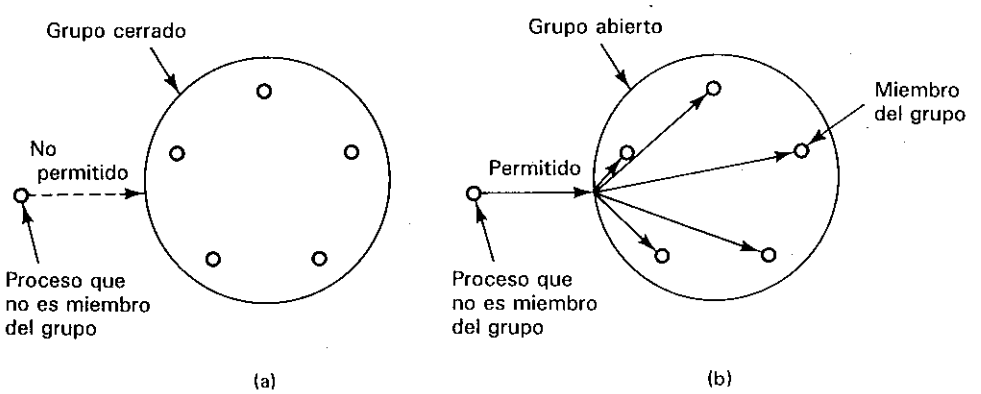


Figura 2-31. (a) Los extraños no pueden hacer envíos a un grupo cerrado. (b) Los extraños pueden hacer envíos a un grupo abierto.

La decisión si el sistema soporta grupos cerrados o abiertos se relaciona por lo general con la razón primaria por la que se soportan los grupos. Los grupos cerrados se utilizan en general para el procesamiento paralelo. Por ejemplo, una colección de procesos que trabajan de manera conjunta para jugar una partida de ajedrez podrían formar un grupo cerrado. Tienen su propio objetivo y no interactúan con el mundo exterior.

Por otro lado, cuando la idea de grupos pretende soportar servidores duplicados, entonces es importante que los procesos que no sean miembros (clientes) puedan enviar mensajes hacia el grupo. Además, podría ser necesario que los miembros del grupo utilizaran la comunicación en grupo; por ejemplo, para decidir quién debe llevar a cabo una solicitud particular. La distinción entre los grupos abiertos y cerrados se hace por lo general por razones de implantación.

Grupos de compañeros *vs.* grupos jerárquicos

La distinción entre los grupos cerrados y abiertos se relaciona con la pregunta de quién se puede comunicar con el grupo. Otra distinción importante tiene que ver con la estructura interna del grupo. En algunos grupos, todos los procesos son iguales. Nadie es el jefe y todas las decisiones se toman en forma colectiva. En otros grupos, existe cierto tipo de jerarquía. Por ejemplo, un proceso es el coordinador y todos los demás son trabajadores. En este modelo, si se genera una solicitud de un trabajo, ya sea de un cliente externo o uno de los trabajadores, ésta se envía al coordinador. Éste decide entonces cuál de los trabajadores es el más adecuado para llevarla a cabo y se envía. Por supuesto, también son posibles jerarquías más complejas. Estos patrones de comunicación se muestran en la figura 2-32.

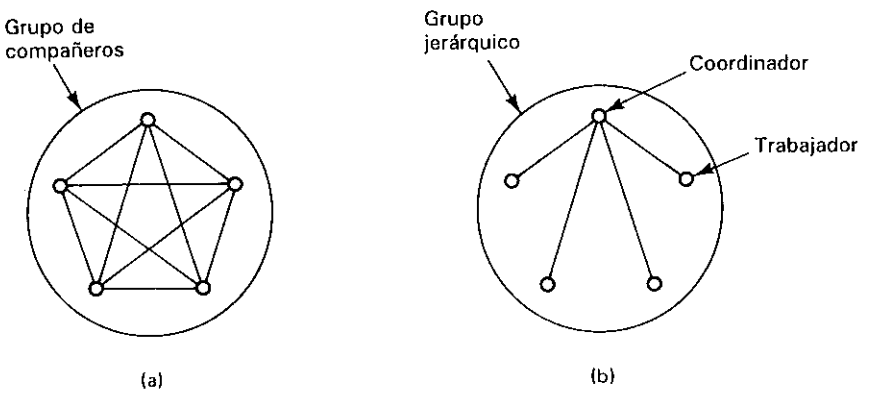


Figura 2-32. (a) La comunicación en un grupo de compañeros. (b) La comunicación en un simple grupo jerárquico.

Cada una de estas organizaciones tiene sus propias ventajas y desventajas. El grupo de compañeros es simétrico y no tiene punto de falla. Si uno de los procesos falla, el grupo sólo se vuelve más pequeño, pero puede continuar. Una desventaja es que la toma de decisiones es más difícil. Para tomar una decisión, hay que pedir un voto, lo que produce cierto retraso y costo.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador lleva a todo el grupo a un agobiante alto, pero mientras se mantenga en ejecución, puede tomar decisiones sin molestar a los demás. Por ejemplo, un grupo jerárquico podría ser adecuado

para un programa de ajedrez en paralelo. El coordinador toma el tablero actual, genera todos los movimientos válidos a partir de él y los envía a los trabajadores para su evaluación. Durante ésta, se generan nuevos tableros y envían de regreso al coordinador. Cuando un trabajador está inactivo, solicita al coordinador un nuevo tablero en el cual trabajar. De esta forma, el coordinador controla la estrategia de búsqueda y desarrolla el árbol del juego (por ejemplo, mediante el método de búsqueda alfa-beta), pero deja a los trabajadores la evaluación real.

Membresía del grupo

Si se utiliza la comunicación en grupo, se requiere cierto método para la creación y eliminación de grupos, así como para permitir a los procesos que se unan o dejen grupos. Un posible método es tener un **servidor de grupos** al cual se pueden enviar todas las solicitudes. El servidor de grupos puede mantener entonces una base de datos de todos los grupos y sus membresías exactas. Este método es directo, eficiente y fácil de implantar. Por desgracia, comparte una desventaja fundamental con todas las técnicas centralizadas: un punto de falla. Si el servidor de grupos falla, deja de existir el manejo de los mismos. Es probable que la mayoría o todos los grupos deban reconstruirse a partir de cero, terminando con todo el trabajo realizado hasta entonces.

El método opuesto es manejar la membresía de grupo en forma distribuida. En un grupo abierto, un extraño puede enviar un mensaje a todos los miembros del grupo para anunciar su presencia. En un grupo cerrado se necesita algo similar (de hecho, incluso los grupos cerrados deben estar abiertos a la opción de admitir otro miembro). Para salir de un grupo, basta que el miembro envíe un mensaje de despedida a todos.

Hasta aquí, todo es directo. Sin embargo, existen dos aspectos asociados con la membresía que tienen unos cuantos trucos. En primer lugar, si un miembro falla, en realidad sale del grupo. El problema es que no existe un anuncio apropiado de este hecho como en el caso en que un proceso salga del grupo en forma voluntaria. Los demás miembros deben descubrir esto en forma experimental, al observar que el miembro ya no responde. Una vez verificado que el miembro en realidad está inactivo, puede eliminarse del grupo.

Otro aspecto problemático es que la salida y la entrada al grupo debe sincronizarse con el envío de mensajes. En otras palabras, a partir del momento en que un proceso se ha unido a un grupo, debe recibir todos los mensajes que se envíen al mismo. De manera similar, tan pronto un proceso salga del grupo, no debe recibir más mensajes de éste y los otros miembros no deben recibir más mensajes de dicho proceso. Una forma de garantizar que una entrada o salida se integra al flujo de mensajes en el lugar correcto es convertir esta operación en un mensaje a todo el grupo.

Un aspecto final relacionado con la membresía es la acción a realizar si fallan tantas máquinas que el grupo ya no pueda funcionar. Se necesita cierto protocolo para reconstruir el grupo. De manera invariable, alguno de los procesos deberá tomar la iniciativa, pero ¿qué ocurre si dos o tres lo intentan al mismo tiempo? El protocolo debe poder resolver esto.

Direccionamiento al grupo

Para enviar un mensaje a un grupo, un proceso debe tener una forma de especificar dicho grupo. En otras palabras, los grupos deben poder direccionarse, al igual que los procesos. Una forma es darle a cada grupo una dirección, parecida a una dirección de proceso. Si la red soporta la multitransmisión, la dirección del grupo se puede asociar con una dirección de multitransmisión, de forma que cada mensaje enviado a la dirección del grupo se pueda multitransmitir. De esta forma, el mensaje será enviado a todas las máquinas que lo necesiten y a ninguna más.

Si el hardware no soporta la multitransmisión pero sí la transmisión simple, el mensaje se puede transmitir. Cada núcleo lo recibirá y extraerá de él la dirección del grupo. Si ninguno de los procesos en la máquina es un miembro del grupo, entonces se descarta la transmisión. En caso contrario, se transfiere a todos los miembros del grupo.

Por último, si no se soporta la multitransmisión o la transmisión simple, el núcleo de la máquina emisora debe contar con una lista de las máquinas que tienen procesos pertenecientes al grupo, para entonces enviar a cada una un mensaje puntual. Estos tres métodos de implantación se muestran en la figura 2-33. El detalle importante a observar es que en los tres casos, un proceso envía un mensaje a una dirección de grupo, que se entrega a todos los miembros. La forma en que ocurre esto es labor del sistema operativo. El emisor no conoce el tamaño del grupo o si la comunicación se implanta mediante multitransmisión, transmisión simple o unitransmisión.

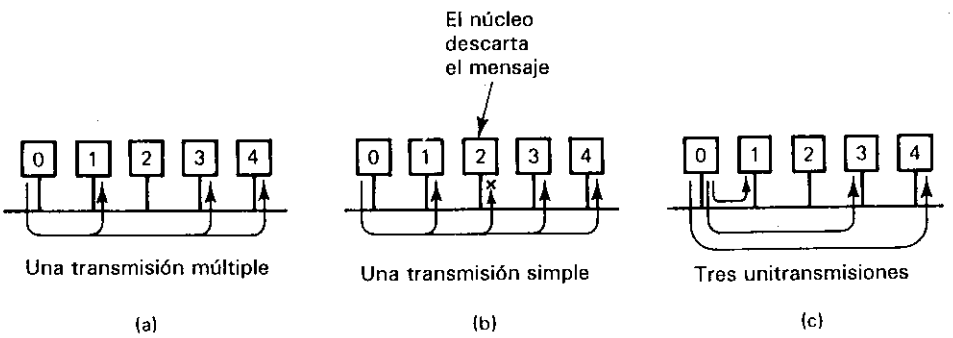


Figura 2-33. El proceso 0 enviado a un grupo que consta de los procesos 1, 3 y 4. (a) Implantación de transmisión múltiple. (b) Implantación de transmisión simple. (c) Implantación de unitransmisión.

Un segundo método de direccionamiento de grupo consiste en pedir al emisor una lista explícita de todos los destinos (por ejemplo, direcciones IP). Si se utiliza este método, el parámetro de la llamada *send* que especifica el destino es un apuntador a una lista de direcciones. Este método tiene la seria desventaja de que obliga a los procesos del usuario (es decir, a los miembros del grupo) a ser conscientes de quién es miembro de cada grupo. En otras palabras, no es transparente. Además, cuando cambia la membresía del grupo, los

procesos del usuario deben actualizar sus listas de miembros. En la figura 2-33, los núcleos pueden realizar con facilidad esta administración para ocultarla de los procesos del usuario.

La comunicación en grupo también permite un tercer método, un tanto novedoso, de direccionamiento, que llamaremos **direccionamiento de predicados**. Con este sistema, se envía cada mensaje a todos los miembros del grupo (o tal vez a todo el sistema) mediante uno de los métodos ya descritos, pero con nuevo giro. Cada mensaje contiene un predicado (expresión booleana) para ser evaluado. El predicado puede utilizar el número de máquina del receptor, sus variables locales u otros factores. Si el valor del predicado es verdadero, se acepta el mensaje. Si es falso, el mensaje se descarta. Mediante este esquema, es posible, por ejemplo, enviar un mensaje sólo a aquellas máquinas que tengan al menos 4M de memoria libre y estén dispuestas a ocuparse de un nuevo proceso.

Primitivas send y receive

En forma ideal, la comunicación puntual y la comunicación en grupo deberían combinarse en un conjunto de primitivas. Sin embargo, si RPC es el mecanismo usual de comunicación del usuario, en vez de los simples *send* y *receive*, entonces es difícil combinar RPC y la comunicación en grupo. El envío de un mensaje a un grupo no se puede modelar como llamada a un procedimiento. La principal dificultad es que, con RPC, el cliente envía un mensaje al servidor y obtiene de regreso una respuesta. Con la comunicación en grupo, existen en potencia *n* respuestas diferentes. ¿Cómo podría trabajar una llamada de procedimiento con *n* respuestas? En consecuencia, un método común es abandonar el modelo solicitud/respuesta (en los dos sentidos) subyacente en RPC y regresar a las llamadas explícitas para el envío y recepción (modelo de un sentido).

Los procedimientos de biblioteca llamados por los procesos para realizar la comunicación en grupo pueden ser iguales o distintos a los que se utilizan para la comunicación puntual. Si el sistema se basa en RPC, los procesos usuario nunca llaman en forma directa a *send* y *receive*, por lo que existen menos incentivos para la fusión de las primitivas puntuales y de grupo. Si los programas usuario llaman en forma directa a *send* y *receive*, hay que hacer algo para realizar la comunicación en grupo con estas primitivas ya existentes, en lugar de inventar un conjunto nuevo.

Supongamos, por el momento, que queremos combinar las dos formas de comunicación. Para enviar un mensaje, uno de los parámetros de *send* indica el destino. Si es una dirección de un proceso, se envía un mensaje a ese proceso en particular. Si es una dirección de grupo (o un apuntador a una lista de destinos), se envía un mensaje a todos los miembros del grupo. Un segundo parámetro de *send* apunta hacia el mensaje por enviar.

La llamada se puede o no almacenar en un buffer, puede o no utilizar bloqueo, ser confiable o no confiable, para cualquiera de los casos puntual o de grupo. Por lo general, los diseñadores del sistema eligen entre estas opciones y las dejan fijas, en vez de ser elegibles para cada mensaje. La introducción de la comunicación en grupo no modifica esto.

En forma análoga, *receive* indica una disposición para aceptar un mensaje y es posible que se bloquee hasta disponer de uno. Si se combinan las dos formas de comunicación, entonces *receive* termina su labor cuando llega un mensaje puntual o un mensaje de grupo. Sin embargo, puesto que estas dos formas de comunicación se utilizan con frecuencia para fines distintos, algunos sistemas introducen nuevos procedimientos de biblioteca, por ejemplo, *group_send* y *group_receive*, para que un proceso pueda indicar si desea un mensaje puntual o de grupo.

En el diseño recién descrito, la comunicación tiene un sentido. Las respuestas son mensajes independientes y no están asociadas con solicitudes previas. A veces es deseable esta asociación, para intentar tener un poco de RPC. En este caso, después de enviar un mensaje, se necesita un proceso *getreply* para llamarlo varias veces y coleccionar todas las respuestas, una a la vez.

Atomicidad

Una característica de la comunicación en grupo a la que hemos aludido varias veces es la propiedad del todo o nada. La mayoría de los sistemas de comunicación en grupo están diseñados de forma que, cuando se envíe un mensaje a un grupo, éste llegue de manera correcta a todos los miembros del grupo o a ninguno de ellos. No se permiten situaciones en las que ciertos miembros reciben un mensaje y otros no. La propiedad del todo o nada en la entrega se llama **atomicidad** o **transmisión atómica**.

La atomicidad es deseable, puesto que facilita la programación de los sistemas distribuidos. Si un proceso envía un mensaje al grupo, no tiene que preocuparse por qué hacer si alguno de ellos no lo obtiene. Por ejemplo, en un sistema distribuido duplicado de una base de datos, supongamos que un proceso envía un mensaje a todas las máquinas de las bases de datos para crear un nuevo registro en la base y que posteriormente envía un segundo mensaje para actualizarlo. Si alguno de los miembros pierde el mensaje de creación del registro, no podrá llevar a cabo la actualización y la base de datos será inconsistente. La vida será más simple si el sistema garantiza que cada mensaje se entrega a todos los miembros del grupo; o bien, si eso no es posible, que no se le entregue a ninguno, además de que esa falla se debe informar de regreso al emisor, de modo que pueda llevar a cabo la acción adecuada para la recuperación.

La implantación de la transmisión atómica no es tan sencilla como parece. El método de la figura 2-33 falla, puesto que es posible la sobre-ejecución del receptor en una o varias máquinas. La única forma de garantizar que cada destino recibe todos sus mensajes es pedirle que envíe de regreso un reconocimiento después de recibir el mensaje. Mientras las máquinas no fallen, este método funciona.

Sin embargo, muchos sistemas distribuidos contemplan la tolerancia de fallas, por lo que para ellos es esencial que se cumpla la atomicidad incluso con la presencia de fallas de máquina. En vista de esto, todos los métodos de la figura 2-33 son inadecuados, puesto que alguno de los mensajes iniciales podría no llegar a su destino, debido a la sobre-ejecución del receptor seguida por una falla del emisor. En estas circunstancias, algunos miembros

del grupo habrán recibido el mensaje y otros no, que es precisamente la situación inaceptable. Peor aún, los miembros del grupo que no recibieron el mensaje ni siquiera saben que les falta algo, por lo que no pueden pedir una retransmisión. Por último, si el emisor falla, aunque lo supieran, no hay forma de proporcionar el mensaje.

Sin embargo, hay una esperanza. He aquí un sencillo algoritmo que demuestra al menos la posibilidad de la transmisión atómica. El emisor comienza con el envío de un mensaje a todos los miembros del grupo. Los cronómetros se activan y se envían las retransmisiones en los casos necesarios. Cuando un proceso recibe un mensaje, si no ha visto ya este mensaje particular, lo envía también a todos los miembros del grupo (de nuevo con cronómetros y retransmisiones en los casos necesarios). Si ya ha visto el mensaje, este paso no es necesario y el mensaje se descarta. No importa el número de máquinas que fallen o el número de paquetes perdidos; en cierto momento, todos los procesos sobrevivientes obtendrán el mensaje. Más adelante describiremos algoritmos más eficientes para garantizar la atomicidad.

Ordenamiento de mensajes

Para que la comunicación en grupo sea fácil de comprender y utilizar, se necesitan dos propiedades. La primera es la transmisión atómica, ya analizada. Ésta garantiza que un mensaje enviado al grupo llegue a todos los miembros o a ninguno. La segunda propiedad se refiere al ordenamiento de mensajes. Para ver lo que es ese aspecto, consideremos la figura 2-34, en la que tenemos cinco máquinas, cada una con un proceso. Los procesos 0, 1, 3 y 4 pertenecen al mismo grupo. En forma simultánea, los procesos 0 y 4 desean enviar un mensaje al grupo. Supongamos que no se dispone de multitransmisión o transmisión simple, por lo que cada proceso debe enviar tres mensajes independientes (unitransmisión). El proceso 0 envía al 1, 3 y 4; el proceso 4 envía al 0, 1 y 3. Estos seis mensajes se muestran intercalados en el tiempo en la figura 2-34(a).

El problema es que cuando dos procesos contienden por el acceso a una LAN, el orden de envío de los mensajes no es determinista. En la figura 2-34(a), vemos que (por accidente), el proceso 0 ha ganado la primera ronda y envía hacia el proceso 1. Entonces, el proceso 4 gana tres rondas seguidas y envía a los procesos 0, 1 y 3. Por último, el proceso 0 envía al 3 y 4. El orden de estos seis mensajes se muestra de distintas formas en las dos partes de la figura 2-34.

Consideremos ahora la situación vista por los procesos 1 y 3, como se muestra en la figura 2-34(b). En primer lugar, el proceso 1 recibe un mensaje de 0, para recibir después uno de 4. En un principio, el proceso 3 no recibe nada, para después recibir mensajes de 4 y 0, en ese orden. Así, los dos mensajes llegan en un orden distinto. Si los procesos 0 y 4 intentan actualizar el mismo registro de una base de datos, 1 y 3 terminarán con distintos valores finales. No es necesario decir que esta situación es tan mala como aquella en la que un mensaje (multitransmisión verdadera en hardware) enviado al grupo llegue a algunos miembros y a otros no (falla de atomicidad). Así, para hacer razonable la programación, un

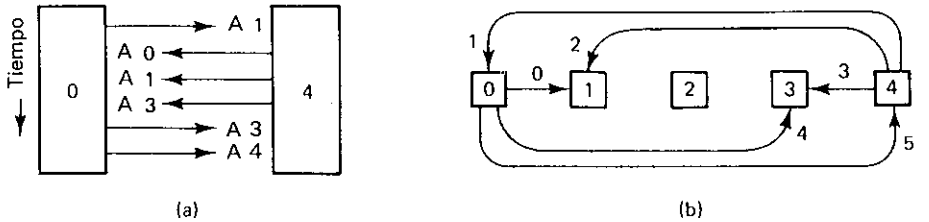


Figura 2-34. (a) Los tres mensajes enviados por el proceso 0 y 4 se intercalan al paso del tiempo. (b) Representación gráfica de los seis mensajes, donde se muestra el orden de llegada.

sistema debe tener una semántica bien definida con respecto al orden de entrega de los mensajes.

La mejor garantía es la entrega inmediata de todos los mensajes, en el orden en que fueron enviados. Si el proceso 0 envía el mensaje *A* y un poco después el proceso 4 envía el mensaje *B*, el sistema debe entregar en primer lugar *A* a todos los miembros del grupo y después entregar *B* a todos los miembros del grupo. De esta forma, todos los receptores obtiene todos los mensajes en el mismo orden. Este patrón de entrega es algo comprensible para los programadores y en el cual basan su software. Lo llamaremos **ordenamiento con respecto al tiempo global**, puesto que entrega todos los mensajes en el orden preciso con el que fueron enviados (aquí ignoramos por conveniencia el hecho de que, de acuerdo con la teoría de la relatividad de Einstein, no existe una cosa tal como el tiempo global absoluto).

El ordenamiento del tiempo absoluto no siempre es fácil de implantar, por lo que ciertos sistemas ofrecen distintas variantes moderadas. Una de éstas es el ordenamiento consistente, en el cual, si dos mensajes *A* y *B* se envían muy cercanos en el tiempo, el sistema elige uno de ellos como el "primero" y lo envía a todos los miembros del grupo, seguido por el otro. Podría ocurrir que el proceso elegido como primero no lo era en realidad, pero como nadie lo sabe, el comportamiento del sistema no debería depender de él. En efecto, se garantiza que los mensajes lleguen a todos los miembros del grupo en el mismo orden, pero ese orden podría no ser aquel en el que fueron enviados.

Se han utilizado otros ordenamientos más débiles. En una sección posterior de este capítulo dedicada a ISIS, estudiaremos uno de éstos, basado en el sistema de causalidad.

Grupos traslapados

Como hemos mencionado, un proceso puede ser miembro de varios grupos a la vez. Este hecho puede provocar un nuevo tipo de inconsistencia. Para ver el problema, observemos la figura 2-35, la cual muestra dos grupos, 1 y 2. Los procesos *A*, *B* y *C* son miembros del grupo 1 y los procesos *B*, *C* y *D* son miembros del grupo 2.

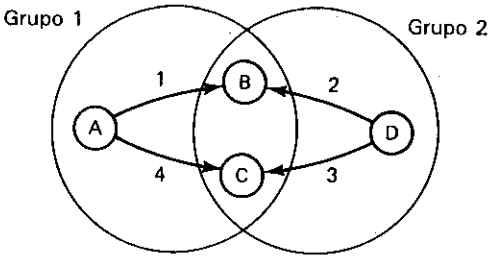


Figura 2-35. Cuatro procesos, *A, B, C* y *D* y cuatro mensajes. Los procesos *B* y *C* obtienen los mensajes de *A* y *D* en orden diferente.

Supongamos ahora que cada uno de los procesos *A* y *D* decide de manera simultánea enviar un mensaje a sus grupos respectivos y que el sistema utiliza el ordenamiento con respecto al tiempo global dentro de cada grupo. Se utiliza la unitransmisión como en nuestro ejemplo anterior. El orden de los mensajes se muestra en la figura 2-35 mediante los números 1 al 4. De nuevo tenemos la situación en que dos procesos, en este caso, *B* y *C*, reciben los mensajes en un orden distinto. *B* obtiene primero un mensaje de *A* y después uno de *D*. *C* los recibe en el orden contrario.

El problema aquí es que, aunque existe un ordenamiento con respecto al tiempo global dentro de cada grupo, no es necesario que exista coordinación entre varios grupos. Algunos sistemas soportan un ordenamiento bien definido entre los grupos traslapados y otros no. (Si los grupos son ajenos, este problema no aparece.) Con frecuencia, es difícil implantar el orden con respecto al tiempo entre los distintos grupos, por lo que surge la pregunta de si es importante.

Escalabilidad

Nuestro aspecto final del diseño es la escalabilidad. Muchos algoritmos funcionan bien mientras todos los grupos tengan unos cuantos miembros, pero ¿qué ocurre cuando existen decenas, centenas o incluso miles de miembros por grupo? ¿O miles de grupos? Además, ¿qué ocurre si el sistema es tan grande que no cabe en una LAN, de modo que se necesiten varias LAN y compuertas? ¿Qué ocurre si los grupos están diseminados en varios continentes?

La presencia de compuertas puede afectar muchas propiedades de la implantación. Para comenzar, la multitransmisión es más complicada. Consideremos, por ejemplo, la red que se muestra en la figura 2-36. Consta de cuatro LAN y cuatro compuertas, con el fin de protegerse contra la falla de cualquier compuerta.

Imaginemos que una de las máquinas de la LAN 2 ejecuta una multitransmisión. Cuando el paquete de multitransmisión llega a las compuertas *G1* y *G3*, ¿qué deben hacer éstas? Si

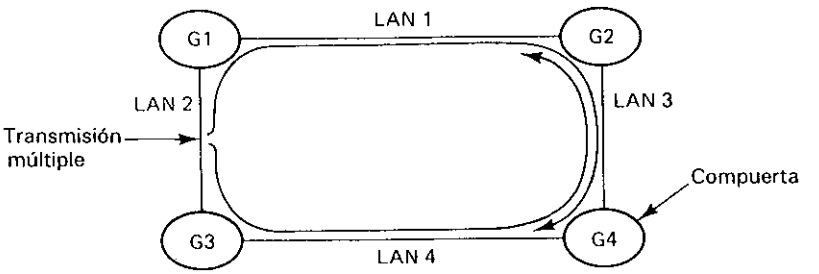


Figura 2-36. La transmisión múltiple en una red puede provocar problemas.

lo descartan, la mayoría de las máquinas nunca lo verán, lo cual destruye su valor como multitransmisión. Sin embargo, si el algoritmo sólo tiene compuertas hacia todas las multitransmisiones, entonces el paquete se copiará a la LAN 1 y la LAN 4 y poco después a la LAN 3 dos veces. Peor aún, la compuerta *G2* verá la multitransmisión de *G4*, la copiará a la LAN 2 y viceversa. Es claro que se necesita un algoritmo más complejo, que mantenga un registro de los paquetes anteriores, con el fin de evitar un crecimiento exponencial en el número de multitransmisiones de paquetes.

Otro problema con una red es que algunos métodos de la comunicación en grupo aprovechan el hecho de que un paquete puede estar en una LAN en un instante dado. En efecto, el orden de la transmisión de paquetes define un orden absoluto en el tiempo global que, como ya hemos visto, es con frecuencia crucial. Con las compuertas y las redes múltiples, es posible que dos paquetes estén "en la línea" en forma simultánea, lo que destruye esta útil propiedad.

Por último, ciertos algoritmos no se pueden escalar bien al uso de componentes centralizados u otros factores, debido a su complejidad computacional.

2.5.3. Comunicación en grupo en ISIS

Como ejemplo de comunicación en grupo, analizaremos el sistema ISIS desarrollado en Cornell (Birman, 1993; Birman y Joseph, 1987a, 1987b; y Birman y Van Renesse, 1994). ISIS es un conjunto de herramientas para la construcción de aplicaciones distribuidas; por ejemplo, para coordinar el intercambio de acciones entre corredores de bolsa de cierta empresa en Wall Street. ISIS no es un sistema operativo completo, sino más bien un conjunto de programas que se ejecutan sobre UNIX o algún otro sistema operativo ya existente. Es interesante estudiarlo, puesto que se ha descrito con amplitud en la bibliografía y se ha utilizado para numerosas aplicaciones reales. En el capítulo 7 estudiaremos la comunicación en grupo de Amoeba, que tiene un punto de vista un poco distinto.

La idea fundamental en ISIS es la **sincronía** y las primitivas fundamentales de comunicación son diversas formas de la transmisión atómica. Antes de analizar cómo lleva a cabo ISIS esta transmisión atómica, es necesario examinar en primer lugar las distintas

formas de sincronía que distingue. Un **sistema síncrono** es donde los eventos ocurren estrictamente en forma secuencial, donde cada evento (por ejemplo, una transmisión simple) tarda en esencia un tiempo nulo en llevarse a cabo. Por ejemplo, si el proceso *A* envía un mensaje a los procesos *B*, *C* y *D*, como se muestra en la figura 2-37(a), el mensaje llega en forma instantánea a todos los destinos. En forma análoga, un mensaje posterior de *D* a los demás también tarda un tiempo nulo en ser entregado. Desde el punto de vista de un observador externo, el sistema consta de eventos discretos, ninguno de los cuales se traslapa con los demás. Esta propiedad facilita la comprensión del comportamiento del sistema.

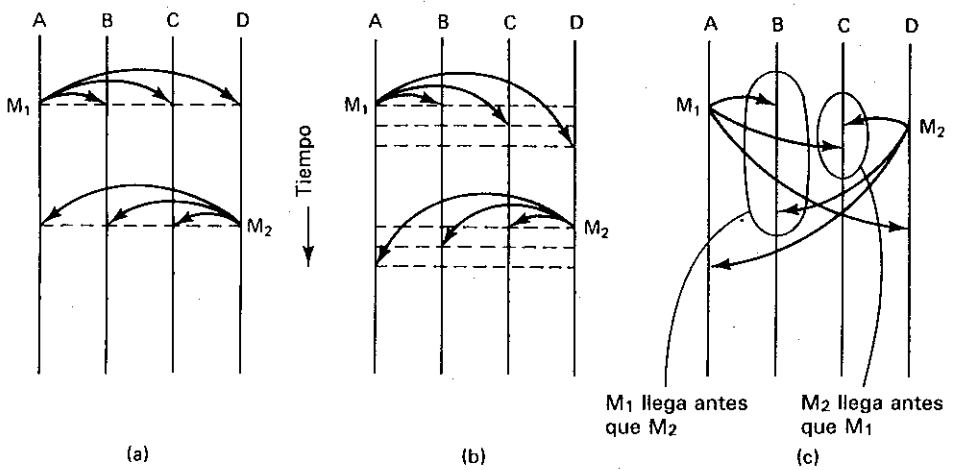


Figura 2-37. (a) Un sistema síncrono. (b) Una sincronía vaga. (c) Sincronía virtual.

Los sistemas síncronos son imposibles de construir, por lo que necesitamos investigar otro tipo de sistemas, con requisitos más débiles en cuanto al tiempo. Un **sistema vagamente síncrono** es como el que se muestra en la figura 2-37(b), donde los eventos tardan un tiempo finito, pero todos los eventos aparecen en el mismo orden para todas las partes. En particular, todos los procesos reciben todos los mensajes en el mismo orden. Anteriormente analizamos la misma idea en esencia, bajo el nombre de ordenamiento consistente según el tiempo.

Se pueden construir tales sistemas, pero para ciertas aplicaciones se puede utilizar una semántica todavía más débil y la esperanza es que se pueda capitalizar esta semántica débil para lograr un mejor desempeño. La figura 2-37(c) muestra un **sistema virtualmente síncrono**, donde las restricciones de orden se relajan, pero de forma que esto no importe bajo circunstancias elegidas con cuidado.

Analicemos estas circunstancias. En un sistema distribuido, se dice que dos eventos están **relacionados de manera causal** si el primero puede influir de alguna forma en la naturaleza del comportamiento del segundo. Así, si *A* envía un mensaje a *B*, el cual lo

examina y después envía un nuevo mensaje a *C*, el segundo mensaje se relaciona de manera causal con el primero, puesto que su contenido podría haberse obtenido del primero. El hecho de que esto ocurra en realidad no es importante. La relación es válida si *puede* existir tal influencia.

Dos eventos no relacionados son **concurrentes**. Si *A* envía un mensaje a *B* y, casi al mismo tiempo, *C* envía un mensaje a *D*, estos eventos son concurrentes, puesto que ninguno de ellos influye en el otro. Lo que en realidad significa la sincronía virtual es que si dos mensajes están relacionados de manera causal, todos los procesos *deben* recibirlos en el mismo orden (el correcto). Sin embargo, si son concurrentes, no hay garantías y el sistema es libre de entregarlos con un orden distinto a procesos diferentes, si esto es más sencillo. Así, cuando es importante, los mensajes se entregan siempre en el mismo orden, pero cuando esto no importa, pueden o no entregarse en el mismo orden.

Primitivas de comunicación en ISIS

Pasemos ahora a las primitivas de comunicación simple que se utilizan en ISIS. Se han definido tres de ellas: ABCAST, CBCAST y GBCAST, cada una con distinta semántica. ABCAST proporciona la comunicación vagamente síncrona y se utiliza para la transmisión de datos a los miembros de un grupo. CBCAST proporciona la comunicación virtualmente síncrona y también se utiliza para el envío de datos. GBCAST se parece a ABCAST, excepto que se usa para el manejo de la membresía en vez del envío de datos ordinarios.

En un principio, ABCAST utilizó una forma de protocolo de compromiso de dos fases, que funcionaba de la manera siguiente. El emisor *A* asignaba una marca de tiempo (en realidad, solo un número secuencial) al mensaje y lo enviaba a todos los miembros del grupo (al nombrarlos de manera explícita). Cada uno de ellos elegía su propia marca de tiempo, mayor que cualquier otra marca que hubiese enviado o recibido y enviaba ésta de regreso a *A*. Al recibir todas estas marcas, *A* elegía la mayor y enviaba un mensaje *Commit* (compromiso) a todos los miembros que la contuvieran de nuevo. Los mensajes comprometidos se entregaban a los programas de aplicación según el orden de las marcas de tiempo. Se puede mostrar que este protocolo garantiza la entrega de todos los mensajes a todos los procesos en el mismo orden.

También se puede mostrar que este protocolo es complejo y caro. Por esta razón, los diseñadores de ISIS inventaron la primitiva CBCAST, que sólo garantiza la entrega ordenada de los mensajes relacionados de manera causal. (El protocolo ABCAST recién descrito ha sido sustituido por otro, pero incluso éste es mucho más lento que CBCAST.) El protocolo CBCAST funciona de la manera siguiente. Si un grupo tiene *n* miembros, cada proceso mantiene un vector con *n* componentes, uno por cada miembro del grupo. El *i*-ésimo componente de este vector es el número del último mensaje recibido desde el proceso *i*. Los vectores son controlados por el sistema de tiempo de ejecución, no por los propios procesos del usuario y se inician en 0, como se muestra en la parte superior de la figura 2-38.

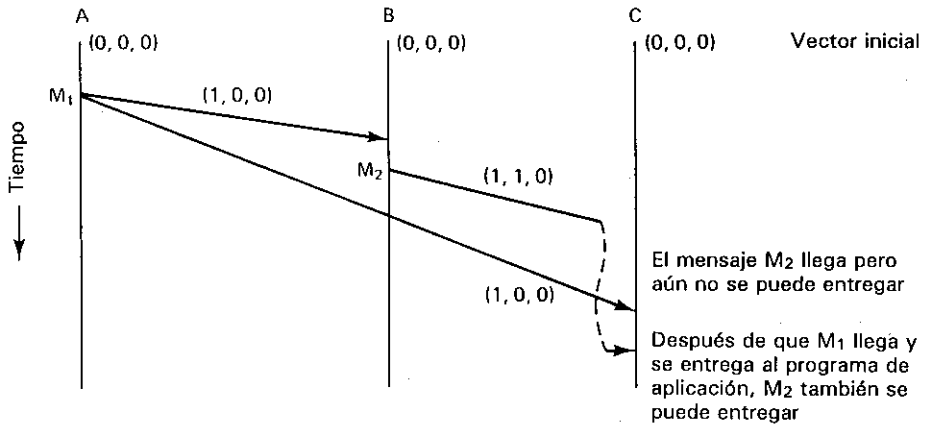


Figura 2-38. Los mensajes sólo se pueden entregar si todos los mensajes anteriores desde el punto de vista causal ya se han entregado.

Cuando un proceso tiene que enviar un mensaje, incrementa su espacio en el vector y envía este vector como parte del mensaje. Cuando M_1 de la figura 2-38 llega a B, se verifica si depende de algo no visto por B. El primer componente del vector es una unidad mayor que el primer componente de B, que espera (y necesita) un mensaje de A y los demás son iguales, por lo que el mensaje se acepta y trasfiere al miembro del grupo que se ejecuta en B. Si algún otro componente del vector recibido ha sido mayor que el componente correspondiente del vector B, el mensaje no podría entregarse aún.

Ahora, B envía su propio mensaje M_2 a C, el cual llega antes de M_1. Mediante el análisis del vector, C ve que B ya ha recibido un mensaje de A antes de enviar M_2; puesto que no ha recibido nada de A, M_2 se guarda hasta que llegue un mensaje de A. La entrega de M_2 no debe ocurrir antes del mensaje de A.

Podemos enunciar ahora el algoritmo general para decidir si se transfiere un mensaje recibido al proceso usuario o bien, si se retrasa. Sea V_i el i-ésimo componente del vector en el mensaje recibido y L_i el i-ésimo componente del vector almacenado en la memoria del receptor. Supongamos que j envía un mensaje. La primera condición para aceptar un mensaje es $V_j = L_j + 1$. Esto quiere decir que éste es el siguiente mensaje en orden secuencial de j; es decir, que no se han perdido mensajes. (Los mensajes del mismo emisor siempre se relacionan de manera causal.) La segunda condición para aceptarlo es $V_i \leq L_i$ para toda $i \neq j$. Esta condición indica que el emisor no ha visto un mensaje perdido por el receptor. Si el mensaje recibido aprueba estos criterios, el sistema de tiempo de ejecución puede transferirlo al proceso usuario sin retraso. En caso contrario, debe esperar.

En la figura 2-39 mostramos un ejemplo más detallado del mecanismo vectorial. En este caso, el proceso 0 envía un mensaje con el vector (4, 6, 8, 2, 1, 5) a los otros cinco miembros de su grupo. El proceso 1 ha visto con exactitud los mismos mensajes que el

proceso 0, excepto por el mensaje 7, enviado por el propio proceso 1, por lo que el mensaje recibido pasa la prueba, es aceptado y se puede transferir a los procesos usuario. El proceso 2 pierde el mensaje 6 enviado por el proceso 1, por lo que el mensaje debe retrasarse. El proceso 3 ha visto lo mismo que el emisor, además de un mensaje 7 del proceso 1, que en apariencia no ha llegado al proceso 0, por lo que el mensaje es aceptado. El proceso 4 pierde el anterior mensaje de 0. Esta omisión es seria, por lo que el nuevo mensaje deberá esperar. Por último, también el proceso 5 está un poco adelantado con respecto al proceso 0, por lo que el mensaje puede ser aceptado de manera inmediata.

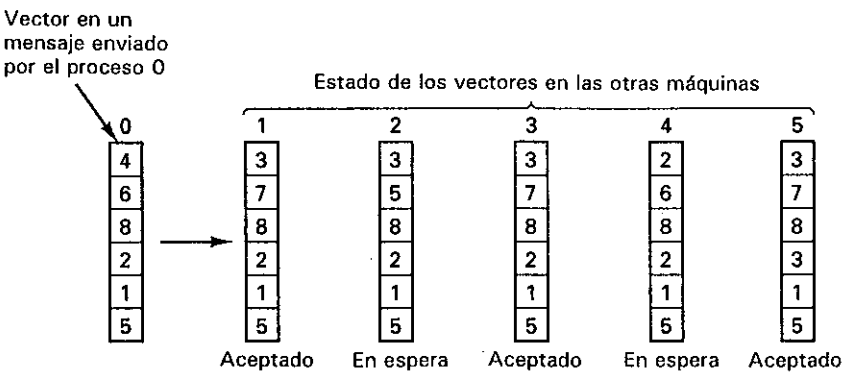


Figura 2-39. Ejemplo de los vectores utilizados por CBCAST.

ISIS también proporciona la tolerancia de fallas y soporta el ordenamiento de mensajes para grupos traslapados mediante CBCAST. Los algoritmos que se utilizan aquí son un poco complejos. Para los detalles, véase (Birman *et al.*, 1991).

2.6. RESUMEN

La diferencia fundamental entre un sistema operativo centralizado y uno distribuido es la importancia de la comunicación en el segundo caso. Se han propuesto e implantado varios métodos para la comunicación en los sistemas distribuidos. Para los sistemas distribuidos de área amplia relativamente lentos, se utilizan los protocolos con capas orientadas hacia la conexión, como OSI y TCP/IP, puesto que el problema principal por resolver es el transporte confiable de los bits a través de líneas físicas pobres.

Para los sistemas distribuidos basados en LAN, los protocolos con capas se utilizan muy poco. En vez de ellos, se adopta por lo general un modelo mucho más sencillo, donde el cliente envía un mensaje al servidor y éste envía de regreso una respuesta al cliente. Se puede lograr un desempeño mucho mejor al eliminar la mayor parte de las capas. Muchos de los aspectos del diseño en estos sistemas de transferencia de mensajes se refieren a las primitivas de comunicación: bloqueo *vs.* no bloqueo, almacenamiento en buffers *vs.* no almacenamiento, confiable *vs.* no confiable, etcétera.

El problema con el modelo básico cliente-servidor es que, desde el punto de vista conceptual, la comunicación entre procesos se maneja como E/S. Para presentar mejor abstracción, se utiliza mucho la llamada a procedimientos remotos (RPC). Con RPC, un proceso cliente que se ejecuta en una máquina llama a un procedimiento que se ejecuta en otra. El sistema de tiempo de ejecución, inmerso en los procedimientos de resguardo, maneja la recolección de parámetros, la construcción de mensajes y la interfaz con el núcleo para el desplazamiento real de los bits.

Aunque RPC está un paso adelante de la simple transferencia de mensajes, tiene sus propios problemas. Hay que localizar al servidor correcto. Es difícil la transferencia de los apuntadores y estructuras de datos complejos. Es difícil utilizar las variables globales. La semántica precisa de RPC es un tanto truculenta, puesto que los clientes y los servidores pueden fallar en forma independiente entre sí. Por último, la implantación eficiente de RPC no es directa y requiere de una reflexión cuidadosa.

RPC se limita a aquellas situaciones en las que un cliente desea comunicarse con un servidor. Cuando una colección de procesos, por ejemplo, servidores duplicados de archivo, tiene que comunicarse con otra como grupo, se necesita algo más. Los sistemas como ISIS proporcionan una nueva abstracción con este fin: la comunicación en grupo. ISIS ofrece una variedad de primitivas, de las cuales la más importante es CBCAST. CBCAST ofrece una semántica de comunicación débil con base en la causalidad y se implanta mediante la inclusión de vectores de números secuenciales en cada mensaje, para permitir al receptor que revise si el mensaje se debe entregar de manera inmediata o retrasarse hasta que lleguen algunos mensajes anteriores.

PROBLEMAS

1. En muchos protocolos con capas, cada una de éstas tiene su propio encabezado. Con seguridad, sería más eficiente tener un encabezado al frente de cada mensaje con todo el control, en vez de todos estos encabezados ajenos. ¿Por qué no se hace esto?
2. ¿Qué se entiende por un *sistema abierto*? ¿Por qué algunos sistemas no son abiertos?
3. ¿Cuál es la diferencia entre un protocolo de comunicación orientado a la conexión y otro sin conexiones?
4. Un sistema ATM transmite las celdas con la tasa OC-3. Cada paquete tiene una longitud de 48 bytes, por lo que cabe dentro de una celda. Una interrupción tarda 1 μseg. ¿Qué proporción del CPU está dedicado al manejo de la interrupción? Repita ahora el problema para los paquetes de 1024 bytes.
5. ¿Cuál es la probabilidad de que un encabezado ATM por completo desordenado sea aceptado como correcto?
6. Sugiera una sencilla modificación de la figura 2-9 que reduzca el tráfico en la red.
7. Si las primitivas de comunicación en un sistema cliente-servidor no utilizan bloqueo, una llamada a *send* terminará antes de que el mensaje se envíe en realidad. Para reducir

3

Sincronización en sistemas distribuidos

En el capítulo anterior vimos la forma en que los procesos se comunican entre sí en un sistema distribuido. Los métodos que analizamos fueron los protocolos con capas, transferencia de mensajes solicitud/respuesta (RPC incluida) y comunicación en grupo. Aunque la comunicación es importante, no es todo lo que hay que considerar. Íntimamente relacionada con esto está la forma en que los procesos cooperan y se sincronizan entre sí. Por ejemplo, la forma de implantar las regiones críticas o asignar recursos en un sistema distribuido. En este capítulo estudiaremos éstos y otros aspectos de la cooperación y sincronización entre procesos en los sistemas distribuidos.

En los sistemas con un CPU, los problemas relativos a las regiones críticas, la exclusión mutua y la sincronización se resuelven en general mediante métodos tales como los semáforos y los monitores. Estos métodos no son adecuados para su uso en los sistemas distribuidos, puesto que siempre se basan (de manera implícita) en la existencia de la memoria compartida. Por ejemplo, dos procesos que interactúan mediante un semáforo deben tener acceso a éste. Si se ejecutan en la misma máquina, pueden compartir el semáforo al guardarlo en el núcleo y realizar llamadas al sistema para tener acceso a él. Sin embargo, si se ejecutan en máquinas distintas, este método ya no funciona, por lo que se necesitan otras técnicas. Incluso las cuestiones más sencillas, como el hecho de determinar si el evento *A* ocurrió antes o después del evento *B* requiere una reflexión cuidadosa.

Comenzaremos con el tiempo y la forma de medirlo, puesto que éste juega un papel fundamental en algunos modelos de sincronización. Después analizaremos los algoritmos de exclusión mutua y de elección. También estudiaremos una técnica de sincronización de alto nivel, las transacciones atómicas. Por último, analizaremos los bloqueos, esta vez para el caso de los sistemas distribuidos.

3.1. SINCRONIZACIÓN DE RELOJES

La sincronización es más compleja en los sistemas distribuidos que en los centralizados, puesto que los primeros deben utilizar algoritmos distribuidos. Por lo general, no es posible (o recomendable) reunir toda la información relativa al sistema en un lugar y después dejar que cierto proceso la examine y tome una decisión, como se hace en el caso centralizado. En general, los algoritmos distribuidos tienen las siguientes propiedades:

1. La información relevante se distribuye entre varias máquinas.
2. Los procesos toman las decisiones sólo con base en la información disponible en forma local.
3. Debe evitarse un punto de fallo en el sistema.
4. No existe un reloj común o alguna otra fuente precisa del tiempo global.

Los primeros tres puntos indican que es inaceptable reunir toda la información en un lugar para su procesamiento. Por ejemplo, para llevar a cabo la asignación de recursos (asignar los dispositivos de E/S en una forma libre de bloqueos), por lo general no es aceptable enviar todas las solicitudes a un administrador de procesos, el cual examina a todos y otorga o niega las solicitudes con base en la información de sus tablas. En un sistema de gran tamaño, tal solución pone en predicamentos a dicho proceso.

Además, el hecho de que sólo exista un punto de fallo como éste hace que el sistema no sea confiable. Lo ideal es que un sistema distribuido debería ser más confiable que las máquinas individuales. Si alguna de ellas falla, el resto puede continuar su funcionamiento. Lo último que quisiéramos es una situación donde una máquina falle (por ejemplo, el asignador de recursos) y con ello detenga a un gran número de máquinas diversas. Para lograr la sincronización sin centralización necesitamos algo distinto al caso de los sistemas operativos tradicionales.

El último punto de la lista también es crucial. En un sistema centralizado, el tiempo no tiene ambigüedades. Cuando un proceso desea conocer la hora, llama al sistema y el núcleo se la dice. Si el proceso *A* pide la hora y un poco después, el proceso *B* también la pide, el valor obtenido por *B* es mayor o igual que el valor obtenido por *A*. Ciertamente, no debe ser menor. En un sistema distribuido, no es trivial poner de acuerdo a todas las máquinas en la hora.

Por ejemplo, pensemos por un momento las implicaciones de la carencia de un tiempo global en el programa *make* de UNIX. En UNIX, los programas de gran tamaño se dividen por lo general en varios archivos fuentes, de modo que un cambio al archivo fuente sólo necesite una nueva compilación de un archivo y no de todos. Si un programa consta de 100 archivos y no hay que volverlo a compilar por la modificación de un archivo, la velocidad de trabajo de los programadores aumenta en gran medida.

La forma de funcionamiento de *make* es muy sencilla. Cuando el programador termina de modificar todos los archivos fuentes, inicia *make*, el cual examina las horas en que todos los archivos fuentes y objetos fueron modificados por última vez. Si el archivo fuente *input.c* tiene la hora 2 151 y el correspondiente archivo objeto *input.o* tiene la hora 2 150, *make* sabe que *input.c* tiene modificaciones desde la creación de *input.o*, por lo que entonces hay que volver a compilar *input.c*. Por otro lado, si *output.c* tiene la hora 2 144 y *output.o* tiene la hora 2 145, entonces no es necesario volver a compilar. Así, *make* revisa todos los archivos fuentes para determinar aquellos que deban volverse a compilar y llama al compilador para que realice esta tarea.

Imaginemos ahora lo que podría ocurrir en un sistema distribuido donde no existe un acuerdo global en el tiempo. Supongamos que *output.o* tiene de nuevo la hora 2144 y que poco después se modifica *output.c*, pero se le asigna la hora 2143 debido a que el reloj de esta máquina es un poco lento, como se muestra en la figura 3-1. *Make* no llamará al compilador. El programa en binario ejecutable resultante contendrá una mezcla de archivos objetos de las fuentes anteriores y nuevas. Tal vez no funcione y el programador se vuelva loco intentando descubrir la parte incorrecta del código.

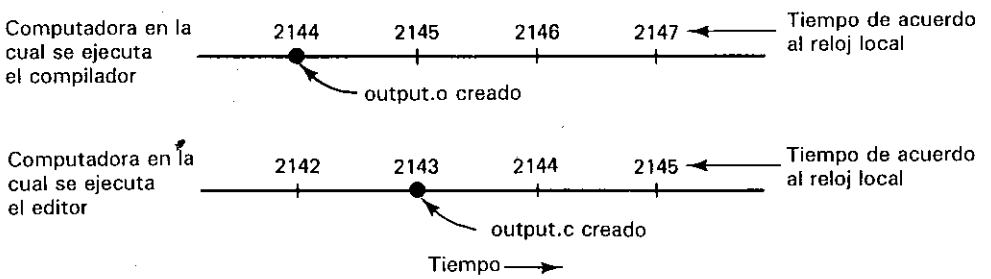


Figura 3-1. Cuando cada máquina tiene su propio reloj, podría asignarse un tiempo anterior a un evento ocurrido después de otro.

Puesto que el tiempo es fundamental para la forma de pensar de la gente y el efecto de carecer de sincronización en los relojes puede ser muy drástico, como hemos visto, podemos iniciar nuestro estudio de la sincronización con la siguiente sencilla pregunta: "¿Es posible sincronizar todos los relojes en un sistema distribuido?"

3.1.1. Relojes lógicos

Casi todas las computadoras tienen un circuito para el registro del tiempo. A pesar del uso generalizado de la palabra "reloj" para hacer referencia a dichos dispositivos, en realidad no son relojes en el sentido usual. **Cronómetro** sería una mejor palabra. Un cronómetro de computadora es por lo general un cristal de cuarzo trabajado con precisión. Cuando se mantiene sujeto a tensión, un cristal de cuarzo oscila con frecuencia bien definida, que depende del tipo de cristal, la forma en que se corte y la magnitud de la tensión. A cada

cristal se le asocian dos registros, un **contador** y un **registro mantenedor**. Cada oscilación del cristal disminuye en 1 al contador. Cuando el contador toma el valor 0, se genera una interrupción y el contador se vuelve a cargar mediante el registro mantenedor. De esta forma, es posible programar un cronómetro de modo que genere una interrupción 60 veces por cada segundo o con cualquier frecuencia que se desee. Cada interrupción recibe el nombre de **marca de reloj**.

Cuando se arranca por vez primera el sistema, por lo general se pide al operador que escriba la fecha y la hora, las cuales se convierten al número de marcas después de cierta fecha conocida y se guardan en la memoria. En cada marca de reloj, el procedimiento de servicio de interrupciones añade 1 al tiempo guardado en memoria. De esta forma, el reloj (de software) se mantiene actualizado.

En el caso de una computadora y un reloj, no importa si éste se desfasa un poco. Puesto que todos los procesos de la máquina utilizan el mismo reloj, tendrán consistencia interna. Por ejemplo, si el archivo *input.c* tiene la hora 2 151 y el archivo *input.o* tiene la hora 2 150, *make* vuelve a compilar el archivo fuente, aunque el reloj se desfase en 2 y los tiempos reales sean 2 153 y 2 152, respectivamente. Todo lo que importa son los tiempos relativos.

Tan pronto se comienza a trabajar con varias máquinas, cada una con su propio reloj, la situación es distinta. Aunque la frecuencia de un oscilador de cristal es muy estable, es imposible garantizar que los cristales de computadoras distintas oscilen precisamente con la misma frecuencia. En la práctica, cuando un sistema tiene *n* computadoras, los *n* cristales correspondientes oscilarán a tasas un poco distintas, lo que provoca una pérdida de sincronía en los relojes (de software) y que al leerlos tengan valores distintos. La diferencia entre los valores del tiempo se llama **distorsión del reloj**. Como consecuencia de esta distorsión, podrían fallar los programas que esperan que el tiempo asociado a un archivo, objeto, proceso o mensaje sea correcto e independiente del sitio donde haya sido generado (es decir, del reloj utilizado), como hemos visto en el ejemplo anterior de *make*.

Esto nos regresa a nuestra pregunta original, saber si es posible sincronizar todos los relojes para obtener un estándar de tiempo único, sin ambigüedades. En un artículo clásico, Lamport (1978) demostró que la sincronización de relojes es posible y presentó un algoritmo para lograr esto. Amplió su trabajo en (Lamport, 1990).

Lamport señaló que la sincronización de relojes no tiene que ser absoluta. Si dos procesos no interactúan, no es necesario que sus relojes estén sincronizados, puesto que la carencia de sincronización no sería observable y por tanto no podría provocar problemas. Además, señaló que lo que importa por lo general, no es que todos los procesos concuerden de manera exacta en la hora, sino que coincidan en el orden en que ocurren los eventos. En el ejemplo anterior de *make*, lo que importa es si *input.c* es anterior o posterior a *input.o* y no la hora exacta en que fueron creados.

Para la mayoría de los fines, basta que todas las máquinas coincidan en la misma hora. No es esencial que esta hora también coincida con la hora real, como se anuncia en la radio cada hora. Por ejemplo, para ejecutar *make,* es adecuado que todas las máquinas estén de acuerdo en que sean las 10:00, aunque en realidad sean las 10:02. Así, para una cierta clase

de algoritmos, lo que importa es la consistencia interna de los relojes, no su particular cercanía al tiempo real. Para estos algoritmos, conviene hablar de los relojes como **relojes lógicos**.

Cuando existe la restricción adicional de que los relojes no sólo deben ser iguales, sino que además no se desvíen del tiempo real más allá de cierta magnitud, los relojes reciben el nombre de **relojes físicos**. En esta sección analizaremos el algoritmo de Lamport, el cual sincroniza los relojes lógicos. En las siguientes secciones presentaremos el concepto de tiempo físico y mostraremos la forma en que se pueden sincronizar los relojes físicos.

Para sincronizar los relojes lógicos, Lamport definió una relación llamada **ocurre antes de**. La expresión $a \rightarrow b$ se lee: "a ocurre antes de b" e indica que todos los procesos coinciden en que primero ocurre el evento a y después el evento b. La relación "ocurre antes de" se puede observar de manera directa en dos situaciones:

1. Si a y b son eventos en el mismo proceso y a ocurre antes de b, entonces $a \rightarrow b$ es verdadero.

2. Si a es el evento del envío de un mensaje por un proceso y b es el evento de la recepción del mensaje por otro proceso, entonces $a \rightarrow b$ también es verdadero. Un mensaje no se puede recibir antes de ser enviado o al mismo tiempo en que se envía, puesto que tarda en llegar una cantidad finita de tiempo.

"Ocurre antes de" es una relación transitiva, de modo que si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$. Si dos eventos, x y y, están en procesos diferentes que no intercambian mensajes (ni siquiera en forma indirecta por medio de un tercero), entonces $x \rightarrow y$ no es verdadero, ni tampoco lo es $y \rightarrow x$. Se dice que estos eventos son **concurrentes**, lo que significa que nada se puede decir (o se necesita decir) acerca del momento en el que ocurren o cuál de ellos es el primero.

Lo que necesitamos es una forma de medir el tiempo tal que, a cada evento a, le podamos asociar un valor del tiempo $C(a)$ en el que todos los procesos estén de acuerdo. Estos valores del tiempo deben tener la propiedad de que si $a \rightarrow b$, entonces $C(a) < C(b)$. En otros términos, si a y b son dos procesos dentro del mismo evento y a ocurre antes de b, entonces $C(a) < C(b)$. De manera análoga, si a es el envío de un mensaje por un proceso y b la recepción de ese mensaje por otro proceso, entonces $C(a)$ y $C(b)$ deben ser asignados de tal forma que todos estén de acuerdo en los valores de $C(a)$ y $C(b)$ con $C(a) < C(b)$. Además, el tiempo del reloj, C, siempre debe ir hacia adelante (creciente), y nunca hacia atrás (decreciente). Se pueden hacer correcciones al tiempo al sumar un valor positivo al reloj, pero nunca se le debe restar un valor positivo.

Analizaremos ahora el algoritmo propuesto por Lamport para la asignación de tiempos a eventos. Consideremos los tres procesos que se muestran en la figura 3-2(a). Los procesos se ejecutan en diferentes máquinas, cada una con su propio reloj y velocidad. Como se puede ver en la figura, cuando el reloj marca 6 en el proceso 0, ha marcado 8 en el proceso 1 y 10 en el proceso 2. Cada reloj corre a una razón constante, sólo que las razones son diferentes debido a las diferencias en los cristales.

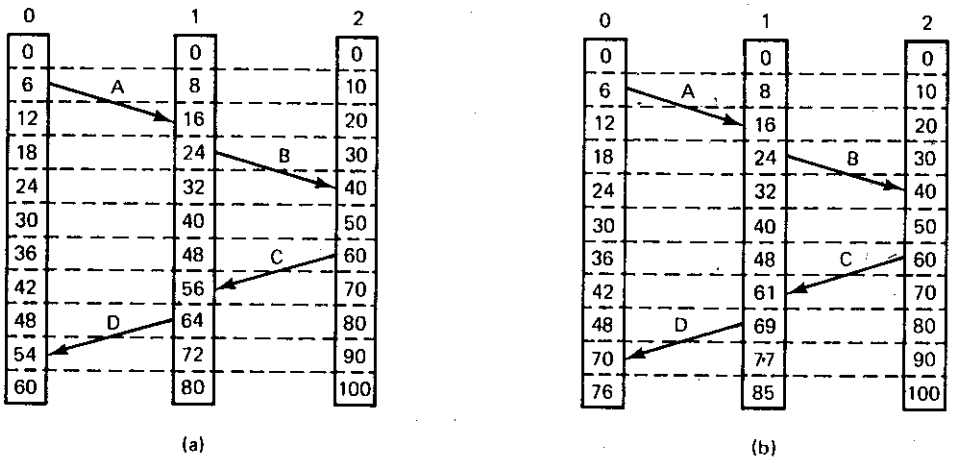


Figura 3-2. (a) Tres procesos, cada uno con su propio reloj. Los relojes corren a diferentes velocidades. (b) El algoritmo de Lamport corrige los relojes.

Al tiempo 6, el proceso 0 envía el mensaje *A* al proceso 1. El tiempo que tarda este mensaje en llegar depende del reloj elegido. En cualquier caso, el reloj del proceso 1 lee 16 cuando el mensaje llega. Si el mensaje transporta el tiempo de inicio 6, entonces el proceso 1 concluirá que tardó 10 marcas de reloj en hacer el viaje. Este valor es posible. De acuerdo con este razonamiento, el mensaje *B* de 1 a 2 ocupa 16 marcas, que de nuevo es un valor plausible.

Ahora viene lo divertido. El mensaje *C*, de 2 a 1, sale en 60 y llega en 56. En forma análoga el mensaje *D*, de 1 a 0, sale en 64 y llega en 54. Es claro que estos valores son imposibles. Esta situación es la que hay que evitar.

La solución de Lamport es consecuencia directa de la relación "ocurre antes de". Puesto que *C* sale en 60, debe llegar en 61 o en un tiempo posterior. Por lo tanto, cada mensaje trae consigo el tiempo de envío, de acuerdo con el reloj del emisor. Cuando un mensaje llega y el reloj del receptor muestra un valor anterior al tiempo en que se envió el mensaje, rápidamente el receptor adelanta su reloj para que tenga una unidad más que el tiempo de envío. En la figura 3-2(b), vemos que *C* llega ahora a 61. En forma análoga, *D* llega en 70.

Con una pequeña adición, este algoritmo cumple nuestras necesidades para el tiempo global. La adición es que entre cualesquiera dos eventos, el reloj debe marcar al menos una vez. Si un proceso envía o recibe dos mensajes en serie muy rápida, avanza su reloj en (al menos) una marca entre ellos.

En ciertas situaciones, existe un requisito adicional: dos eventos no ocurren exactamente al mismo tiempo. Para lograr esto, podemos asociar el número del proceso en que ocurre el evento y el extremo inferior del tiempo, separados por un punto decimal. Así, si ocurren

eventos en los procesos 1 y 2, ambos en el tiempo 40, entonces el primero se convierte en 40.1 y el segundo en 40.2.

Por medio de este método, tenemos ahora una forma de asignar un tiempo a todos los eventos en un sistema distribuido, con las siguientes condiciones:

1. Si a ocurre antes de b en el mismo proceso, $C(a) < C(b)$.

2. Si a y b son el envío y la recepción de un mensaje, $C(a) < C(b)$.

3. Para todos los eventos a y b, $C(a) \neq C(b)$.

Este algoritmo es una forma de obtener un orden total de todos los eventos en el sistema. Muchos otros algoritmos distribuidos necesitan tal orden para evitar las ambigüedades, de modo que el algoritmo es citado de forma amplia en la bibliografía existente.

3.1.2. Relojes físicos

Aunque el algoritmo de Lamport proporciona un orden de eventos sin ambigüedades, los valores de tiempo asignados a los eventos no tienen que ser cercanos a los tiempos reales en los que ocurren. En ciertos sistemas (por ejemplo, los sistemas de tiempo real), es importante la hora real del reloj. Para estos sistemas se necesitan relojes físicos externos. Por razones de eficiencia y redundancia, por lo general son recomendables varios relojes físicos, lo cual implica dos problemas: (1) ¿Cómo los sincronizamos con los relojes del mundo real? y (2) ¿Cómo sincronizamos los relojes entre sí?

Antes de responder estas preguntas, vamos a detenernos un poco para ver la forma en que se mide en realidad el tiempo. No es tan sencillo como uno podría pensar, en particular CUANDO se requiere alta precisión. Desde la invención de los relojes mecánicos en el siglo XVII, el tiempo se ha medido de manera astronómica. Cada día, el sol parece levantarse en el horizonte del este, sube hasta una altura máxima en el cielo y desciende en el oeste. El evento en que el sol alcanza su punto aparentemente más alto en el cielo se llama **tránsito del sol**. Este evento ocurre aproximadamente a las doce del día de cada día. El intervalo entre dos tránsitos consecutivos del sol se llama el **día solar**. Puesto que existen 24 horas en un día, cada una de las cuales contiene 3 600 segundos, el **segundo solar** se define exactamente como 1/86 400 de un día solar. La geometría del cálculo del día medio solar se muestra en la figura 3-3.

En la década de 1940, se estableció que el período de rotación de la tierra no es constante. La tierra está disminuyendo su velocidad, debido a la fricción tidal y el arrastre atmosférico. Con base en estudios de los patrones de crecimiento del coral antiguo, los geólogos piensan ahora que hace 300 millones de años existían cerca de 400 días al año. La duración del año, es decir, el tiempo que tarda la tierra en dar una vuelta alrededor del sol, no ha cambiado; sólo ocurre que los días se hacen más largos. Además de esta tendencia a largo plazo, también ocurren pequeñas variaciones a corto plazo en la duración del día, lo cual probablemente se deba a una turbulencia originada en el centro de acero fundido de la Tierra.

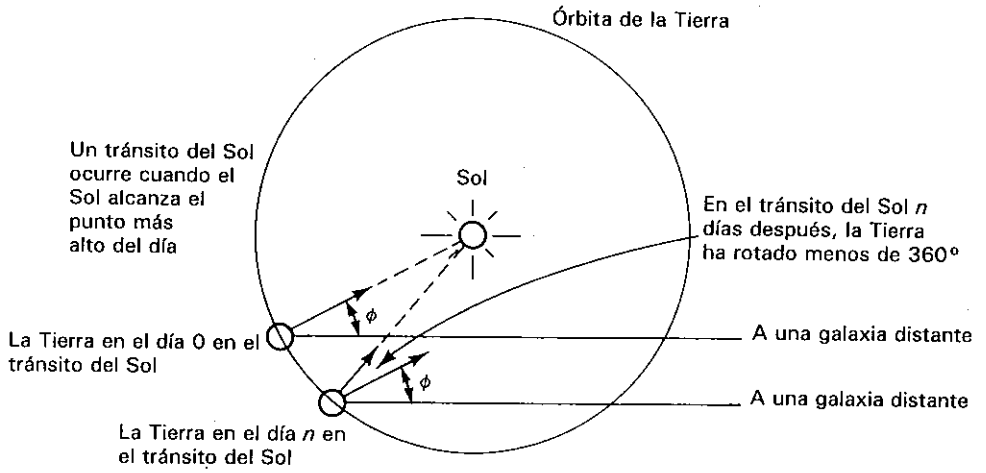


Figura 3-3. Cálculo de un día solar promedio.

Estas revelaciones han llevado a los astrónomos a calcular la longitud del día mediante la medición de un gran número de días y tomar su promedio antes de dividir entre 86 400. La cantidad resultante se llama el **segundo solar promedio**.

Con la invención del reloj atómico en 1948, fue posible medir el tiempo de manera mucho más exacta y en forma independiente de todo el ir y venir de la Tierra, al contar las transiciones del átomo de cesio 133. Los físicos retomaron de los astrónomos la tarea de medir el tiempo y definieron el segundo como el tiempo que tarda el átomo de cesio 133 en hacer exactamente 9 192 631 770 transiciones. La elección de 9 192 631 770 fue hecha para que el segundo atómico fuera igual al segundo solar promedio en el año de su introducción. Actualmente, cerca de 50 laboratorios en el mundo tienen relojes de cesio 133. En forma periódica, cada laboratorio le indica a la oficina internacional de la hora en París (BIH) el número de marcas de su reloj. La oficina hace un promedio de estos números para producir el **tiempo atómico internacional**, que se abrevia **TAI**. Así, TAI es el promedio de las marcas de los relojes de cesio 133 a partir de la medianoche del primero de enero de 1958 (principio del tiempo), dividido entre 9 192 631 770.

Aunque TAI es muy estable y disponible para todos los que quieran comprarse un reloj de cesio, existe un serio problema con él; 86 400 segundos TAI son ahora cerca de 3 milisegundos menos que un día solar medio (puesto que este día solar promedio es cada vez más grande). El uso de TAI para el registro del tiempo significaría que, con el paso de los años, el mediodía sería cada vez más temprano, hasta llegar al momento en que el mediodía ocurriría en la mañana. Las personas notarían esto y podrían estar en el mismo tipo de situación ocurrida en 1582, cuando el Papa Gregorio XIII decretó que se deberían omitir del calendario diez días. Este suceso provocó revueltas en las calles, puesto que los

terratenientes demandaron una renta de todo un mes y los banqueros un interés de todo el mes, mientras que los patrones se rehusaron a pagar a los trabajadores por los diez días que no trabajaron, por mencionar unos cuantos de los conflictos. Los países protestantes, por cuestiones de principio, rechazaron todo lo que tenía que ver con los decretos papales y no aceptaron el calendario Gregoriano durante 170 años.

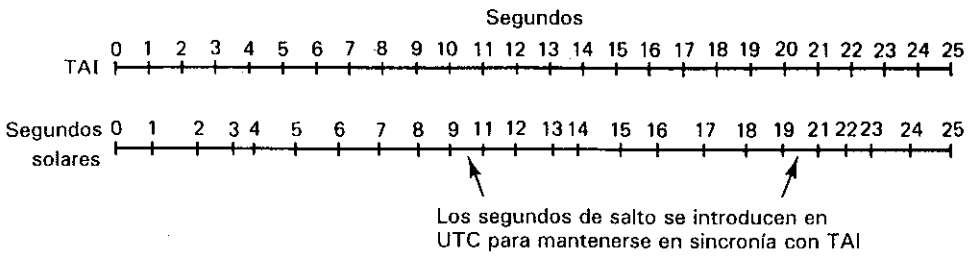


Figura 3-4. Los segundos TAI son de longitud constante, a diferencia de los segundos solares. Los segundos de salto se introducen cuando es necesario mantenerse en fase con el Sol.

La BIH resolvió el problema mediante la introducción de **segundos de salto**, siempre que la discrepancia entre TAI y el tiempo solar creciera hasta 800 milisegundos. El uso de los segundos de salto se ilustra en la figura 3-4. Esta corrección da lugar a un sistema de tiempo basado en los segundos constantes TAI, pero que permanece en fase con el movimiento aparente del Sol. Se le llama **tiempo coordenado universal, UTC**. UTC es la base de todo el sistema de mantenimiento moderno de la hora. En lo esencial, ha remplazado al estándar anterior, el tiempo del meridiano de Greenwich, que es un tiempo astronómico.

La mayoría de las compañías que proporcionan la luz eléctrica basan sus relojes de 60 Hz o 50 Hz en el UTC, por lo que cuando BIH anuncia un segundo de salto, las compañías elevan su frecuencia a 61 Hz o 51 Hz durante 60 o 50 segundos, para que avancen todos sus relojes del área de distribución. Puesto que un segundo es un intervalo notable para una computadora, un sistema operativo que necesite mantener un tiempo exacto durante un periodo de algunos años debe tener un software especial para registrar los segundos de salto conforme sean anunciados (a menos que utilicen la línea de corriente eléctrica para medir su tiempo, lo cual es demasiado burdo). El número total de segundos de salto introducidos en UTC hasta ahora es cercano a 30.

Para proporcionar UTC a las personas que necesitan un tiempo preciso, el Instituto Nacional del Tiempo Estándar (NIST) opera una estación de radio de onda corta con las siglas WWV desde Fort Collins, Colorado. WWV transmite un pulso corto al inicio de cada segundo UTC. La precisión del propio WWV es de cerca de ±1 milisegundo, pero debido a la fluctuación atmosférica aleatoria que puede afectar la longitud de la trayectoria de la señal, en la práctica la precisión no es mejor que ±10 milisegundos. En Inglaterra, la

estación MSF que opera desde Rugby, condado de Warwick, proporciona un servicio similar, al igual que otras estaciones en varios países.

Varios satélites terrestres también ofrecen un servicio UTC. El Satélite de Ambiente Operacional Geoestacionario (GEOS) puede proporcionar UTC con una precisión de hasta 0.5 milisegundos y algunos otros satélites lo hacen incluso mejor.

El uso del radio de onda corta o los servicios de satélite requiere de un conocimiento preciso de la posición relativa del emisor y el receptor, para compensar el retraso de la propagación de la señal. Se dispone en forma comercial de radioreceptores de WWV, GEOS y las otras fuentes UTC. El costo varía desde unos cuantos cientos de dólares hasta decenas de cientos de dólares cada uno, donde se paga más por las mejores fuentes. UTC también se puede obtener de manera más barata, pero menos exacta, por vía telefónica, desde NIST en Fort Collins, pero aquí también hay que hacer una corrección por la ruta de la señal y la velocidad del módem. Esta corrección introduce por lo general cierta imprecisión, lo que dificulta la obtención del tiempo con una precisión extremadamente alta.

3.1.3. Algoritmos para la sincronización de relojes

Si una máquina tiene un receptor WWV, entonces el objetivo es hacer que todas las máquinas se sincronicen con ella. Si ninguna máquina tiene receptores WWV, entonces cada máquina lleva el registro de su propio tiempo y el objetivo es mantener el tiempo de todas las máquinas tan cercano como sea posible. Se han propuesto muchos algoritmos para lograr esta sincronización (por ejemplo, Cristian, 1989; Drummond y Babaoglu, 1993; y Kopetz y Ochsenreiter, 1987). Un panorama de ellos está dado en (Ramanathan *et al.*, 1990b).

Todos los algoritmos tienen el mismo modelo subyacente del sistema, que describiremos a continuación. Se supone que cada máquina tiene un cronómetro, el cual provoca una interrupción H veces por cada segundo. Cuando este cronómetro se detiene, el controlador de interrupciones suma 1 a un reloj en software, el cual mantiene el registro del número de marcas (interrupciones) a partir de cierta fecha acordada en el pasado. Llamaremos al valor de este reloj C. Más precisamente, cuando el tiempo UTC es t, el valor del reloj en la máquina p es $C_p(t)$. En un mundo perfecto, tendríamos $C_p(t) = t$ para toda p y toda t. En otras palabras, dC/dt debería ser 1.

Los cronómetros reales no realizan interrupciones exactamente H veces por cada segundo. En teoría, un cronómetro con $H = 60$ generaría 216 000 marcas por hora. En la práctica, el error relativo que se puede obtener con los circuitos de cronómetros modernos es de cerca de 10^{-5}, lo que significa que una máquina particular puede obtener un valor en el margen que va de 215 998 a 216 002 marcas por hora. Más precisamente, si existe una constante tal que

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

se dice que el cronómetro trabaja dentro de su especificación. La constante ρ es especificada por el fabricante y se conoce como la **tasa máxima de alejamiento**. En la figura 3-5 se muestra un reloj lento, otro perfecto y otro rápido.

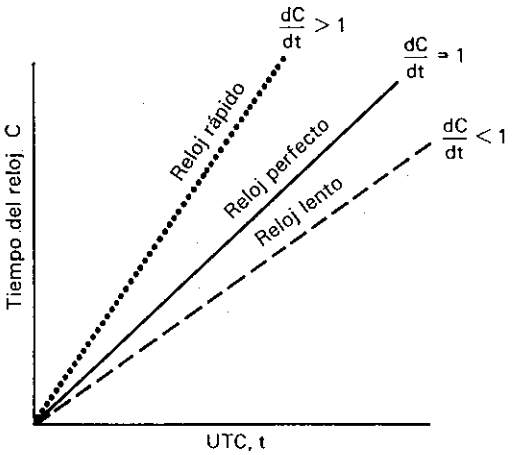


Figura 3-5. No todos los relojes marcan precisamente a la velocidad correcta.

Si dos relojes se alejan de UTC en la dirección opuesta, en un instante Δt después de que fueron sincronizados, podrían estar tan alejados como $2\rho\ \Delta t$. Si los diseñadores del sistema operativo desean garantizar que dos relojes cualesquiera no difieran más de δ, entonces los relojes deben volverse a sincronizar (en software) al menos cada $\delta/2\rho$ segundos. Los distintos algoritmos difieren en la forma precisa en que se realiza esta resincronización.

Algoritmo de Cristian

Comenzaremos con un algoritmo adecuado para los sistemas en los que una máquina tiene un receptor WWV y el objetivo es hacer que todas las máquinas se sincronicen con ella. Llamaremos a la máquina con el receptor WWV un **servidor del tiempo**. Nuestro algoritmo se basa en el trabajo de Cristian (1989) y trabajos anteriores. En forma periódica, en un tiempo que no debe ser mayor que $\delta/2\rho$ segundos, cada máquina envía un mensaje al servidor para solicitar el tiempo actual. Esa máquina responde tan pronto como puede con un mensaje que contiene el tiempo actual, C_{UTC}, como se muestra en la figura 3-6.

Como primera aproximación, cuando el emisor obtiene la respuesta, puede hacer que su tiempo sea C_{UTC}. Sin embargo, este algoritmo tiene dos problemas, uno mayor y otro menor. El problema mayor es que el tiempo nunca debe correr hacia atrás. Si el reloj del emisor es rápido, C_{UTC} será menor que el valor actual de C del emisor. El simple traslado de C_{UTC} podría provocar serios problemas, tales como tener un archivo objeto compilado justo después del cambio de reloj y que tenga un tiempo anterior al del archivo fuente, modificado justo antes del cambio del reloj.

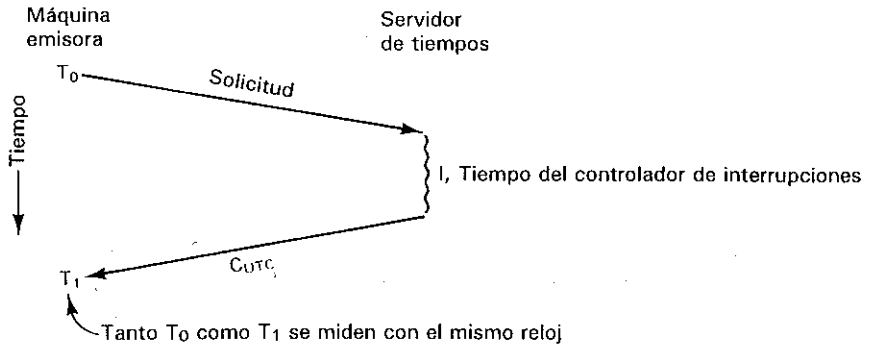


Figura 3-6. Obtención de la hora actual por medio de un servidor de tiempo.

Tal cambio se debe introducir de manera gradual. Una forma es la siguiente. Supongamos que el cronómetro se ajusta para que genere 100 interrupciones por segundo. Lo normal sería que cada interrupción añadiera 10 milisegundos al tiempo. Al reducir la velocidad, la rutina de interrupción sólo añade 9 milisegundos cada vez, hasta que se haga la corrección. De manera similar, el reloj se puede adelantar de manera gradual, sumando 11 milisegundos en cada interrupción en vez de saltar hacia adelante de una vez.

El problema menor es que el tiempo que tarda el servidor en responder al emisor es distinto de cero. Peor aún, este retraso puede ser de gran tamaño y varía con la carga de la red. La forma de enfrentar este problema por parte de Cristian es intentar medirlo. Es muy fácil para el emisor registrar de manera precisa el intervalo entre el envío de la solicitud al servidor de tiempo y la llegada de la respuesta. Tanto el tiempo de inicio, T_0, como el tiempo final, T_1, se miden con el mismo reloj, por lo que el intervalo será relativamente preciso, aunque el reloj del emisor esté alejado de UTC por una magnitud sustancial.

En ausencia de otra información, la mejor estimación del tiempo de propagación del mensaje es de $(T_1-T_0)/2$. Al llegar la respuesta, el valor en el mensaje puede ser incrementado por esta cantidad para dar una estimación del tiempo actual del servidor. Si se conoce el tiempo mínimo de propagación teórico, se pueden calcular otras propiedades de la estimación del tiempo.

Esta estimación se puede mejorar si se conoce con cierta aproximación el tiempo que tarda el servidor del tiempo en manejar la interrupción y procesar el mensaje recibido. Llamaremos al tiempo para el manejo de interrupción I. Entonces la cantidad del tiempo desde T_0 hasta T_1 dedicada a la propagación del mensaje es $T_1 - T_0 - I$, por lo que una mejor estimación del tiempo de propagación en un sentido es la mitad de esta cantidad. Existen sistemas en los que los mensajes de A a B siguen de manera sistemática diferentes rutas de B a A, por lo que tienen un tiempo de propagación distinto, pero aquí no consideraremos estos sistemas.

Para mejorar la precisión, Cristian sugirió hacer no una medición, sino varias. Cualquier medición en la que $T_1 - T_0$ exceda cierto valor límite se descarta, por ser considerada víctima del congestionamiento en la red y por tanto no confiable. Las estimaciones obtenidas de

las demás pruebas se pueden promediar para obtener mejor valor. Otra alternativa es que el mensaje que regrese más rápido se considere como el más preciso, pues supuestamente encontraría menos tráfico y por lo tanto sería el más representativo del tiempo de propagación puro.

El algoritmo de Berkeley

En el algoritmo de Cristian, el servidor de tiempo es pasivo. Otras máquinas le piden el tiempo de manera periódica y todo lo que hace es responder a sus solicitudes. En el UNIX de Berkeley, se sigue el método diametralmente opuesto (Gusella y Zatti, 1989). En este caso, el servidor de tiempo (en realidad, un demonio para el tiempo) está activo y realiza un muestreo periódico de todas las máquinas para preguntarles el tiempo. Con base en las respuestas, calcula un tiempo promedio y le indica a todas las demás máquinas que avancen su reloj a la nueva hora o que disminuyan la velocidad del mismo hasta lograr cierta reducción específica. Este método es adecuado para un sistema donde no exista un receptor de WWV. La hora del demonio para el tiempo debe ser establecida en forma manual por el operador, de manera periódica. El método se muestra en la figura 3-7.

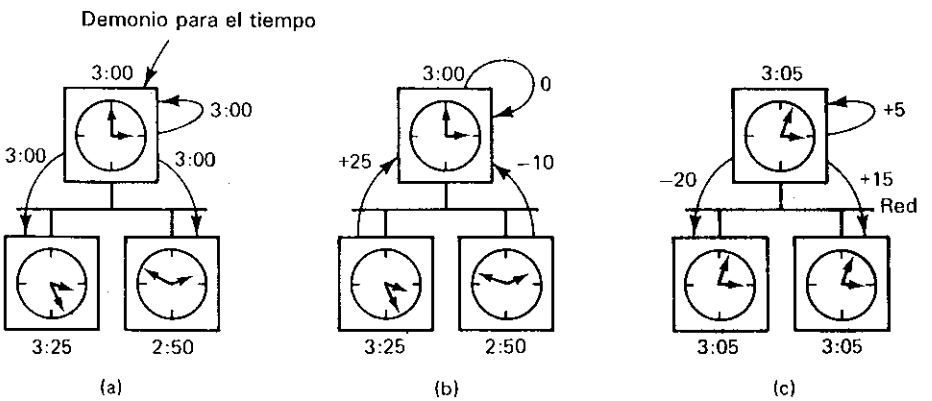


Figura 3-7. (a) El demonio para el tiempo pregunta a todas las demás máquinas el valor de sus relojes. (b) Las máquinas contestan. (c) El demonio para el tiempo les indica la forma de ajustar sus relojes.

En la figura 3-7(a), a las 3:00, el demonio para el tiempo indica a las demás máquinas su hora y les pregunta las suyas. En la figura 3-7(b), las máquinas responden con la diferencia de sus horas con la del demonio. Con estos números, el demonio calcula el tiempo promedio y le dice a cada máquina cómo ajustar su reloj [véase la figura 3-7(c)].

Algoritmos con promedio

Los dos métodos anteriores son muy centralizados, con las desventajas usuales. También se conocen algunos algoritmos descentralizados. Una clase de algoritmos descentralizados trabaja al dividir el tiempo en intervalos de resincronización de longitud fija. El i-ésimo intervalo inicia en $T_0 + iR$ y va hasta $T_0 + (i+1)R$, donde T_0 es un momento ya acordado en el pasado y R es un parámetro del sistema. Al inicio de cada intervalo, cada máquina transmite el tiempo actual según su reloj. Puesto que los relojes de las diversas máquinas no funcionan con exactamente la misma velocidad, estas transmisiones no ocurrirán en forma simultánea.

Después de que una máquina transmite su hora, inicia un cronómetro local para reunir las demás transmisiones que lleguen en cierto intervalo S. Cuando llegan todas las transmisiones, se ejecuta un algoritmo para calcular una nueva hora para ellos. El algoritmo más sencillo consiste en promediar los valores de todas las demás máquinas. Una ligera variación de este tema es descartar primero los m valores más grandes y los m más pequeños, y promediar el resto. El hecho de descartar los valores extremos se puede considerar como autodefensa contra m relojes fallidos que envían mensajes sin sentido.

Otra variación intenta corregir cada mensaje al añadir una estimación del tiempo de propagación desde la fuente. Esta estimación se puede hacer a partir de la topología conocida de la red o al medir el tiempo que tardan en regresar ciertos mensajes de prueba.

Otros algoritmos de sincronización de relojes se analizan en la bibliografía (por ejemplo, Lundelius-Welch y Lynch, 1988; Ramanathan *et al.*, 1990a; Srikanth y Toueg, 1987).

Varias fuentes externas de tiempo

Para los sistemas que requieren una sincronización en extremo precisa con UTC, es posible equiparlos con varios receptores de WWV, GEOS o algunas otras fuentes de UTC. Sin embargo, debido a la imprecisión inherente en la propia fuente de tiempo, así como a las fluctuaciones en la ruta de la señal, lo mejor que puede hacer el sistema operativo es establecer un rango (intervalo de tiempo) en el que caiga UTC. En general, las distintas fuentes de tiempo producirán distintos márgenes, lo que requiere un acuerdo entre las máquinas conectadas a ellas.

Para lograr este acuerdo, cada procesador con fuente UTC puede transmitir su rango en forma periódica; digamos, en el preciso inicio de cada minuto UTC. Ninguno de los procesadores obtendrá los paquetes de tiempo de forma instantánea. Peor aun, el retraso entre la transmisión y recepción depende de la distancia del cable y el número de compuertas que deben atravesar los paquetes, que es diferente para cada pareja (fuente de UTC, procesador). También pueden jugar un papel otros factores, como los retrasos debidos a las colisiones que ocurren cuando varias máquinas intentan transmitir en Ethernet al mismo tiempo. Además, si un procesador está ocupado con un paquete anterior, podría no examinar el paquete de tiempos durante un número considerable de milisegundos, lo que introduce

cierta incertidumbre en el tiempo. En el capítulo 10 analizaremos la forma de sincronizar los relojes en DCE de OSF.

3.1.4. Uso de relojes sincronizados

Hasta hace poco, se dispone del hardware y software necesarios para la sincronización de relojes a gran escala (es decir, en todo Internet). Con esta nueva tecnología, es posible mantener millones de relojes sincronizados con UTC, con precisión de unos cuantos milisegundos. Apenas comienzan a aparecer nuevos algoritmos que utilizan relojes sincronizados. Adelante resumimos dos de los ejemplos analizados por Liskov (1993).

Entrega de mensajes a lo más uno

Nuestro primer ejemplo se refiere a la forma de reforzar la entrega de a lo más un mensaje a un servidor, incluso en presencia de fallas. El método tradicional es que cada mensaje tenga un número de mensaje, y que cada servidor guarde los números de los mensajes que ha visto, de modo que pueda establecer diferencia entre los mensajes nuevos y las retransmisiones. El problema con el algoritmo es que si un servidor falla y vuelve a arrancar, pierde su tabla con los números de mensajes. Además, ¿por cuánto tiempo deben conservarse los números de mensajes?

Con el uso del tiempo, el algoritmo se puede modificar de la siguiente manera. En este caso, cada mensaje tiene un identificador de conexión (elegido por el emisor) y una marca de tiempo. Para cada conexión, el servidor registra en una tabla la marca de tiempo más reciente que haya visto. Si el número de cualquier mensaje recibido en cierta conexión es menor que la marca de tiempo guardada en esa conexión, el mensaje se rechaza como un duplicado.

Para eliminar las marcas de tiempo anteriores, cada servidor mantiene de manera continua una variable global

$$G = TiempoActual - TiempoMáximodeVida - DesviaciónMáximaDelReloj$$

en donde *TiempoMáximoDeVida* es el tiempo máximo de vida de un mensaje y *DesviaciónMáximaDelReloj* es la peor distancia de alejamiento entre UTC y el reloj. Cualquier marca de tiempo anterior a G se puede eliminar con seguridad de la tabla, pues todos los mensajes con esa edad ya han muerto. Si un mensaje recibido tiene un identificador de conexión desconocido, se acepta si su marca de tiempo es más reciente que G y rechazado si su marca de tiempo es anterior a G, puesto que todo lo anterior seguramente es un duplicado. De hecho, G es un resumen de los números de mensaje de todos los mensajes anteriores. Cada ΔT, el tiempo actual se escribe en disco.

Cuando un servidor falla y vuelve a arrancar, vuelve a cargar G del tiempo guardado en el disco y lo incrementa según el periodo de actualización, ΔT. Cualquier mensaje recibido con una marca de tiempo anterior a G se rechaza como duplicado. Como consecuencia, se rechaza cada mensaje que podría ser aceptado antes de la falla. Algunos de los mensajes nuevos podrían rechazarse de manera incorrecta, pero bajo todas las condiciones, el algoritmo conserva la semántica a lo más uno.

Consistencia del caché con base en el reloj

Nuestro segundo ejemplo se refiere a la consistencia del caché en un sistema distribuido de archivos. Por razones de desempeño, es deseable que los clientes puedan ocultar archivos de manera local. Sin embargo, el ocultamiento introduce una potencial inconsistencia si dos clientes modifican el mismo archivo al mismo tiempo. La solución usual consiste en distinguir entre ocultar un archivo para su lectura o para su escritura. La desventaja de este esquema es que si un cliente tiene un archivo oculto para lectura, antes de que otro cliente pueda obtener una copia para escritura, el servidor tiene que solicitar al cliente de lectura que invalide su copia, incluso si la copia se realizó unas cuantas horas antes. Este costo adicional se puede eliminar mediante los relojes sincronizados.

La idea básica es que, cuando un cliente desea un archivo, se le da en **renta**, especificando el tiempo de validez de la copia (Gray y Cheriton, 1989). Cuando la renta está a punto de expirar, el cliente puede solicitar su renovación. Si la renta expira, la copia del caché ya no puede ser utilizada. De esta manera, cuando un cliente necesita leer un archivo una vez, puede solicitarlo. Cuando la renta expira, sólo se desecha; no hay que enviar un mensaje explícito indicando al servidor que ha sido eliminado del caché.

Si una renta expira y el archivo (aún en el caché) se necesita un poco de tiempo después, el cliente puede preguntar al servidor si la copia que tiene (identificada por una marca de tiempo) sigue siendo la activa. En tal caso, se genera una nueva renta, pero sin retransmitir el archivo.

Si uno o más clientes tienen un archivo en caché para su lectura y entonces otro cliente desea escribir en el archivo, el servidor debe solicitar a los lectores que terminen sus rentas de manera prematura. Si uno o más de ellos ha sufrido una falla, el servidor puede sólo esperar hasta que expire la renta del servidor muerto. En el algoritmo tradicional, donde el permiso para guardar el caché debe regresarse de manera explícita del cliente al servidor, ocurre un problema si el servidor pide al cliente o clientes que regresen el archivo (es decir, que lo eliminen de su caché) y no hay respuesta. El servidor no puede determinar si el cliente está muerto o sólo es lento. Con el algoritmo basado en un cronómetro, el servidor puede esperar y dejar que expire la renta.

Además de estos dos algoritmos, Liskov (1993) describe la forma de utilizar los relojes sincronizados para que expiren los boletos utilizados en autenticación de un sistema distribuido, y controlar el compromiso en las transacciones atómicas. Conforme mejore la sincronización de los cronómetros, no hay duda de que aparecerán nuevas aplicaciones.

3.2. EXCLUSIÓN MUTUA

Con frecuencia, los sistemas con varios procesos se programan más fácil mediante las regiones críticas. Cuando un proceso debe leer o actualizar ciertas estructuras de datos compartidas, primero entra a una región crítica para lograr la exclusión mutua y garantizar que ningún otro proceso utilice las estructuras de datos al mismo tiempo. En los sistemas con un procesador, las regiones críticas se protegen mediante semáforos, monitores y construcciones similares. Analizaremos ahora algunos ejemplos de implantación de las regiones críticas y la exclusión mutua en los sistemas distribuidos. Para una taxonomía y bibliografía de otros métodos, véase (Raynal, 1991). Otros trabajos se analizan en (Agrawala y El Abbadi, 1991; Chandy *et al.*, 1983; y Sanders, 1987).

3.2.1. Un algoritmo centralizado

La forma más directa de lograr la exclusión mutua en un sistema distribuido es similar a la forma en que se lleva a cabo en un sistema con un procesador. Se elige un proceso como el coordinador (por ejemplo, aquel que se ejecuta en la máquina con la máxima dirección en la red). Siempre que un proceso desea entrar a una región crítica, envía un mensaje de solicitud al coordinador, donde indica la región crítica a la que desea entrar y pide permiso. Si ningún otro proceso está por el momento en esa región crítica, el coordinador envía una respuesta otorgando el permiso, como se muestra en la figura 3-8(a). Cuando llega la respuesta, el proceso solicitante entra a la región crítica.

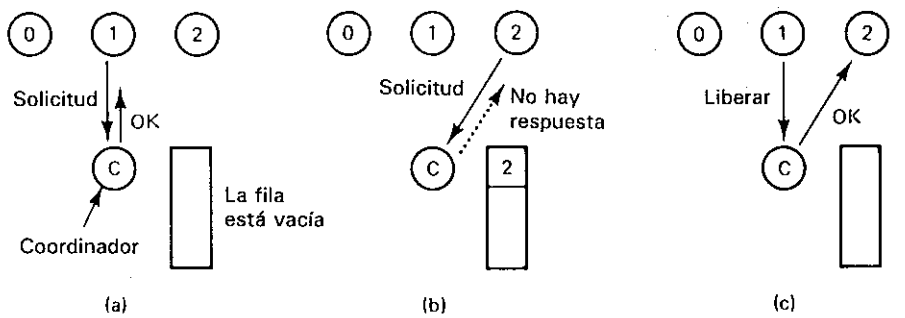


Figura 3-8. (a) El proceso 1 pide permiso al coordinador para entrar en una región crítica. El permiso es concedido. (b) El proceso 2 pide entonces permiso para entrar a la misma región crítica. El coordinador no le responde. (c) Cuando el proceso 1 sale de la región crítica, se lo dice al coordinador, el cual responde entonces a 2.

Supongamos ahora que otro proceso, 2 en la figura 3-8(b), pide permiso para entrar a la misma región crítica. El coordinador sabe que un proceso distinto ya se encuentra en esta región, por lo que no puede otorgar el permiso. El método exacto utilizado para negar

el permiso depende del sistema. En la figura 3-8(b), el coordinador sólo se abstiene de responder, con lo cual se bloquea el proceso 2, que espera una respuesta. Otra alternativa consiste en enviar una respuesta que diga "permiso negado". De cualquier manera, forma en una fila la solicitud de 2 por el momento.

Cuando el proceso 1 sale de la región crítica, envía un mensaje al coordinador para liberar su acceso exclusivo, como se muestra en la figura 3-8(c). El coordinador extrae el primer elemento de la fila de solicitudes diferidas y envía a ese proceso un mensaje otorgando el permiso. Si el proceso estaba bloqueado (es decir, éste es el primer mensaje que se le envía), elimina el bloqueo y entra a la región crítica. Si ya se envió un mensaje explícito negando el permiso, entonces el proceso debe hacer un muestreo del tráfico recibido o bloquearse posteriormente. De cualquier forma, cuando ve el permiso, puede entrar a la región crítica.

Es fácil ver que el algoritmo garantiza la exclusión mutua: el coordinador deja que un proceso esté en cada región crítica a la vez. También es justo, puesto que las solicitudes se aprueban en el orden en que se reciben. Ningún proceso espera por siempre (no hay inanición). Este esquema también es fácil de implantar y sólo requiere tres mensajes por cada uso de una región crítica (solicitud, otorgamiento, liberación). También se puede utilizar para una asignación de recursos más general, en vez de usarlo sólo para el manejo de las regiones críticas.

El método centralizado también tiene limitaciones. El coordinador es un punto de falla, por lo que si se descompone, todo el sistema se puede venir abajo. Si los procesos se bloquean por lo general después de realizar una solicitud, no pueden distinguir entre un coordinador muerto de un "permiso negado", puesto que en ambos casos no reciben respuesta. Además, en un sistema de gran tamaño, un coordinador puede convertirse en un cuello de botella para el desempeño.

3.2.2. Un algoritmo distribuido

Con frecuencia, el hecho de tener un punto de falla es inaceptable, por lo cual los investigadores han buscado algoritmos distribuidos de exclusión mutua. El artículo de 1978 de Lamport relativo a la sincronización de los relojes presentó el primero de ellos. Ricart y Agrawala (1981) lo hicieron más eficiente. En esta sección describiremos su método.

El algoritmo de Ricart y Agrawala requiere de la existencia de un orden total de todos los eventos en el sistema. Es decir, para cualquier pareja de eventos, como los mensajes, debe quedar claro cuál de ellos ocurrió primero. El algoritmo de Lamport presentado en la sección 3.1.1 es una forma de lograr este orden y se puede utilizar para proporcionar marcas de tiempo para la exclusión mutua distribuida.

El algoritmo funciona como sigue. Cuando un proceso desea entrar a una región crítica, construye un mensaje con el nombre de ésta, su número de proceso y la hora actual. Entonces envía el mensaje a todos los demás procesos y de manera conceptual a él mismo. Se supone que el envío de mensajes es confiable; es decir, cada mensaje tiene un reconocimiento. Si

se dispone de una comunicación en grupo confiable, entonces ésta se puede utilizar en vez del envío de mensajes individuales.

Cuando un proceso recibe un mensaje de solicitud de otro proceso, la acción que realice depende de su estado con respecto de la región crítica nombrada en el mensaje. Hay que distinguir tres casos:

1. Si el receptor no está en la región crítica y no desea entrar a ella, envía de regreso un mensaje *OK* al emisor.
2. Si el receptor ya está en la región crítica, no responde, sino que forma la solicitud en una fila.
3. Si el receptor desea entrar a la región crítica, pero no lo ha logrado todavía, compara la marca de tiempo en el mensaje recibido con la marca contenida en el mensaje que envió a cada uno. La menor de las marcas gana. Si el mensaje recibido es menor, el receptor envía de regreso un mensaje *OK*. Si su propio mensaje tiene una marca menor, el receptor forma la solicitud en una fila y no envía nada.

Después de enviar las solicitudes que piden permiso para entrar a una región crítica, un procesador espera hasta que alguien más obtiene el permiso. Tan pronto llegan todos los permisos, puede entrar a la región crítica. Cuando sale de ella, envía mensajes *OK* a todos los procesos en su fila y elimina a todos los elementos de la fila.

Intentemos comprender el funcionamiento del algoritmo. Si no existe conflicto, es claro que funciona. Sin embargo, supongamos que dos procesos intentan entrar a la misma región crítica en forma simultánea, como se muestra en la figura 3-9(a).

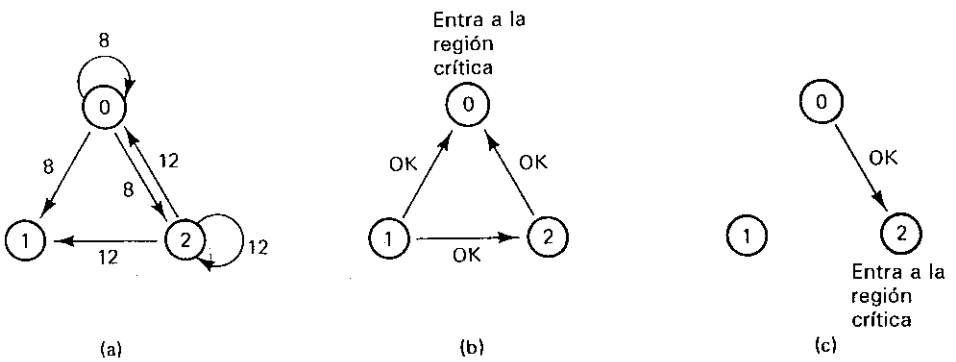


Figura 3-9. (a) Dos procesos desean entrar a la misma región crítica en el mismo momento. (b) El proceso 0 tiene una marca de tiempo menor, por lo que gana. (c) Cuando termina el proceso 0, envía un *OK*, por lo que 2 puede ahora entrar en la región crítica.

El proceso 0 envía a todos una solicitud con la marca de tiempo 8, mientras que al mismo tiempo, el proceso 2 envía a todos una solicitud con la marca de tiempo 12. El

proceso 1 no se interesa en entrar a la región crítica, por lo que envía *OK* a ambos emisores. Los procesos 0 y 2 ven el conflicto y comparan las marcas. El proceso 2 ve que ha perdido, por lo que otorga el permiso a 0 al enviar un mensaje *OK*. El proceso 0 forma ahora la solicitud de 2 en una fila para posterior procesamiento y entra a la región crítica, como se muestra en la figura 3-9(b). Cuando termina, retira la solicitud de 2 de la fila y envía un mensaje *OK* al proceso 2, lo que permite a éste entrar a su región crítica, como se muestra en la figura 3-9(c). El algoritmo funciona, puesto que en caso de un conflicto, gana la menor de las marcas y todos coinciden en el orden de las marcas de tiempo.

Observe que la situación de la figura 3-9 sería en esencia distinta si el proceso 2 hubiese enviado un mensaje en un tiempo anterior, de modo que el proceso 0 lo haya recibido y otorgado permiso antes de hacer su propia solicitud. En este caso, 2 habría notado que él mismo estaba en una región crítica al momento de la solicitud y se formaría en una fila en vez de enviar una respuesta.

Como en el caso del algoritmo centralizado ya analizado, la exclusión mutua queda garantizada sin bloqueo ni inanición. El número de mensajes necesarios por entrada es ahora $2(n-1)$, donde n es el número total de procesos en el sistema. Lo mejor es que no existe un punto de falla.

Por desgracia, el único punto de falla es remplazado por n puntos de falla. Si cualquier proceso falla, no podrá responder a las solicitudes. Este silencio será interpretado (incorrectamente) como negación del permiso, con lo que se bloquearán los siguientes intentos de los demás procesos por entrar a todas las regiones críticas. Puesto que la probabilidad de que uno de los n procesos falle es n veces mayor que la probabilidad de que falle un coordinador, hemos trabajado para remplazar un algoritmo pobre con otro que es n veces peor y que requiere mayor tráfico en la red para funcionar.

El algoritmo se puede mejorar mediante el mismo truco propuesto antes. Al llegar una solicitud, el receptor siempre envía una respuesta, otorgando o negando el permiso. Siempre que pierda una solicitud o una respuesta, el emisor espera y sigue intentando hasta que regresa una respuesta o el emisor concluye que el destino está muerto. Después de negar una solicitud, el emisor debe bloquearse en espera de un mensaje *OK* posterior.

Otro problema con este algoritmo es que se debe utilizar una primitiva de comunicación en grupo; o bien, cada proceso debe mantener por sí mismo la lista de membresía del grupo, donde se incluyan los procesos que ingresan al grupo, los que salen de él y los que fallan. El método funciona mejor con grupos pequeños de procesos que nunca cambian sus membresías de grupo.

Por último, recordemos que uno de los problemas con el algoritmo centralizado es que al hacer que maneje todas las solicitudes, esto puede conducir a un cuello de botella. En el algoritmo distribuido, *todos* los procesos participan en *todas* las decisiones referentes a la entrada en las regiones críticas. Si un proceso no puede manejar esta tarea, es poco probable que obligar a todos a que realicen lo mismo en paralelo sea de mucha ayuda.

Es posible mejorar un poco este algoritmo. Por ejemplo, la obtención del permiso de todos para entrar a una región crítica es en realidad redundante. Todo lo que se necesita es un método para evitar que dos procesos entren a la misma región crítica al mismo tiempo. El algoritmo se puede modificar para permitir que un proceso entre a una región crítica cuando ha conseguido el permiso de una mayoría simple de los demás procesos, en vez de todos ellos. Por supuesto, en esta variante, después de que un proceso ha otorgado el permiso a otro para entrar a una región crítica, no puede otorgar el mismo permiso a otro proceso hasta que el primero lo libere. Son posibles otras mejoras (por ejemplo, Maekawa *et al.*, 1987).

Sin embargo, este algoritmo es más lento, más complejo, más caro y menos robusto que el algoritmo centralizado original. ¿Por qué estudiarlo bajo estas condiciones? En primer lugar, muestra la posibilidad de un algoritmo distribuido, algo que no era obvio en un principio. Además, al señalar las limitaciones, podríamos estimular a futuros teóricos para que intenten producir algoritmos que en realidad sean útiles. Por último, al igual que los hechos de comer espinacas y aprender latín en el bachillerato, se dice que algunas cosas son buenas para uno, en cierta forma abstracta.

3.2.3. Un algoritmo de anillo de fichas

Un método por completo distinto para lograr la exclusión mutua en un sistema distribuido se muestra en la figura 3-10. Aquí tenemos una red basada en un bus, como se muestra en la figura 3-10(a) (por ejemplo, Ethernet), sin un orden inherente en los procesos. En software, se construye un anillo lógico y a cada proceso se le asigna una posición en el anillo, como se muestra en la figura 3-10(b). Las posiciones en el anillo se pueden asignar según el orden numérico de las direcciones de la red o mediante algún otro medio. No importa cómo sea el orden. Lo importante es que cada proceso sepa quién es el siguiente en la fila después de él.

Al iniciar el anillo, se le da al proceso 0 una **ficha**, la cual circula en todo el anillo. Se trasfiere del proceso k al proceso $k+1$ (módulo el tamaño del anillo) en mensajes puntuales. Cuando un proceso obtiene la ficha de su vecino, verifica si intenta entrar a una región crítica. En ese caso, el proceso entra a la región, hace todo el trabajo necesario y sale de la región. Después de salir, pasa la ficha a lo largo del anillo. No se permite entrar a una segunda región crítica con la misma ficha.

Si un proceso recibe la ficha de su vecino y no está interesado en entrar a una región crítica, sólo la vuelve a pasar. En consecuencia, cuando ninguno de los procesos desea entrar a una región crítica, la ficha sólo circula a gran velocidad en el anillo.

Es evidente que el algoritmo es correcto. En un instante dado, sólo uno de los procesos tiene la ficha, por lo que sólo un proceso puede estar en una región crítica. Puesto que las fichas circulan entre los procesos en orden bien definido, no puede existir la inanición. Una vez que un proceso decide entrar a una región crítica, lo peor que puede ocurrir es que deba esperar a que los demás procesos entren y salgan de ella.

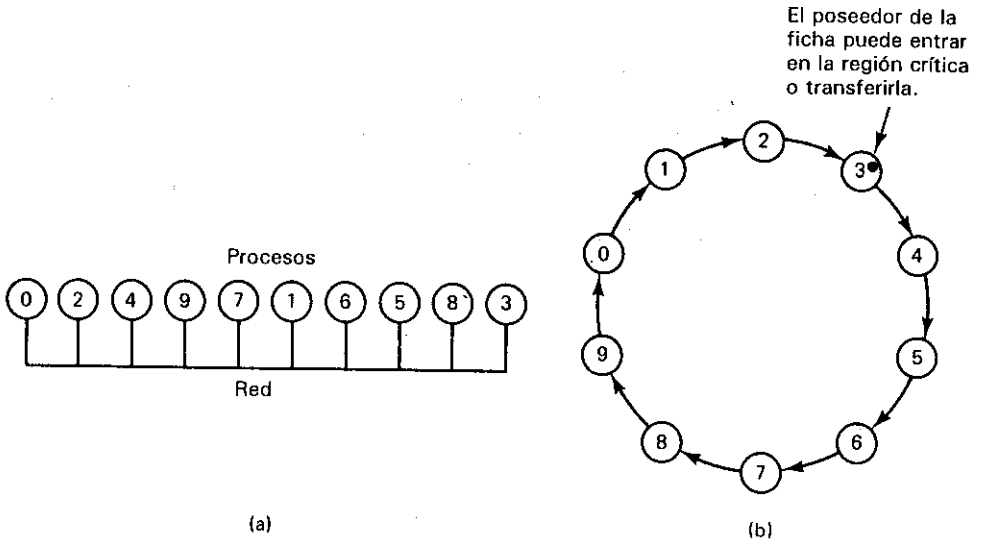


Figura 3-10. (a) Un grupo no ordenado de procesos en una red. (b) Un anillo lógico construido en software.

Como es usual, también este algoritmo tiene problemas. Si la ficha llega a perderse, debe ser regenerada. De hecho, es difícil detectar su pérdida, puesto que la cantidad de tiempo entre las apariciones sucesivas de la ficha en la red no está acotada. El hecho de que la ficha no se haya observado durante una hora no significa su pérdida; tal vez alguien la esté utilizando.

El algoritmo también tiene problemas si falla un proceso, pero la recuperación es más sencilla que en los demás casos. Si pedimos un reconocimiento a cada proceso que reciba la ficha, entonces se detectará un proceso muerto si su vecino intenta darle la ficha y fracasa en el intento. En ese momento, el proceso muerto se puede eliminar del grupo y el poseedor de la ficha puede enviar ésta por encima de la cabeza del proceso muerto al siguiente miembro, y así sucesivamente, en caso necesario. Por supuesto, esto requiere que todos mantengan la configuración actual del anillo.

3.2.4. Comparación de los tres algoritmos

Es instructiva una breve comparación de los tres algoritmos para la exclusión mutua que hemos analizado. En la figura 3-11 enumeramos los algoritmos y tres propiedades fundamentales: el número de mensajes necesarios para que un proceso entre y salga de una región crítica, el retraso antes de que pueda ocurrir una entrada (suponiendo que los mensajes se transfieren de manera secuencial en una LAN) y algunos de los problemas asociados con cada algoritmo.

El algoritmo centralizado es el más sencillo y también el más eficiente. Sólo requiere de tres mensajes para entrar y salir de una región crítica: una solicitud y otorgamiento para entrar y una liberación para salir. El algoritmo distribuido necesita $n-1$ mensajes de solicitud, uno para cada uno de los demás procesos y $n-1$ mensajes de otorgamiento, para un total de $2(n-1)$. Este número es variable con el algoritmo del anillo de fichas. Si todos los procesos desean constantemente entrar a una región crítica, entonces cada paso de la ficha provocará una entrada y salida, para un promedio de un mensaje por cada región crítica a la que se ha entrado. En el otro extremo, a veces la ficha podría circular durante horas, sin que nadie se interese en ella. En este caso, el número de mensajes por entrada en una región crítica no es acotado.

Algoritmo	Mensajes por dato/salida	Retraso antes del dato (en tiempo de mensajes)	Problemas
Centralizado	3	2	Fallo del coordinador
Distribuido	2 (n – 1)	2 (n – 1)	Fallo de cualquier proceso
Anillo de elementos	1 a ∞	0 a n – 1	Ficha perdida, falla del proceso

Figura 3-11. Comparación de tres algoritmos de exclusión mutua.

El retraso desde el momento en que un proceso necesita entrar a una región crítica hasta su entrada real también varía en los tres algoritmos. Cuando las regiones críticas son cortas y se utilizan pocas veces, el factor dominante en el retraso es el mecanismo real para la entrada a una región crítica. Cuando las regiones son de gran tamaño y de uso frecuente, el factor dominante es la espera para tomar su turno. En la figura 3-11 mostramos el primer caso. En el caso centralizado, sólo se necesitan dos tiempos de mensaje para entrar a la región crítica, pero en el caso distribuido son $2(n-1)$ tiempos de mensajes, si la red sólo puede manejar un mensaje a la vez. Para el caso del anillo de fichas, el tiempo varía desde 0 (la ficha acaba de llegar) hasta $n-1$ (la ficha acaba de salir).

Por último, los tres algoritmos sufren en caso de fallas. Se pueden utilizar medidas especiales y complejidad adicional, para evitar que una falla haga que todo el sistema se venga abajo. Es un poco irónico que los algoritmos distribuidos sean más sensibles a las fallas que los centralizados. En un sistema tolerante de fallas, ninguno de éstos sería adecuado, pero si las fallas son poco frecuentes, todos son aceptables.

3.3. ALGORITMOS DE ELECCIÓN

Muchos de los algoritmos distribuidos necesitan que un proceso actúe como coordinador, iniciador, secuenciador o que desempeñe de cierta forma algún papel especial. Ya hemos visto varios ejemplos, como el coordinador en el algoritmo centralizado de exclusión

mutua. En general, no importa cuál de los procesos asuma esta responsabilidad especial, pero uno de ellos debe hacerlo. En esta sección analizaremos los algoritmos para la elección de un coordinador (utilizaremos éste como nombre genérico del proceso especial).

Si todos los procesos son idénticos, sin alguna característica que los distinga, no existe forma de elegir uno de ellos como especial. En consecuencia, supondremos que cada proceso tiene un número único; por ejemplo, su dirección en la red (para hacer más sencilla la exposición, supondremos que existe un proceso por cada máquina). En general, los algoritmos de elección intentan localizar al proceso con el máximo número de proceso y designarlo como coordinador. Los algoritmos difieren en la forma en que lo llevan a cabo.

Además, también supondremos que cada proceso conoce el número de proceso de todos los demás. Lo que el proceso desconoce es si los procesos están activos o inactivos. El objetivo de un algoritmo de elección es garantizar que al inicio de una elección, ésta concluya con el acuerdo de todos los procesos con respecto a la identidad del nuevo coordinador. Se conocen varios algoritmos; por ejemplo (Fredrickson y Lynch, 1987; García-Molina, 1982; y Singh y Kurose, 1994).

3.3.1. El algoritmo del grandulón

Como primer ejemplo, consideremos al **algoritmo del grandulón**, diseñado por García-Molina (1982). Cuando un proceso observa que el coordinador ya no responde a las solicitudes, inicia una elección. Un proceso *P* realiza una elección de la siguiente manera:

1. *P* envía un mensaje *ELECCIÓN* a los demás procesos con un número mayor.

2. Si nadie responde, *P* gana la elección y se convierte en el coordinador.

3. Si uno de los procesos con un número mayor responde, toma el control. El trabajo de *P* termina.

En cualquier momento, un proceso puede recibir un mensaje *ELECCIÓN* de uno de sus compañeros con un número menor. Cuando llega dicho mensaje, el receptor envía de regreso un mensaje *OK* al emisor para indicar que está vivo y que tomará el control. El receptor realiza entonces una elección, a menos que ya esté realizando alguna. En cierto momento, todos los procesos se rinden, menos uno, el cual se convierte en el nuevo coordinador. Anuncia su victoria al enviar un mensaje a todos los procesos para indicarles que a partir de ese momento es el nuevo coordinador.

Si un proceso inactivo se activa, realiza una elección. Si ocurre que es el proceso en ejecución con el número máximo, ganará la elección y tomará para sí el trabajo del coordinador. Así, siempre gana el tipo más grande del pueblo, de ahí el nombre "algoritmo del grandulón".

En la figura 3-12 vemos un ejemplo del funcionamiento de este algoritmo. El grupo consta de ocho procesos, numerados del 0 al 7. Antes, el proceso 7 era el coordinador, pero

acaba de fallar. El proceso 4 es el primero en notarlo, por lo que envía mensajes *ELECCIÓN* a todos los procesos mayores que él, 5, 6 y 7, como se muestra en la figura 3-12(a). Los procesos 5 y 6 responden con *OK*, como se muestra en la figura 3-12(b). Al recibir la primera de estas respuestas, 4 sabe que su trabajo ha terminado. Sabe que alguno de estos grandulones tomará el control y se convertirá en el coordinador. Sólo se sienta y espera para ver quién es el ganador (aunque en este punto puede hacer una estimación relativamente buena).

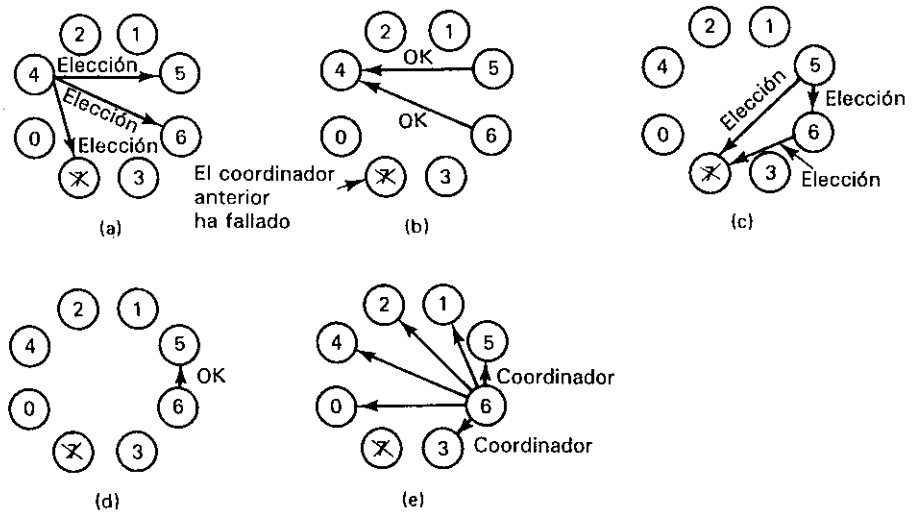


Figura 3-12. El algoritmo del grandulón para la elección. (a) El proceso 4 hace una elección. (b) Los procesos 5 y 6 responden e indican a 4 que se detenga. (c) Ahora, 5 y 6 hacen una elección. (d) El proceso 6 indica a 5 que se detenga. (e) El proceso 6 gana y se lo informa a todos.

En la figura 3-13(c), tanto 5 como 6 realizan elecciones y cada uno envía mensajes a los demás procesos mayores que él. En la figura 3-13(d), el proceso 6 indica a 5 que tomará el control. En este punto, 6 sabe que 7 está muerto y que él (6) es el ganador. Si existe cierta información de estado a recoger del disco o alguna otra parte abandonada por el coordinador, 6 debe hacer lo que sea necesario. Cuando está listo para asumir el control, 6 anuncia esto al enviar un mensaje *COORDINADOR* a todos los procesos en ejecución. Cuando 4 recibe este mensaje, puede continuar entonces con la operación que intentaba llevar a cabo cuando descubrió la muerte de 7, sólo que ahora utiliza a 6 como coordinador. De esta forma, la falla de 7 se controla y el trabajo puede continuar.

Si el proceso 7 vuelve al inicio, tan sólo enviará a los demás un mensaje *COORDINADOR* y los someterá.

3.3.2. Un algoritmo de anillo

Otro algoritmo de elección se basa en el uso de un anillo, sólo que a diferencia del anillo anterior, éste no utiliza una ficha. Suponemos que los procesos tienen un orden, físico o lógico, de modo que cada proceso conoce a su sucesor. Cuando algún proceso observa que el coordinador no funciona, construye un mensaje *ELECCIÓN* con su propio número de proceso y envía el mensaje a su sucesor. Si éste está inactivo, el emisor pasa sobre el sucesor y va hacia el siguiente número del anillo o al siguiente de éste, hasta que localiza un proceso en ejecución. En cada paso, el emisor añade su propio número de proceso a la lista en el mensaje.

En cierto momento, el mensaje regresa a los procesos que lo iniciaron. Ese proceso reconoce este evento cuando recibe un mensaje de entrada con su propio número de proceso. En este punto, el mensaje escrito cambia a *COORDINADOR* y circula de nuevo, esta vez para informar a todos los demás quién es el coordinador (el miembro de la lista con el número máximo) y quiénes son los miembros del nuevo anillo. Una vez circulado el mensaje, se elimina y todos se ponen a trabajar.

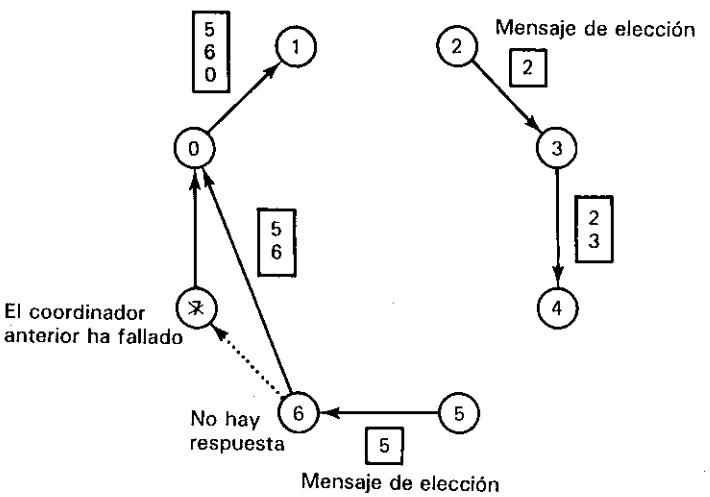


Figura 3-13. Algoritmo de elección mediante un anillo.

En la figura 3-13 vemos lo que ocurre si dos procesos, 2 y 5, descubren en forma simultánea que ha fallado el anterior coordinador, el proceso 7. Cada uno de ellos construye un mensaje *ELECCIÓN* y comienza a circularlo. En cierto momento, ambos mensajes habrán dado una vuelta completa y ambos se convertirán en mensajes *COORDINADOR*, con los mismos miembros y en el mismo orden. Cuando han dado una vuelta completa, ambos serán eliminados. No hace daño la circulación de mensajes adicionales; a lo más se desperdicia un poco de ancho de banda.

3.4. TRANSACCIONES ATÓMICAS

Todas las técnicas de sincronización estudiadas hasta el momento son en esencia de bajo nivel, como los semáforos. Necesitan que el programador se enfrente de forma directa con los detalles de la exclusión mutua, el manejo de las regiones críticas, prevención de bloqueos y recuperación de una falla. Lo que quisiéramos en realidad es una abstracción de mayor nivel, que oculte estos aspectos técnicos y permita a los programadores concentrarse en los algoritmos y la forma en que los procesos trabajan juntos en paralelo. Tal abstracción existe y se utiliza con amplitud en los sistemas distribuidos. La llamaremos **transacción atómica**, o sólo transacción. También se utiliza el término **acción atómica**. En esta sección examinaremos el uso, diseño e implantación de las transacciones atómicas.

3.4.1. Introducción a las transacciones atómicas

El modelo original de la transacción atómica proviene del mundo de los negocios. Supongamos que la Compañía Internacional Dingbat necesita un lote de ciertos artefactos. Se comunican con un posible proveedor, Artefactos, S.A., conocido ampliamente por la calidad de sus artefactos, para que les proporcione 100 000 artefactos de color púrpura de 10 cm cada uno, para entregarse en junio. Artefactos, S.A. ofrece 100 000 artefactos de 4 pulgadas, en color malva, para entregarse en diciembre. Dingbat está de acuerdo con el precio, pero no le gusta el color malva, los desea en julio e insiste en la medida de 10 cm para sus clientes internacionales. Artefactos responde con un ofrecimiento de artefactos lavanda de 3 15/16 pulgadas en octubre. Después de varias negociaciones, coinciden en artefactos violeta, de 3 959/1024 pulgadas, para entregarse el 15 de agosto.

Hasta este momento, ambas partes son libres de terminar la discusión, en cuyo caso el mundo regresa al estado en que se encontraba antes de comenzar las pláticas; sin embargo, una vez que las compañías firman un contrato, están obligadas legalmente a concluir la venta, pase lo que pase. Así, antes de que ambas partes firmen en la línea punteada, cualquiera puede retroceder como si nada hubiese pasado; pero al momento en que ambos firman, pasan un punto sin retorno y la transacción debe llevarse a cabo.

El modelo de la computadora es similar. Un proceso anuncia que desea comenzar una transacción con uno o más procesos. Pueden negociar varias opciones, crear y eliminar objetos y llevar a cabo ciertas operaciones durante unos momentos. Entonces, el iniciador anuncia que desea que todos los demás se comprometan con el trabajo realizado hasta entonces. Si todos coinciden, los resultados se vuelven permanentes. Si uno o más procesos se niegan (o fallan antes de expresar su acuerdo), entonces la situación regresa al estado que presentaba antes de comenzar la transacción, sin que existan efectos colaterales en los objetos, archivos, bases de datos, etc. Esta propiedad del todo o nada facilita el trabajo del programador.

El uso de transacciones en los sistemas de cómputo data de la década de 1960. Antes de la existencia de los discos y las bases de datos en línea, todos los archivos se mantenían en cintas magnéticas. Imaginemos un supermercado con un sistema automatizado para los

inventarios. Cada día, después del cierre, la ejecución de la computadora se realizaba mediante dos cintas de entrada. La primera contenía todo el inventario al tiempo de la apertura, en la mañana. La segunda contenía una lista de las actualizaciones del día: los productos vendidos a los clientes y los productos entregados por los proveedores. La computadora leía ambas cintas y producía una nueva cinta maestra de inventario, como se muestra en la figura 3-14.

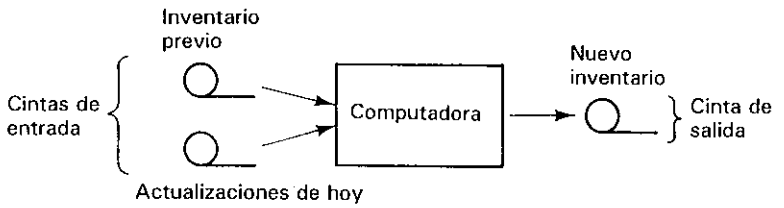


Figura 3-14. La actualización de una cinta maestra es tolerante de fallas.

La gran belleza de este esquema (aunque las personas que lo vivieron no se daban cuenta de ello) es que si una ejecución fallaba por cierta razón, todas las cintas se rebobinaban y el trabajo volvía a comenzar sin que hubiera algún daño. Aún siendo primitivo, el antiguo sistema de cintas magnéticas tenía la propiedad del todo o nada de la transacción atómica.

Analicemos ahora una aplicación bancaria moderna, la cual actualiza una base de datos en línea. El cliente llama al banco mediante una PC con un módem, con la intención de retirar dinero de una cuenta y depositarlo en otra. La operación se lleva a cabo en dos etapas:

1. Retiro(cantidad, cuenta1).
2. Depósito(cantidad, cuenta2).

Si la conexión telefónica falla después de la primera etapa pero antes de la segunda, la primera cuenta tendrá un retiro y la otra no recibirá el dinero; éste se ha desvanecido en el aire.

El problema se resolvería mediante la agrupación de las dos operaciones en una transacción atómica. O las dos terminarían, o bien ninguna de las dos. La clave es regresar al estado inicial si la transacción no puede concluir. Lo que deseamos en realidad es una forma para rebobinar la base de datos, como en el caso de las cintas magnéticas. Esta capacidad es lo que debe ofrecer la transacción atómica.

3.4.2. El modelo de transacción

Ahora desarrollaremos un modelo más preciso de una transacción y sus propiedades. Supondremos que el sistema consta de varios procesos independientes, cada uno de los cuales puede fallar de manera aleatoria. La comunicación es por lo general no confiable,

en el sentido de que se pueden perder mensajes, pero los niveles inferiores pueden utilizar un protocolo de tiempos de espera y retransmisión para recuperarse de la pérdida de mensajes. Así, en el curso del análisis supondremos que el software subyacente maneja de manera transparente los errores de comunicación.

Almacenamiento estable

El almacenamiento tiene tres categorías. En primer lugar, tenemos la memoria RAM ordinaria, que se limpia al fallar el suministro de energía o en un fallo de la máquina. A continuación tenemos el almacenamiento en disco, que sobrevive a las fallas del CPU, pero que se puede perder debido a problemas con la cabeza lectora del disco.

Por último tenemos el **almacenamiento estable**, diseñado para sobrevivir a todo, excepto catástrofes mayores, como las inundaciones y terremotos. El almacenamiento estable se puede implantar con una pareja de discos ordinarios, como se muestra en la figura 3-15(a). Cada bloque en la unidad 2 es una copia exacta del bloque correspondiente en la unidad 1. Cuando se actualiza un bloque, primero se actualiza y verifica el bloque de la unidad 1, para después encargarse del mismo bloque de la unidad 2.

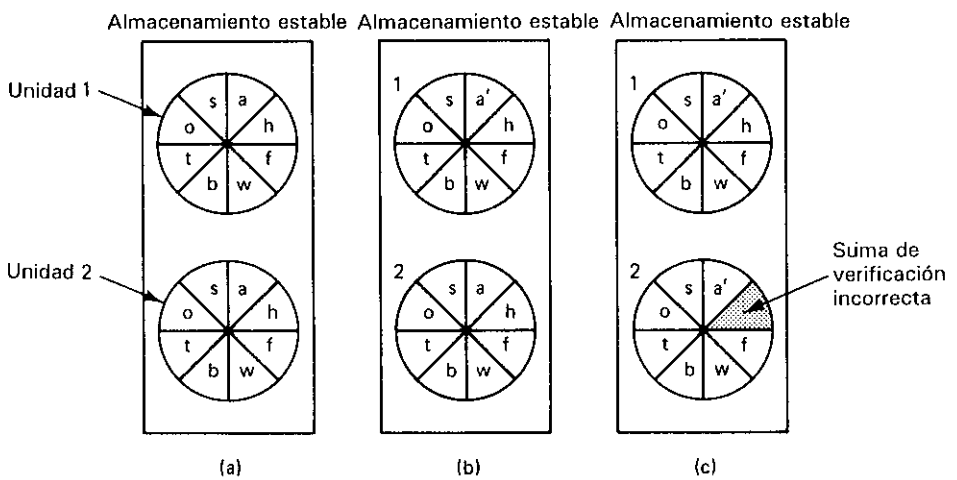


Figura 3-15. (a) Almacenamiento estable. (b) Falla después de la actualización de la unidad 1. (c) Punto defectuoso.

Supongamos que el sistema falla después de la actualización de la unidad 1, pero antes de actualizar la unidad 2, como se muestra en la figura 3-15(b). Después de la recuperación, el disco se puede comparar bloque por bloque. En caso que dos bloques correspondientes difieran, se puede suponer que la unidad 1 es la correcta (puesto que la unidad 1 siempre se actualiza antes de la unidad 2), por lo que el nuevo bloque se copia de la unidad 1 a la unidad 2. Al terminar el proceso de recuperación, ambas unidades vuelven a ser idénticas.

Otro problema potencial es el deterioro espontáneo de un bloque. Las partículas de polvo, el uso o rasgado pueden provocar un error en la suma de verificación de un bloque anteriormente válido, sin causa o advertencia alguna, como se muestra en la figura 3-15(c). Cuando se detecta un error de este tipo, se puede regenerar el bloque defectuoso por medio del bloque correspondiente en la otra unidad.

Como consecuencia de esta implantación, el almacenamiento estable es adecuado para las aplicaciones que requieren de un alto grado de tolerancia de fallas, como las transacciones atómicas. Cuando se escriben los datos en un almacenamiento estable y después se leen de nuevo para verificar si fueron escritos de manera correcta, la probabilidad de que se pierdan es muy pequeña.

Primitivas de transacción

La programación con uso de transacciones requiere de primitivas especiales, las cuales deben ser proporcionadas por el sistema operativo o por el compilador del lenguaje. Algunos ejemplos son:

1. BEGIN_TRANSACTION: Señala el inicio de una transacción.
2. END_TRANSACTION: Termina la transacción y se intenta un compromiso.
3. ABORT_TRANSACTION: Se elimina la transacción; se recuperan los valores anteriores.
4. READ: Se leen datos de un archivo (o algún otro objeto).
5. WRITE: Se escriben datos en un archivo (o algún otro objeto).

La lista exacta de primitivas depende del tipo de objetos que se utilicen en la transacción. En un sistema de correo, podrían existir primitivas para el envío, recepción y direccionamiento del correo. En un sistema de contabilidad, podrían ser un poco distintas. Sin embargo, READ y WRITE son ejemplos típicos. Dentro de una transacción se permiten también enunciados ordinarios, llamadas a procedimientos, etcétera.

BEGIN_TRANSACTION y END_TRANSACTION se utilizan para establecer los límites de una transacción. Las operaciones entre ellas forman el cuerpo de la transacción. Todas o ninguna de ellas deben ejecutarse. Estas operaciones pueden ser llamadas al sistema, procedimientos de biblioteca o enunciados encapsulados en un lenguaje, según la implantación.

Consideremos por ejemplo el proceso de reservación de un asiento para volar de White Plains, Nueva York a Malindi, Kenia, en un sistema de reservaciones en una línea aérea. Una ruta es de White Plains al aeropuerto JFK de Nueva York, del JFK a Nairobi y de Nairobi a Malindi. En la figura 3-16(a) vemos las reservaciones de estos tres vuelos independientes llevadas a cabo como tres acciones. Supongamos ahora que se han reservado los dos primeros vuelos, pero que para el tercero no hay boletos. La transacción aborta y los resultados de las dos primeras acciones no se registran; la base de datos de la línea aérea regresa a su estado antes de iniciar la transacción [(véase la figura 3-16(b)]. Es como si nada hubiera pasado.

```
BEGIN_TRANSACTION
    reserve WP-JFK;
    reserve JFK-Nairobi;
    reserve Nairobi-Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
    reserve WP-JFK;
    reserve JFK-Nairobi;
    Nairobi-Malindi full => ABORT_TRANSACTION;
```

(b)

Figura 3-16. (a) Transacción para reservar tres compromisos de vuelo. (b) La transacción aborta cuando el tercer vuelo no está disponible.

Propiedades de las transacciones

Las transacciones tienen cuatro propiedades fundamentales. Las transacciones son:

1. Atómicas: Para el mundo exterior, la transacción ocurre de manera indivisible.
2. Consistentes: La transacción no viola los invariantes del sistema.
3. Aisladas: Las transacciones concurrentes no interfieren entre sí.
4. Durables: Una vez comprometida una transacción, los cambios son permanentes.

Utilizamos las siglas **ACID** (las iniciales en inglés) para referirnos a estas propiedades.

La primera propiedad fundamental de todas las transacciones es que son **atómicas**. Esta propiedad garantiza que cada transacción no ocurre o bien, se realiza en su totalidad; en este segundo caso, se presenta como acción instantánea e indivisible. Mientras se desarrolla una transacción, otros procesos (ya sea que estén o no relacionados con las transacciones) no pueden ver los estados intermedios.

Supongamos, por ejemplo, que cierto archivo tiene una longitud de 10 bytes cuando una transacción comienza a añadirle datos. Si otros procesos leen el archivo mientras se lleva a cabo la transacción, sólo verán los 10 bytes originales, sin importar el número de bytes que haya agregado hasta ese momento la transacción. Si la transacción se realiza con éxito, el archivo crece de manera instantánea hasta su nuevo tamaño al momento de hacer el compromiso, sin estados intermedios y sin importar el número de operaciones realizadas para llegar a ese punto.

La segunda propiedad dice que son **consistentes**. Lo que esto significa es que si el sistema tiene ciertos invariantes que deben conservarse siempre, si éstos se conservan antes de la transacción, entonces también deben conservarse después de ellos. Por ejemplo, en un sistema bancario, un invariante fundamental es la conservación del dinero. Después de cualquier transferencia interna, la cantidad de dinero en el banco debe ser la misma que había antes de la transferencia, y durante un breve momento durante la misma, este invariante podría violarse. Sin embargo, la violación no es visible fuera de la transacción.

La tercera propiedad dice que las transacciones son **aisladas** o **serializables**. Lo que esto significa es que si dos o más transacciones se ejecutan al mismo tiempo, para cada una

de ellas y para los demás procesos, el resultado final aparece como si todas las transacciones se ejecutasen de manera secuencial en cierto orden (dependiente del sistema).

En la figura 3-17(a)-(c) tenemos tres procesos que ejecutan en forma simultánea tres transacciones. Si la ejecución fuese secuencial, el valor final de x sería 1, 2 o 3, según el proceso que se ejecutara al último (x podría ser una variable, un archivo o algún otro tipo de objeto compartido). En la figura 3-17(d) vemos varios ordenamientos, llamados **esquemas de planificación**, según los cuales se pueden intercalar los procesos. El esquema 1 es en realidad serializado. En otras palabras, las transacciones se ejecutan en forma estricta secuencial, de modo que satisfaga por definición la hipótesis de serialización. El esquema 2 no está serializado, pero es válido, puesto que produce un valor de x que se podría obtener al ejecutar las transacciones de manera estricta secuencial. El tercero no es válido, puesto que establece el valor de x en 5, algo que no se podría obtener con ningún orden secuencial de las transacciones. El sistema se encarga de garantizar que las operaciones individuales se intercalen en la forma correcta. Si se permite al sistema la libertad de elegir cualquier orden de las operaciones (siempre y cuando se obtenga la respuesta correcta), eliminamos la necesidad de que los programadores realicen su propia exclusión mutua, con lo cual se facilita la programación.

```
BEGIN_TRANSACTION
   x = 0;
   x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
   x = 0;
   x = x + 3;
END_TRANSACTION
```

(c)

Tiempo ⟶

Esquema 1	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = 0;	x = x + 3;	Válido
Esquema 2	x = 0;	x = 0;	x = x + 1;	x = x + 2;	x = 0;	x = x + 3;	Válido
Esquema 3	x = 0;	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = x + 3;	No válido

(d)

Figura 3-17. (a)–(c) Tres transacciones. (d) Esquemas posibles de planificación.

La cuarta propiedad dice que las transacciones son **durables**. Se refiere al hecho de que una vez comprometida una transacción, no importa lo que ocurra, la transacción sigue adelante y los resultados se vuelven permanentes. Ninguna falla después del compromiso puede deshacer los resultados o provocar la pérdida de los mismos.

Transacciones anidadas

Las transacciones pueden contener subtransacciones, a menudo llamadas **transacciones anidadas**. La transacción de nivel superior puede producir hijos que se ejecuten en

paralelo entre sí, en procesadores distintos, con el fin de mejorar el desempeño o hacer más sencilla la programación. Cada uno de estos hijos puede ejecutar una o más subtransacciones o bien producir sus propios hijos.

Las subtransacciones dan lugar a un problema sutil, pero importante. Imaginemos que una transacción inicia varias subtransacciones en paralelo y que una de ellas realiza un compromiso, lo cual hace que los resultados sean visibles para la transacción padre. Después de algunos cálculos, el padre aborta, lo que regresa todo el sistema al estado que tenía antes de iniciar la transacción de más alto nivel. En consecuencia, los resultados de la transacción comprometida también deben deshacerse. Así, la permanencia ya mencionada sólo se aplica a las transacciones del nivel más alto.

Puesto que las transacciones se pueden anidar con un nivel de profundidad arbitrario, es necesaria una considerable administración para que todo sea correcto. Sin embargo, la semántica es clara. Al iniciar una transacción o subtransacción, conceptualmente se le da una copia particular de todos los objetos del sistema, para que los maneje como desee. Si aborta, sólo se desvanece su universo particular, como si nunca hubiese existido. Si se compromete, su universo particular remplaza al del padre. Así, si una subtransacción se compromete y después inicia una nueva subtransacción, la segunda ve los resultados producidos por la primera.

3.4.3. Implantación

Las transacciones parecen una gran idea, pero ¿cómo se implantan? Ésa es la cuestión que trataremos en esta sección. Debe quedar claro por el momento que si cada proceso que ejecuta una transacción sólo actualiza los objetos utilizados (archivos, registros de bases de datos, etc.), entonces las transacciones no serán atómicas y los cambios no desaparecerán si la transacción aborta. Además, los resultados de la ejecución de varias transacciones no serán serializables. Es claro que se necesita otro método de implantación. Se utilizan comúnmente dos métodos, los cuales analizaremos a continuación.

Espacio de trabajo privado

Desde un punto de vista conceptual, cuando un proceso inicia una transacción, se le otorga un espacio de trabajo privado, el cual contiene todos los archivos (y otros objetos) a los cuales tiene acceso. Hasta que la transacción se comprometa o aborte, todas sus lecturas y escrituras irán al espacio de trabajo particular, en vez del espacio "real", donde éste último es el sistema de archivos normal. Esta observación conduce de manera directa al primer método de implantación: otorgar un espacio de trabajo particular a cada proceso en el momento en que inicie una transacción.

El problema con esta técnica es el costo prohibitivo de copiar todo a un espacio de trabajo particular, pero se pueden hacer ciertas optimizaciones. La primera de ellas se basa en el hecho de que, cuando un proceso lee un archivo, pero no lo modifica, no existe necesidad de una copia particular. Puede utilizar el archivo verdadero (a menos que haya sido modificado después del inicio de la transacción). En consecuencia, cuando un proceso inicia una transacción, basta crear un espacio de trabajo particular para él que sea vacío,

excepto por un apuntador de regreso al espacio de trabajo de su padre. Cuando la transacción se encuentra en el nivel superior, el espacio de trabajo del padre es el sistema de archivos "real". Cuando el proceso abre un archivo para su lectura, se rastrean los apuntadores hasta localizar el archivo en el espacio de trabajo del padre (o algún otro de sus antecesores).

Cuando se abre un archivo para la escritura, se puede localizar de la misma manera que en el caso de la lectura, excepto que ahora se copia en primer lugar al espacio de trabajo particular. Sin embargo, una segunda optimización elimina la mayoría del copiado, incluso en este caso. En vez de copiar todo el archivo, sólo se copia el índice del archivo en el espacio de trabajo particular. El índice es el bloque de datos asociado a cada archivo, en el cual se indica la localización de sus bloques en el disco. En UNIX, el índice es el nodo-i. Por medio del índice particular se puede leer el archivo de la manera usual, puesto que las direcciones en disco contenidas en él son las correspondientes a los bloques originales en disco. Sin embargo, cuando un bloque de un archivo se modifica por primera vez, se hace una copia del bloque y la dirección de la copia se inserta en el índice, como se muestra en la figura 3-18. El bloque se puede actualizar entonces sin afectar al original. También se manejan de esta forma los bloques añadidos. Los nuevos bloques reciben a menudo el nombre de **bloques sombra**.

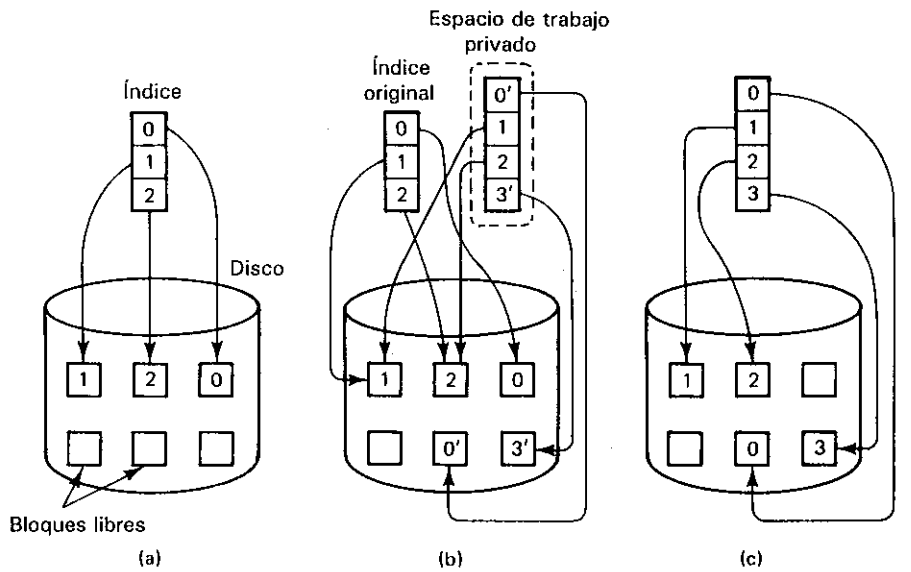


Figura 3-18. (a) El índice de archivo y los bloques de disco para un archivo de tres bloques. (b) La situación después de que una transacción ha modificado el bloque 0 y añadido el bloque 3. (c) Después del registro.

Como se puede ver en la figura 3-18(b), el proceso que ejecuta la transacción ve el archivo modificado, pero todos los demás procesos ven el archivo original. En una trans-

acción más compleja, el espacio de trabajo particular podría contener gran número de archivos, en vez de uno. Si la transacción aborta, el espacio de trabajo particular sólo se elimina y todos los bloques particulares a los cuales apunta se colocan de nuevo en la lista de bloques libres. Si la transacción se compromete, los índices particulares se desplazan al espacio de trabajo del padre de manera atómica, como se muestra en la figura 3-18(c). Los bloques que no son alcanzables se colocan en la lista de bloques libres.

Bitácora de escritura anticipada

El otro método común para implantar las transacciones es la **bitácora de escritura anticipada**, a veces conocida como la **lista de intenciones**. Con este método, los archivos en realidad se modifican, pero antes de cambiar cualquier bloque, se escribe un registro en la bitácora de escritura anticipada en un espacio de almacenamiento estable para indicar la transacción que realiza el cambio, el archivo y bloque modificados y los valores anterior y nuevo. Sólo después de que se puede escribir en la bitácora se realiza el cambio en el archivo.

La figura 3-19 da un ejemplo del funcionamiento de la bitácora. En la figura 3-19(a) tenemos una transacción sencilla, que utiliza dos variables (u otros objetos) compartidas, *x* y *y*, con valores iniciales 0. Para cada uno de los tres enunciados dentro de la transacción, se escribe un registro en la bitácora antes de ejecutar el enunciado que proporciona los valores anterior y actual, separados por una diagonal.

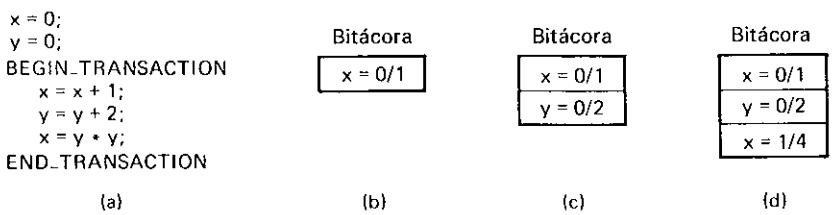


Figura 3-19. (a) Una transacción. (b)–(d) La bitácora antes de ejecutar cada enunciado.

Si la transacción tiene éxito y se establece un compromiso, se escribe un registro de este último en la bitácora, pero las estructuras de datos no tienen que modificarse, puesto que ya han sido actualizadas. Si la transacción aborta, se puede utilizar la bitácora para respaldar el estado original. A partir del final y hacia atrás, se lee cada registro de la bitácora y se deshace cada cambio descrito en él. Esta acción se llama **retroalimentación**.

La bitácora también se puede utilizar para la recuperación de las fallas. Supongamos que el proceso que realiza la transacción falla justo antes de escribir el último registro de bitácora de la figura 3-19(d), pero antes de modificar *x*. Después de volver a arrancar esa máquina, se verifica la bitácora para revisar las transacciones que se encontraban en proceso al momento de la falla. Cuando se lee el último registro y se ve que el valor actual de *x* es 1, es claro que la falla ocurrió *antes* de hacer la actualización, por lo que *x* toma el valor 4.

Si, por otro lado, x es 4 al momento de la recuperación, también es claro que el fallo ocurrió *después* de la actualización, por lo que no hay que realizar modificaciones. Por medio de la bitácora, es posible ir hacia adelante (realizar la transacción) o hacia atrás (deshacer la transacción).

Protocolo de compromiso de dos fases

Como lo hemos señalado en repetidas ocasiones, la acción de establecer un compromiso con una transacción debe llevarse a cabo de manera atómica; es decir, de forma instantánea e indivisible. En un sistema distribuido, el compromiso puede necesitar la cooperación de varios procesos en distintas máquinas, cada uno de los cuales contenga algunas de las variables, archivos y bases de datos y otros objetos modificados por la transacción. En esta sección estudiaremos un protocolo para lograr un compromiso atómico en un sistema distribuido.

El protocolo que analizaremos es el **protocolo de compromiso de dos fases** (Gray, 1978). Aunque no es el único protocolo de su tipo, es tal vez el más utilizado. La idea fundamental se muestra en la figura 3-20. Uno de los procesos que intervienen en este caso funciona como el coordinador. Por lo general, éste es quien ejecuta la transacción. El protocolo de compromiso comienza cuando el coordinador escribe una entrada en la bitácora para indicar que inicia dicho protocolo, seguido del envío de un mensaje a cada uno de los procesos implicados (subordinados) para que estén listos para el compromiso.

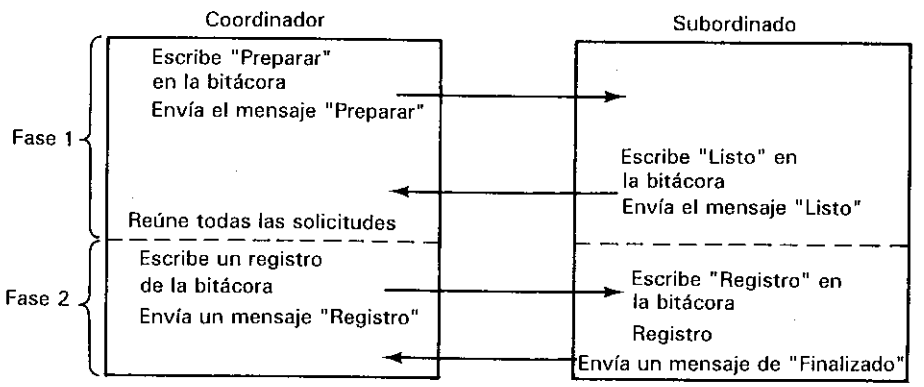


Figura 3-20. El protocolo de compromiso de dos fases cuando tiene éxito.

Cuando un subordinado recibe el mensaje, verifica si está listo para comprometerse, escribe una entrada en la bitácora y envía de regreso su decisión. Cuando el coordinador ha recibido todas las respuestas, sabe si establece el compromiso o aborta. Si todos los procesos están listos para comprometerse, entonces se cierra la transacción. Si uno o más procesos no se comprometen (o no responden), la transacción se aborta. De cualquier modo, el coordinador escribe una entrada en la bitácora y envía entonces un mensaje a cada subor-

dinado para informarle de la decisión. Es esta escritura en la bitácora la que en realidad cierra la transacción y hace que proceda sin importar ya lo ocurrido antes.

Debido al uso de la bitácora en el espacio de almacenamiento estable, este protocolo resiente mucho la presencia de (varias) fallas. Si el coordinador falla después de escribir el registro inicial en la bitácora, una vez recuperado podrá continuar a partir del punto en que se quedó, repitiendo el mensaje inicial en caso necesario. Si falla después de escribir el resultado de la votación en la bitácora, una vez recuperado puede volver a informar a todos los subordinados del resultado. Si un subordinado falla antes de responder al primer mensaje, el coordinador seguirá enviando mensajes hasta darse por vencido. Si falla después, puede revisar en la bitácora el lugar donde se encontraba y ver qué es lo que debe hacer.

3.4.4. Control de concurrencia

Cuando se ejecutan varias transacciones de manera simultánea en distintos procesos (o distintos procesadores), se necesita cierto mecanismo para mantener a cada uno lejos del camino del otro. Este mecanismo se llama **algoritmo de control de concurrencia**. En esta sección analizaremos tres algoritmos distintos.

Cerradura

El algoritmo más antiguo y de más amplio uso es la **cerradura**. En su forma más sencilla, cuando un proceso necesita leer o escribir en un archivo (u otro objeto) como parte de una transacción, primero cierra el archivo. La cerradura se puede hacer mediante un controlador centralizado de cerraduras, o bien con un controlador local de cerraduras en cada máquina, que maneje los archivos locales. En ambos casos, el controlador de cerraduras mantiene una lista de los archivos cerrados y rechaza todos los intentos de otro proceso por cerrarlos. Puesto que los procesos bien comportados no intentan tener acceso a un archivo antes de ser cerrado, el establecimiento de una cerradura en un archivo mantiene a todos lejos de éste, lo cual garantiza que no será modificado durante la transacción. El sistema de transacciones es el que por lo general adquiere y libera las cerraduras y no necesita acción alguna por parte del programador.

Este esquema básico es muy restrictivo y se puede mejorar al distinguir las cerraduras para lectura de las cerraduras para escritura. Si se establece una cerradura para lectura en cierto archivo, se permiten otras cerraduras para lectura. Este tipo de cerraduras se establecen para garantizar que el archivo no tendrá cambio alguno (es decir, se excluyen todos los escritores), pero no existe razón para prohibir otras transacciones a partir de la lectura del archivo. En contraste, cuando se cierra un archivo con respecto a la escritura, no se permiten cerraduras de otro tipo. Así, las cerraduras para lectura se comparten, pero las cerraduras para escritura deben ser exclusivas.

Para hacer más sencilla nuestra exposición, hemos supuesto que la unidad de cerradura es todo el archivo. En la práctica, podría ser un elemento más pequeño, como un registro o una página individual, o bien un elemento mayor, como toda una base de datos. El aspecto

relativo al tamaño del elemento por cerrar se llama la **granularidad de la cerradura**. Mientras más fina sea la granularidad, puede ser más precisa la cerradura y lograr un mayor paralelismo (por ejemplo, no bloquear un proceso que desee utilizar el final de un archivo sólo porque otro proceso esté utilizando el principio). Por otro lado; la cerradura de grano fino necesita un número mayor de cerraduras, es más cara y es más probable que ocurran bloqueos.

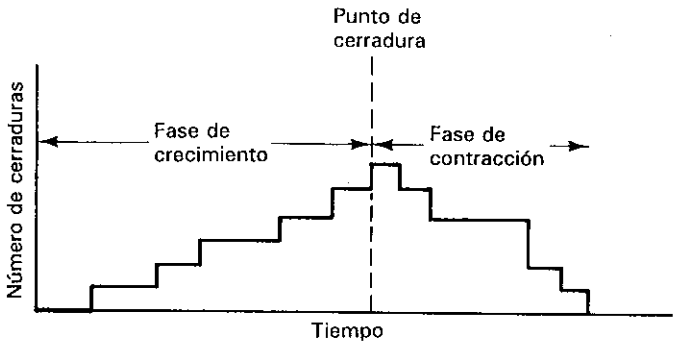


Figura 3-21. Cerradura de dos fases.

La adquisición y liberación de las cerraduras en el preciso momento en que se necesiten o se dejen de necesitar puede conducir a cierta inconsistencia y a bloqueos. En vez de esto, la mayoría de las transacciones que se implantan mediante cerraduras utilizan la llamada **cerradura de dos fases**. En este caso, que se muestra en la figura 3-21, el proceso adquiere en primer lugar todas las cerraduras necesarias durante la **fase de crecimiento** y después las libera en la **fase de contracción**. Si el proceso se abstiene de actualizar todos los archivos hasta que pasa a la segunda fase, entonces el problema de no poder adquirir una cerradura se puede resolver tan sólo con liberar todas las cerraduras, esperar un poco y comenzar de nuevo. Además, se puede demostrar (Eswaran *et al.*, 1976) que si todas las transacciones utilizan la cerradura de dos fases, entonces todos los esquemas de planificación formados mediante intercalación son serializables. Ésta es la razón del uso tan amplio de la cerradura de dos fases.

En muchos sistemas, la fase de contracción no se realiza sino hasta que la transacción ha terminado su ejecución y se ha comprometido o abortado. Esta política, llamada **cerradura estricta de dos fase**, tiene dos ventajas principales. La primera es que una transacción siempre lee un valor escrito por una comprometida; por lo tanto, nunca tiene que abortar una transacción debido a que sus cálculos se basaron en un archivo que no debía haber visto. En segundo lugar, la adquisición y liberación de cerraduras pueden controlarse mediante el sistema, sin que la transacción tenga conciencia de ello; las cerraduras se adquieren siempre que se necesite el acceso a un archivo y se liberan cuando termina la transacción. Esta política elimina los **abortos en cascada**, es decir, el hecho de tener que deshacer una transacción comprometida debido a que se ha visto un archivo que no debía verse.

La cerradura, e incluso la cerradura de dos fases, puede provocar bloqueos. Si dos procesos intentan adquirir la misma pareja de cerraduras, pero en el orden opuesto, puede

ocurrir un bloqueo. Aquí podemos aplicar las técnicas usuales, como la adquisición de todas las cerraduras en cierto orden canónico, para evitar ciclos tipo detenerse y esperar. También se pueden detectar los bloqueos mediante el mantenimiento de una gráfica explícita con los procesos que tienen cerraduras y los procesos que desean cerraduras, para entonces verificar que la gráfica no tenga ciclos. Por último, cuando se sabe de antemano que una cerradura no se puede mantener durante más de T segundos, se puede utilizar un esquema con tiempos de espera: si una cerradura permanece bajo el mismo propietario durante más de T segundos, debe existir un bloqueo.

Control optimista de la concurrencia

Un segundo método para el manejo de varias transacciones al mismo tiempo es el **control optimista de la concurrencia** (Kung y Robinson, 1981). La idea detrás de esta técnica sorprende por sencilla: sólo se continúa y se hace todo lo que se deba llevar a cabo, sin prestar atención a lo que hacen los demás. Si existe un problema, hay que preocuparse por él después. (También muchos políticos utilizan este algoritmo.) En la práctica, los conflictos son sumamente raros, por lo que la mayoría del tiempo todo funciona muy bien.

Aunque los conflictos pueden ser raros, no son imposibles, por lo que se necesita manejarlos de alguna forma. Lo que hace el control optimista de la concurrencia es mantener un registro de los archivos leídos o en los que se ha escrito algo. En el momento del compromiso, se verifican todas las demás transacciones para ver si alguno de los archivos ha sido modificado desde el inicio de la transacción. Si esto ocurre, la transacción aborta. Si no, se realiza el compromiso.

El control optimista de la concurrencia se ajusta mejor a la implantación con base en espacios de trabajo particulares. De esa manera, cada transacción modifica sus archivos en forma privada, sin interferencia de los demás. Al final, los nuevos archivos quedan comprometidos o liberados.

Las grandes ventajas del control optimista de la concurrencia son la ausencia de bloqueos y que permite un paralelismo máximo, puesto que ningún proceso tiene que esperar una cerradura. La desventaja es que a veces puede fallar, en cuyo caso la transacción tiene que ejecutarse de nuevo. En condiciones de carga pesada, la probabilidad de fallas puede crecer de manera sustancial, lo que convierte al control optimista de la concurrencia en una mala opción.

Marcas de tiempo

Un método por completo distinto al control de la concurrencia consiste en asociar a cada transacción una marca de tiempo, al momento en que realiza BEGIN_TRANSACTION (Reed, 1983). Mediante el algoritmo de Lamport, podemos garantizar que las marcas son únicas, lo cual es importante en este caso. Cada archivo del sistema tiene asociadas una marca de tiempo para la lectura y otra para la escritura, las cuales indican la última transacción comprometida que realizó la lectura o la escritura, respectivamente. Si las transac-

ciones son breves y muy espaciadas con respecto del tiempo, entonces ocurrirá de manera natural que cuando un proceso intente tener acceso a un archivo, las marcas de tiempo de lectura y escritura sean menores (más antiguas) que la marca de la transacción activa. Este orden quiere decir que las transacciones se procesan en el orden adecuado, por lo que todo funciona bien.

Cuando el orden es incorrecto, esto indica que una transacción iniciada posteriormente a la transacción activa ha intentado entrar al archivo, tenido acceso a éste y ha realizado un compromiso. Esta situación indica que la transacción activa se ha realizado tarde, por lo que se aborta. En cierto sentido, este mecanismo también es optimista, como el de Kung y Robinson, aunque los detalles son un poco distintos. En el método de Kung y Robinson, esperamos que las transacciones concurrentes no utilicen los mismos archivos. En el método de las marcas, no nos preocupa que las transacciones concurrentes utilicen los mismos archivos, siempre que la transacción con el número más pequeño esté en primer lugar.

Es fácil explicar el método de las marcas de tiempo mediante un ejemplo. Imaginemos que existen tres transacciones, alfa, beta y gamma. Alfa se ejecutó hace tiempo y utilizó todos los archivos necesarios para beta y gamma, de modo que sus archivos tienen marcas de tiempo para lectura y escritura, asociadas a la marca de tiempo de alfa. Beta y gamma inician su ejecución en forma concurrente, donde beta tiene una marca menor que gamma (pero mayor que alfa, por supuesto).

Consideremos ahora lo que ocurre cuando alfa escribe un archivo. T es su marca de tiempo y las marcas de lectura y escritura son T_{RD} y T_{WR}, respectivamente. A menos que gamma se haya comprometido, tanto T_{RD} como T_{WR} tendrán la marca de tiempo de alfa, que por tanto será menor que T. En la figura 3-22(a) y (b) vemos que T es mayor que T_{RD} y T_{WR} (gamma no se ha comprometido), de modo que se acepta y realiza de manera tentativa la escritura. Será permanente cuando beta se comprometa. La marca de tiempo de beta se registra entonces en el archivo como una escritura tentativa.

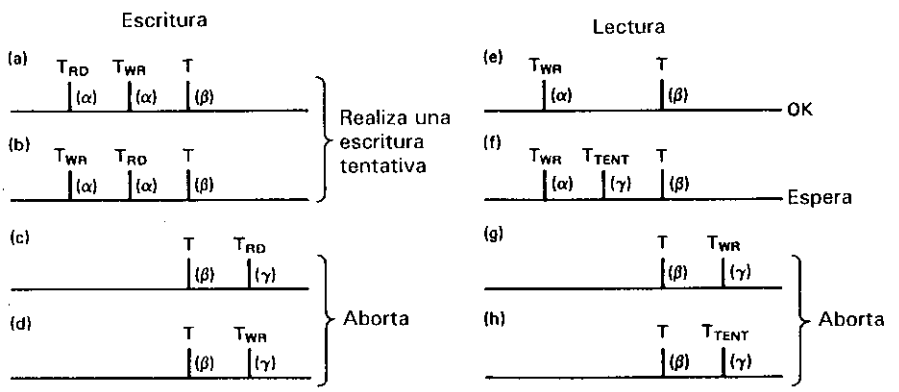


Figura 3-22. Control de concurrencia mediante marcas de tiempo.

En la figura 3-22(c) y (d), beta no tiene suerte. Gamma ha leído (c) o escrito (d) en el archivo y se ha comprometido. La transacción de beta aborta. Sin embargo, puede solicitar una nueva marca de tiempo y comenzar de nuevo.

Analizaremos ahora las lecturas. En la figura 3-22(e), no existe conflicto, de modo que la lectura se puede llevar a cabo de manera inmediata. En la figura 3-22(f), existe cierto intruso que intenta leer el archivo. La marca del intruso es menor que la de beta, por lo que beta sólo espera hasta que el intruso se comprometa, momento en el cual puede leer el nuevo archivo y continuar.

En la figura 3-22(g), gamma ha modificado el archivo y ya se ha comprometido. De nuevo, beta debe abortar. En la figura 3-22(h), gamma está en el proceso de modificar el archivo, aunque no se ha comprometido todavía. Aún así, beta está demasiado atrasado y debe abortar.

Las marcas de tiempo tienen propiedades distintas a las cerraduras. Cuando una transacción encuentra una marca mayor (posterior), aborta, mientras que con las cerraduras, en las mismas circunstancias, podría esperar o poder continuar de manera inmediata. Por otro lado, es libre de bloqueos, que es un gran punto a su favor.

En resumen, todas las transacciones ofrecen distintas ventajas y son una técnica promisoria para la construcción de sistemas distribuidos confiables. Su problema principal es la enorme complejidad de su implantación, lo que provoca un bajo desempeño. Se trabaja en estos problemas y es probable que sean resueltos en un corto plazo.

3.5. BLOQUEOS EN SISTEMAS DISTRIBUIDOS

Los bloqueos en los sistemas distribuidos son similares a los bloqueos en los sistemas con un procesador, sólo que peores. Son más difíciles de evitar, prevenir e incluso detectar, además de ser más difíciles de curar cuando se les sigue la pista, puesto que toda la información relevante está dispersa en muchas máquinas. En ciertos sistemas, como los sistemas distribuidos de bases de datos, los bloqueos pueden ser en extremo serios, por lo que es importante comprender sus diferencias con los bloqueos ordinarios y lo que se puede hacer con ellos.

Ciertas personas hacen una distinción entre dos tipos de bloqueos distribuidos: bloqueos de comunicación y bloqueos de recursos. Un ejemplo de bloqueo de comunicación es el que aparece cuando el proceso *A* intenta enviar un mensaje al proceso *B*, el cual a su vez intenta enviar uno al proceso *C*, el cual intenta enviar uno a *A*. Existen diversas circunstancias en las que esta situación puede conducir a un bloqueo, como en donde no se dispone de buffers. Un bloqueo de recursos ocurre cuando los procesos pelean por el acceso exclusivo a los dispositivos de E/S, archivos, cerraduras u otro tipo de recursos.

Aquí no haremos esa distinción, puesto que los canales de comunicación, buffers, etc. también son recursos y se pueden modelar como bloqueos de recursos, puesto que los procesos pueden solicitarlos y liberarlos. Además, los patrones de comunicación circular del tipo ya descrito son algo raros en la mayoría de los sistemas. Por ejemplo, en los sistemas cliente-servidor, un cliente podría enviar un mensaje (o llevar a cabo una RPC) con un servidor de archivos, el cual podría enviar un mensaje a un servidor de discos. Sin embargo, no es probable que el servidor de discos, que actúa como un cliente, envíe un mensaje al cliente original, en espera de que actúe como un servidor. Así, es poco probable que la condición para la espera circular se cumpla como resultado de la simple comunicación.

Se utilizan varias estrategias para el manejo de los bloqueos. Cuatro de las más conocidas se enumeran y analizan a continuación.

1. El algoritmo del avestruz (ignorar el problema).
2. Detección (permitir que ocurran los bloqueos, detectarlos e intentar recuperarse de ellos).
3. Prevención (lograr estáticamente que los bloqueos sean imposibles desde el punto de vista estructural).
4. Evitarlos (evitar los bloqueos mediante la asignación cuidadosa de los recursos).

Las cuatro estrategias son aplicables de manera potencial a los sistemas distribuidos. El algoritmo del avestruz es tan bueno y tan popular en los sistemas distribuidos como en los sistemas con un procesador. En los sistemas distribuidos que se utilizan para la programación, automatización de oficinas, control de procesos y muchas otras aplicaciones, no existe un mecanismo de bloqueo en todo el sistema, aunque las aplicaciones individuales, como las bases de datos distribuidas, pueden implantar los suyos propios si lo necesitan.

La detección y recuperación de los bloqueos también es popular, principalmente porque es muy difícil prevenirlos y evitarlos. Más adelante analizaremos varios algoritmos para la detección de bloqueos.

También es posible prevenir los bloqueos, aunque es más difícil que en los sistemas con un procesador. Sin embargo, si se cuenta con las transacciones atómicas, existen ciertas opciones disponibles. Adelante analizaremos dos algoritmos.

Por último, en los sistemas distribuidos nunca se evitan los bloqueos. Ni siquiera en el caso de los sistemas con un procesador, por lo que no existe razón para su uso en el caso más difícil de los sistemas distribuidos. El problema es que el algoritmo del banquero u otros algoritmos similares necesitan conocer (de antemano) la proporción de cada recurso que necesitará cada proceso. Sin embargo, es muy raro disponer de esta información, si es que existe. Así, nuestro análisis de los bloqueos en los sistemas distribuidos se centrará sólo en dos de las técnicas: detección y prevención.

3.5.1. Detección distribuida de bloqueos

El descubrimiento de métodos generales para prevenir o evitar bloqueos distribuidos parece un tanto difícil, por lo que muchos investigadores han intentado resolver el problema más sencillo consistente en la detección de bloqueos, en vez de tratar de inhibir su aparición.

Sin embargo, la presencia de las transacciones atómicas en ciertos sistemas distribuidos es una diferencia conceptual fundamental. Cuando se detecta un bloqueo en un sistema operativo convencional, la forma de resolverlo es eliminar uno o más procesos. Esto produce uno o varios usuarios infelices. Cuando se detecta un bloqueo en un sistema basado en transacciones atómicas, se resuelve abortando una o más transacciones. Pero como vimos antes en detalle, las transacciones se han diseñado para soportar ser abortadas. Cuando se aborta una transacción por contribuir a un bloqueo, el sistema restaura en primer lugar el

estado que tenía antes de iniciar la transacción, punto en el cual puede comenzar de nuevo la misma. Con un poco de suerte, tendrá éxito la segunda vez. Así, la diferencia es que las consecuencias de la eliminación de un proceso son mucho menos severas si se utilizan las transacciones que en caso de que no se utilicen.

Detección centralizada de bloqueos

Como primer intento, podemos utilizar un algoritmo centralizado para la detección de bloqueos y tratar de imitar al algoritmo no distribuido. Aunque cada máquina mantiene la gráfica de recursos de sus propios procesos y recursos, un coordinador central mantiene la gráfica de recursos de todo el sistema (la unión de todas las gráficas individuales). Cuando el coordinador detecta un ciclo, elimina uno de los procesos para romper el bloqueo.

A diferencia del caso centralizado, donde se dispone de toda la información de manera automática en el lugar correcto, en un sistema distribuido esta información se debe enviar de manera explícita. Cada máquina mantiene la gráfica de sus propios procesos y recursos. Existen varias posibilidades para llegar ahí. En primer lugar, siempre que se añada o elimine un arco a la gráfica de recursos, se puede enviar un mensaje al coordinador para informar de la actualización. En segundo lugar, cada proceso puede enviar de manera periódica una lista de los arcos añadidos o eliminados desde la última actualización. Este método necesita un número menor de mensajes que el primero. En tercer lugar, el coordinador puede pedir la información cuando la necesite.

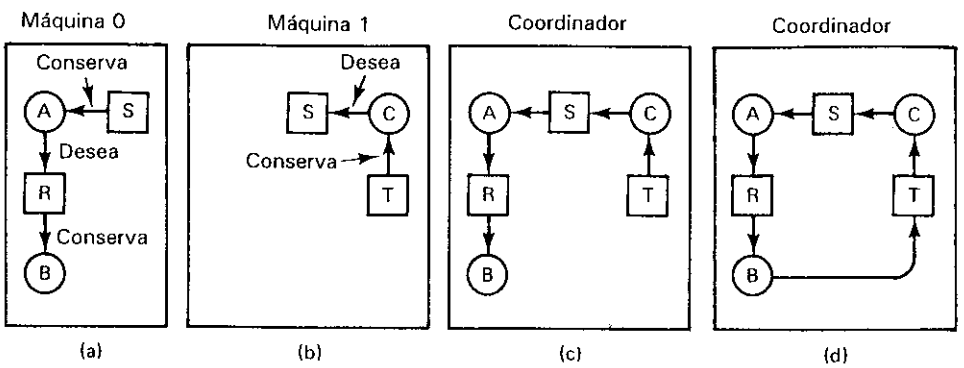


Figura 3-23. (a) Gráfica de recursos iniciales para la máquina 0. (b) Gráfica de recursos iniciales para la máquina 1. (c) Visión del mundo por parte del coordinador. (d) La situación después del mensaje retrasado.

Por desgracia, ninguno de estos métodos funciona bien. Consideremos un sistema donde los procesos *A* y *B* se ejecutan en la máquina 0 y el proceso *C* se ejecuta en la máquina 1. Existen tres recursos, *R*, *S* y *T*. En un principio, la situación es como se muestra en la figura 3-23(a) y (b): *A* conserva a *S* pero desea a *R*, que no puede tener debido a que *B* lo utiliza. C tiene a *T* y también desea a *S*. La visión desde la perspectiva del coordinador es como se

muestra en la figura 3-23(c). Esta configuración es segura. Tan pronto termina *B*, *A* puede obtener *R* y terminar, con lo que libera *S* para *C*.

Después de un rato, *B* libera *R* y pide *T*, un intercambio por completo válido y seguro. La máquina 0 envía un mensaje al coordinador para avisarle de la liberación de *R* y la máquina 1 envía un mensaje al coordinador para anunciar el hecho de que *B* espera su recurso *T*. Por desgracia, el mensaje de la máquina 1 llega en primer lugar, lo que hace que el coordinador construya la gráfica de la figura 3-23(d). Con ella, el coordinador concluye, de manera errónea, que existe un bloqueo y elimina un proceso. Tal situación recibe el nombre de **falso bloqueo**. Muchos de los algoritmos de bloqueos en los sistemas distribuidos producen este tipo de falsos bloqueos debido a información incompleta o con retraso.

Una posible solución es utilizar el algoritmo de Lamport para disponer de un tiempo global. Puesto que el mensaje de la máquina 1 al coordinador es activado por la solicitud de la máquina 0, el mensaje de la máquina 1 al coordinador tendrá una marca de tiempo posterior a la del mensaje de la máquina 0 al coordinador. Cuando el coordinador obtiene el mensaje de la máquina 1 que lo hace sospechar de un bloqueo, podría enviar el siguiente mensaje a todas las máquinas del sistema: "Acabo de recibir un mensaje con marca *T* que conduce a un bloqueo. Si alguien tiene un mensaje para mí con una marca anterior, haga favor de enviármelo de manera inmediata." Cuando cada máquina conteste, en forma afirmativa o negativa, el coordinador verá que el arco de *R* a *B* ha desaparecido, por lo que el sistema seguirá siendo seguro. Aunque este método elimina el falso bloqueo, necesita un tiempo global y es caro. Además, pueden existir otras situaciones donde sea más difícil eliminar el falso bloqueo.

Detección distribuida de bloqueos

Se han publicado muchos algoritmos para la detección distribuida de bloqueos. Existen reseñas del tema en Knapp (1987) y Singhal (1989). Examinaremos aquí un algoritmo típico, el algoritmo de Chandy-Misra-Haas (Chandy *et al.*, 1983). En este algoritmo, se permite que los procesos soliciten varios recursos (por ejemplo, cerraduras) al mismo tiempo, en vez de uno cada vez. Al permitir las solicitudes simultáneas de varios procesos, la fase de crecimiento de una transacción se puede realizar más rápido. La consecuencia de este cambio al modelo es que un proceso puede esperar ahora a dos o más recursos en forma simultánea.

En la figura 3-24 presentamos una gráfica de recursos modificada, donde sólo se muestran los procesos. Cada arco pasa a través de un recurso, como es usual, sólo que para hacer más sencilla la figura hemos omitido los recursos. Observemos que el proceso 3 de la máquina 1 espera dos recursos, uno de los cuales lo tiene el proceso 4 y el otro lo tiene el proceso 5.

Algunos de los procesos esperan recursos locales, como el proceso 1; pero otros, como el proceso 2, esperan recursos localizados en una máquina distinta. Precisamente estos arcos a través de las máquinas son los que dificultan la búsqueda de ciclos. El algoritmo Chandy-Misra-Haas se utiliza cuando un proceso debe esperar cierto recurso; por ejemplo,

el proceso 0 se bloquea debido al proceso 1. En este momento, se genera un mensaje especial **de exploración**, el cual se envía al proceso (o procesos) que detienen los recursos necesarios. El mensaje consta de tres números: el proceso recién bloqueado, el proceso que envía el mensaje y el proceso al cual se envía. El mensaje inicial de 0 a 1 contiene la tercia (0 0 1).

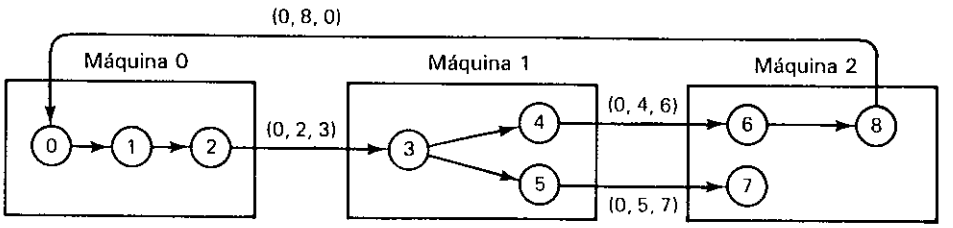


Figura 3-24. El algoritmo Chandy-Misra-Haas para la detección distribuida de bloqueos.

Al llegar el mensaje, el receptor verifica si él mismo espera a algunos procesos. En este caso, el mensaje se actualiza, conservando el primer campo pero reemplazando el segundo por su propio número de proceso y el tercero por el número del proceso al cual espera. El mensaje se envía entonces al proceso debido al cual se bloquea. Si se bloquea debido a varios procesos, a todos ellos se les envían mensajes (diferentes). Este algoritmo se lleva a cabo sin importar si el recurso es local o remoto. En la figura 3-24 vemos los mensajes remotos (0,2,3), (0,4,6), (0,5,7) y (0,8,0). Si un mensaje recorre todo el camino y regresa a su emisor original, es decir, al proceso enumerado en el primer campo, existe un ciclo y el sistema está bloqueado.

Existen varias formas de romper un bloqueo. Una forma es que el proceso que inició la exploración se comprometa a suicidarse. Sin embargo, este método tiene problemas si varios procesos llaman en forma simultánea al algoritmo. Por ejemplo, en la figura 3-24, imaginemos que 0 y 6 se bloquean al mismo tiempo y que ambos inician exploraciones. Cada uno de ellos descubriría el bloqueo y se eliminaría a sí mismo. Esto es demasiada eliminación. Con que uno desaparezca es suficiente.

Un algoritmo alternativo consiste en que cada proceso añada su identidad al final del mensaje de exploración, de modo que cuando regrese al emisor inicial, se enumere todo el ciclo. El emisor puede ver entonces cuál de los procesos tiene el número más grande y eliminarlo, o bien enviar un mensaje que le pida que se elimine a sí mismo. De cualquier forma, si varios procesos descubren el mismo ciclo al mismo tiempo, todos elegirán la misma víctima.

Existen pocas áreas en la ciencia de la computación donde la teoría y la práctica divergen tanto como en los algoritmos de detección distribuida de bloqueos. El descubrimiento de otro algoritmo para la detección de bloqueos es el objetivo de muchos investigadores. Por desgracia, estos métodos tienen a menudo poca relación con la realidad. Por ejemplo, algunos de los algoritmos necesitan que los procesos envíen exploraciones cuando están bloqueados. Sin embargo, no es trivial poder enviar un mensaje mientras se está bloqueado.

Muchos de los artículos en el área contienen elaborados análisis del desempeño del nuevo algoritmo, señalando por ejemplo, que mientras el nuevo algoritmo sólo necesita recorrer dos veces el ciclo, utiliza mensajes más breves y se ve si estos factores se equilibran de cierta manera. Sin duda, los autores se sorprenden con el hecho de que un mensaje "corto" típico (20 bytes) en una LAN tarda cerca de 1 milisegundo y un mensaje "largo" típico (100 bytes) en la misma LAN tarda 1.1 milisegundos. Tampoco existe duda alguna en que es un choque para estas personas el hecho de darse cuenta de que las mediciones experimentales han mostrado que el 90% de los ciclos de bloqueo ocurren en el caso de dos procesos (Gray *et al.*, 1981).

Lo peor de todo es que una enorme proporción de todos los algoritmos publicados en el área sólo son incorrectos, incluidos aquellos que han mostrado ser correctos Knapp (1987) y Singhal (1989) señalan algunos ejemplos. Ocurre con frecuencia que poco después de idearse un algoritmo, demostrar que es correcto y ser publicado, alguien encuentra un contraejemplo. Así, aquí tenemos un área de investigación activa en la que el modelo del problema no corresponde a la realidad, las soluciones que se determinan son por lo general poco prácticas, los análisis de desempeño carecen de sentido y los resultados demostrados son con frecuencia incorrectos. Para concluir con una observación positiva, ésta es un área que ofrece grandes oportunidades para su mejoramiento.

3.5.2. Prevención distribuida de bloqueos

La prevención de bloqueos consiste en el diseño cuidadoso del sistema, de modo que los bloqueos sean imposibles, desde el punto de vista estructural. Entre las distintas técnicas se incluye el permitir a los procesos que sólo conserven un recurso a la vez, exigir a los procesos que soliciten todos sus recursos desde un principio y hacer que liberen todos sus recursos cuando soliciten uno nuevo. Todo esto es difícil de manejar en la práctica. Un método que a veces funciona es ordenar todos los recursos y exigir a los procesos que los adquieran en orden estrictamente creciente. Este método significa que un proceso nunca puede conservar un recurso y pedir otro menor, por lo que son imposibles los bloqueos.

Sin embargo, en un sistema distribuido con tiempo global y transacciones atómicas, son posibles otros algoritmos prácticos. Ambos se basan en la idea de asociar a cada transacción una marca de tiempo global al momento de su inicio. Como en muchos de los algoritmos basados en las marcas de tiempo, en estos dos es esencial que ninguna pareja de transacciones tengan asociada la misma marca de tiempo. Como hemos visto, el algoritmo de Lamport garantiza la unicidad (mediante el uso de los números de proceso para evitar los empates).

La idea detrás de este algoritmo es que cuando un proceso está a punto de bloquearse en espera de un recurso que está utilizando otro proceso, se verifica cuál de ellos tiene la marca de tiempo mayor (es decir, es más joven). Podemos permitir entonces la espera sólo si el proceso en estado de espera tiene marca inferior (más antiguo) que el proceso esperado. De esta forma, al seguir cualquier cadena de procesos en espera, las marcas siempre aparecen en forma creciente, de modo que los ciclos son imposibles. Otra alternativa consiste en

permitir la espera de procesos sólo si éste tiene una marca mayor (es más joven) que el proceso esperado, en cuyo caso las marcas aparecen en la cadena de forma descendente.

Aunque ambos métodos previenen los bloqueos, es más sabio dar prioridad a los procesos más viejos. Se han ejecutado durante más tiempo, por lo que el sistema ha invertido mucho en ellos y es probable que conserven más recursos. Además, un proceso joven eliminado en esta etapa llegará en cierto momento al punto en que sea el más antiguo del sistema, de modo que esta opción elimina la inanición. Como hemos señalado antes, la eliminación de una transacción es relativamente inofensiva, ya que por definición puede volver a iniciar más tarde.

Para aclarar este algoritmo, consideremos la situación de la figura 3-25. En (a), un proceso antiguo desea un recurso que mantiene un proceso joven. En (b), un proceso joven desea un recurso que mantiene un proceso antiguo. En el primer caso debemos permitir al proceso que continúe; en el otro lo eliminamos. Supongamos que nombramos (a) *muerte* y (b) *espera*. Entonces eliminamos un proceso antiguo que intenta utilizar un recurso conservado por un proceso joven, lo cual es ineficiente. Así, los nombramos en el orden inverso, como se muestra en la figura. En estas condiciones, las flechas siempre apuntan en la dirección creciente de los números de transacción, lo que impide la existencia de ciclos. Este algoritmo se llama **espera-muerte**.

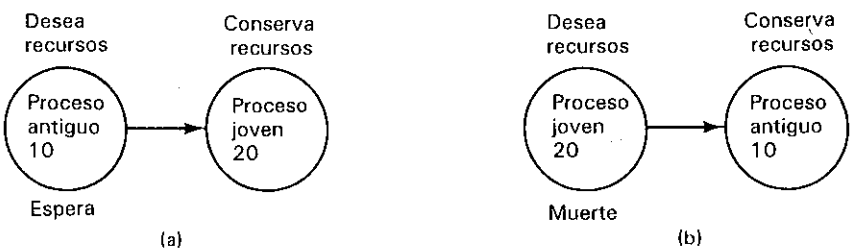


Figura 3-25. El algoritmo espera-muerte para la prevención de bloqueos.

Una vez que hemos supuesto la existencia de transacciones, podemos hacer algo que antes estaba prohibido: tomar los recursos de los procesos en ejecución. En los hechos, estamos diciendo que al surgir un conflicto, en vez de eliminar el proceso que hace la solicitud, podemos eliminar los procesos propietarios. Sin las transacciones, la eliminación de un proceso podría tener severas consecuencias, puesto que el proceso podría haber modificado ciertos archivos, por ejemplo. Con las transacciones, estos efectos se desvanecen en el aire si la transacción muere.

Consideremos ahora la situación de la figura 3-26, donde vamos a permitir las prioridades. Dado que nuestro sistema venera a los ancestros, como hemos analizado, no queremos que un joven mequetrefe tenga prioridad sobre un viejo y venerable sabio, de modo que la figura 3-26(a) y no la figura 3-26(b) recibe el nombre de *Derecho de prioridad.*

Ahora podemos llamar con seguridad a la figura 3-26(b) *espera*. Este algoritmo se llama **herida-espera**, puesto que se supone que una de las transacciones queda herida (en realidad se le elimina) y la otra espera. No es probable que este algoritmo sea un candidato para el Salón de la Fama de las Nomenclaturas.

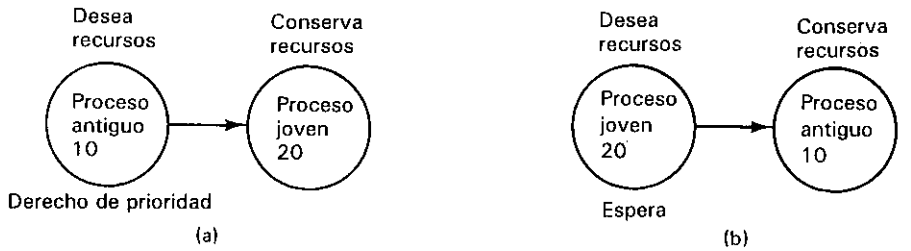


Figura 3-26. El algoritmo herida-espera para la prevención de bloqueos.

Si un proceso antiguo desea un recurso mantenido por uno joven, el proceso antiguo ejerce su derecho de prioridad sobre el joven, cuya transacción es eliminada, como se muestra en la figura 3-26(a). Es probable que el joven vuelva a iniciar su ejecución de manera inmediata, con lo que intentaría adquirir de nuevo el recurso; esto lleva a la situación de la figura 3-26(b), obligándolo a esperar. Compare este algoritmo con el de espera-muerte. En él, si un proceso antiguo desea un recurso conservado por un joven insolente, el antiguo espera cortésmente. Sin embargo, si el joven desea un recurso del antiguo, se elimina al joven. Sin duda, volverá a iniciar su ejecución y volverá a ser eliminado. Este ciclo puede continuar muchas veces hasta que el antiguo libere el recurso. La herida-espera no tiene esta desagradable propiedad.

3.6. RESUMEN

Este capítulo trató de la sincronización en los sistemas distribuidos. Comenzamos con el algoritmo de Lamport para la sincronización de relojes sin hacer referencia a fuentes externas de tiempo y posteriormente vimos la utilidad de este algoritmo. Vimos cómo se pueden utilizar los relojes físicos para la sincronización en los casos donde es importante el tiempo real.

A continuación analizamos la exclusión mutua en los sistemas distribuidos y estudiamos tres algoritmos. El centralizado mantiene toda la información en un lugar. El distribuido ejecutaba el cómputo en varios sitios, de manera paralela. El del anillo de fichas transfiere el control a lo largo del anillo. Cada uno tiene sus puntos fuertes y sus puntos débiles.

Muchos algoritmos distribuidos necesitan un coordinador, por lo que vimos dos formas para la elección de éste, el algoritmo del grandulón y otro algoritmo de anillo.

Aunque lo anterior es interesante e importante, todos son conceptos de bajo nivel. Las transacciones son un concepto de alto nivel que facilita a los programadores el manejo de

la exclusión mutua, las cerraduras, la tolerancia de fallas y los bloqueos en un sistema distribuido. Analizamos el modelo de transacción, la forma de implantar las transacciones y tres esquemas de control de concurrencia: cerradura, control optimista de la concurrencia y marcas de tiempo.

Por último, revisamos el problema de los bloqueos y vimos algunos algoritmos para su detección y prevención en los sistemas distribuidos.

PROBLEMAS

1. Añada un nuevo mensaje a la figura 3-2(b) que sea concurrente con el mensaje *A*; es decir, que no ocurra antes o después de *A*.

2. Cite al menos tres fuentes de retraso que se pueden presentar entre la transmisión de la hora por WWV y la puesta a tiempo de los relojes internos de los procesadores en un sistema distribuido.

3. Consideremos el comportamiento de dos máquinas en un sistema distribuido. Ambas poseen relojes, los cuales deben marcar 1 000 veces por cada milisegundo. Uno de ellos las hace, pero el otro sólo marca 900 veces por cada milisegundo. Si las actualizaciones de UTC se reciben una vez cada minuto, ¿cuál es la máxima desviación que puede ocurrir?

4. En el método de rentas para la consistencia del caché, ¿es en realidad esencial que los relojes estén sincronizados? En caso contrario, ¿qué es lo que se necesita?

5. En el método centralizado correspondiente a la exclusión mutua (figura 3-8), al recibir un mensaje de un proceso que libera su acceso exclusivo a la región crítica que había utilizado, el coordinador otorga el permiso generalmente al primer proceso de la fila. Dé otro posible algoritmo para uso del coordinador.

6. Consideremos de nuevo la figura 3-8. Supongamos que el coordinador falla. ¿Provoca esto siempre la falla del sistema? En caso contrarió, ¿bajo qué condiciones ocurre esto? ¿Existe alguna forma de evitar el problema y hacer que el sistema tolere las fallas del coordinador?

7. El algoritmo de Ricart y Agrawala tiene el problema de que si un proceso falla y no responde a la solicitud de otro proceso para entrar a una región crítica, la carencia de respuesta se interpreta como una negación del permiso. Sugerimos que todas las solicitudes se respondan de manera inmediata, para facilitar la detección de los procesos fallidos. ¿Existen circunstancias donde este método siga siendo insuficiente? Analice.

8. Un sistema distribuido puede tener varias regiones críticas independientes entre sí. Imaginemos que el proceso 0 desea entrar a la región crítica *A* y que el proceso 1 desea

entrar a la región crítica *B*. ¿Pueden ocurrir bloqueos con el algoritmo de Ricart y Agrawala? Explique su respuesta.

9. En la figura 3-12 es posible una pequeña optimización. ¿Cuál es?

10. Supongamos que dos procesos detectan la muerte del coordinador en forma simultánea y que ambos deciden hacer una elección mediante el algoritmo del grandulón. ¿Qué ocurre?

11. En la figura 3-13 tenemos dos mensajes *ELECCIÓN* que circulan al mismo tiempo. Aunque no causa daño alguno tener dos mensajes de este tipo, sería más elegante si uno de ellos se pudiera eliminar. Diseñe un algoritmo para esto, sin afectar la operación del algoritmo básico de elección.

12. En la figura 3-14 vimos una forma de actualizar en forma atómica una lista de inventario, mediante una cinta magnética. Puesto que una cinta se puede simular con facilidad como un archivo de un disco, ¿por qué cree usted que este método ya no se utiliza?

13. Para el caso de ciertas aplicaciones ultrasensibles, es concebible que no sea lo bastante confiable el almacenamiento estable implantado con dos discos. ¿Se puede ampliar la idea a tres discos? En tal caso, ¿cómo funcionaría? En caso contrario, ¿por qué?

14. En la figura 3-17(d) se muestran tres esquemas de planificación, dos válidos y otro no. Para las mismas transacciones, dé una lista completa de todos los valores finales posibles de x y establezca cuáles de ellos son válidos y cuáles no.

15. Si se utiliza un espacio de trabajo privado para la implantación de las transacciones, puede suceder que un gran número de índices de archivo se copien de regreso en el espacio de trabajo del padre. ¿Cómo se puede hacer esto sin conducir a condiciones de competencia?

16. En la bitácora de escritura anticipada, tanto el valor nuevo como el anterior se guardan en las entradas de la bitácora. ¿No es adecuado guardar solamente la entrada nueva? ¿Cuál es la utilidad de la entrada anterior?

17. En la figura 3-20, ¿en qué instante se alcanza el punto sin retorno? Es decir, ¿cuándo se lleva a cabo en realidad el compromiso atómico?

18. Dé el algoritmo completo para que un intento de cerrar un archivo tenga éxito o fracaso. Considere las cerraduras para lectura y escritura y la posibilidad de que el archivo no estuviera cerrado, que esté cerrado para lectura o que esté cerrado para escritura.

19. Los sistemas que utilizan las cerraduras para el control de concurrencia distinguen por lo general entre las cerraduras para lectura y las cerraduras para escritura. ¿Qué ocurre

si un proceso que ya ha adquirido una cerradura para lectura desea ahora cambiarla a una cerradura para escritura? ¿Qué ocurre con el cambio de una cerradura para escritura a una cerradura para lectura?

20. ¿Es más (o menos) restrictivo el control optimista de la concurrencia que el uso de las marcas de tiempo? ¿Por qué?

21. ¿Garantiza la serialización el uso de las marcas de tiempo para el control de la concurrencia? Analice.

22. Hemos dicho varias veces que si se aborta una transacción, el mundo regresa a su estado anterior, como si la transacción nunca hubiera ocurrido. Mentimos. Dé un ejemplo en donde la restauración del mundo es imposible.

23. El algoritmo centralizado para la detección de bloqueos, descrito en el texto, produce al principio un falso bloqueo, pero que después se arregla mediante el uso del tiempo global. Supongamos que se ha decidido no mantener el tiempo global (es demasiado caro). Diseñe una forma alternativa de arreglar el problema en el algoritmo.

24. Un proceso con marca de tiempo de la transacción igual a 50 necesita un recurso que mantiene un proceso con marca 100. ¿Qué ocurre si se utiliza en:

(a) ¿Espera-muerte?
(b) ¿Herida-espera?

4

Procesos y procesadores en sistemas distribuidos

En los dos capítulos anteriores analizamos dos temas relacionados entre sí: la comunicación y la sincronización en sistemas distribuidos. En este capítulo cambiaremos a un tema diferente: procesos. Aunque los procesos son un concepto importante en los sistemas con un procesador, en este capítulo daremos énfasis a los aspectos del manejo de procesos que a menudo no se estudian en el contexto de los sistemas operativos clásicos. En particular, veremos cómo se enfrenta la existencia de muchos procesadores.

En muchos sistemas distribuidos, es posible tener muchos hilos de control dentro de un proceso. Esta capacidad tiene algunas ventajas importantes, pero también introduce varios problemas. Primero estudiaremos estos aspectos. Después analizaremos el tema de la organización de los procesos y los procesadores y veremos que pueden existir varios modelos. Después analizaremos la asignación de procesadores y la planificación en los sistemas distribuidos. Por último, analizaremos dos tipos especiales de sistemas distribuidos, los sistemas tolerantes de fallas y los de tiempo real.

4.1. HILOS

En la mayoría de los sistemas operativos tradicionales, cada proceso tiene un espacio de direcciones y un hilo de control. De hecho, ésa es casi la definición de un proceso. Sin embargo, con frecuencia existen situaciones en donde se desea tener varios hilos de control que compartan un espacio de direcciones, pero que se ejecutan de manera cuasi-paralela, como si fuesen de hecho procesos independientes (excepto por el espacio de direcciones compartido). En esta sección analizaremos estas situaciones y sus implicaciones.

4.1.1. Introducción a los hilos

Consideremos, por ejemplo, un servidor de archivos que debe bloquearse en forma ocasional, en espera de acceso al disco. Si el servidor tiene varios hilos de control, se podría ejecutar un segundo hilo mientras el primero duerme. El resultado neto sería mejor rendimiento y desempeño. No es posible lograr este objetivo si se crean dos procesos servidores independientes, puesto que deben compartir un buffer caché común, lo que implica que deben estar en el mismo espacio de direcciones. Así, se necesita un nuevo mecanismo, uno que históricamente no se encontraba en los sistemas operativos de un procesador.

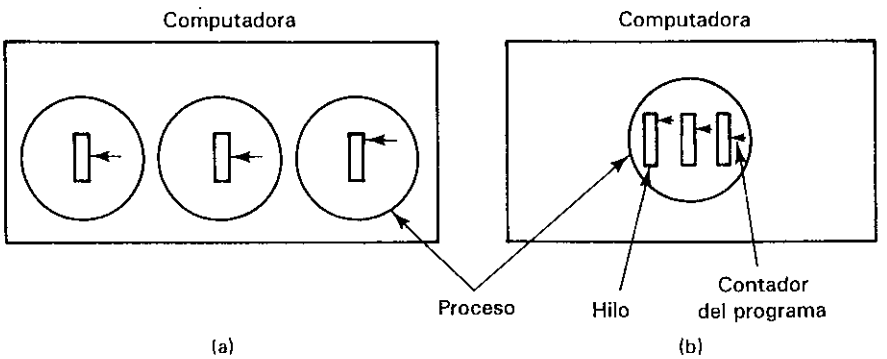


Figura 4-1. (a) Tres procesos con un hilo cada uno. (b) Un proceso con tres hilos.

En la figura 4-1(a) vemos una máquina con tres procesos. Cada uno de ellos tiene su contador del programa, su pila, su conjunto de registros y su espacio de direcciones. Los procesos no tienen nada que ver entre sí, excepto que podrían comunicarse mediante las primitivas de comunicación entre procesos del sistema, como los semáforos, monitores o mensajes. En la figura 4-1(b) vemos otra máquina, con un proceso, sólo que ahora este proceso tiene varios hilos de control, los cuales por lo general se llaman sólo **hilos** o a veces **procesos ligeros**. En muchos sentidos, los hilos son como pequeños miniprocesos. Cada hilo se ejecuta en forma estrictamente secuencial y tiene su contador de programa y una pila para llevar un registro de su posición. Los hilos comparten el CPU, de la misma forma que lo hacen los procesos: primero, se ejecuta un hilo y después otro (tiempo compartido). Sólo en un multiprocesador se pueden ejecutar en realidad en paralelo. Los hilos pueden crear hilos hijos y se pueden bloquear en espera de que se terminen sus llamadas al sistema, al igual que los procesos regulares. Mientras un hilo está bloqueado, se puede ejecutar otro hilo del mismo proceso, en la misma forma en que, cuando se bloquea un proceso, se puede ejecutar en la misma máquina otro proceso. La analogía "hilo es a proceso como proceso es a máquina" es válida en muchos sentidos.

Sin embargo, los distintos hilos de un proceso no son tan independientes como los procesos distintos. Todos los hilos tienen el mismo espacio de direcciones, lo que quiere decir que comparten también las mismas variables globales. Puesto que cada hilo puede tener acceso a cada dirección virtual, un hilo puede leer, escribir o limpiar de manera completa la pila de otro hilo. No existe protección entre los hilos debido a que (1) es imposible y (2) no debe ser necesaria. A diferencia de los procesos distintos, que pueden ser de diversos usuarios y que pueden ser hostiles entre sí, un proceso siempre es poseído por un usuario, quien supuestamente ha creado varios hilos para que éstos cooperen y no luchen entre sí. Además de compartir un espacio de direcciones, todos los hilos comparten el mismo conjunto de archivos abiertos, procesos hijos, cronómetros, señales, etc. como se muestra en la figura 4-2. Así, la organización de la figura 4-1(a) se utilizaría en el caso en que los tres procesos en esencia no estuvieran relacionados, mientras que la figura 4-1(b) sería adecuada cuando los tres hilos fueran parte del mismo trabajo y cooperaran de manera activa y cercana entre sí.

Elementos por hilo
Contador del programa Pila Conjunto de registros Hilos hijos Estado

Elementos por proceso
Espacio de direcciones Variables globales Archivos abiertos Procesos hijos Cronómetros Señales Semáforos Información contable

Figura 4-2. Conceptos por hilo y por proceso.

Como los procesos tradicionales (es decir, los procesos con un hilo), los hilos pueden tener uno de los siguientes estados: en ejecución, bloqueado, listo o terminado. Un hilo en ejecución posee CPU y está activo. Un hilo bloqueado espera que otro elimine el bloqueo (por ejemplo, en un semáforo). Un hilo listo está programado para su ejecución, la cual se llevará a cabo tan pronto le llegue su turno. Por último, un hilo terminado es aquel que ha hecho su salida, pero que todavía no es recogido por su padre (en términos de UNIX, el hilo padre no ha realizado un WAIT).

4.1.2. Uso de hilos

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema. Consideremos de nuevo nuestro ejemplo del servidor de archivos. En la figura 4-3(a) se muestra una posible organización. En ésta, un hilo, el **servidor**, lee las solicitudes de trabajo del buzón del sistema. Después de examinar

la solicitud elige a un **hilo trabajador** inactivo (es decir, bloqueado) y le envía la solicitud, lo cual se puede realizar al escribir un apuntador al mensaje en una palabra especial asociada a cada hilo. El servidor despierta entonces al trabajador dormido (por ejemplo, lleva a cabo un UP en el semáforo en donde duerme).

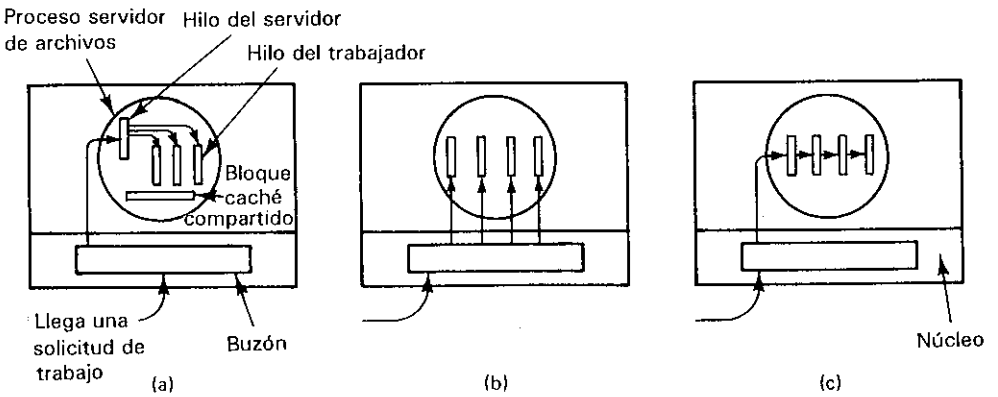


Figura 4-3. Tres organizaciones de hilos en un proceso. (a) Modelo del servidor/trabajador. (b) Modelo de equipo. (c) Modelo de entubamiento.

Cuando el trabajador despierta, verifica si puede satisfacer la solicitud por medio del bloque caché compartido, al que tienen acceso todos los hilos. Si no puede, envía un mensaje al disco para obtener el bloque necesario (si se trata de un READ) y se duerme en espera de la conclusión de la operación del disco. Se llama entonces al planificador y se inicia otro hilo, que tal vez sea el servidor, para pedir más trabajo; o bien, otro trabajador listo para realizar un trabajo.

Pensemos ahora cómo se podría escribir al servidor de archivos en ausencia de los hilos. Una posibilidad es que opere como un hilo. El ciclo principal del servidor de archivos obtiene una solicitud, la examina y concluye antes de obtener la siguiente. Mientras espera al disco, el servidor está inactivo y no procesa otras solicitudes. Si el servidor de archivos se ejecuta en una máquina exclusiva para él, como es lo común, el CPU está inactivo mientras el servidor espera al disco. El resultado neto es que se pueden procesar menos solicitudes/segundo. Así, los hilos ganan un desempeño considerable, pero cada uno de ellos se programa en forma secuencial, de la manera usual.

Hasta aquí hemos visto dos posibles diseños: un servidor de archivos con varios hilos y otro con un hilo. Supongamos que no se dispone de los hilos, pero los diseñadores del sistema consideran inaceptable la pérdida de desempeño debida al uso de un hilo. Una tercera posibilidad es ejecutar el servidor como una máquina de estado finito de gran tamaño. Al recibir una solicitud, el único hilo la examina. Si se puede satisfacer mediante el caché, está bien; pero si no, envía un mensaje al disco.

Sin embargo, en vez del bloqueo, registra el estado de la solicitud actual en una tabla, va hacia ella y obtiene el siguiente mensaje, que puede ser una solicitud de nuevo trabajo o una respuesta del disco con respecto a una operación anterior. En el caso del nuevo trabajo, éste comienza. Si es una respuesta del disco, se busca la información relevante en la tabla y se procesa la respuesta. Puesto que no se permite en este caso enviar un mensaje y bloquearse en espera de una respuesta, no se puede utilizar RPC. Las primitivas deben ser llamadas sin bloqueo a *send* y *receive*.

En este diseño, se pierde el modelo de "proceso secuencial" que teníamos en los primeros dos casos. Hay que guardar de forma explícita el estado del cómputo y restaurarlo en la tabla, para cada mensaje enviado o recibido. En efecto, simulamos los hilos y sus pilas de la manera difícil. El proceso es operado como una máquina de estado finito que obtiene un evento y entonces reacciona a esto, dependiendo de lo que hay en él.

Debe ser claro ahora lo que los hilos ofrecen. Ellos mantienen la idea de procesos secuenciales que hacen llamadas al sistema con bloqueo (por ejemplo, RPC para comunicarse al disco) y aun así logran un paralelismo. Este tipo de llamadas al sistema facilita la programación y el paralelismo mejora el desempeño. El servidor de un hilo mantiene el uso de estas llamadas al sistema, pero no aumenta el desempeño. El método de la máquina de estado finito logra mejor desempeño a través del paralelismo, pero utiliza llamadas que no bloquean y por lo tanto es difícil de programar. Estos modelos se resumen en la figura 4-4.

Modelo	Características
Hilos	Paralelismo, llamadas al sistema con bloqueo
Proceso de un solo hilo	Sin paralelismo, llamadas al sistema con bloqueo
Máquina de estado finito	Paralelismo, llamadas al sistema sin bloqueo

Figura 4-4. Tres formas para construir un servidor.

La estructura del servidor en la figura 4-3(a) no es la única manera de organizar un proceso de muchos hilos. El modelo de equipo de la figura 4-3(b) es también una posibilidad. Aquí todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. No hay servidor. A veces llega trabajo que un hilo no puede manejar, en particular si cada hilo se especializa en manejar cierto tipo de trabajo. En este caso, se puede utilizar una cola de trabajo, la cual contiene todos los trabajos pendientes. Con este tipo de organización, un hilo debe verificar primero la cola de trabajo antes de buscar en el buzón del sistema.

Los hilos también se pueden organizar mediante el modelo de **entubamiento** de la figura 4-3(c). En este modelo, el primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Ésta puede ser una buena opción en ciertos problemas, como el de los

productores y los consumidores, aunque no es adecuado para servidores de archivos. Los entubamientos se utilizan con amplitud en muchas áreas de los sistemas de cómputo, desde la estructura interna de los CPU RISC hasta las líneas de comandos de UNIX.

Con frecuencia, los hilos también son útiles para los clientes. Por ejemplo, si un cliente desea copiar un archivo en varios servidores, puede hacer que un hilo se comunique con cada servidor. Otro uso de los hilos clientes es el manejo de señales, como las interrupciones desde el teclado (DEL o BREAK). En vez de dejar que la señal interrumpa al proceso, se dedica un hilo de tiempo completo para la espera de señales. Por lo general, éste se bloquea; pero cuando llega una señal, despierta y la procesa. Así, el uso de hilos puede eliminar la necesidad de las interrupciones a nivel usuario.

Otro argumento para los hilos no tiene que ver con RPC o la comunicación. Ciertas aplicaciones son más fáciles de programar mediante procesos paralelos; por ejemplo, el problema de los productores y los consumidores. El hecho de que el productor y el consumidor se ejecuten en paralelo es secundario. Se les programa de esa manera debido a que esto hace más simple el diseño del software. Como comparten un buffer común, no funcionaría el hecho de tenerlos en procesos ajenos. Los hilos se ajustan perfectamente a este caso.

Por último, aunque no estamos analizando este tema de manera explícita, en un sistema multiprocesador es posible que los hilos de un espacio de direcciones se ejecuten en realidad en paralelo, en varios CPU. De hecho, ésta es una de las formas principales para compartir los objetos en tales sistemas. Por otro lado, un programa diseñado de manera adecuada y que utilice hilos debe funcionar bien, tanto en un CPU con hilos compartidos, como en un verdadero multiprocesador, por lo que los aspectos del software son casi iguales de cualquier forma.

4.1.3. Aspectos del diseño de paquetes de hilos

Un conjunto de primitivas relacionadas con los hilos (por ejemplo, llamadas a biblioteca) disponibles para los usuarios se llama un **paquete de hilos**. En esta sección analizaremos algunos de los aspectos relacionados con la arquitectura y funcionalidad de estos paquetes. En la siguiente veremos la forma de implantarlos.

El primer aspecto que analizaremos es el manejo de los hilos. Aquí se tienen dos alternativas, los hilos dinámicos y los estáticos. En un diseño estático, se elige el número de hilos al escribir el programa o durante su compilación. Cada uno de ellos tiene asociada una pila fija. Este método es simple, pero inflexible.

Un método más general consiste en permitir la creación y destrucción de los hilos durante la ejecución. La llamada para la creación de hilos determina el programa principal del hilo (como un apuntador a un procedimiento) y un tamaño de pila, así como otros posibles parámetros; por ejemplo, una prioridad de planificación. La llamada regresa por lo general un identificador de hilo, para utilizarlo en las llamadas posteriores relacionadas con el hilo. En este modelo, un proceso se inicia (de manera implícita) con un hilo, pero puede crear el número necesario de ellos y éstos pueden expirar al terminar.

Los hilos pueden concluir de dos maneras. Un hilo puede hacer su salida por su cuenta, al terminar su trabajo, o puede ser eliminado desde el exterior. En este aspecto, los hilos son como los procesos. En muchas situaciones, como en el caso de los servidores de archivos de la figura 4-3, los hilos se crean después de que se inicia el programa y nunca se eliminan.

Puesto que los hilos comparten una memoria común, pueden y de hecho la utilizan, para guardar datos que comparten los distintos hilos, como los buffers en un sistema de productores y consumidores. El acceso a los datos compartidos se programa por lo general mediante regiones críticas, para evitar que varios hilos intenten tener acceso a los mismos datos al mismo tiempo. La implantación de las regiones críticas es más fácil si se utilizan los semáforos, monitores u otras construcciones similares. Una técnica de uso común en los paquetes de hilos es el **mútex**, que es un cierto tipo de semáforo moderado. Un mútex sólo tiene dos estados, cerrado y no cerrado. Se definen dos operaciones de mútex. La primera, LOCK, intenta cerrar el mútex. Si el mútex no está cerrado, LOCK tiene éxito y el mútex se cierra en una acción atómica. Si dos hilos intentan cerrar al mismo mútex en el mismo instante, evento que sólo es posible en un multiprocesador, en el cual existen varios hilos en ejecución en distintos CPU, uno de ellos gana y el otro pierde. Si un hilo intenta cerrar un mútex ya cerrado, se bloquea.

La operación UNLOCK elimina la cerradura de un mútex. Si uno o más hilos esperan a un mútex, se libera exactamente uno de ellos. El resto continúa su espera.

Otra de las operaciones que se tienen en ciertos casos es TRYLOCK, que intenta cerrar un mútex. Si el mútex no está cerrado, TRYLOCK regresa un código de estado que indica el éxito. Sin embargo, si el mútex está cerrado, TRYLOCK no bloquea el hilo, sino que regresa un código de estado que indica la falla.

Los mútex son como los semáforos binarios (es decir, semáforos que sólo tienen los valores 0 y 1). No son como los semáforos de conteo. Esta limitación facilita su implantación.

Otra característica de sincronización que a veces está disponible en los paquetes de hilos es la **variable de condición**, similar a la variable de condición que se utiliza para la sincronización en el caso de los monitores. Por lo general, se asocia una variable de condición a un mútex cuando éste se crea. La diferencia entre el mútex y la variable de condición es que el primero se utiliza para una cerradura a corto plazo, principalmente para proteger la entrada a las regiones críticas. Las variables de condición se utilizan para una espera a largo plazo hasta que un recurso esté disponible.

La siguiente situación es muy común. Un hilo cierra un mútex para obtener la entrada a una región crítica. Una vez dentro de ella, examina las tablas del sistema y encuentra que cierto recurso necesario para él está ocupado. Si sólo cierra un segundo mútex (asociado al recurso), el mútex exterior permanecerá cerrado y el hilo que conserva el recurso no podrá entrar a la región crítica para liberarlo. Ocurre un bloqueo. Si se elimina la cerradura del mútex exterior, otros hilos pueden entrar a la región crítica, lo cual provoca un caos, de modo que esta solución no es aceptable.

Una solución es el uso de las variables de condición para adquirir el recurso, como se muestra en la figura 4-5(a). En este caso, la espera de la variable de condición se define de modo que ejecute la espera y elimine la cerradura en forma atómica. Más adelante, cuando

el hilo que conservaba el recurso lo libera, como se muestra en la figura 4-5(b), llama a *wakeup*, que se define de forma que despierte a un hilo o bien a todos los hilos que esperan a la variable de condición especificada. El uso de WHILE en vez de IF en la figura 4-5(a) evita que el hilo se despierte pero que alguien más se apropie del recurso antes de que se ejecute el hilo.

```
lock mutex;
    check data structures;
    white(resource busy)
        wait(condition variable);
    mark resource as busy;
unlock mutex;
```

(a)

```
lock mutex;
    mark resource as free;
unlock mutex;
wakeup(condition variable);
```

(b)

Figura 4-5. Uso de mútex y variables de condición.

La necesidad de poder despertar a todos los hilos, en vez de uno, se demuestra en el problema de los lectores y los escritores. Al terminar un escritor, puede optar por despertar a todos los escritores pendientes o a todos los lectores pendientes. Si elige los lectores, debe despertarlos a todos y no a uno. La flexibilidad necesaria se obtiene mediante las primitivas de hilos para despertar con exactitud un hilo o despertarlos a todos.

El código de un hilo consta por lo general de varios procedimientos, al igual que un proceso. Pueden tener variables locales, variables globales y parámetros del procedimiento. Las variables locales y los parámetros no provocan problema alguno, pero las variables globales de un hilo que no son globales en todo el programa pueden provocar ciertas dificultades.

Por ejemplo, consideremos la variable *errno* de UNIX. Cuando un proceso (o hilo) hace una llamada al sistema y ésta falla, se coloca el código de error en *errno*. En la figura 4-6, el hilo 1 ejecuta la llamada al sistema ACCESS para saber si tiene permiso para tener acceso a cierto archivo. El sistema operativo regresa la respuesta en una variable global *errno*. Después de que el control regresa al hilo 1, pero antes de poder leer *errno*, el planificador decide que el hilo 1 ha tenido el tiempo suficiente de CPU y decide cambiar al hilo 2. El hilo 2 ejecuta una llamada OPEN que falla, lo que provoca la escritura en *errno* y que se pierda para siempre el código de acceso al hilo 1. Si el hilo 1 se inicia más tarde, leerá el valor incorrecto y se comportará también de forma incorrecta.

Es posible dar varias soluciones a este problema. Una es prohibir las variables globales en su conjunto. Aunque esto podría ser lo ideal, hay un conflicto con gran parte del software existente, como UNIX. Otra solución consiste en asignarle a cada hilo sus propias variables globales particulares, como se muestra en la figura 4-7. De esta manera, cada hilo tiene su copia de *errno* y de otras variables globales, por lo que se evitan los conflictos. De hecho, esta decisión crea un nuevo nivel de visibilidad, donde las variables son visibles a todos los procedimientos de un hilo, además de los niveles de visibilidad correspondientes a las variables visibles en un procedimiento específico y las variables visibles en todo el programa.

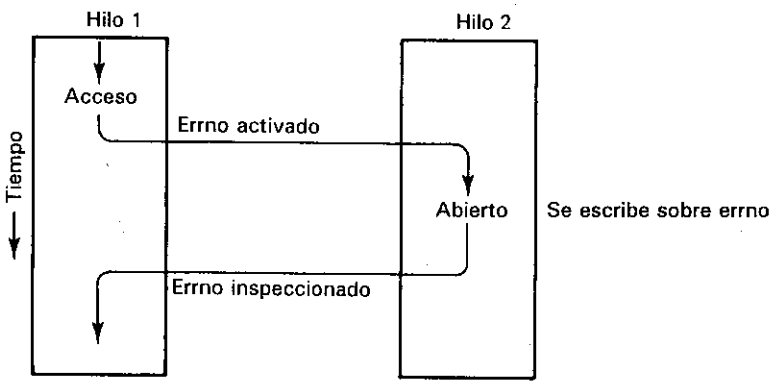


Figura 4-6. Conflictos entre los hilos sobre el uso de una variable global.

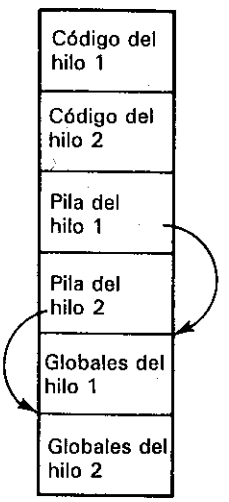


Figura 4-7. Los hilos pueden tener variables globales particulares.

Sin embargo, el acceso a las variables globales particulares es un tanto truculento, puesto que la mayoría de los lenguajes de programación tienen forma de expresar las variables locales y las globales, pero no tienen formas intermedias. Es posible asignar un bloque de memoria a las variables globales y transferirlo a cada procedimiento del hilo, como un parámetro adicional. Aunque ésta no es una solución elegante, funciona.

En otra alternativa, se pueden introducir nuevos procedimientos de biblioteca para crear, dar valores y leer estas variables globales a lo largo de todo un hilo. La primera llamada se parecería a:

```
create_global("bufptr");
```

Asigna un espacio de almacenamiento a un apuntador llamado *bufptr* en una pila o un área especial de almacenamiento, reservado para el hilo que hizo la llamada. Sin importar la posición del espacio de almacenamiento, el hilo que llamó es el único que tiene acceso a la variable global. Si otro hilo crea una variable global con el mismo nombre, obtiene un espacio de almacenamiento distinto, que no entra en conflicto con el ya existente.

Se necesitan dos llamadas para tener acceso a las variables globales; una para escribirlas y otra para leerlas. Para la escritura, funciona algo como

```
set_global("bufptr", &buf);
```

Guarda el valor de un apuntador en el espacio de almacenamiento creado en la llamada a *create_global*. Para leer una variable global, la llamada sería algo así como

```
bufptr = read_global("bufptr");
```

Esta llamada regresa la dirección almacenada en la variable global, de modo que se pueda tener acceso al valor del dato.

Nuestro último aspecto del diseño relativo a los hilos es la planificación. Los hilos se pueden planificar mediante distintos algoritmos, entre los que se encuentran la prioridad, round robin y otros. Los paquetes de hilos proporcionan a menudo ciertas llamadas para que el usuario pueda especificar el algoritmo de planificación y establecer las prioridades, en su caso.

4.1.4. Implantación de un paquete de hilos

Existen dos métodos principales para implantar un paquete de hilos: en el espacio del usuario y en el espacio del núcleo. La elección tiene una controversia moderada y también existe una implantación híbrida. En esta sección describiremos estos métodos, sus ventajas y desventajas.

Implantación de los hilos en el espacio del usuario

El primer método consiste en colocar todo el paquete de hilos en el espacio del usuario. El núcleo no sabe de su existencia. En lo que respecta a éste, maneja procesos ordinarios con un hilo. La primera ventaja, que también es la más obvia, es que un paquete de hilos implantado en el espacio del usuario se puede implantar en un sistema operativo que no

tiene que soportar dichos hilos. Por ejemplo, UNIX no soporta hilos, pero para él se han escrito distintos paquetes de hilos en el espacio del usuario.

Todas estas implantaciones tienen la misma estructura general, la cual se muestra en la figura 4-8(a). Los hilos se ejecutan en la parte superior de un sistema al tiempo de ejecución, el cual es una colección de procedimientos que manejan los hilos. Cuando un hilo ejecuta una llamada al sistema, se duerme, desarrolla una operación en un semáforo o mútex, o bien lleva a cabo cualquier acción que pueda provocar su suspensión, entonces llama a un procedimiento del sistema de tiempo de ejecución. Este procedimiento verifica si hay que suspender al hilo. En tal caso, guarda los registros del hilo (es decir, los propios) en una tabla, busca un hilo no bloqueado para ejecutarlo y vuelve a cargar los registros de la máquina con los valores resguardados del nuevo hilo. Tan pronto se intercambien el apuntador a la pila y el contador del programa, el nuevo hilo vuelve a la vida. Si la máquina tiene una instrucción para almacenar todos los registros y otra para cargarlos, todo el intercambio de hilos se puede llevar a cabo en una operación. Este intercambio de hilo es al menos de un orden de magnitud más rápido que los señalamientos al núcleo y es un fuerte argumento en favor de los paquetes de hilos en el espacio del usuario.

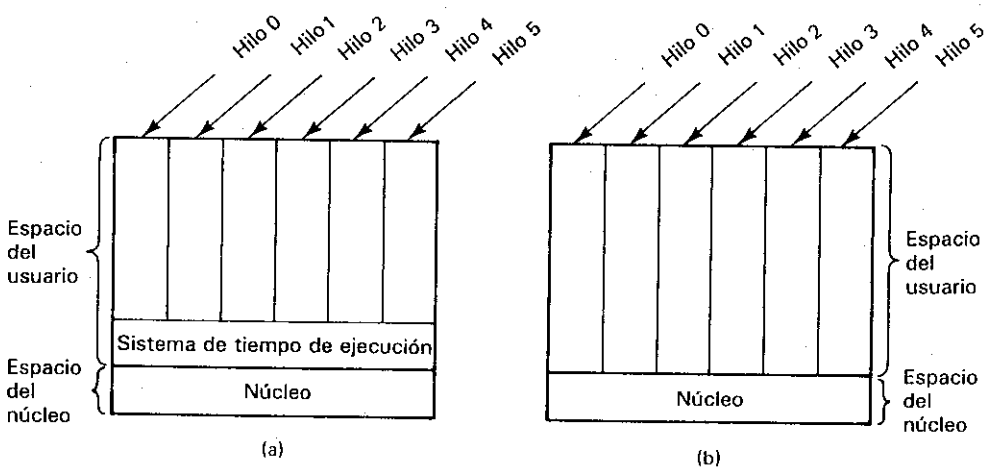


Figura 4-8. (a) Un paquete de hilos a nivel usuario. (b) Paquete de hilos manejado por el núcleo.

Los hilos a nivel usuario también tienen otras ventajas. Permiten que cada proceso tenga su algoritmo de planificación adaptado. Por ejemplo, para ciertas aplicaciones, aquellas que cuentan con un hilo recolector de basura, es muy bueno no tener que preocuparse por el hecho de que un hilo se detenga en un momento inconveniente. Tienen mejor escalabilidad, puesto que los hilos del núcleo requieren de manera invariable algún espacio para sus tablas y su pila en el núcleo, lo cual puede ser un problema si existe un número muy grande de hilos.

A pesar de su mejor desempeño, los paquetes de hilos a nivel usuario tienen ciertos problemas fundamentales. El primero de ellos es el problema de la implantación de las llamadas al sistema con bloqueo. Supongamos que un hilo lee de un entubamiento vacío o

que hace algo que provoque un bloqueo. No se puede permitir que el hilo realice en realidad la llamada al sistema, puesto que esto detendría a todos los hilos. Uno de los objetivos principales de contar con los hilos es permitir a cada uno de ellos que utilicen llamadas con bloqueo, pero evitando que un hilo bloqueado afecte a los demás. Con las llamadas al sistema con bloqueo no se puede lograr este objetivo.

Las llamadas al sistema se pueden modificar para que no utilicen el bloqueo (por ejemplo, una lectura en un entubamiento vacío puede sólo provocar una falla), pero pedirle cambios al sistema operativo no es atractivo. Además, uno de los argumentos para el uso de los hilos a nivel usuario era precisamente que se pudieran ejecutar con los sistemas operativos *existentes*. Por último, la modificación de la semántica de READ requeriría cambios a muchos programas del usuario.

Se dispone de otra alternativa cuando es posible decir de antemano si una llamada se bloqueará. En ciertas versiones de UNIX, existe una llamada SELECT, la cual permite decidir si un entubamiento está vacío, etc. Cuando se dispone de esta llamada, se puede remplazar el procedimiento de biblioteca *read* con otro que primero realice una llamada SELECT y después realice la llamada READ sólo en caso de que sea segura (es decir, que no se bloquee). Si la llamada READ se bloquea, no se lleva a cabo, sino que se ejecuta otro hilo. La siguiente vez que el sistema de tiempo de ejecución obtiene el control, puede volver a verificar si la llamada READ es segura. Este método requiere escribir de nuevo parte de la biblioteca de llamadas al sistema, es ineficiente y poco elegante, pero existen pocas opciones. El código que se coloca junto a la llamada al sistema para hacer la verificación recibe el nombre de **jacket**.

Algo análogo al problema del bloqueo de las llamadas del sistema es el de las fallas de página. Si un hilo causa una falla de página, el núcleo, aun desconociendo la existencia de hilos, bloquea naturalmente todo el proceso hasta que la página necesaria ha sido incluida aunque se puedan correr otros hilos.

Otro problema con los paquetes de hilos a nivel usuario es que si un hilo comienza su ejecución, ninguno de los demás hilos de ese proceso puede ejecutarse, a menos que el primer hilo entregue en forma voluntaria al CPU. Dentro de un proceso, no existen interrupciones del reloj, lo que imposibilita la planificación round robin. El planificador no tendrá oportunidad alguna, a menos que un hilo entre al sistema de tiempo de ejecución por voluntad propia.

Un área en la que la ausencia de interrupciones de reloj es crucial es la sincronización. En las aplicaciones distribuidas, es común que un hilo inicie una actividad a la que debe responder otro hilo, y después se siente a esperar en un ciclo que verifica si ha habido una respuesta. Esta situación es una **espera ocupada** (o **spin lock**). Este método es en particular atractivo cuando se espera una respuesta rápida y el costo del uso de semáforos es alto. Si los hilos se vuelven a planificar de manera automática después de algunos milisegundos con base en las interrupciones del reloj, este método trabaja bien. Sin embargo, si los hilos se ejecutan hasta bloquearse, este método es una receta para obtener un bloqueo.

Una posible solución al problema de los hilos con ejecución infinita es que el sistema de tiempo de ejecución solicite una señal al reloj (interrupción) una vez cada segundo, para obtener el control, pero esto es muy difícil de programar. Las interrupciones periódicas del

reloj con mayor frecuencia no siempre son posibles; y aunque lo fueran, el costo excesivo sería sustancial. Además, un hilo podría necesitar una interrupción del reloj, lo que interfiere con el sistema de tiempo de ejecución.

Otro argumento en contra de los hilos a nivel usuario, que tal vez es el más devastador, es que por lo general los programadores desean los hilos en aplicaciones donde aquéllos se bloquean a menudo, como por ejemplo, un servidor de archivos con varios hilos. Estos hilos realizan constantes llamadas al sistema. Una vez que se hace un señalamiento al núcleo para llevar a cabo una llamada al sistema, es probable que no haya mayor trabajo para el núcleo que el de la conmutación de hilos, si el primero de ellos está bloqueado; si el núcleo lo hace, no hay necesidad de la constante verificación de la seguridad de las llamadas al sistema. Para las aplicaciones que tienen limitaciones esenciales para el uso del CPU y que pocas veces se bloquean, ¿cuál es la razón para tener hilos? Nadie podría proponer en forma seria el cálculo de los primeros *n* números primos o un juego de ajedrez mediante hilos, puesto que no se gana nada haciéndolo de esa manera.

Implantación de hilos en el núcleo

Ahora consideremos que el núcleo sabe de la existencia de los hilos y cómo manejarlos. No se necesita un sistema de tiempo de ejecución, como se muestra en la figura 4-8(b). En vez de ello, cuando un hilo desea crear un nuevo hilo o destruir uno existente, hace una llamada al núcleo, el que realiza entonces la creación o la destrucción.

Para controlar todos los hilos, el núcleo tiene una tabla con una entrada por cada hilo, con los registros, estado, prioridades y demás información relativa al hilo. La información es la misma que en el caso de los hilos a nivel usuario, sólo que ahora se encuentra en el espacio del núcleo y no en el espacio del usuario (dentro del sistema de tiempo de ejecución). Esta información también es la misma que los núcleos tradicionales conservan acerca de sus procesos con un hilo, es decir, el estado del proceso.

Todas las llamadas que pueden bloquear un hilo, como la sincronización entre hilos mediante semáforos, se implantan como llamadas al sistema, con un costo considerable mayor que una llamada a un procedimiento del sistema de tiempo de ejecución. Cuando un hilo se bloquea, el núcleo puede optar por ejecutar otro hilo del mismo proceso (si alguno está listo) o un hilo de un proceso distinto. Con los hilos a nivel usuario, el sistema de tiempo de ejecución mantiene en ejecución los hilos de su proceso hasta que el núcleo les retira el CPU (o bien cuando no existan hilos listos para su ejecución).

Debido al costo relativo mayor de la creación y destrucción de hilos en el núcleo, algunos sistemas adoptan un método correcto desde el punto de vista ambiental y reciclan sus hilos. Al destruir un hilo, se marca como no ejecutable, pero las estructuras de datos de su núcleo no se ven afectadas. Posteriormente, cuando se debe crear un nuevo hilo, se reactiva un hilo anterior, lo que ahorra ciertos costos. El reciclaje de hilos también es posible para los hilos a nivel usuario, pero puesto que el costo de administración de los hilos es mucho menor, existen menos incentivos para realizarlo.

Los hilos del núcleo no requieren nuevas llamadas al sistema sin bloqueo, ni conducen a bloqueos cuando se utiliza la espera ocupada. Además, si un hilo de un proceso provoca

una falla, el núcleo puede ejecutar con facilidad otro hilo mientras que espera que la página requerida sea traída del disco (o de la red). Su principal desventaja es que el costo de una llamada a sistema es sustancial, de modo que si las operaciones con los hilos (creación, eliminación, sincronización, etc.) son comunes, se incurrirá en más costos.

Además de todos los problemas específicos de los hilos de usuario o los hilos del núcleo, existen otros que aparecen en ambos. Para comenzar, muchos procedimientos de biblioteca no son re-entrantes. Por ejemplo, el envío de un mensaje a través de la red se puede programar de forma que primero organice el mensaje en un buffer fijo y que después señale al núcleo que lo envíe. ¿Qué ocurre si un hilo ha organizado un mensaje en un buffer, para que después una interrupción del reloj provoque una conmutación con un segundo hilo que de manera inmediata escriba en el buffer su mensaje encima del anterior? De manera similar, cuando se completa una llamada al sistema, puede ocurrir un cambio de hilos antes de que el hilo anterior pueda leer el estado de error (*errno*, ya analizado). Además, los procedimientos para la asignación de memoria, como *malloc* de UNIX, juegan con tablas cruciales, sin preocuparse por establecer y utilizar regiones críticas protegidas, puesto que fueron escritas en ambientes con un hilo, donde esto no era necesario. Para poder arreglar todos estos problemas de manera adecuada habría que escribir toda la biblioteca.

Otra solución consiste en proporcionar a cada quien un jacket que cierre un semáforo o mútex global al iniciar el procedimiento. De esta forma, sólo puede estar activo un hilo en la biblioteca en un instante dado. De hecho, la biblioteca se convierte en un enorme monitor.

Las señales también presentan dificultades. Supongamos que un hilo desea atrapar una señal particular (por ejemplo, el hecho de que el usuario oprima la tecla DEL) y que otro hilo desea esta señal para terminar el proceso. Esta situación puede surgir si uno o más hilos ejecutan procedimientos estándar de biblioteca y otros utilizan procedimientos escritos por el usuario. Es claro que estos deseos son incompatibles. En general, es difícil manejar las señales en un ambiente con un hilo. El paso a un ambiente con varios hilos no facilita su manejo. Las señales son un concepto típico por proceso y no por hilo, en especial si el núcleo no está consciente de la existencia de los hilos.

Activaciones del planificador

Varios investigadores han intentando combinar la ventaja de los hilos de usuario (buen desempeño) con la ventaja de los hilos de núcleo (no tener que utilizar muchos trucos para que las cosas funcionen). Adelante describiremos uno de tales métodos, diseñado por Anderson *et al.* (1991), llamado **activaciones del planificador**. El trabajo relativo a este método se analiza en Edler *et al.* (1988) y Scott *et al.* (1990).

Los objetivos del trabajo de activación del planificador son imitar la funcionalidad de los hilos del núcleo, pero con el mejor desempeño y mayor flexibilidad asociados por lo general con los paquetes de hilos implantados en el espacio del usuario. En particular, los hilos del usuario no tienen que realizar llamadas especiales al sistema sin bloqueo o verificar de antemano si es seguro realizar ciertas llamadas al sistema. Sin embargo, cuando un hilo se bloquea debido a una llamada al sistema o por fallo de página, debe ser posible ejecutar otros hilos dentro del mismo proceso, si existen.

La eficiencia se logra al evitar las transiciones innecesarias entre el espacio del usuario y el espacio del núcleo. Por ejemplo, si un hilo se bloquea por un semáforo local, no existe razón para implicar al núcleo. El sistema de tiempo de ejecución en el espacio del usuario puede bloquear el hilo de sincronización y planear otro nuevo por sí mismo.

Cuando se utilizan las activaciones del planificador, el núcleo asigna cierta cantidad de procesadores virtuales a cada proceso y permite que el sistema de tiempo de ejecución (en el espacio del usuario) asigne los hilos a los procesadores. Este mecanismo también se puede utilizar en un multiprocesador, donde los procesadores virtuales podrían ser CPU reales. La cantidad de procesadores virtuales asignados a un proceso es 1, al principio, pero el proceso puede solicitar más y también puede regresar los procesadores que ya no necesite. El núcleo puede retomar los procesadores virtuales ya asignados para asignarlos a otros procesos más necesitados.

La idea básica que permite que este esquema funcione es que cuando el núcleo sabe que un hilo se ha bloqueado (por ejemplo, al ver que ha ejecutado una llamada al sistema con bloqueo o haber provocado un fallo de página), el núcleo informa al sistema de tiempo de ejecución del proceso, pasando como parámetros sobre la pila el número del hilo en cuestión y una descripción del evento ocurrido. El aviso ocurre al activar el núcleo el sistema de tiempo de ejecución en una dirección de inicio conocida, algo similar a una señal en UNIX. Este mecanismo es una **llamada (upcall)**.

Una vez activado de esta manera, el sistema de tiempo de ejecución puede replanificar sus hilos, señalando por lo general al hilo activo como bloqueado y tomar otro hilo de la lista, configurar sus registros y reiniciarlo. Después, cuando el núcleo observa que el hilo original puede ejecutarse de nuevo (por ejemplo, el entubamiento que quería leer ya contiene datos, o la página faltante ha sido traída del disco), hace otra llamada al sistema de tiempo de ejecución para informarle de este evento. El sistema de tiempo de ejecución, a discreción, puede reiniciar el hilo bloqueado de inmediato, o colocarlo en la lista de hilos preparados para su ejecución posterior.

Cuando la interrupción de hardware ocurre mientras se ejecuta un hilo del usuario, el CPU interrumpido conmuta a modo núcleo. Si la interrupción es causada por un evento que no es de interés para el proceso interrumpido, como la terminación de la E/S de otro proceso, al concluir el controlador de interrupciones, coloca de nuevo el hilo interrumpido en su estado antes de la interrupción. Sin embargo, si el proceso está interesado en la misma, como la llegada de una página necesaria para uno de los hilos del proceso, el hilo interrumpido no se reinicia. En vez de esto, se suspende el hilo interrumpido y el sistema de tiempo de ejecución se inicia en tal CPU virtual, con el estado del hilo interrumpido en la pila. Entonces, el sistema de tiempo de ejecución debe decidir cuál de los hilos debe planificar para tal CPU: el interrumpido, el nuevo o algún otro.

Aunque las activaciones del planificador resuelven el problema de cómo transferir el control a un hilo no bloqueado en un proceso del cual uno de sus hilos acaba de bloquearse, crea un nuevo problema. Éste consiste en que un hilo interrumpido podría haber estado ejecutando una operación de semáforo en el momento de su suspensión, en cuyo caso podría tener una cerradura sobre la lista de hilos listos para su ejecución. Si el sistema de tiempo

de ejecución iniciado por la llamada intenta entonces adquirir esta cerradura, para colocar un hilo listo para su ejecución en la lista, no podrá adquirirla y ocurrirá un bloqueo. El problema se puede resolver llevando un registro del momento en que los hilos están o no en las regiones críticas, pero la solución es compleja y lejos de ser elegante.

Otra objeción a las activaciones del planificador es la confianza fundamental en las llamadas (upcall), un concepto que viola la estructura inherente en los sistemas con capas. Por lo general, la capa n ofrece ciertos servicios que pueden ser llamados por la capa $n + 1$, pero la capa n no puede llamar a los procedimientos de la capa $n + 1$.

4.1.5. Hilos y RPC

Es común que los sistemas distribuidos utilicen tanto la RPC como los hilos. Puesto que los hilos se idearon como alternativa menos costosa a los procesos estándar (pesados), es natural que los investigadores analicen las RPC más de cerca en este contexto, para ver si éstas también se pueden aligerar. En esta sección analizaremos algunos trabajos interesantes en esta área.

Bershad *et al.* (1990) observaron que incluso en un sistema distribuido, un gran número de RPC corresponden a procesos donde una misma máquina es quien hace las llamadas (por ejemplo, al administrador de ventanas). Es evidente que este resultado depende del sistema, pero es muy común como para tomarlo en cuenta. Ellos propusieron un nuevo esquema que hace posible la llamada de un hilo en un proceso a un hilo de otro proceso en la misma máquina, de manera mucho más eficiente que la usual.

La idea funciona como sigue. Al iniciar un hilo servidor, S, éste exporta su interfaz al informarle de ésta al núcleo. La interfaz define los procedimientos que puede llamar, sus parámetros, etc. Al iniciar un hilo cliente, C, éste importa la interfaz del núcleo y se le proporciona un identificador especial para utilizarlo en la llamada. El núcleo sabe ahora que C llamará posteriormente a S y crea estructuras de datos especiales con el fin de prepararse para la llamada.

Una de estas estructuras de datos es una pila de argumentos compartida por C y S y que se asocia de manera lectura/escritura en ambos espacios de direcciones. Para llamar al servidor, C coloca sus argumentos en la pila compartida, mediante el procedimiento normal de transferencia y después hace un señalamiento al núcleo, al colocar un identificador especial en un registro. El núcleo se da cuenta de esto y sabe que la llamada es local. (Si hubiese sido remota, el núcleo trabajaría con ella de forma normal para el caso de las llamadas remotas.) Después modifica el mapa de memoria del cliente para colocar éste en el espacio de direcciones del servidor e inicia el hilo cliente, al ejecutar el procedimiento del servidor. La llamada se lleva a cabo de tal forma que los argumentos se encuentren ya en su lugar, de modo que no sea necesario el copiado o el ordenamiento. El resultado neto es que la RPC local se puede realizar más rápido de esta manera.

Otra técnica de uso común para agilizar la RPC se basa en la observación de que, al bloquearse un hilo servidor en espera de una nueva solicitud, en realidad no tiene que disponer de información importante relativa al contexto. Por ejemplo, es raro que deba tener variables locales y lo usual es que no haya algo importante en sus registros. Por consiguiente, cuando un hilo termina de atender una solicitud, se desvanece y se descartan su pila e información del contexto.

Al llegar un mensaje nuevo a la máquina servidora, el núcleo crea en ese momento un nuevo hilo para darle servicio a la solicitud. Además, asocia el mensaje con el espacio de direcciones del servidor y configura la pila del nuevo hilo para tener acceso al mensaje. Este esquema se llama de **recepción implícita** y contrasta con el hilo convencional que realiza una llamada al sistema para recibir un mensaje. El hilo que se crea de manera espontánea para el manejo de la RPC recibida se conoce a menudo como **hilo de aparición instantánea (pop-up thread)**. La idea se muestra en la figura 4-9.

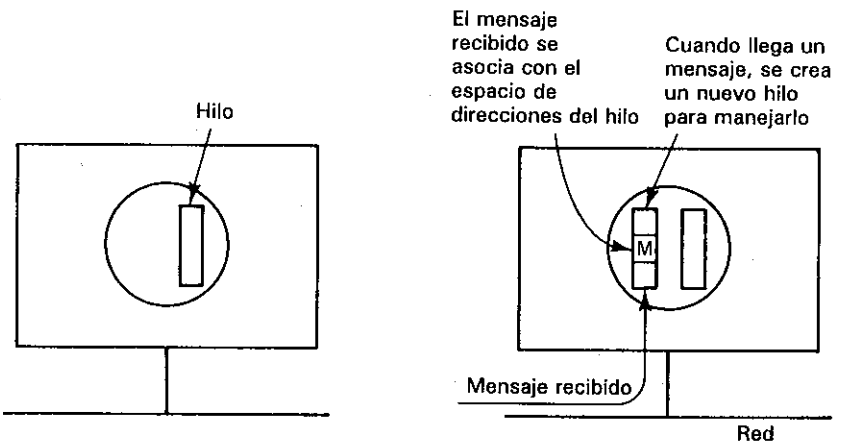


Figura 4-9. Creación de un hilo al llegar un mensaje.

El método tiene varias ventajas fundamentales sobre la RPC convencional. En primer lugar, los hilos no tienen que bloquearse en espera de más trabajo. Por esto, no hay que guardar información relativa al contexto. En segundo lugar, la creación de un nuevo hilo es más económica que la restauración de uno existente, puesto que no hay que restaurar el contexto. Por último, se ahorra tiempo al no tener que copiar los mensajes recibidos a un buffer dentro de un hilo servidor. Se pueden utilizar otros métodos para reducir el costo excesivo. En resumen, se puede lograr una mejora sustancial en la velocidad.

Los hilos son un tema actual de investigación. Algunos otros resultados se presentan en (Marsh *et al.*, 1991 y Draves *et al.*, 1991).

4.2. MODELOS DE SISTEMAS

Los procesos se ejecutan en procesadores. En un sistema tradicional, sólo existe un procesador, por lo que no viene a cuento la pregunta de cómo debe utilizarse éste. En un sistema distribuido, con varios procesadores, éste es un aspecto fundamental del diseño. Los procesadores de un sistema distribuido se pueden organizar de varias formas. En esta sección analizaremos las principales, el modelo de estación de trabajo y el modelo de la pila de procesadores, así como un híbrido que toma ciertas características de cada uno. Estos modelos se basan en filosofías diferentes en lo fundamental de lo que debe ser un sistema distribuido.

4.2.1. El modelo de estación de trabajo

El modelo de estación de trabajo es directo: el sistema consta de estaciones de trabajo (computadoras personales para usuarios finales) dispersas en un edificio o campus y conectadas entre sí por medio de una LAN de alta velocidad, como se muestra en la figura 4-10. Algunas de las estaciones de trabajo pueden estar en oficinas, con lo que de manera implícita, cada una de ellas se dedica a un usuario, mientras que otras pueden estar en áreas públicas y tener distintos usuarios en el transcurso del día. En ambos casos, en un instante dado, una estación de trabajo puede tener un usuario conectado a ella y tener entonces un "poseedor" (aunque sea temporal) o estar inactiva.

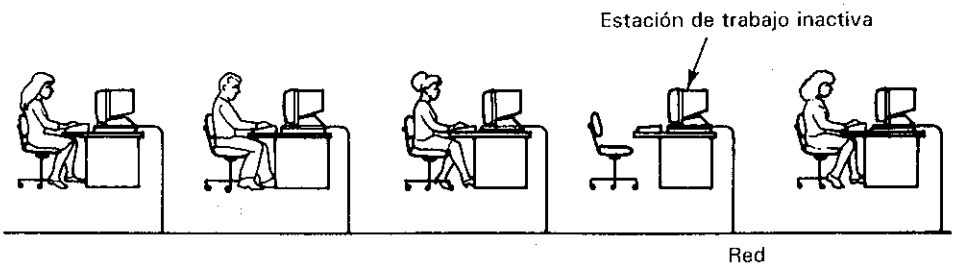


Figura 4-10. Una red de estaciones de trabajo personales, cada una con un sistema local de archivos.

En ciertos sistemas, las estaciones de trabajo pueden tener discos locales o en otros no. Los últimos reciben el nombre universal de **estaciones de trabajo sin disco**, pero las otras tienen nombres variados, como **estaciones de trabajo con disco,** o aun nombres más extraños. Si las estaciones de trabajo carecen de disco, el sistema de archivos debe ser implantado mediante uno o varios servidores de archivos en la red. Las solicitudes de lectura o escritura se envían a un servidor de archivos, el cual realiza el trabajo y envía de regreso las respuestas.

Las estaciones de trabajo sin disco son populares en las universidades y empresas por varias razones, una de las cuales es su precio. Las estaciones de trabajo equipadas con pequeños y lentos discos son generalmente más caras que si se tienen uno o dos servidores

de archivos equipados con discos enormes y rápidos a los cuales se tiene acceso mediante la LAN.

Una segunda razón para la popularidad de las estaciones de trabajo sin disco es su fácil mantenimiento. Cuando se lanza una nueva versión de cierto programa, digamos, un compilador, los administradores del sistema pueden instalarlo con facilidad en un pequeño número de servidores de archivos en el cuarto de máquinas. Su instalación en docenas o cientos de máquinas en todo el edificio o campus es otro problema por completo distinto. El respaldo y el mantenimiento del hardware también es más sencillo en un disco de 5 gigabytes en un lugar céntrico que en cincuenta discos de 100 megabytes dispersos en todo un edificio.

Otro punto en contra de los discos es que tienen ventiladores y hacen ruido. A muchas personas les disgusta ese ruido y no lo quieren en sus oficinas.

Por último, las estaciones de trabajo sin disco proporcionan simetría y flexibilidad. Un usuario puede caminar hacia cualquier estación de trabajo y entrar al sistema. Puesto que todos sus archivos están dentro del servidor de archivos, una estación de trabajo sin disco es tan buena como cualquier otra. Por el contrario, si todos los archivos se guardan en discos locales, el uso de la estación de trabajo de otra persona implica que usted tendrá fácil acceso a *sus* archivos, pero el uso de los propios requerirá de un esfuerzo adicional y claramente distinto del uso de su estación de trabajo.

Si la estación de trabajo tiene sus propios discos, éstos se pueden utilizar al menos de cuatro maneras:

1. Paginación y archivos temporales.
2. Paginación, archivos temporales y binarios del sistema.
3. Paginación, archivos temporales, binarios del sistema y sistema de ocultamiento de archivos.
4. Un sistema local de archivos completo.

El primer diseño se basa en la observación de que, aunque es conveniente mantener todos los archivos del usuario en los servidores centrales de archivos (para facilitar el respaldo, el mantenimiento, etc.), también se necesitan discos para la paginación (o intercambio) y los archivos temporales. En este modelo, los discos locales se utilizan de forma exclusiva para la paginación y los archivos temporales no compartidos que se puedan eliminar al final de la sesión. Por ejemplo, la mayoría de los compiladores constan de varias etapas, cada una de las cuales crea un archivo temporal que se lee en la siguiente etapa. Una vez leído el archivo, se le descarta. Los discos locales son ideales para el almacenamiento de tales archivos.

El segundo modelo es una variante del primero, en el que los discos locales también contienen los programas en binario (ejecutables), como los compiladores, editores de texto y controladores del correo electrónico. Cuando se llama a alguno de estos programas, se le busca en el disco local y no en el servidor de archivos, lo cual reduce la carga en la red. Puesto que es raro que estos programas sean modificados, se les puede instalar en todos los discos locales y mantenerlos ahí durante largos periodos. Cuando se dispone de una

nueva versión del programa, se transmite a todas las máquinas. Sin embargo, si ocurre que la máquina no funciona cuando se envía el programa, perderá éste y continuará ejecutando la versión anterior. Por ello, se necesita cierta administración para mantener un registro de las versiones de cada programa.

Un tercer método consiste en utilizar los discos locales como cachés explícitos (además de utilizarlos para la paginación, los archivos temporales y los binarios). En este modo de operación, los usuarios pueden cargar archivos desde los servidores de archivos hasta sus propios discos, leerlos y escribir en ellos de manera local y después regresar los archivos modificados al final de la sesión. El objetivo de esta arquitectura es mantener centralizado el almacenamiento a largo plazo, pero reducir la carga en la red al mantener los archivos en forma local mientras se utilizan. Una desventaja es poder mantener consistentes los cachés. ¿Qué ocurre si dos usuarios cargan el mismo archivo y lo modifican de manera distinta? Este problema no es fácil de resolver y lo analizaremos con mayor detalle en una sección posterior de este libro.

En cuarto lugar, cada máquina puede tener su sistema de archivos autocontenido, con la posibilidad de montar o tener acceso a los sistemas de archivos de otras máquinas. La idea aquí es que cada máquina esté autocontenida en lo fundamental y que el contacto con el mundo exterior sea limitado. Esta organización proporciona un tiempo de respuesta uniforme y garantizado para el usuario y pone poca carga en la red. La desventaja es que es difícil compartir algunos aspectos y el sistema resultante se parece más a un sistema operativo de red que a un verdadero sistema operativo distribuido y transparente.

Los modelos sin disco y con disco que hemos analizado se resumen en la figura 4-11. De arriba hacia abajo, se comienza desde la total dependencia de los servidores de archivos hasta la total independencia.

Las ventajas del modelo de estación de trabajo son variadas y claras. Ciertamente, el modelo es fácil de comprender. Los usuarios tienen una cantidad fija de poder de cómputo exclusivo, con lo que tienen un tiempo de respuesta garantizado. Los programas gráficos complejos pueden ser muy rápidos, puesto que tienen acceso directo a la pantalla. Cada usuario tiene alto grado de autonomía y puede asignar los recursos de su estación de trabajo como lo juzgue necesario. Los discos locales favorecen esta independencia y hacen posible que el trabajo continúe en mayor o menor grado si el servidor de archivos falla.

Sin embargo, el modelo también tiene dos problemas. En primer lugar, si los circuitos procesadores siguen abaratándose, pronto será posible, desde el punto de vista económico, proporcionar a cada usuario 10 CPU, o más adelante 100. Con 100 estaciones de trabajo en la oficina, será difícil ver la ventana de su oficina. En segundo lugar, la mayor parte del tiempo, los usuarios no utilizan sus estaciones de trabajo, las cuales se encuentran inactivas, mientras que es probable que otros usuarios necesiten una capacidad de cómputo adicional y no puedan obtenerla. Desde la perspectiva de todo el sistema, es ineficiente la asignación de recursos de modo que algunos usuarios reciban recursos que no necesitan mientras que otros usuarios sí.

El primer problema se puede enfrentar al hacer de cada estación de trabajo un multiprocesador personal. Por ejemplo, cada ventana de la pantalla puede tener su CPU para la ejecución exclusiva de sus programas. Sin embargo, la evidencia preliminar de algunos de

Dependencia de los servidores de archivos ↑

Uso del disco	Ventajas	Desventajas
(Sin disco)	Bajo costo, fácil mantenimiento del hardware y el software, simetría y flexibilidad	Gran uso de la red; los servidores de archivos se pueden convertir en cuellos de botella
Paginación, archivos de tipo borrador	Reduce la carga de la red comparado con el caso sin discos	Un costo alto debido al gran número de discos necesarios
Paginación, archivos de tipo borrador, binarios	Reduce todavía más la carga sobre la red	Alto costo; complejidad adicional para actualizar los binarios
Paginación, archivos de tipo borrador, binarios, ocultamiento de archivos	Una carga aún menor en la red; también reduce la carga en los servidores de archivos	Alto costo; problemas de consistencia del caché
Sistema local de archivos completo	Escasa carga en la red; elimina la necesidad de los servidores de archivos	Pérdida de transparencia

Figura 4-11. Uso de los discos en las estaciones de trabajo.

los primeros multiprocesadores personales, como la Firefly de DEC, sugieren que el promedio de CPU utilizados rara vez es mayor que 1, puesto que es raro que los usuarios tengan activos más de un proceso a la vez. De nuevo, éste es un uso ineficiente de los recursos, pero conforme se abaratan los CPU al mejorar la tecnología, su desperdicio podría dejar de ser un pecado.

4.2.2. Uso de estaciones de trabajo inactivas

El segundo problema, las estaciones de trabajo inactivas, ha sido el tema de numerosas investigaciones, principalmente porque muchas universidades tienen un número importante de estaciones de trabajo personales, algunas de las cuales están inactivas (¿una estación de trabajo inactiva es el patio de juegos del diablo?). Diversas mediciones muestran que en los períodos pico del mediodía, hasta un 30% de las estaciones de trabajo están inactivas en un instante dado. En la tarde, un número todavía mayor permanece inactivo. Se han propuesto varios esquemas para el uso de estaciones de trabajo inactivas o subutilizadas (Litzkow *et al.*, 1988; Nichols, 1987; Theimer *et al.*, 1985). En esta sección describiremos los principios fundamentales en relación con este trabajo.

El primer intento por permitir el uso de las estaciones de trabajo inactivas fue el programa *rsh* proporcionado junto con el UNIX de Berkeley. Este programa se llama con

```
rsh machine command
```

donde el primer argumento es el nombre de una máquina y el segundo un comando para ejecutarse en ella. Lo que hace *rsh* es ejecutar el comando específico en la máquina dada. Aunque se utiliza con amplitud, este programa tiene serios problemas. El primero de ellos es que el usuario debe indicar la máquina que desea utilizar, lo que coloca al usuario en el predicamento de mantener un registro de las máquinas inactivas. El segundo problema es

que el programa se ejecuta en el ambiente de la máquina remota, el cual es por lo general distinto del ambiente local. Por último, si alguien debe entrar a una máquina inactiva que ejecuta un proceso remoto, el proceso continúa su ejecución y el nuevo usuario debe aceptar el desempeño menor o encontrar otra máquina.

La investigación en el tema de las estaciones de trabajo inactivas se ha centrado en la solución de estos problemas. Los aspectos claves son:

1. ¿Cómo encontrar una estación de trabajo inactiva?
2. ¿Cómo lograr que un proceso remoto se ejecute en forma transparente?
3. ¿Qué ocurre si regresa el poseedor de la máquina?

Estudiaremos estos tres aspectos, uno a la vez.

¿Cómo encontrar una estación de trabajo inactiva? Para comenzar, ¿qué es una estación de trabajo inactiva? A primera vista, parecería que una estación de trabajo sin una persona utilizando la consola es una estación de trabajo inactiva, pero con los modernos sistemas de cómputo las cosas no son tan sencillas. En muchos sistemas, una estación donde ninguna persona está frente a ella puede ejecutar docenas de procesos, como demonios de reloj, de correo, de noticias y todos los demás demonios posibles. Por otro lado, un usuario que entre al sistema por la mañana pero que después no toque la computadora durante horas no coloca una carga adicional en dicho sistema. Los distintos sistemas toman diversas decisiones acerca del significado de la palabra "inactiva", pero por lo general se dice que la estación de trabajo está inactiva cuando nadie toca el teclado o el ratón durante varios minutos y no se ejecuta algún proceso iniciado por el usuario. En consecuencia, pueden existir diferencias sustanciales en la carga de una estación de trabajo inactiva y otra, debido, por ejemplo, al volumen de correo recibido en la primera pero no en la segunda.

Los algoritmos que se utilizan para localizar las estaciones de trabajo inactivas se pueden dividir en dos categorías: controlados por el servidor y controlados por el cliente. En la primera categoría, cuando una estación de trabajo está inactiva y por lo tanto se convierte en un servidor potencial, anuncia su disponibilidad. Puede hacer esto al proporcionar su nombre, dirección en la red y propiedades en un archivo de registros (o base de datos). Posteriormente, cuando un usuario desee ejecutar un comando en una estación de trabajo inactiva, puede escribir algo como:

```
remote command
```

y el programa *remote* busca en el registro una estación de trabajo adecuada. Por razones de confiabilidad, también es posible contar con varias copias del registro.

Otra alternativa consiste en que la estación inactiva anuncie el hecho de que no tiene trabajo, al colocar un mensaje que se transmite en toda la red. Todas las demás estaciones registran esto. De hecho, cada máquina tiene su copia del registro. La ventaja de esto es un menor costo en la búsqueda de una estación de trabajo y mayor redundancia. La desventaja es pedir a todas las máquinas que se encarguen de mantener el registro.

Ya sea que existan muchos o pocos registros, existe un peligro potencial de que aparezcan condiciones de competencia. Si dos usuarios llaman al mismo tiempo al comando *remote* y ambos descubren que la misma máquina está inactiva, ambos intentarán iniciar procesos al mismo tiempo. Para detectar y evitar esta situación, el programa *remote* verifica la estación de trabajo inactiva, la cual, si continúa libre, se elimina a sí misma del registro y da la señal de continuar. En este momento, quien hizo la llamada puede enviar su ambiente e iniciar el proceso remoto, como se muestra en la figura 4-12.

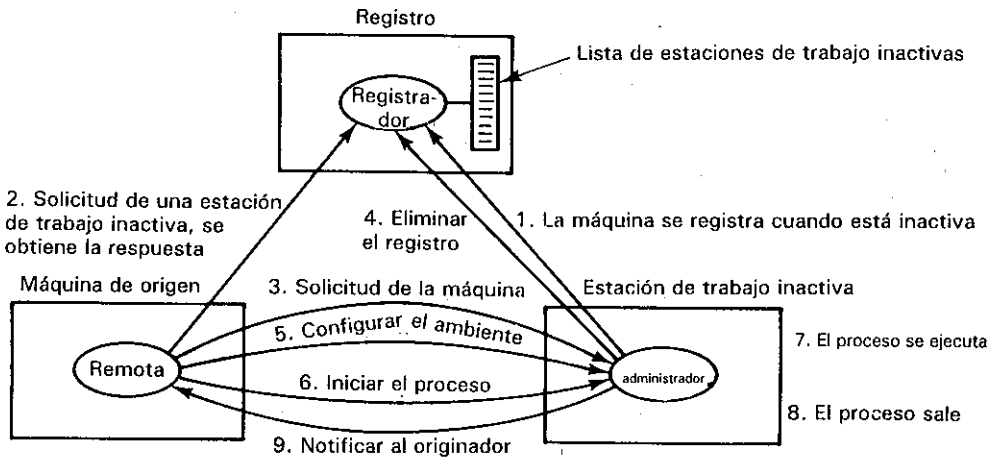


Figura 4-12. Un algoritmo basado en registros para la búsqueda y uso de las estaciones de trabajo inactivas.

La otra forma de localizar las estaciones inactivas utiliza un método controlado por el cliente. Al llamar a *remote*, éste transmite una solicitud donde indica el programa que desea ejecutar, la cantidad de memoria necesaria, si requiere o no un circuito coprocesador de punto flotante, etc. Estos detalles no son necesarios si todas las estaciones de trabajo son idénticas, pero son esenciales si el sistema es heterogéneo y no todos los programas se pueden ejecutar en todas las estaciones. Al regresar la respuesta, *remote* elige una de ellas y la configura. Un cambio interesante es que las estaciones "inactivas" retrasen un poco sus respuestas, con un retraso proporcional a la carga actual. De esta forma, la respuesta de la máquina con menor carga llega primero y se selecciona.

La búsqueda de la estación de trabajo es sólo el primer caso. Ahora hay que ejecutar el proceso ahí. El desplazamiento del código es fácil. El truco consiste en configurar el proceso remoto de modo que vea el mismo ambiente que tendría en el caso local, en la **estación de trabajo de origen** y llevar a cabo el cómputo de la misma forma que en el caso local.

Para comenzar, necesita la misma visión del sistema de archivos, el mismo directorio de trabajo y las mismas variables del ambiente (variables del shell), si es que existen.

Después de configurar esto, el programa puede iniciar su ejecución. El problema comienza cuando se ejecuta la primera llamada al sistema; digamos, un READ. ¿Qué debe hacer el núcleo? La respuesta depende en mucho de la arquitectura del sistema. Si el sistema no tiene disco y todos los archivos se localizan en el servidor de archivos, el núcleo puede enviar sólo la solicitud al servidor apropiado, que es lo mismo que hubiera hecho la máquina de origen si el proceso se hubiese ejecutado en ella. Por otro lado, si el sistema tiene discos locales, cada uno con su sistema de archivos, entonces hay que dirigir la solicitud de regreso a la máquina de origen para su ejecución.

Algunas de las llamadas al sistema se deben regresar a la máquina de origen sin hacer distinción alguna; ni siquiera en el caso de que todas las máquinas no posean discos. Por ejemplo, la lectura del teclado y la escritura en la pantalla nunca se pueden ejecutar en la máquina remota. Sin embargo, existen otras llamadas al sistema que se pueden realizar en forma remota bajo todas las condiciones. Por ejemplo, las llamadas SBRK (para ajustar el tamaño del segmento de datos), NICE (para establecer la prioridad de planificación del CPU) y PROFIL (permitir la configuración del contador del programa) de UNIX no se pueden ejecutar en la máquina de origen. Además, todas las llamadas al sistema que soliciten el estado de la máquina deben realizarse en la máquina donde se ejecuta el proceso. Esto incluye el preguntar el nombre de la máquina, su dirección en la red, la memoria disponible, etcétera.

Las llamadas al sistema relacionadas con el tiempo son un problema, puesto que los relojes de las diversas máquinas no tienen que estar sincronizados. En el capítulo 3 vimos lo difícil que es lograr la sincronización. El uso del tiempo de la máquina remota provoca que los programas dependientes del mismo, como *make*, proporcionen resultados incorrectos. Sin embargo, si todas las llamadas relacionadas con el tiempo hacen referencia al tiempo de la máquina de origen, se introduce un retraso, lo que también provoca problemas con el tiempo.

Para complicar aún más las cosas, ciertos casos particulares de llamadas que por lo general deberían hacer referencia al tiempo de la máquina de origen, como la creación y escritura en un archivo temporal, se pueden realizar de manera más eficiente en la máquina remota. Además, el seguimiento del ratón y la propagación de las señales deben pensarse con cuidado. Los programas que escriben de manera directa en dispositivos de hardware, como el buffer del marco para la pantalla, los discos flexibles y las cintas magnéticas no se pueden ejecutar de modo remoto. En resumen, es posible lograr que los programas se ejecuten en las máquinas remotas como si se ejecutaran en las máquinas de origen, pero esto es un asunto complejo y truculento.

La última pregunta de nuestra lista original es ¿qué hacer si regresa el poseedor de la máquina (es decir, alguien entra al sistema o un usuario previamente inactivo toca el teclado o el ratón)? Lo más fácil es no hacer nada, pero esto tiende a destruir la idea de las estaciones de trabajo "personales". Si otras personas intentan ejecutar programas en su estación de trabajo al mismo tiempo en que usted desea utilizarla, se diluye la respuesta garantizada.

Otra posibilidad es eliminar el proceso intruso. La forma más sencilla de lograr esto es hacerlo de manera abrupta y sin previo aviso. La desventaja de esta estrategia es que se perderá todo el trabajo y el sistema de archivos tendrá un estado caótico. Es mejor darle al

proceso una advertencia, mediante una señal que permita detectar una catástrofe inminente y hacer esto con bondad (escribir los buffers editados en un disco, cerrar archivos, etc.) Si no sale después de unos cuantos segundos, es terminado. Por supuesto, hay que escribir el programa de modo que espere y pueda manejar esta señal, algo que no hacen la mayoría de los programas existentes.

Un método por completo distinto es hacer que el proceso emigre a otra máquina, ya sea a la máquina de origen o a alguna otra estación de trabajo inactiva. La migración se realiza poco en la práctica, puesto que el mecanismo real es complejo. La parte difícil no es mover el código y los datos del usuario, sino encontrar y recolectar todas las estructuras de datos del núcleo relativas al proceso de salida. Por ejemplo, podría tener archivos abiertos, ejecutar cronómetros, tener una cola de mensajes recibidos, y otras partes de información diseminadas en el núcleo. Todo esto debe eliminarse con cuidado de la máquina fuente e instalarse con éxito en la máquina destino. No existen problemas teóricos en este aspecto, pero las dificultades prácticas de ingeniería son sustanciales. Para mayor información, véase (Arsty y Finkel, 1989; Douglis y Ousterhout, 1987; y Zayas, 1987).

En ambos casos, al irse el proceso, debe dejar la máquina en el mismo estado en que la encontró, para evitar molestar al poseedor. Entre otros aspectos, esto significa que el proceso no sólo debe irse, sino también sus hijos y los hijos de éstos. Además, hay que eliminar los buzones de correo, conexiones en la red y otras estructuras de datos relativas a todo el sistema; tomar todas las previsiones necesarias para ignorar las respuestas de RPC y otros mensajes que lleguen al proceso después de haberse ido. Si existe un disco local, hay que eliminar los archivos temporales y, de ser posible, restaurar los archivos que hayan sido eliminados de su caché.

4.2.3. El modelo de la pila de procesadores

Aunque el uso de las estaciones de trabajo inactivas añade cierto poder de cómputo al sistema, no enfrenta un aspecto todavía más fundamental: ¿Qué ocurre cuando es posible proporcionar 10 o 100 veces más CPU que el número de usuarios activos? Ya hemos visto una solución, la cual consiste en dar a cada quien un multiprocesador personal. Sin embargo, éste es un diseño algo ineficiente.

Otro método consiste en construir una **pila de procesadores**, repleta de CPU, en el cuarto de las máquinas, los cuales se pueden asignar de manera dinámica a los usuarios según la demanda. El método de la pila de procesadores se muestra en la figura 4-13. En vez de darle a los usuarios estaciones de trabajo personales, en este modelo se les dan terminales gráficas de alto rendimiento, como las terminales X (aunque también las pequeñas estaciones de trabajo se pueden utilizar como terminales). Esta idea se basa en la observación de que lo que en realidad quieren muchos usuarios es una interfaz gráfica de alta calidad y un buen desempeño. Desde un punto de vista conceptual, este método es mucho más parecido al tiempo compartido tradicional que al modelo de la computadora personal, aunque se construye con la tecnología moderna (microprocesador de bajo costo).

La motivación para la idea de la pila de procesadores proviene de dar un paso más adelante en la idea de las estaciones de trabajo sin disco. Si el sistema de archivos se debe

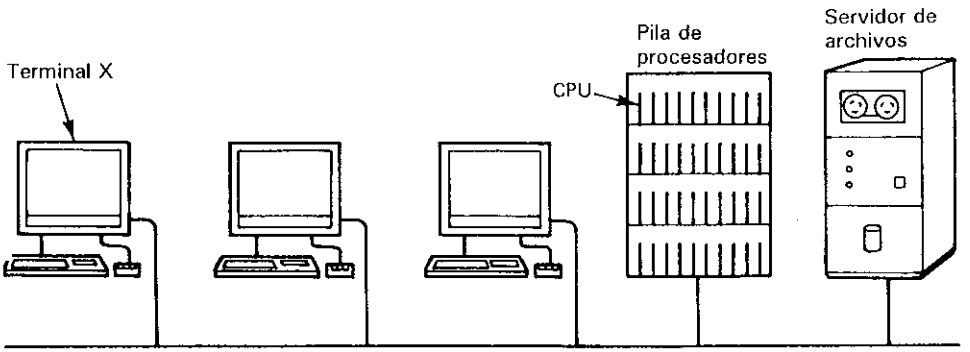


Figura 4-13. Sistema basado en el modelo de la pila de procesadores.

concentrar en un pequeño número de servidores de archivos para mayor economía, debe ser posible hacer lo mismo con los servidores de cómputo. Si colocamos todos los CPU en un gabinete de gran tamaño dentro del cuarto de máquinas, se pueden reducir los costos de suministro de energía y otros costos de empaquetamiento, lo cual produce un mayor poder de cómputo para una cantidad fija de dinero. Además, permite el uso de terminales X más baratas (o incluso terminales ASCII ordinarias) y elimina la asociación entre el número de usuarios y el número de estaciones de trabajo. El modelo también facilita el crecimiento por incrementos. Si la carga de cómputo se incrementa en un 10%, sólo se compra 10% más procesadores y se coloca en la pila.

De hecho, convertimos todo el poder de cómputo en "estaciones de trabajo inactivas" a las que se puede tener acceso de manera dinámica. Los usuarios obtienen tantos CPU como sea necesario, durante períodos cortos, después de lo cual regresan a la pila, de modo que otros usuarios puedan disponer de ellos. En este caso no existe el concepto de propiedad: todos los procesadores pertenecen por igual a todos.

El principal argumento para la centralización del poder de cómputo como pila de procesadores proviene de la teoría de colas. Un sistema de colas es una situación donde los usuarios generan en forma aleatoria solicitudes de trabajo a un servidor. Cuando el servidor está ocupado, los usuarios se forman para el servicio y se procesan según su turno. Algunos de los ejemplos comunes de sistemas de colas son las panaderías, los contadores para ingreso a los aeropuertos, contadores de salida en los supermercados y muchos otros más. Los fundamentos de esto se muestran en la figura 4-14.

Los sistemas de colas son útiles, puesto que es posible modelarlos de manera analítica. Llamemos λ a la tasa de entradas totales de solicitudes por segundo de todos los usuarios combinados. Sea μ la tasa de procesamiento de solicitudes por parte del servidor. Para una operación estable, debemos tener $\mu > \lambda$. Si el servidor puede manejar 100 solicitudes por segundo, pero los usuarios generan de manera continua 110 solicitudes por segundo, la cola crecerá sin límite alguno. (Se pueden permitir pequeños intervalos de tiempo en los que la

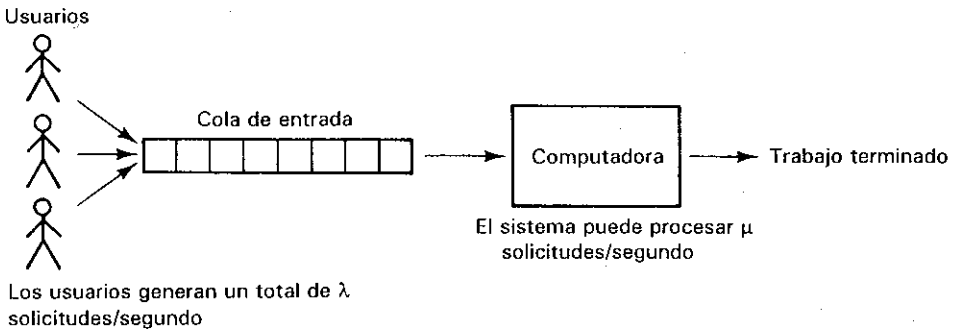


Figura 4-14. Un sistema básico con colas.

tasa de entrada exceda a la tasa de servicio, siempre que la tasa media de entrada sea menor que la tasa de servicio y que exista el espacio suficiente en los buffers.)

Se puede demostrar (Kleinrock, 1974) que T, el promedio de tiempo entre la emisión de una solicitud y la obtención de una respuesta completa está relacionado con λ y μ mediante la fórmula

$$T = \frac{1}{\mu - \lambda}$$

Consideremos, por ejemplo, un servidor de archivos que puede manejar 50 solicitudes/segundo y obtiene 40 respuestas/segundo. El tiempo promedio de respuesta será de 1/10 segundo o 100 milisegundos. Observe que cuando λ tiende a 0 (no existe carga), el tiempo de respuesta del servidor de archivos no tiende a 0, sino a 1/50 segundo o 20 milisegundos. La razón es obvia, una vez que se conoce. Si el servidor de archivos sólo puede procesar 50 solicitudes/segundo, debe tardar 20 milisegundos en procesar cada solicitud, incluso en ausencia de competencia, por lo que el tiempo de respuesta, que incluye el tiempo de procesamiento, nunca puede ser menor de 20 milisegundos.

Supongamos que tenemos n multiprocesadores personales, cada uno con cierto número de CPU y que cada uno forma su sistema de colas, con una tasa de llegada de solicitudes λ y tasa de procesamiento de los CPU μ. El tiempo promedio de respuesta, T, estará dado como antes. Consideremos ahora lo que ocurre si reunimos todos los CPU y los colocamos en una pila de procesadores. En vez de tener n pequeños sistemas de colas ejecutándose en paralelo, ahora sólo tenemos uno grande, con una tasa de entrada $n\lambda$ y una tasa de servicio $n\mu$. Llamemos T_1 al tiempo promedio de respuesta de este sistema combinado. De la fórmula anterior tenemos que

$$T_1 = \frac{1}{n\mu - n\lambda} = T/n$$

Este sorprendente resultado nos dice que si reemplazamos n pequeños recursos por uno grande que sea n veces más poderoso, podemos reducir el tiempo promedio de respuesta n veces.

Este resultado es más general y se aplica a gran variedad de sistemas. Es una de las principales razones por las que las líneas aéreas prefieren volar un 747 de 300 asientos cada 5 horas que volar un jet empresarial de 10 asientos cada 10 minutos. Este efecto surge porque la división del poder de cómputo en servidores pequeños (por ejemplo, las estaciones de trabajo personales), cada una con un usuario, no concuerda con una carga de trabajo consistente con solicitudes que llegan de manera arbitraria. La mayor parte del tiempo, pocos servidores están ocupados, e incluso sobrecargados, pero la mayoría están inactivos. Este tiempo desperdiciado se elimina en el modelo de la pila de procesadores y es la razón de su mejor desempeño general. El concepto de uso de las estaciones de trabajo inactivas es un débil intento por recuperar los ciclos desperdiciados, pero es complejo y tiene muchos problemas, como ya lo hemos visto.

De hecho, este resultado de la teoría de colas es uno de los principales argumentos en contra de los propios sistemas distribuidos. Si se debe elegir entre un CPU centralizado de 1000 MIPS o 100 CPU de uso exclusivo y particular, de 10 MIPS, el tiempo promedio de respuesta del primero será 100 veces mejor, puesto que no se desperdiciarán ciclos. La máquina sólo está inactiva si ningún usuario tiene trabajo que realizar. Este hecho argumenta en favor de la máxima concentración posible del poder de cómputo.

Sin embargo, el tiempo promedio de respuesta no lo es todo. También existen argumentos a favor del cómputo distribuido. El costo es uno de ellos. Si un CPU de 1000 MIPS es mucho más caro que 100 estaciones de trabajo de 10 MIPS, la relación precio/desempeño del segundo caso será mucho mejor. Incluso podría no ser posible construir una máquina tan grande al precio que sea. La confiabilidad y la tolerancia de fallas también son otros factores por considerar.

Además, las estaciones de trabajo tienen una respuesta uniforme, independiente de lo que hagan las demás personas (excepto cuando la red o los servidores de archivos están saturados). Para ciertos usuarios, una ligera variación en el tiempo de respuesta puede ser más importante que el propio tiempo promedio de respuesta. Por ejemplo, consideremos una edición en una estación de trabajo particular, donde la solicitud de exhibición de la siguiente página siempre tarda 500 milisegundos. Consideremos ahora una edición en una enorme computadora, compartida y centralizada, en la que la misma acción tarda 5 milisegundos el 95% del tiempo y 5 segundos una vez por cada 20. Incluso aunque el promedio aquí sea el doble de bueno que en la estación de trabajo, los usuarios podrían considerar intolerable este desempeño. Por otro lado, para el usuario que deba ejecutar una simulación de gran tamaño, la computadora grande gana con los brazos cruzados.

Hasta aquí hemos supuesto de manera implícita que una pila de n procesadores es igual a un procesador, que es n veces más rápido que un procesador. En realidad, esta hipótesis sólo se justifica si todas las solicitudes se pueden dividir de manera que se puedan ejecutar en todos los procesadores en forma paralela. Si un trabajo sólo se puede dividir en, digamos, 5 partes, entonces el modelo de pila de procesadores sólo tiene un tiempo de servicio efectivo 5 veces mejor que el de un procesador y no n veces mejor.

Aún así, el modelo de pila de procesadores es una forma más limpia de obtener un poder de cómputo adicional que la búsqueda de estaciones inactivas o husmear por ahí

mientras nadie observa. Si se parte de la hipótesis de que ningún procesador pertenece a alguien, obtenemos un diseño con base en el concepto de solicitud de las máquinas a la pila, su uso posterior y su regreso al terminar. Tampoco hay necesidad de regresar algo a una máquina de origen, puesto que ésta no existe. Tampoco hay peligro de que el poseedor regrese, puesto que no existen poseedores.

Por último, todo llega hasta la naturaleza misma de la carga de trabajo. Si todas las personas hacen una sencilla edición y de manera ocasional envían uno o dos mensajes de correo electrónico, tal vez sea suficiente con estaciones de trabajo personales. Si, por otro lado, los usuarios están implicados en un gran proyecto de desarrollo de software y ejecutan con frecuencia *make* en grandes directorios, o intentan obtener la inversa de grandes matrices ralas, o llevar a cabo enormes simulaciones, ejecutar grandes programas de inteligencia artificial o de ruteo VLSI, la búsqueda constante de un gran número de estaciones de trabajo no será nada divertido. En todas estas situaciones, la idea de la pila de procesadores es fundamentalmente más sencilla y atractiva.

4.2.4. Un modelo híbrido

Se puede establecer una mediación al proporcionar a cada usuario una estación de trabajo personal y además tener una pila de procesadores. Aunque esta solución es más cara que cualquiera de los dos modelos puros, combina las ventajas de ambos.

El trabajo interactivo se puede llevar a cabo en las estaciones de trabajo, con una respuesta garantizada. Sin embargo, las estaciones inactivas no se utilizan, lo cual hace más sencillo el diseño del sistema. Sólo se dejan sin utilizar. En vez de esto, todos los procesos no interactivos se ejecutan en la pila de procesadores, así como todo el cómputo pesado en general. Este modelo proporciona una respuesta interactiva más rápida, un uso eficiente de los recursos y un diseño sencillo.

4.3. ASIGNACIÓN DE PROCESADORES

Por definición, un sistema distribuido consta de varios procesadores. Éstos se pueden organizar como colección de estaciones de trabajo personales, una pila pública de procesadores o alguna forma híbrida. En todos los casos, se necesita cierto algoritmo para decidir cuál proceso hay que ejecutar y en qué máquina. Para el modelo de estaciones de trabajo, la pregunta es cuándo ejecutar el proceso de manera local y cuándo buscar una estación inactiva. Para el modelo de la pila de procesadores, hay que tomar una decisión por cada nuevo proceso. En esta sección estudiaremos los algoritmos que se utilizan para determinar cuál proceso se asigna a cuál procesador. Seguiremos la tradición y nos referiremos a este tema como "asignación de procesadores" en vez de "asignación de procesos", aunque también se puede analizar desde este punto de vista.

4.3.1. Modelos de asignación

Antes de analizar los algoritmos específicos, o incluso los principios de diseño, es importante decir algo del modelo subyacente, hipótesis y objetivos del trabajo de asignación

de procesadores. Casi todo el trabajo en esta área supone que todas las máquinas son idénticas, o que al menos son compatibles en el código y que difieren a lo más en la velocidad. Un artículo ocasional supone que el sistema consta de varias pilas ajenas de procesadores, cada una de las cuales es homogénea. Estas hipótesis son válidas por lo general y simplifican el problema, pero dejan sin respuesta por el momento preguntas tales como si un comando para iniciar un programa que dé formato a un texto se debe iniciar en una 486, SPARC o MIPS CPU, suponiendo que se dispone de todos los binarios para ello.

Casi todos los modelos publicados suponen que el sistema está por completo interconectado; es decir, que cada procesador se puede comunicar con los demás. Aquí también supondremos esto. Esta hipótesis no quiere decir que cada máquina tenga un cable hacia cualquier otra máquina, sino que se pueden establecer conexiones de transporte entre cualquier pareja de máquinas. El hecho de que los mensajes puedan saltar de una máquina a otra, en una secuencia de máquinas, sólo es de interés para las capas inferiores. Algunas redes soportan la transmisión simple o la multitransmisión y algunos algoritmos utilizan estas capacidades.

Se genera un nuevo trabajo cuando un proceso en ejecución decide realizar una bifurcación o crear un subproceso. En ciertos casos, el proceso que se bifurca es el intérprete de comandos (shell) que inicia un nuevo trabajo en respuesta a un comando del usuario. En otros, el propio proceso usuario crea uno o más hijos; por ejemplo, para tener mejor desempeño con la ejecución en paralelo de todos los hijos.

Las estrategias de asignación de procesadores se pueden dividir en dos categorías amplias. En la primera, que llamaremos **no migratoria**, al crearse un proceso, se toma una decisión acerca de dónde colocarlo. Una vez colocado en una máquina, el proceso permanece ahí hasta que termina. No se puede mover, no importa lo sobrecargada que esté la máquina ni que existan muchas otras máquinas inactivas. Por el contrario, con los algoritmos **migratorios**, un proceso se puede trasladar aunque haya iniciado su ejecución. Mientras que las estrategias migratorias permiten un mejor balance de la carga, son más complejas y tienen un efecto fundamental en el diseño del sistema.

Un algoritmo que asigne procesos a los procesadores lleva implícito el intento por optimizar algo. Si éste no fuera el caso, sólo haríamos la asignación en forma aleatoria o en orden numérico. Sin embargo, lo que hay que optimizar varía de un sistema a otro. Un posible objetivo podría ser maximizar el **uso de los CPU**; es decir, el número de ciclos de CPU que se ejecutan en beneficio de los trabajos del usuario, por cada hora de tiempo real. La maximización del uso del CPU es otra forma de decir que hay que evitar a todo costo el tiempo inactivo del CPU. Cuando exista la duda, hay que garantizar que cada CPU tenga algo que hacer.

Otro objetivo importante es la minimización del **tiempo promedio de respuesta**. Consideremos, por ejemplo, los dos procesos y procesadores de la figura 4-15. El procesador 1 ejecuta 10 MIPS; el procesador 2 ejecuta 100 MIPS, pero tiene una lista de espera de procesos atrasados, la cual tardará 5 segundos en terminar. El proceso *A* tiene 100 millones de instrucciones y el proceso *B* tiene 300 millones. En la figura se muestran los tiempos de respuesta para cada proceso en cada procesador (tiempo de espera incluido). Si asignamos

el proceso *A* al procesador 1 y *B* al procesador 2, el tiempo promedio de respuesta será de (10 + 8)/2 = 9 segundos. Si los asignamos al revés, el tiempo promedio de respuesta será de (30 + 6)/2 = 18 segundos. Es claro que la primera asignación es mejor, en términos de minimizar el tiempo promedio de respuesta.

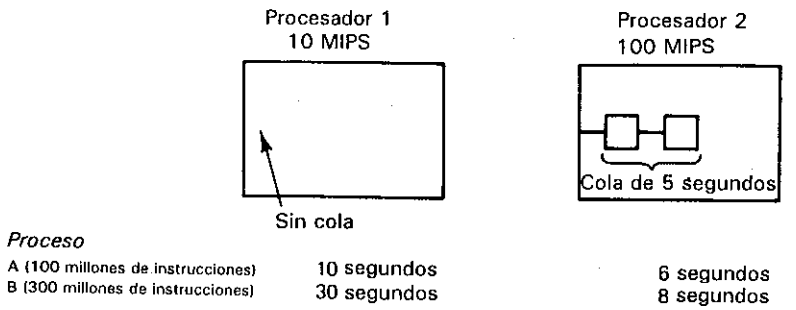


Figura 4-15. Tiempos de respuesta de dos procesos en dos procesadores.

Una variante de la minimización del tiempo de respuesta es la minimización de la **tasa de respuesta**, la cual se define como la cantidad de tiempo necesaria para ejecutar un proceso en cierta máquina, dividido entre el tiempo que tardaría en ejecutarse en cierto procesador de referencia, no cargado. Para muchos usuarios, la tasa de respuesta es una métrica más útil que el tiempo de respuesta, puesto que toma en cuenta el hecho de que los trabajos de gran tamaño tardan más que los pequeños. Para ver esto, ¿cuál es mejor, un trabajo de 1 segundo que tarda 5 segundos o un trabajo de 1 minuto que tarda 70 segundos? Mediante el tiempo de respuesta, el primero es mejor, pero con la tasa de respuesta, el segundo es mucho mejor, puesto que 5/1 >> 70/60.

4.3.2. Aspectos del diseño de algoritmos de asignación de procesadores

Con el paso de los años, se han propuesto un gran número de algoritmos para la asignación de procesadores. En esta sección analizaremos algunas de las opciones claves en estos algoritmos y señalaremos los distintos puntos intermedios. Las principales decisiones que deben tomar los diseñadores se pueden resumir en cinco aspectos:

1. Algoritmos deterministas *vs.* heurísticos.
2. Algoritmos centralizados *vs.* distribuidos.
3. Algoritmos óptimos *vs.* subóptimos.
4. Algoritmos locales *vs.* globales.
5. Algoritmos iniciados por el emisor *vs.* iniciados por el receptor.

También hay que tomar otras decisiones, pero éstas son las principales. Analizaremos cada una de ellas en su turno.

Los algoritmos deterministas son adecuados cuando se sabe de antemano todo acerca del comportamiento de los procesos. Imaginemos que tenemos una lista completa de todos los procesos, sus necesidades de cómputo, de archivos, de comunicación, etc. Con esta información, es posible hacer una asignación perfecta. En teoría, uno podría intentar todas las posibles asignaciones y tomar la mejor.

En pocos, si no es que en ninguno de los sistemas, se tiene un conocimiento total de antemano, pero a veces se puede obtener una aproximación razonable. Por ejemplo, en los bancos, aseguradoras o en las reservaciones de las líneas aéreas, el trabajo de un día es similar al del día anterior. Las líneas aéreas tienen una muy buena idea de la cantidad de personas que desean viajar desde Nueva York hasta Chicago, el lunes por la mañana a principios de la primavera, de modo que la naturaleza de la carga de trabajo se puede caracterizar con cierta precisión, al menos de forma estadística, lo que posibilita el uso de los algoritmos de asignación determinista.

En el otro extremo están los sistemas donde la carga es por completo impredecible. Las solicitudes de trabajo dependen de quién esté haciendo qué, y puede variar de manera drástica cada hora, e incluso cada minuto. La asignación de procesadores en tales sistemas no se puede hacer de manera determinista o matemática, sino que por necesidad utiliza técnicas *ad hoc* llamadas **heurísticas**.

El segundo aspecto del diseño es centralizado *vs.* distribuido. Este tema ha aparecido varias veces en este libro. La recolección de toda la información en un lugar permite tomar una mejor decisión, pero menos robusta y coloca una carga pesada en la máquina central. Son preferibles los algoritmos descentralizados, pero se han propuesto algunos algoritmos centralizados por la carencia de alternativas descentralizadas adecuadas.

El tercer aspecto está relacionado con los dos anteriores: ¿Intentamos encontrar la mejor asignación, o sólo una que sea aceptable? Se pueden obtener las soluciones óptimas tanto en los sistemas centralizados como en los descentralizados, pero por regla son más caros que los subóptimos. Hay que recolectar más información y procesarla un poco más. En la práctica, la mayoría de los sistemas distribuidos reales buscan soluciones subóptimas, heurísticas y distribuidas, debido a la dificultad para obtener las óptimas.

El cuarto aspecto se relaciona con lo que se llama a menudo **política de transferencia**. Cuando se está a punto de crear un proceso, hay que tomar una decisión para ver si se ejecuta o no en la máquina que lo genera. Si esa máquina está muy ocupada, hay que transferir a otro lugar al nuevo proceso. La opción en este aspecto consiste en basar o no la decisión de transferencia por completo en la información local. Una escuela de pensamiento prefiere un algoritmo (local) sencillo: si la carga de la máquina está por debajo de cierta marca, se conserva al nuevo proceso; en caso contrario, se deshace de él. Otra escuela dice que esta heurística es muy cruda. Es mejor recolectar información (global) acerca de la carga antes de decidir si la máquina local está o no muy ocupada para otro proceso. Cada una de estas opciones tiene sus puntos a favor. Los algoritmos locales son sencillos, pero están muy lejos de ser los óptimos, mientras que los globales sólo dan un resultado poco mejor a un costo mayor.

El último aspecto de nuestra lista trata de la **política de localización**. Una vez que la política de transferencia ha decidido deshacerse de un proceso, la política de localización debe decidir dónde enviarlo. Es claro que esta política no puede ser local. Necesita información de la carga en todas partes para tomar una decisión inteligente. Sin embargo, esta información se puede dispersar de dos maneras. En uno de los métodos, los emisores inician el intercambio de información. En el otro, el receptor toma la iniciativa.

Como ejemplo sencillo, observemos la figura 4-16(a). En ésta, una máquina sobrecargada envía una solicitud de ayuda a las demás máquinas, con la esperanza de que descarguen el nuevo proceso en alguna otra máquina. En este ejemplo, el emisor toma la iniciativa para localizar más ciclos del CPU. Por el contrario, en la figura 4-16(b), una máquina inactiva o subcargada anuncia a las demás que tiene poco trabajo y está preparada para más. Su objetivo es localizar una máquina dispuesta a darle trabajo.

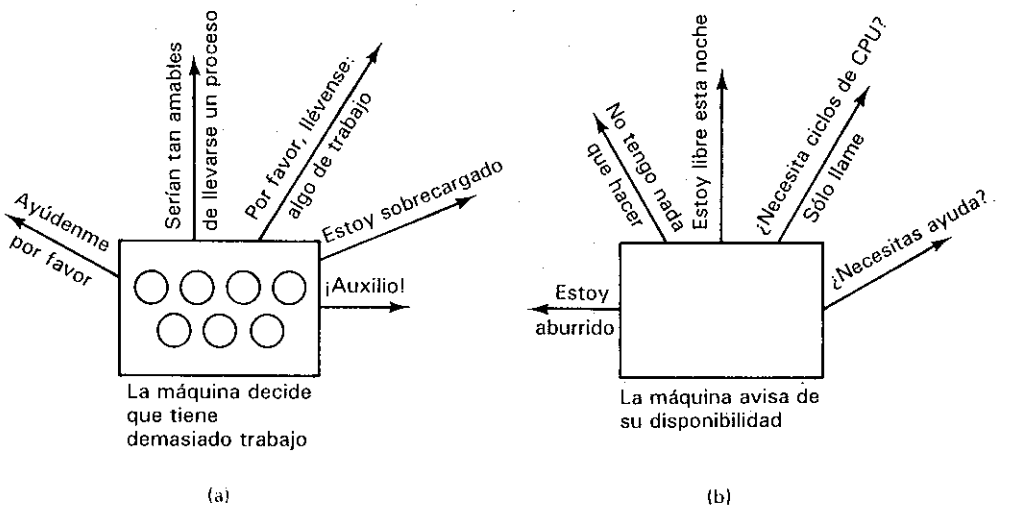


Figura 4-16. (a) Un emisor en búsqueda de una máquina inactiva. (b) Un receptor en búsqueda de trabajo por realizar.

Para ambos casos, el de los procesos iniciados por el emisor o por el receptor, los distintos algoritmos tienen estrategias diversas para decidir a quién examinar, el tiempo que durará dicho examen y qué hacer con los resultados. Sin embargo, en estos momentos debe quedar clara la diferencia entre ambos enfoques.

4.3.3. Aspectos de la implantación de algoritmos de asignación de procesadores

Todos los puntos señalados en la sección anterior son aspectos con claro corte teórico que podrían debatirse de manera ilimitada. En esta sección analizaremos otros aspectos,

más relacionados con los detalles de la implantación real de los algoritmos para la asignación de procesadores que con los grandes principios detrás de ellos.

Para comenzar, casi todos los algoritmos suponen que las máquinas conocen su carga, de modo que pueden decir si están subcargados o sobrecargados y pueden informar a las demás máquinas de su estado. La medición de la carga no es tan sencilla como parece. Un método consiste en contar el número de procesos en cada máquina y utilizar ese número como la carga. Sin embargo, como ya hemos señalado antes, incluso en un sistema inactivo pueden ejecutarse muchos procesos, como los demonios de correo o noticias, administradores de ventanas y otros. Así, el contador del proceso casi no dice nada de la carga actual.

El siguiente paso consiste en contar sólo los procesos en ejecución o listos. Después de todo, cada proceso en ejecución o que se pueda ejecutar impone cierta carga a la máquina, incluso aunque sea un proceso secundario. Sin embargo, muchos de estos demonios despiertan de forma periódica, verifican si ocurre algo interesante y, en caso contrario, vuelven a dormir. La mayoría sólo pone una pequeña carga en el sistema.

Una medida más directa, aunque requiere de un mayor trabajo para su registro, es la fracción de tiempo que el CPU está ocupado. Es claro que una máquina con 20% de uso del CPU tiene una carga mayor que la de una máquina con 10% de uso del CPU, sin importar si ejecuta programas del usuario o demonios. Una forma de medir el uso del CPU es configurar un cronómetro y dejarlo que interrumpa a la máquina en forma periódica. En cada interrupción, se observa el estado del CPU. De esta forma, se puede observar la fracción de tiempo que se gasta en el ciclo inactivo.

Un problema con las interrupciones por medio de cronómetros es que cuando el núcleo ejecuta un código crítico, éste desactiva por lo general todas las interrupciones, entre las cuales se encuentra la interrupción del cronómetro. Así, si el tiempo del cronómetro se termina mientras el núcleo está activo, la interrupción se retrasa hasta que el núcleo termina. Si el núcleo estaba en el proceso de bloquear los últimos procesos activos, el tiempo del cronómetro no se agotará sino hasta que el núcleo termine (y entre al ciclo inactivo). Este efecto tiende a subestimar el verdadero uso del CPU.

Otro aspecto de la implantación es el enfrentamiento con el costo excesivo. Muchos de los algoritmos teóricos para la asignación de procesos ignoran el costo de recolectar medidas y desplazar los procesos de aquí para allá. Si un algoritmo descubre que el traslado de un proceso recién creado a una máquina distante puede mejorar el desempeño del sistema en un 10%, tal vez sería mejor no hacer nada, puesto que el costo del traslado del proceso puede engullirse todo el beneficio. Un algoritmo adecuado tomaría en cuenta el tiempo de CPU, uso de memoria y el ancho de banda de la red utilizada por el propio algoritmo para asignación de procesadores. Pocos lo hacen, lo cual se debe principalmente a que no es fácil.

Nuestra siguiente consideración en torno a la implantación es la complejidad. Casi todos los investigadores miden la calidad de sus algoritmos mediante el análisis de datos analíticos, simulados o experimentales del uso del CPU y de la red, así como el tiempo de respuesta. Pocas veces se considera también la complejidad del software en cuestión, a pesar de las obvias implicaciones para el desempeño, precisión y consistencia del sistema.

Rara vez ocurre que alguien escriba un nuevo algoritmo, demuestra lo bueno que es su desempeño y después concluye que el algoritmo no tiene importancia, debido a que su desempeño es tan sólo un poco mejor que los algoritmos ya existentes pero que es mucho más complejo de implantar (o tiene una ejecución más lenta).

A este respecto, un estudio de Eager *et al.* (1986) arroja cierta luz en el tema de la búsqueda de algoritmos complejos y óptimos. Ellos estudiaron tres algoritmos. En todos los casos, cada máquina del sistema mide su carga y decide por sí misma si está subcargada. Al crearse un nuevo proceso, la máquina que lo crea verifica si está sobrecargada. En tal caso, verifica una máquina remota en la cual iniciar el nuevo proceso. Los tres algoritmos difieren en la localización de la máquina candidato.

El algoritmo 1 elige una máquina de manera aleatoria y tan sólo envía ahí al nuevo proceso. Si la propia máquina receptora está sobrecargada, elige una máquina al azar y le envía el proceso. Esto se repite hasta que alguien está dispuesto a tomar el proceso o que se exceda un contador de tiempo, en cuyo caso ya no se permite que avance.

El algoritmo 2 elige una máquina en forma aleatoria y le envía una prueba para ver si está subcargada o sobrecargada. Si la máquina admite estar subcargada, obtiene el nuevo proceso; en caso contrario se repite la prueba. Este ciclo se repite hasta que se encuentra una máquina adecuada o se excede el número de pruebas, en cuyo caso permanece en el sitio de su creación.

El algoritmo 3 analiza k máquinas para determinar sus cargas exactas. El proceso se envía entonces a la máquina con la carga más pequeña.

De forma intuitiva, si ignoramos el costo excesivo de las pruebas y las transferencias de procesos, uno esperaría que el algoritmo 3 tuviese el mejor desempeño, lo cual en realidad ocurre. Sin embargo, la ganancia en desempeño del algoritmo 3 sobre el algoritmo 2 es muy pequeña, aunque la complejidad y cantidad del trabajo adicional necesarios son más grandes. Eager *et al.* concluyen que si el uso de un algoritmo sencillo proporciona casi la misma ganancia que uno más caro y más complejo, es mejor utilizar el más sencillo.

Nuestro última observación en esta sección es que la estabilidad también es un aspecto importante. Las diversas máquinas ejecutan sus algoritmos en forma asíncrona uno del otro, de modo que, desde el punto de vista práctico, el sistema nunca alcanza el equilibrio. Es posible llegar a situaciones donde ni A ni B tienen la información actualizada y cada uno de ellos piensa que el otro tiene carga menor, lo cual provoca que un pobre proceso sea enviado de ida y regreso un gran número de veces. El problema es que la mayoría de los algoritmos que intercambian información son correctos después de intercambiar la información y que todo se ha asentado, pero se puede decir muy poco de su operación mientras las tablas continúan su actualización. Es en estas situaciones de no equilibrio que surgen a menudo problemas inesperados.

4.3.4. Ejemplo de algoritmos de asignación de procesadores

Para tener idea de cómo se lleva a cabo en realidad la asignación de procesadores, en esta sección analizaremos varios algoritmos diferentes, los cuales han sido seleccionados para cubrir un amplio margen de posibilidades, pero existen otros más.

Un algoritmo determinista según la teoría de gráficas

Una clase de algoritmos analizada con amplitud es para los sistemas que constan de procesos con requerimientos conocidos de CPU y memoria, además de una matriz conocida con el tráfico promedio entre cada pareja de procesos. Si el número k de CPU es menor que el número de procesos, habrá que asignar varios procesos al mismo CPU. La idea es llevar a cabo esta asignación de forma que se minimice el tráfico en la red.

El sistema se puede representar como gráfica con pesos, donde cada nodo es un proceso y cada arco representa el flujo de mensajes entre dos procesos. Desde el punto de vista matemático, el problema se reduce entonces a encontrar una forma de partir (es decir, cortar) la gráfica en dos subgráficas ajenas, sujetas a ciertas restricciones (por ejemplo, el total de requerimientos de CPU y memoria deberá estar por debajo de ciertos límites para cada subgráfica). Para cada solución que cumpla las restricciones, los arcos contenidos por completo dentro de una subgráfica representan la comunicación entre las máquinas y se puede ignorar. Los arcos que van de una subgráfica a la otra representan el tráfico en la red. El objetivo es entonces encontrar una partición que minimice el tráfico en la red, a la vez que satisfaga todas las restricciones. La figura 4-17 muestra dos formas de partir la misma gráfica, lo cual produce dos cargas distintas en la red.

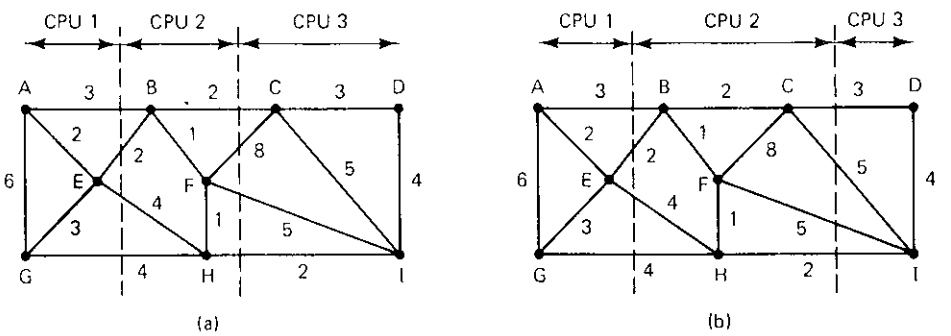


Figura 4-17. Dos formas de asignar 9 procesos a 3 procesadores.

En la figura 4-17(a) hemos partido la gráfica con los procesos A, E y G en un procesador; los procesos B, F y H en un segundo; los procesos C, D e I en el tercero. El tráfico total en la red es la suma de los arcos intersecados por las líneas punteadas, que en este caso es 30 unidades. En la figura 4-17(b) tenemos una partición distinta, la cual sólo tiene 28 unidades de tráfico en la red. Si suponemos que cumple con todas las restricciones de memoria y CPU, ésta es una mejor opción puesto que utiliza menor comunicación.

De manera intuitiva, lo que hacemos es buscar unidades de asignación fuertemente acopladas (flujo intenso de tráfico dentro de las unidades de asignación), pero que interactúen poco con las demás unidades (poco flujo de tráfico entre las unidades). Algunos de los artículos que analizan el problema son (Chow y Abraham, 1982; Stone y Bokhari, 1978; y Lo, 1984).

Un algoritmo centralizado

Los algoritmos de teoría de gráficas del tipo analizado tienen poca aplicabilidad, puesto que necesitan información completa de antemano, por lo que veremos ahora un algoritmo heurístico que no necesita dicha información. Este algoritmo, llamado **arriba-abajo** (Mutka y Livny, 1987) es centralizado, en el sentido de que un coordinador mantiene una **tabla de uso**, con una entrada por cada estación de trabajo personal (es decir, por usuario), con un valor inicial de 0. Cuando ocurren eventos significativos, se pueden enviar mensajes al coordinador para actualizar la tabla. Las decisiones de asignación se basan en esta tabla. Estas decisiones se toman cuando ocurren eventos de planificación: se realiza una solicitud, se libera un procesador, o bien el reloj hace una marca de tiempo.

La parte poco común de este algoritmo y la razón de ser centralizado es que en vez de intentar maximizar el uso del CPU, se preocupa por darle a cada poseedor de una estación de trabajo una parte justa del poder de cómputo. Mientras que otros algoritmos pueden otorgarle todas las máquinas a un usuario, con la única condición de que las mantenga ocupadas (es decir, lograr un alto uso del CPU), este algoritmo está diseñado para evitar eso precisamente.

Cuando se va a crear un proceso y la máquina donde se crea decide que el proceso se debe ejecutar en otra parte, le pide al coordinador de la tabla de usos que le asigne un procesador. Si existe uno disponible y nadie más lo desea, se otorga el permiso. Si no existen procesadores libres, la solicitud se niega por el momento y se toma nota de ella.

Cuando el poseedor de una estación de trabajo ejecuta procesos en las máquinas de otras personas, acumula puntos de penalización, un número fijo por cada segundo, como se muestra en la figura 4-18. Estos puntos se añaden a su entrada en la tabla de usos. Cuando tiene solicitudes pendientes no satisfechas, los puntos de penalización se restan de su entrada en la tabla de usos. Si no existen solicitudes pendientes y ningún procesador está en uso, la entrada de la tabla de usos se desplaza un cierto número de puntos hacia el cero, hasta que llega ahí. De esta forma, su puntuación se mueve hacia arriba o hacia abajo; de ahí el nombre del algoritmo.

Las entradas de la tabla de usos pueden ser positivas, cero o negativas. Una puntuación positiva indica que la estación de trabajo es un usuario de los recursos del sistema, mientras que uno negativo significa que necesita recursos. Una puntuación 0 es neutra.

Podemos dar ahora la heurística utilizada para la asignación de procesadores. Cuando un procesador se libera, gana la solicitud pendiente cuyo poseedor tiene la puntuación más baja. En consecuencia, un usuario que no ocupe procesadores y que tenga pendiente una solicitud durante mucho tiempo siempre vencerá a alguien que utilice muchos procesadores. Esta propiedad es la intención del algoritmo: asignar la capacidad de manera justa.

En la práctica, esto quiere decir que si un usuario tiene carga justa y continua en el sistema, pero otro usuario llega y desea iniciar un proceso, el usuario ligero será favorecido, por encima del usuario pesado. Estudios de simulación (Mutka y Livny, 1987) muestran que el algoritmo funciona como se esperaba bajo una variedad de condiciones de carga.

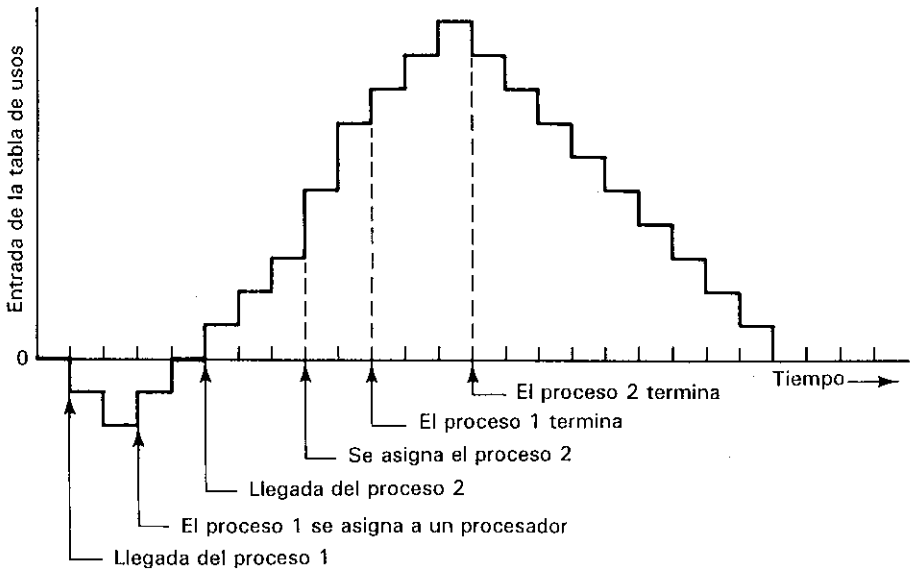


Figura 4-18. Operación del algoritmo arriba-abajo.

Un algoritmo jerárquico

Los algoritmos centralizados, como el de arriba-abajo, no se adaptan bien a los sistemas de gran tamaño. El nodo central se convierte muy rápido en un cuello de botella, por no mencionar la existencia de un punto de falla. Estos problemas se pueden atacar mediante un algoritmo jerárquico en vez de uno centralizado. Los algoritmos jerárquicos mantienen algo de la sencillez de los centralizados, pero se escalan mejor.

Un método propuesto para etiquetar una colección de procesadores es organizarlos mediante una jerarquía lógica, independiente de la estructura física de la red, como en MICROS (Wittie y van Tilborg, 1980). Este método organiza las máquinas como las personas en jerarquías corporativas, militares, académicas y otras del mundo real. Algunas de las máquinas son trabajadores y otras son administradores.

Para cada grupo de k trabajadores, una máquina administrador (el "jefe de departamento") tiene la tarea de mantener un registro de las máquinas ocupadas y las inactivas. Si el sistema es grande, existirá gran número de jefes de departamento, por lo que algunas máquinas funcionarán como "coordinadores", cada uno de los cuales estará por encima de cierto número de jefes de departamento. Si existen muchos coordinadores, éstos también se pueden organizar en forma jerárquica y un "gran jefe" puede controlar una colección de coordinadores. Esta jerarquía se puede extender hasta el infinito, donde el número de niveles necesarios crece en forma logarítmica con el número de trabajadores. Puesto que cada

procesador sólo necesita comunicarse con un superior y unos cuantos subordinados, se puede controlar el flujo de información.

Una pregunta obvia es: ¿Qué ocurre si un jefe de departamento, o peor aún, un gran jefe detiene su funcionamiento? Una respuesta sería promover a uno de los subordinados directos del administrador faltante para que llene el hueco del jefe. La elección se puede llevar a cabo por los propios subordinados, por los compañeros del difunto o, en un sistema más autocrático, por el jefe del administrador enfermo.

Para evitar tener un administrador (vulnerable) en la parte superior del árbol, uno puede truncar éste en su parte superior y tener un comité como la última autoridad, como se muestra en la figura 4-19. Cuando un miembro del comité empieza a fallar, los demás miembros promueven a alguien del nivel inmediato inferior para el reemplazo.

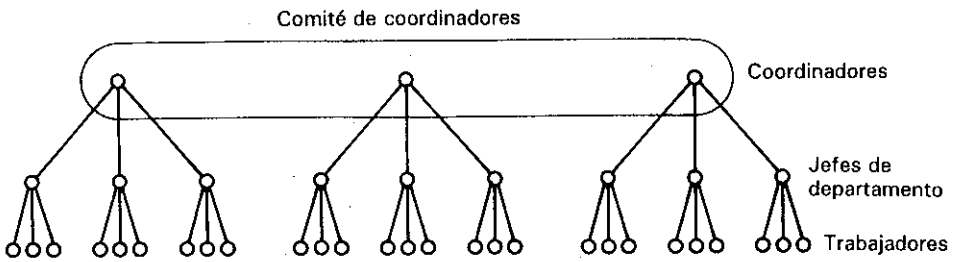


Figura 4-19. Una jerarquía de procesadores se puede modelar como una jerarquía organizacional.

Aunque este esquema no es en realidad distribuido, es factible, y en la práctica funciona bien. En particular, el sistema se puede reparar a sí mismo y puede sobrevivir a fallas ocasionales de los trabajadores y de los administradores, sin efectos a largo plazo.

En MICROS, los procesadores son monoprogramados, de modo que si de pronto aparece una tarea que necesite S procesos, el sistema debe asignarle S procesadores. Las tareas se pueden crear en cualquier parte de la jerarquía. La estrategia que se utiliza es que cada administrador mantenga un registro aproximado del número de trabajadores a su cargo que estén disponibles (es posible que estén varios niveles por debajo de él). Si cree que existe un número suficiente de trabajadores disponibles, reserva cierto número R de ellos, donde $R \geq S$, puesto que la estimación de los trabajadores disponibles podría no ser exacta y algunas máquinas podrían no funcionar.

Si el administrador que recibe la solicitud piensa que tiene pocos procesadores disponibles, transfiere la solicitud hacia arriba, a su jefe. Si el jefe tampoco la puede manejar, la solicitud se sigue propagando hacia arriba, hasta que alcanza un nivel donde tiene un número suficiente de trabajadores a su disposición. En ese momento, el administrador divide la solicitud en partes y las esparce entre los administradores por debajo de él, los cuales a su vez repiten la operación, hasta que la ola de asignación llega al punto inferior. En este nivel

inferior, los procesadores se señalan como "ocupados" y el número de procesadores asignados se informa de regreso hacia arriba del árbol.

Para que esta estrategia funcione bien, *R* debe ser lo bastante grande como para que sea alta la probabilidad de encontrar el número suficiente de trabajadores para manejar todo el trabajo. De otro modo, la solicitud tendrá que desplazarse de nuevo un nivel hacia arriba en el árbol y comenzar de nuevo, lo que desperdicia una cantidad considerable de tiempo y poder de cómputo. Por otro lado, si *R* es muy grande, se podrían asignar demasiados procesadores, lo que desperdiciaría la capacidad de cómputo, mientras la palabra regresa a la parte superior y dichos procesadores sean liberados.

Esta situación se complica mucho por el hecho de que las solicitudes de procesadores se pueden generar de manera aleatoria en cualquier parte del sistema, por lo que en cualquier instante, es probable que varias solicitudes estén en diversas etapas del algoritmo de asignación, lo cual puede conducir a estimaciones no actualizadas del número de trabajadores disponibles, así como a condiciones de competencia, bloqueos y otros problemas más. En Van Wilborg y Wittie (1981) se da un análisis matemático del problema y se abarcan con detalle varios otros aspectos no descritos aquí.

Un algoritmo heurístico distribuido iniciado por el emisor

Los algoritmos anteriores son todos centralizados o semicentralizados. También existen algoritmos distribuidos. Los ejemplos típicos son los descritos por Eager *et al.* (1986). Como ya se mencionó antes, en los algoritmos con mayor efecto en el costo estudiados por ellos, al crearse un proceso, la máquina donde se origina envía mensajes de prueba a una máquina elegida al azar, para preguntar si su carga está por debajo de cierto valor de referencia. En caso afirmativo, el proceso se envía a ese lugar. Si no, se elige otra máquina para la prueba. Las pruebas no se realizan por siempre. Si no se encuentra una máquina adecuada después de *N* pruebas, el algoritmo termina y el proceso se ejecuta en la máquina de origen.

Se ha construido e investigado un modelo analítico de este algoritmo con colas. Mediante este modelo, se estableció que el algoritmo funciona bien y es estable en un amplio margen de parámetros, incluidos diferentes valores de referencia, costos de transferencia y límites de las pruebas.

Sin embargo, hay que observar que bajo condiciones de carga pesada, todas las máquinas enviarán pruebas de manera constante a las demás, en un vano intento por encontrar alguna dispuesta a aceptar más trabajo. Pocos procesos estarán subcargados, pero el intento de esto produciría un costo excesivo.

Un algoritmo heurístico distribuido iniciado por el receptor

Un algoritmo complementario al anterior, iniciado por un emisor sobrecargado, es aquel iniciado por un receptor subcargado. Con este algoritmo, cuando un proceso termina, el sistema verifica si tiene el trabajo suficiente. En caso contrario, elige alguna máquina al

azar y le solicita trabajo. Si esa máquina no tiene nada que ofrecer, se le solicita a una segunda, o a una tercera máquina. Si no encuentra trabajo en *N* pruebas, el receptor deja de preguntar durante cierto tiempo, realiza el trabajo que esté formado en su cola de trabajos e intenta de nuevo, cuando termina el siguiente proceso. Si no hay trabajo disponible, la máquina queda inactiva. Después de cierto lapso, vuelve a probar.

Una ventaja de este algoritmo es que no coloca una carga adicional en el sistema en los tiempos críticos. El algoritmo iniciado por el emisor crea una gran cantidad de pruebas precisamente cuando el sistema puede tolerarlas menos (cuando tiene carga pesada). Con el algoritmo iniciado por el receptor, cuando el sistema tiene carga pesada, la probabilidad de que una máquina tenga trabajo insuficiente es pequeña, pero cuando esto ocurre, será fácil encontrar alguien que se ocupe del trabajo. Por supuesto, cuando hay poco trabajo, el algoritmo iniciado por el receptor crea considerable tráfico de pruebas, ya que todas las máquinas desempleadas buscan un trabajo. Sin embargo, es mejor tener un costo excesivo cuando el sistema está subcargado que cuando está sobrecargado.

También es posible combinar ambos algoritmos y hacer que las máquinas intenten deshacerse del trabajo cuando tienen demasiado y traten de conseguirlo cuando no tienen suficiente. Además, las máquinas podrían realizar una encuesta aleatoria que contenga una historia de las pruebas anteriores para determinar si algunas máquinas están crónicamente subcargadas o sobrecargadas. Se podría intentar con alguna de éstas primero, según si el iniciador quiere deshacerse o hacerse de trabajo.

Un algoritmo de remates

Otra clase de algoritmos intenta convertir el sistema de cómputo en una economía en miniatura, con compradores y vendedores de servicios, además de precios establecidos por la oferta y la demanda (Ferguson *et al.*, 1988). Los actores clave de la economía son los procesos, los cuales deben comprar tiempo de CPU para terminar su trabajo, así como los procesadores, que venden sus ciclos al mejor postor.

Cada procesador anuncia su precio aproximado a través de un archivo que todos pueden leer. Este precio no es garantizado, pero da una indicación de lo que vale el servicio (en realidad es el precio pagado por el último cliente). Los distintos procesadores pueden tener distintos precios, según su velocidad, tamaño de memoria, presencia de hardware de punto flotante y otras características. También se puede publicar una indicación de los servicios prestados, como el tiempo esperado de respuesta.

Cuando un proceso desea iniciar un proceso hijo, verifica si alguien ofrece el servicio que necesita. Entonces determina el conjunto de procesadores que pueden proporcionar sus servicios. De este conjunto, obtiene el mejor candidato, donde la palabra "mejor" puede indicar al más barato, el más rápido, o la mejor relación precio/desempeño, según el tipo de aplicación. Después genera una oferta y envía ésta a su primera opción. La oferta puede ser mayor o menor que el precio anunciado.

Los procesadores reúnen todas las ofertas enviadas a ellos y eligen una, tal vez la mayor de todas. Se informa a los ganadores y los perdedores y se ejecuta el proceso ganador. El precio anunciado se actualiza entonces para reflejar la nueva tasa.

Aunque Ferguson *et al.* no entran en detalles, este tipo de modelo económico hace que surjan gran número de preguntas interesantes: ¿De dónde obtienen los procesos el dinero para hacer sus ofertas? ¿Tienen un salario regular? ¿Tienen todos el mismo salario mensual, o los coordinadores ganan más que los profesores, los cuales a su vez ganan más que los estudiantes? Si entran más usuarios al sistema sin el correspondiente aumento en los recursos, ¿se elevan los precios (inflación)? ¿Pueden formar cárteles los procesadores para extorsionar a los usuarios? ¿Se permiten los sindicatos de usuarios? ¿También se puede cargar a una cuenta el espacio en disco? ¿Qué hay de la salida en una impresora láser? La lista es infinita.

4.4. PLANIFICACIÓN EN SISTEMAS DISTRIBUIDOS

No hay mucho que decir de la planificación en los sistemas distribuidos. Por lo general, cada procesador hace su planificación local (si tiene varios procesos en ejecución), sin preocuparse por lo que hacen los demás procesadores. Lo normal es que este método funcione. Sin embargo, si un grupo de procesos relacionados entre sí y con gran interacción se ejecutan en distintos procesadores, la planificación independiente no es el camino más eficiente.

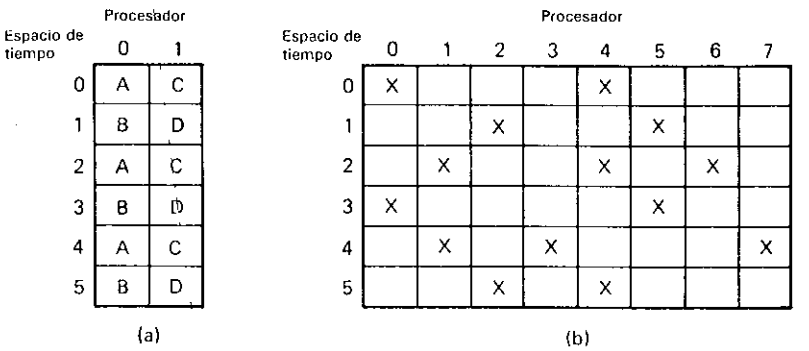


Figura 4-20. (a) Dos procesos que se ejecutan en forma desfasada entre ellos. (b) Matriz de planificación para ocho procesadores, cada uno con seis espacios de tiempo. Las X indican los espacios asignados.

La dificultad básica se puede mostrar mediante un ejemplo, en el cual los procesos *A* y *B* se ejecutan en un procesador y los procesos *C* y *D* en otro. El tiempo de cada procesador se comparte en pedazos de 100 milisegundos, donde *A* y *C* se ejecutan en los pedazos pares y *B* y *D* en los nones, como se muestra en la figura 4-20(a). Supongamos que *A* envía muchos mensajes o lleva a cabo muchas llamadas a procedimientos remotos de *D*. Durante el tiempo 0, *A* inicia y llama de inmediato a *D*, que por desgracia no se ejecuta en ese momento, puesto que es el turno de *C*. Después de 100 milisegundos, se alternan los procesos, *D* obtiene el mensaje de *A*, lleva a cabo el trabajo y responde con rapidez. Puesto

que *B* está ejecutándose, pasarán otros 100 milisegundos antes de que *A* obtenga la respuesta y pueda proseguir. El resultado neto es un intercambio de mensajes cada 200 milisegundos. Lo que se necesita es una forma de garantizar que los procesos con comunicación frecuente se ejecuten de manera simultánea.

Aunque es difícil determinar en forma dinámica los patrones de comunicación entre los procesos, en muchos casos, un grupo de procesos relacionados entre sí iniciarán juntos. Por ejemplo, en general está bien suponer que los filtros de un entubamiento en UNIX se comunicarán entre sí más de lo que lo harán con otros procesos ya iniciados. Supongamos que los procesos se crean en grupos y que la comunicación dentro de los grupos prevalece sobre la comunicación entre los grupos. Supongamos además que se dispone de un número de procesadores lo bastante grande como para manejar al grupo de mayor tamaño y que cada procesador se multiprograma con *N* espacios para los procesos (multiprogramación de nivel *N*).

Ousterhout (1982) propuso varios algoritmos con base en un concepto llamado **coplanificación**, el cual toma en cuenta los patrones de comunicación entre los procesos durante la planificación para garantizar que todos los miembros de un grupo se ejecuten al mismo tiempo. El primer algoritmo utiliza una matriz conceptual, en la que cada columna es la tabla de procesos de un procesador, como se muestra en la figura 4-20(b). Así, la columna 4 consta de todos los procesos que se ejecutan en el procesador 4. El renglón 3 es la colección de todos los procesos que se encuentran en el espacio 3 de algún procesador, a partir del proceso en el espacio 3 del procesador 0, el proceso en el espacio 3 del procesador 1, etc. La esencia de esta idea es que cada procesador utilice un algoritmo de planificación round robin, donde todos los procesadores ejecuten el proceso en el espacio 0 durante un cierto período fijo, para que después todos los procesadores ejecuten el proceso del espacio 1 durante un cierto período fijo, etc. Se puede utilizar un mensaje para indicarle a cada procesador el momento en que debe hacer un intercambio de procesos, para mantener sincronizados los intervalos de tiempo.

Si todos los miembros de un grupo se colocan en el mismo número de espacio, pero en procesadores distintos, se tiene la ventaja del paralelismo de nivel *N*, con la garantía de que todos los procesos se ejecutarán al mismo tiempo, lo cual maximiza el desempeño de la comunicación. Así, en la figura 4-20(b), los cuatros procesos que se deben comunicar entre sí tendrían que colocarse en el espacio 3 de los procesadores 1, 2, 3 y 4, para obtener un desempeño óptimo. Esta técnica de planificación se puede combinar con el modelo jerárquico de administración de procesos utilizado en MICROS al hacer que cada jefe de departamento mantenga la matriz de sus trabajadores, asignando procesos a los espacios en la matriz y transmitiendo las señales de tiempo.

Ousterhout también describió algunas variantes de este método básico para mejorar el desempeño. Una de estas variantes separa la matriz por renglones y concatena los renglones para formar un gran renglón. Con *k* procesadores, cualesquiera *k* entradas consecutivas pertenecen a distintos procesadores. Para asignar un nuevo grupo de procesos a las entradas, se deja una ventana de *k* entradas de ancho en el renglón de gran tamaño, de modo que la entrada del extremo izquierdo esté vacía pero que la entrada justo a la izquierda de la ventana

esté ocupada. Si existe el número suficiente de entradas en dicha ventana, los procesos se asignan a las entradas vacías; o bien, la ventana se desliza a la derecha y se repite el algoritmo. La planificación se lleva a cabo al iniciar la ventana en la orilla izquierda y moviéndola a la derecha una ventana por cada intervalo de tiempo, teniendo cuidado de no dividir los grupos en las ventanas. El artículo de Ousterhout analiza éstos y otros métodos con más detalle y da algunos resultados relativos al desempeño.

4.5. TOLERANCIA DE FALLAS

Decimos que un sistema falla cuando no cumple su especificación. En algunos casos, como en un sistema de ordenamiento distribuido en un supermercado, una falla podría provocar la falta de frijoles enlatados en una tienda. En otros casos, como en un sistema distribuido para el control de tráfico aéreo, una falla podría ser catastrófica. Como las computadoras y los sistemas distribuidos se utilizan cada vez más en misiones donde la seguridad es crítica, la necesidad de evitar las fallas es cada vez mayor. En esta sección examinaremos algunos aspectos relativos a las fallas del sistema y cómo evitarlas. Se puede encontrar material introductorio adicional en (Cristian, 1991; y Nelson, 1990). Gantenbein (1992) ha recopilado una bibliografía sobre el tema.

4.5.1. Fallas de componentes

Los sistemas de cómputo pueden fallar debido a una falla en algún componente, como procesador, la memoria, un dispositivo de E/S, un cable o el software. Una **falla** es un desperfecto, causado tal vez por un error de diseño, un error de fabricación, un error de programación, un daño físico, el deterioro con el curso de tiempo, condiciones ambientales adversas (pudo nevar sobre la computadora), entradas inesperadas, un error del operador, roedores comiendo parte del sistema y muchas otras causas. No todo esto conduce (de inmediato) a fallas del sistema, pero algunas de estas cosas sí.

Las fallas se clasifican por lo general como transitorias, intermitentes o permanentes. Las **fallas transitorias** ocurren una vez y después desaparecen. Si la operación se repite, la falla ya no se presenta. Un pájaro que vuela a través del rayo de un transmisor de microondas provoca la pérdida de bits en una red (por no mencionar un pájaro frito). Si la transmisión expira y se repite, es probable que funcione la segunda vez.

Si ocurre una **falla intermitente**, ésta desaparece, reaparece, etcétera. Un mal contacto de un conector causa con frecuencia una falla intermitente, las cuales son graves por su difícil diagnóstico. Lo usual es que cuando aparece el doctor, el sistema funcione de manera perfecta.

Una **falla permanente** es aquella que continúa existiendo hasta reparar el componente con el desperfecto. Los circuitos quemados, los errores del software y el rompimiento de la cabeza del disco provocan con frecuencia fallas permanentes.

El objetivo del diseño y construcción de sistemas tolerantes de fallas consiste en garantizar que el sistema continúe funcionando de manera correcta como un todo, incluso en

la presencia de fallas. Este propósito es muy diferente al de garantizar que las componentes individuales sean muy confiables, pues permite (e incluso espera) que el sistema falle si alguno de los componentes lo hace.

Las fallas pueden ocurrir en todos los niveles: transistores, circuitos, tarjetas, procesadores, sistemas operativos, programas del usuario, etc. El trabajo tradicional en el área de tolerancia de fallas se ha preocupado principalmente del análisis estadístico de las fallas de los componentes electrónicos. Brevemente, si algún componente tiene una probabilidad p de tener un desperfecto en un segundo dado, la probabilidad de que *no* falle durante k segundos consecutivos y que después falle es $p(1-p)^k$. El tiempo esperado de fallo está dado entonces por la fórmula

$$\text{tiempo promedio de fallo} = \sum_{k=1}^{\infty} kp(1-p)^{k-1}$$

Utilizamos la ecuación conocida para una suma infinita que comienza en $k=1$: $\Sigma\,\alpha^k = \alpha/(1-\alpha)$, sustituimos $\alpha = 1-p$, derivamos ambos lados de la ecuación resultante con respecto de p y multiplicamos por $-p$ para obtener

$$\text{tiempo promedio de falla} = 1/p$$

Por ejemplo, si la probabilidad de un desperfecto es 10^{-6} por segundo, el tiempo promedio de falla es de 10^6 segundos o aproximadamente 11.6 días.

4.5.2. Fallas de sistema

En un sistema distribuido crítico, con frecuencia nos interesa que el *sistema* pueda sobrevivir a las fallas de los componentes (en particular, del procesador), en vez de hacer que las fallas sean poco probables. La confiabilidad de un sistema es en particular importante en un sistema distribuido, debido a la gran cantidad de componentes presentes; de ahí la mayor posibilidad de que falle uno de ellos.

Para el resto de la sección, analizaremos las fallas del procesador, pero esto debe entenderse también como fallas del proceso (por ejemplo, debido a errores del software). Se pueden distinguir dos tipos de fallas del procesador:

1. Fallas silentes.

2. Fallas bizantinas.

Con las **fallas silentes**, un procesador que falla sólo se detiene y no responde a las entradas subsecuentes ni produce más entradas, excepto que puede anunciar que ya no está funcionando. También se llaman **fallas de detención**. Con las **fallas bizantinas**, un procesador que falla continúa su ejecución, proporcionando respuestas incorrectas a las pre-

guntas, y posiblemente trabajando de manera maliciosa junto con otros procesadores que han fallado, para dar la impresión de que todos funcionan de manera correcta aunque no sea así. Los errores no detectados en el software exhiben con frecuencia fallas bizantinas. Es claro que enfrentarse a las fallas bizantinas será más difícil que enfrentarse a las fallas silentes.

El término "bizantino" se refiere al imperio bizantino, época (330-1453) y lugar (Los Balcanes y la Turquía moderna) en los cuales supuestamente hubo innumerables conspiraciones, intrigas e infidelidades en los círculos gobernantes. Las fallas bizantinas fueron analizadas por primera vez por Pease *et al.* (1980) y Lamport *et al.* (1982). Algunos investigadores también consideran combinaciones de estas fallas con las de la línea de comunicación, pero puesto que los protocolos estándar pueden recuperarse de los errores de las líneas de manera predecible, sólo examinaremos las fallas del procesador.

4.5.3. Sistemas síncronos *vs.* asíncronos

Como acabamos de ver, las fallas de los componentes pueden ser transitorias, intermitentes o permanentes, mientras que las fallas del sistema pueden ser silentes o bizantinas. Un tercer eje ortogonal trata del desempeño en un sentido abstracto. Supongamos que tenemos un sistema en el cual, si un procesador envía un mensaje a otro, se garantiza que obtiene una respuesta dentro de un tiempo T conocido de antemano. Si no se obtiene una respuesta, esto significa que el sistema receptor ha fallado. El tiempo T incluye el tiempo suficiente para considerar la pérdida de mensajes (enviándolos hasta n veces).

En el contexto de la investigación relativa a la tolerancia de fallas, un sistema que tiene la propiedad de responder siempre a un mensaje dentro de un límite finito conocido, si está funcionando, es **síncrono**. Un sistema que no tiene esta propiedad es **asíncrono**. Aunque esta terminología es desafortunada, ya que entra en conflicto con usos más tradicionales de los términos, se utiliza con amplitud entre las personas que trabajan con la tolerancia de fallas.

Debe ser claro que los sistemas asíncronos serán más difíciles de tratar que los síncronos. Si un procesador puede enviar un mensaje y sabe que la ausencia de respuesta dentro de T segundos significa que el pretendido receptor ha fallado, puede realizar una acción correctiva. Si no existe una cota superior para el tiempo de la respuesta, será un problema determinar incluso si ha ocurrido una falla.

4.5.4. Uso de redundancia

El método general para la tolerancia de fallas consiste en el uso de redundancia. Existen tres tipos posibles: redundancia de la información, redundancia del tiempo y la redundancia física. Con la redundancia de la información, se agregan algunos bits para poder recuperar los bits revueltos. Por ejemplo, se puede agregar un código Hamming para transmitir los datos y recuperarse del ruido en la línea de transmisión.

Con la redundancia del tiempo, se realiza una acción, y entonces, en caso necesario, se vuelve a realizar. El uso de las transacciones atómicas descrito en el capítulo 3 es un ejemplo de este método. Si una transacción aborta, puede volverse a realizar sin daño alguno. La redundancia de tiempo es de particular utilidad cuando las fallas son transitorias o intermitentes.

Con la redundancia física, se agrega un equipo adicional para permitir que el sistema como un todo tolere la pérdida o el mal funcionamiento de algunos componentes. Por ejemplo, se pueden agregar más procesadores, de modo que si unos pocos de ellos fallan, el sistema pueda seguir funcionando de manera correcta.

Hay dos formas de organizar estos procesadores adicionales: la réplica activa y el respaldo primario. Consideremos el caso de un servidor. Si se utiliza la réplica activa, todos los procesadores se utilizan todo el tiempo como servidores (en paralelo) para ocultar las fallas por completo. Por el contrario, el esquema de respaldo primario sólo utiliza un procesador como servidor, reemplazándolo con un respaldo si falla.

Analizaremos estas dos estrategias más adelante. Para ambas, los aspectos a considerar son:

1. El grado de réplica requerido.

2. El desempeño en el caso promedio y en el peor caso, en ausencia de fallas.

3. El desempeño en el caso promedio y en el peor caso, cuando ocurre una falla.

El análisis teórico de muchos sistemas tolerantes de fallas se puede realizar en estos términos. Para más información véase (Schneider, 1990; y Budhiraja *et al.*, 1993).

4.5.5. Tolerancia de fallas mediante réplica activa

La **réplica activa** es una técnica muy conocida para proporcionar la tolerancia de fallas mediante la redundancia física. Se utiliza en la biología (los mamíferos tienen dos ojos, dos oídos, dos pulmones, etc.), aviación (los 747 tienen cuatro motores pero pueden volar con tres) y los deportes (varios árbitros, en caso de que alguno omita un evento). Algunos autores se refieren a la réplica activa como el **método de la máquina de estados**.

También se ha utilizado durante años para la tolerancia de fallas en los circuitos electrónicos. Por ejemplo, consideremos el circuito de la figura 4-21(a). En este caso las señales pasan por los dispositivos A, B y C, en ese orden. Si uno de ellos falla, el resultado final probablemente sería incorrecto.

En la figura 4-21(b), cada dispositivo se reproduce tres veces. Después de cada etapa del circuito aparece un votante por triplicado. Cada votante es un circuito que tiene tres entradas y una salida. Si dos o tres de las entradas son iguales, la salida es igual a esa entrada. Si las tres entradas son diferentes, la salida queda indefinida. Este tipo de diseño se conoce como **TMR (Triple Modular Redundancy)** [Redundancia Modular Triple].

Supongamos que el elemento A_2 falla. Cada uno de los votantes V_1, V_2 y V_3 obtiene dos entradas correctas (idénticas) y una entrada incorrecta, y cada uno de ellos produce como

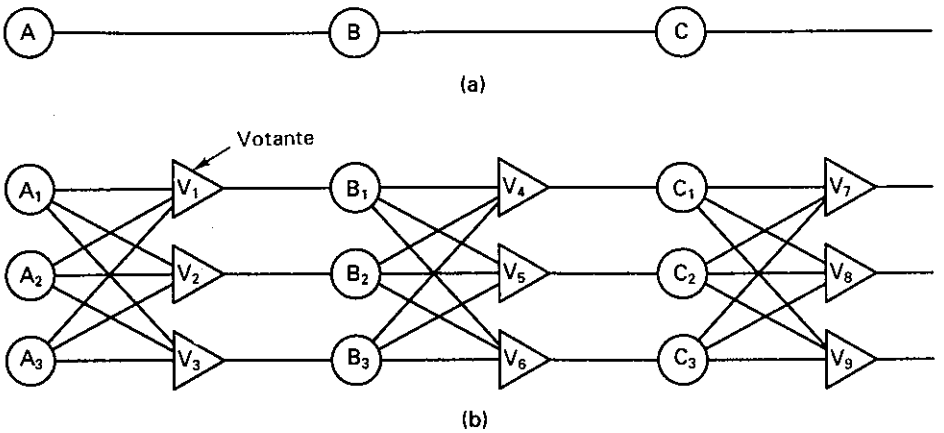


Figura 4-21. Redundancia modular triple.

salida el valor correcto a la segunda etapa. En esencia, el efecto de la falla de A_2 queda por completo encubierto, de modo que las entradas de B_1, B_2 y B_3 son las mismas que hubieran ocurrido sin la falla.

Consideremos ahora lo que ocurre si B_3 y C_1 también falla, además de A_2. Estos efectos también son encubiertos, de modo que los tres resultados finales siguen siendo correctos.

En primera instancia, podría no ser obvia la razón de la necesidad de tres votantes en cada etapa. Después de todo, un votante también podría detectar y aprobar según la mayoría. Sin embargo, un votante también es un componente, y también puede fallar. Por ejemplo, supongamos que V_1 falla. La entrada de B_1 será entonces incorrecta, pero si todo lo demás funciona, B_2 y B_3 producirán la misma salida y V_4, V_5 y V_6 producirán el resultado correcto en la tercera etapa. Efectivamente, una falla en V_1 no es diferente de una falla en B_1. En ambos casos, B_1 produce la salida incorrecta, pero en ambos casos se vota contra ella más adelante.

Aunque no todos los sistemas operativos distribuidos tolerantes de fallas utilizan TMR, la técnica es muy general y debe proporcionar una clara idea de lo que es un sistema tolerante, en oposición a un sistema cuyos componentes individuales son muy confiables, pero cuya organización no tolera las fallas. Por supuesto, TMR se puede aplicar de manera recursiva; por ejemplo, para crear un circuito muy confiable, que utilice TMR de manera interna, lo que desconocen los diseñadores que utilizan el circuito.

Regresando a la tolerancia de fallas en general y la réplica activa en particular, en muchos sistemas, los servidores actúan como grandes máquinas de estado finito: aceptan solicitudes y producen respuestas. La lectura de solicitudes no altera el estado del servidor, pero la escritura de solicitudes sí lo hace. Si cada solicitud cliente se envía a cada servidor y todas son recibidas y procesadas en el mismo orden, entonces, después de procesar cada una, todos los servidores que no han fallado tendrán con exactitud el mismo estado y darán

las mismas respuestas. El cliente o votante puede combinar todos los resultados para enmascarar las fallas.

Un aspecto importante es la cantidad de réplica necesaria. La respuesta depende de la cantidad de tolerancia de fallas deseada. Un sistema es **tolerante de k fallas** si puede sobrevivir a fallas en k componentes y seguir cumpliendo sus especificaciones. Si los componentes (digamos, los procesadores) fallan de manera silente, entonces bastan $k + 1$ de ellos para proporcionar la tolerancia de k fallas. Si k de estos componentes sólo se detienen, se puede utilizar la respuesta del otro componente.

Por otro lado, si los procesadores exhiben fallas bizantinas, continúan su ejecución al enfermar y envían respuestas erróneas o aleatorias, se necesita un mínimo de $2k + 1$ procesadores para lograr la tolerancia de k fallas. En el peor de los casos, los k procesadores que han fallado podrían generar de manera accidental (o incluso intencional) la misma respuesta. Sin embargo, los restantes $k + 1$ siempre producirán la misma respuesta, de modo que el cliente o votante puede sólo creer a la mayoría.

Por supuesto, en teoría está bien decir que un sistema es tolerante de k fallas y solo dejar que las $k + 1$ respuestas idénticas derroten a las k respuestas idénticas, pero en la práctica, es difícil imaginar circunstancias en las que uno pueda decir con certidumbre que k procesadores puedan fallar pero $k + 1$ procesadores no pueden. Así, incluso en un sistema tolerante de fallas, se necesita cierto tipo de análisis estadístico.

Una condición previa implícita para que este modelo de máquina de estado finito sea relevante es que todas las solicitudes lleguen a todos los servidores en el mismo orden, lo que a veces se llama el **problema de transmisión atómica**. En realidad, esta condición se puede relajar, puesto que las lecturas no importan y algunas escrituras pueden conmutar, pero el problema general sigue en pie. Una forma de garantizar que todas las solicitudes se procesen en el mismo orden en todos los servidores consiste en numerarlas de forma global. Se han diseñado varios protocolos para lograr este objetivo. Por ejemplo, todas las solicitudes se podrían enviar primero a un servidor global de números, con el fin de obtener un número consecutivo, pero entonces habría que tomar previsiones para la falla de este servidor (por ejemplo, haciéndolo tolerante de fallas internas).

Otra posibilidad es utilizar los relojes lógicos de Lamport, descritos en el capítulo 3. Si cada mensaje enviado a un servidor recibe una marca de tiempo, y los servidores procesan todas las solicitudes según el orden de dichas marcas, todas las solicitudes serán procesadas en el mismo orden en todos los servidores. El problema con este método es que, cuando un servidor recibe una solicitud, no sabe si las solicitudes anteriores están en camino. De hecho, la mayor parte de las soluciones con marcas de tiempo sufren de este problema. En resumen, la réplica activa no es un asunto trivial. Schneider (1990) analiza los problemas y soluciones con cierto detalle.

4.5.6. Tolerancia de fallas mediante respaldo primario

La idea esencial del método de respaldo primario es que en cualquier instante, un servidor es el primario y realiza todo el trabajo. Si el primario falla, el respaldo ocupa su lugar. En forma ideal, el remplazo debe ocurrir de manera limpia, y ser notado únicamente

por el sistema operativo cliente, no por los programas de aplicación. Como la réplica activa, este esquema se utiliza con amplitud en el mundo. Algunos ejemplos son el gobierno (el vicepresidente), la aviación (los copilotos), los automóviles (llantas de refacción), y los generadores de energía eléctrica con base en diesel en las salas de operación de un hospital.

La tolerancia de fallas con respaldo primario tiene dos ventajas principales sobre la réplica activa. En primer lugar, es más sencilla durante la operación normal, puesto que los mensajes van sólo a un servidor (el primario) y no a todo un grupo. Los problemas asociados con el ordenamiento de estos mensajes también desaparecen. En segundo lugar, en la práctica se requieren menos máquinas, puesto que en cualquier instante se necesitan un primario y un respaldo (aunque cuando un respaldo se pone en servicio como primario, se necesita un nuevo respaldo de manera instantánea). Como desventaja, trabaja mal en presencia de fallas bizantinas, en las que el primario afirma erróneamente que funciona de manera perfecta. Además, la recuperación de una falla del primario puede ser compleja y consumir mucho tiempo.

Como ejemplo de la solución con respaldo primario, consideremos el protocolo simple de la figura 4-22, en donde se muestra una operación de escritura. El cliente envía un mensaje al primario, quien realiza el trabajo y después envía un mensaje de actualización al respaldo. Cuando el respaldo lo recibe, realiza el trabajo y entonces envía un reconocimiento de regreso al primario. Cuando llega el reconocimiento, el primario envía la respuesta al cliente.

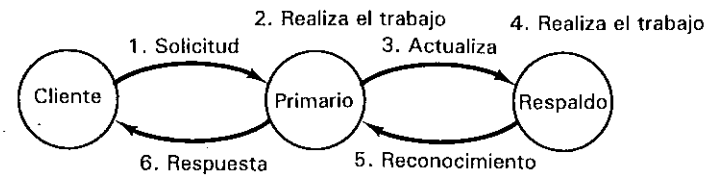


Figura 4-22. Un protocolo simple de respaldo primario en una operación de escritura.

Consideremos ahora el efecto de una falla del primario durante varios momentos durante una RPC. Si el primario falla antes de realizar el trabajo (paso 2), no hay ningún daño. El cliente expira y vuelve a intentar. Si intenta las veces suficientes, entonces de manera eventual llegará al respaldo y el trabajo se realizará con exactitud. Si el primario falla después de realizar el trabajo pero antes de enviar la actualización, cuando el respaldo ocupa su lugar y la solicitud vuelve a llegar, el trabajo se realizará por segunda vez. Si el trabajo tiene efectos colaterales, esto podría ser un problema. Si el primario falla antes del paso 4 pero antes del paso 6, el trabajo podría terminar realizándose tres veces, una por el primario, otra por el respaldo como consecuencia del paso 3 y otra después de que el respaldo se convierte en el primario. Si solicita identificadores de acarreo, entonces podríamos garantizar que el trabajo sólo se realiza dos veces, pero hacer que se realice una vez es difícil o imposible.

Un problema teórico y práctico con el método del respaldo primario es el momento en que debe pasarse del primario al respaldo. En el protocolo anterior, el respaldo podría enviar

el mensaje "¿Estás vivo?" de manera periódica al primario. Si el primario no responde después de cierto tiempo, el respaldo tomaría el control.

Sin embargo, ¿qué ocurre si el primario no ha fallado, sino que sólo es lento (es decir, tenemos un sistema asíncrono)? No existe forma de distinguir entre un primario lento y uno que ha fallado. Aun así, existe la necesidad de garantizar que cuando el respaldo toma el control, el primario se detiene y deja de intentar actuar como tal. Lo ideal es que el respaldo y el primario tengan un protocolo para discutir esto, pero es difícil negociar con un muerto. La mejor solución es un mecanismo de hardware en el que un respaldo pueda obligar a detenerse o volver a arrancar el primario. Observe que todos los esquemas de respaldo primario necesitan un acuerdo, lo que se obtiene de manera truculenta, mientras que la réplica activa no siempre requiere un protocolo de acuerdo (por ejemplo, TMR).

Una variante del método de la figura 4-22 utiliza un disco con puerto dual compartido entre el primario y el secundario. En esta configuración, cuando el primario recibe una solicitud, escribe ésta en el disco antes de realizar cualquier trabajo y también escribe los resultados en el disco. No se necesitan mensajes a o desde el respaldo. Si el primario falla, el respaldo puede ver el estado del mundo al leer el disco. La desventaja de este esquema es que sólo existe un disco, de modo que si falla, todo se pierde. Por supuesto, con el costo adicional de más equipo y un mejor desempeño, el disco también se podría replicar y realizar todas las escrituras en ambos discos.

4.5.7. Acuerdos en sistemas defectuosos

En muchos sistemas distribuidos existe la necesidad de que los procesos coincidan en algo. Algunos ejemplos son la elección de un coordinador, decidir cuándo comprometer una transacción o no, dividir las tareas entre los trabajadores, la sincronización, etc. Cuando la comunicación y los procesadores son todos perfectos, llegar a tales acuerdos es con frecuencia algo directo, pero cuando no lo son, surgen los problemas. En esta sección analizaremos algunos de los problemas y sus soluciones (o la carencia de éstas).

El objetivo general de los algoritmos de acuerdo distribuido es lograr que todos los procesadores no defectuosos alcancen el consenso acerca de cierto tema, en un número finito de pasos. Existen varios casos dependientes de los parámetros del sistema, entre los que están:

1. ¿Se entregan los mensajes de manera confiable todo el tiempo?

2. ¿Pueden fallar los procesos, y, en tal caso, las fallas son silentes o bizantinas?

3. ¿Es el sistema síncrono o asíncrono?

Antes de considerar el caso de los procesadores con falla, analizaremos el "sencillo" caso de los procesadores perfectos pero en el que las líneas de comunicación pueden perder los mensajes. Existe un famoso problema, el **problema de los dos ejércitos**, que ilustra la dificultad de tener dos procesadores perfectos para llegar a un acuerdo en torno a un pedazo

de información. El ejército rojo, con 5 000 soldados, está acampando en un valle. Dos ejércitos azules, cada uno con 3 000 efectivos, acampan en las colinas circundantes, con visión panorámica del valle. Si los dos ejércitos azules pueden coordinar sus ataques sobre el ejército rojo, saldrán victoriosos. Sin embargo, si uno de ellos ataca por su cuenta, será derrotado. El objetivo de los ejércitos azules es lograr un acuerdo acerca del ataque. El problema es que sólo se pueden comunicar mediante un canal no confiable: enviar un mensaje que está sujeto a ser capturado por el ejército rojo.

Supongamos que el comandante del ejército azul 1, el general Alejandro, envía un mensaje al comandante del ejército azul 2, el general Bonaparte, que dice: "Tengo un plan: ataquemos mañana al amanecer." El mensajero logra pasar y Bonaparte le regresa una nota que dice: "Espléndida idea, Alejandro. Nos veremos al amanecer mañana." El mensajero regresa a su base a salvo, entrega el mensaje y Alejandro lo lee y dice a sus tropas que se preparen para la batalla al amanecer.

Sin embargo, en un momento posterior del día, Alejandro se da cuenta de que Bonaparte no sabe si el mensajero regresó a salvo y, al no saber esto, podría no atreverse a atacar. En consecuencia, Alejandro dice al mensajero que vaya y diga a Bonaparte que su mensaje (el de Bonaparte) llegó y que están listos para la batalla.

De nuevo, el mensajero logra llegar y entrega el reconocimiento. Pero ahora Bonaparte se preocupa porque Alejandro no sabe si el reconocimiento llegó. Él razona que si Bonaparte piensa que el mensajero fue capturado, no estará seguro de sus planes (los de Alejandro) y podría no arriesgarse a atacar, de modo que envía al mensajero de regreso.

Aunque el mensajero pueda pasar cada vez, es fácil mostrar que Alejandro y Bonaparte nunca llegarán a un acuerdo, sin importar el número de reconocimientos enviados. Supongamos que existe cierto protocolo que concluye en un número finito de pasos. Eliminemos los pasos adicionales para obtener el protocolo mínimo que funciona. Algún mensaje es entonces el último y es esencial para el acuerdo (puesto que éste es el protocolo mínimo). Si este mensaje no llega, la guerra no se realizará.

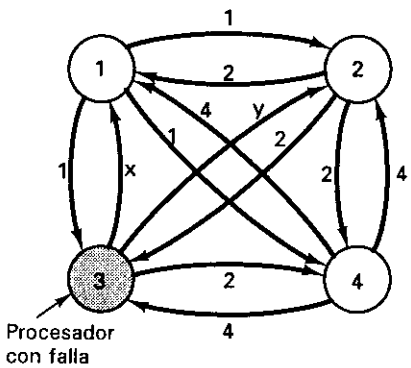
Sin embargo, el emisor del último mensaje no sabe si el último mensaje llegó. Si no lo hizo, el protocolo no se completa y el otro general no atacará. Así, el emisor del último mensaje no puede saber si la guerra está planeada o no, y por lo tanto no puede comprometer de manera segura a sus tropas. Puesto que el receptor del último mensaje sabe que el emisor no puede estar seguro, él tampoco se arriesgará a una muerte y no existe acuerdo alguno. Incluso con los procesadores sin falla (los generales), el acuerdo entre dos procesos no es posible si existe una comunicación no confiable.

Supongamos ahora que la comunicación es perfecta pero que los procesadores no lo son. El problema clásico en este caso también ocurre en un ambiente militar y se llama el **problema de los generales bizantinos**. En este problema, el ejército rojo sigue acampando en el valle, pero n generales azules comandan ejércitos en las colinas cercanas. La comunicación se realiza por parejas mediante líneas telefónicas y es perfecta, pero m de estos generales son traidores (fallan) y de manera activa intentan evitar que los generales leales lleguen a un acuerdo, dándoles información incorrecta y contradictoria (para modelar el

desperfecto de los procesadores). La cuestión es ahora si los generales leales pueden llegar a un acuerdo.

Con el fin de tener cierta generalidad, definiremos el acuerdo aquí de manera un poco distinta. Suponemos que cada general conoce el número de efectivos de que dispone. El objetivo del problema es que los generales intercambien la información de este número de efectivos, de modo que al final del algoritmo, cada general tenga un vector de longitud n correspondiente a todos los ejércitos. Si el general i es leal, entonces el elemento i es su cantidad de efectivos; en caso contrario, está indefinido.

Lamport *et al.*, (1982) diseñaron un algoritmo recursivo que resuelve este problema bajo ciertas condiciones. En la figura 4-23 ilustramos el trabajo de este algoritmo para el caso de $n = 4$ y $m = 1$. Con estos parámetros, el algoritmo opera en cuatro pasos. En el paso uno, cada general envía un mensaje (confiable) a los demás con la información de sus tropas. Los generales leales dicen la verdad; los traidores podrían decir a cada uno de los demás una mentira diferente. En la figura 4-23(a) vemos que el general 1 informa de 1K soldados, el general 2 informa de 2K soldados, el general 3 miente a todos, dado x, y y z, respectivamente, y el general 4 informa de 4K soldados. En el paso 2, los resultados de estos anuncios en el paso 1 se reúnen en la forma de los vectores de la figura 4-23(b).



1 Tenemos	(1, 2, x, 4)	1 Tenemos	2 Tenemos	3 Tenemos
2 Tenemos	(1, 2, y, 4)	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
3 Tenemos	(1, 2, 3, 4)	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
4 Tenemos	(1, 2, z, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(b) (c)

Figura 4-23. El problema de los generales bizantinos para 3 generales leales y un traidor. (a) los generales anuncian el número de soldados (en unidades de 1K). (b) Los vectores que cada general obtiene con base en (a). (c) Los vectores que cada general recibe en el paso 2.

En el paso 3, cada general pasa su vector de la figura 4-23(b) a los demás. En este caso, de nuevo, el general 3 miente, ideando 12 nuevos valores, a a l. Los resultados del paso 3 se muestran en la figura 4-23(c). Por último, en el paso 4, cada general examina el i-ésimo elemento de cada uno de los vectores recién recibidos. Si cualquier valor tiene una mayoría, este valor se coloca en el vector de resultado. Si ningún valor tiene la mayoría, el elemento

correspondiente del vector de resultado se marca como INCÓGNITA. De la figura 4-23(c) vemos que los generales 1, 2 y 4 llegan a un acuerdo de que

(1, 2, INCÓGNITA, 4)

que es el resultado correcto. El traidor no pudo lograr su objetivo.

Revisemos ahora este problema para $m = 3$ y $n = 1$, es decir, sólo dos generales leales y un traidor, como se muestra en la figura 4-24. Aquí vemos que en la figura 4-24(c), ninguno de los generales leales ve una mayoría para el elemento 1, el elemento 2 o el elemento 3, de modo que todos éstos se marcan como INCÓGNITA El algoritmo no ha podido producir un acuerdo.

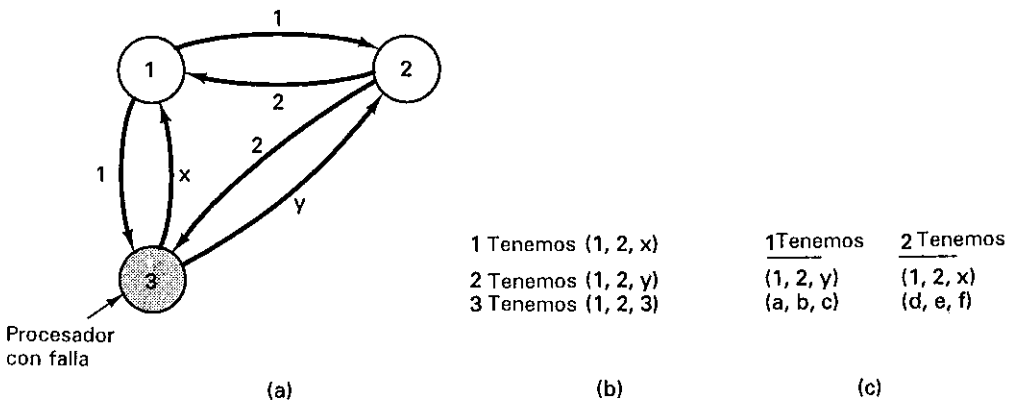


Figura 4-24. Lo mismo de la figura 4-23, pero ahora con 2 generales leales y un traidor.

En su artículo, Lamport *et al.*, (1982) demostraron que en un sistema con m procesadores con falla, el acuerdo se puede lograr sólo si se dispone de $2m + 1$ procesadores que funcionen de manera correcta, para un total de $3m + 1$. Dicho en otras palabras, el acuerdo sólo es posible si *más* de dos terceras partes de los procesadores están funcionando de manera correcta.

Peor aún, Fischer *et al.*, (1985) demostraron que en un sistema distribuido con procesadores asíncronos y retrasos no acotados en la transmisión, no se puede llegar a un acuerdo aunque sólo un procesador esté fallando (e incluso si el procesador falla de manera silente). El problema con los sistemas asíncronos es que los procesadores muy lentos son indistinguibles de los muertos. Se conocen muchos otros resultados teóricos acerca de lograr o no acuerdos. Algunas reseñas de estos resultados son las dadas por Barborak *et al.*, (1993) y Turek y Shasha (1992).

4.6. SISTEMAS DISTRIBUIDOS DE TIEMPO REAL

Los sistemas tolerantes de fallas no son el único tipo de sistemas distribuidos especializados. Los sistemas de tiempo real forman otra categoría. A veces se combinan estos dos tipos para obtener sistemas de tiempo real tolerantes de fallas. Para más información, véase por ejemplo (Burns y Wellings, 1990; Klein *et al.*, 1994; y Shin, 1991).

4.6.1. ¿Qué es un sistema de tiempo real?

Para la mayoría de los programas, el hecho de que sean correctos depende sólo de la secuencia lógica de las instrucciones ejecutadas, no del momento en que se ejecuten. Si un programa en C calcula de manera correcta la función raíz cuadrada en punto flotante con doble precisión en una estación de trabajo de 200 Mhz, también calculará la función de manera correcta en una computadora personal de 4.77 Mhz basada en 8 088, aunque más lento.

Por el contrario, los **programas** (y sistemas) **de tiempo real** interactúan con el mundo exterior de una manera que implica al tiempo. Cuando aparece un estímulo, el sistema responde a éste de cierta manera y antes de cierto momento límite. Si entrega la respuesta correcta, pero después del límite, se considera que el sistema está fallando. El *momento* en que se produce la respuesta es tan importante como *aquello* que produce.

Consideremos un ejemplo sencillo. Un reproductor de discos compactos de audio consta de un CPU que toma los bits que llegan del disco y los procesa para generar música. Supongamos que el CPU es lo bastante rápido como para hacer el trabajo. Ahora imaginemos que un competidor decide construir un reproductor más barato mediante un CPU con un tercio de la velocidad del otro. Si éste guarda todos los bits de llegada y los reproduce con un tercio de la velocidad esperada, las personas se sobrecogerán con el sonido y si sólo reproduce cada tercer nota, la audiencia tampoco quedará satisfecha. A diferencia de nuestro ejemplo anterior con la raíz cuadrada, el tiempo es parte inherente de la especificación de precisión.

Muchas otras aplicaciones relacionadas con el mundo exterior también son de tiempo real de manera inherente. Algunos ejemplos son las computadoras incluidas con los televisores y las grabadoras de video, las computadoras que controlan los alerones y demás partes de los aviones (que se dice vuelan por medio de cables), los subsistemas de los automóviles controlados por computadoras (¿Controlados por medio de cables?), las computadoras militares que controlan los misiles antitanques (¿Disparados por medio de cables?), los sistemas computarizados para el control del tráfico aéreo, los experimentos científicos, desde los aceleradores de partículas hasta los ratones de laboratorio con electrodos en sus cerebros, las fábricas automatizadas, los conmutadores telefónicos, los robots, las unidades médicas de cuidado intensivo, los digitalizadores para la tomografía axial computarizada, los sistemas automáticos para intercambio de acciones y varios otros más.

Muchas aplicaciones y sistemas de tiempo real son muy estructurados, mucho más que los sistemas distribuidos de propósito general. De manera típica, un dispositivo externo (tal

vez un reloj) genera un estímulo para la computadora, la que entonces debe realizar ciertas acciones antes de un momento límite. Al terminar el trabajo solicitado, el sistema queda inactivo hasta que llega el siguiente estímulo.

Con frecuencia, los estímulos son **periódicos**, de modo que un estímulo ocurre de manera regular cada ΔT segundos, como una computadora en un televisor o videocasetera, que recibe un cuadro nuevo cada 1/60 segundos. A veces, los estímulos son **aperiódicos**, lo que significa que son recurrentes, pero no regulares, como en la llegada de un avión al espacio aéreo de un controlador de tráfico aéreo. Por último, algunos estímulos son **esporádicos** (inesperados), como el sobrecalentamiento de un dispositivo.

Aún en un sistema que en gran medida sea periódico, una complicación es que pueden existir muchos tipos de eventos, como entrada de video, entrada de audio y el control de la unidad motora, cada uno con su período y acciones necesarias. La figura 4-25 muestra una situación con tres flujos de eventos periódicos, *A*, *B* y *C*, más un evento esporádico, *X*.

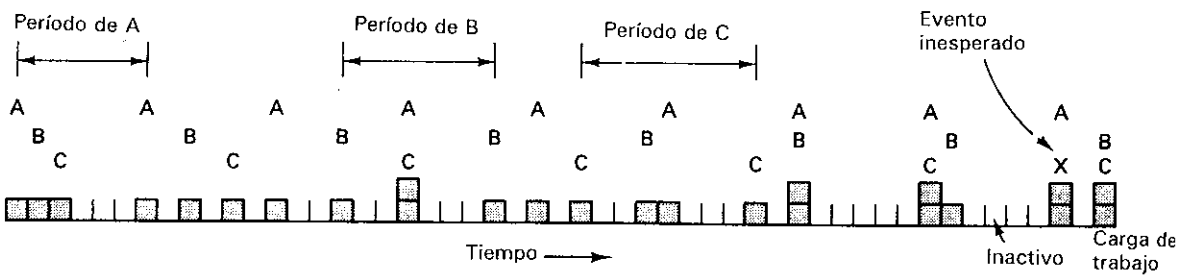


Figura 4-25. Superposición de tres flujos de eventos más un evento esporádico.

A pesar del hecho de que el CPU tiene que trabajar con varios flujos de eventos, no es aceptable que diga: Es cierto que omití el evento *B*, pero no es mi error; yo seguía trabajando en *A* cuando ocurrió *B*. Aunque no es difícil controlar dos o tres flujos de entrada con interrupciones de prioridad, conforme las aplicaciones son cada vez mayores y complejas (por ejemplo, las líneas de ensamblaje en una fábrica automatizada con miles de robots), será cada vez más difícil que una máquina cumpla con todas las horas límite y otras restricciones de tiempo real.

En consecuencia, algunos diseñadores están experimentando con la idea de colocar un microprocesador exclusivo al frente de cada dispositivo de tiempo real para aceptar salida de ella cuando tenga algo que decir, y dar una entrada con la velocidad que requiera. Por supuesto, esto no hace que el carácter de tiempo real se esfume, sino que da lugar a un sistema distribuido de tiempo real, con sus propias características y retos (por ejemplo, la comunicación de tiempo real).

Los sistemas distribuidos de tiempo real pueden estructurarse con frecuencia como se ilustra en la figura 4-26. Aquí vemos una colección de computadoras conectadas mediante

una red. Algunas de éstas se conectan a dispositivos externos que producen o aceptan datos o esperar ser controlados en tiempo real. Las computadoras pueden ser pequeños microcontroladores integrados a los dispositivos, o máquinas independientes. En ambos casos, por lo general tienen sensores para recibir señales de los dispositivos y/o actores a los cuales enviar señales. Los sensores y actores pueden ser digitales o analógicos.

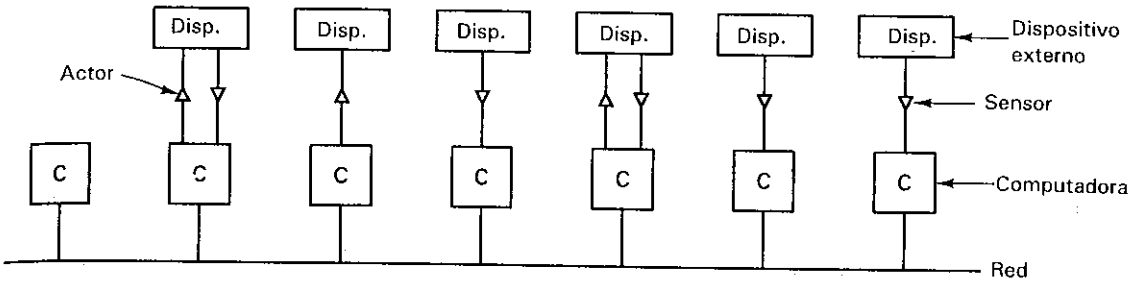


Figura 4-26. Un sistema de cómputo distribuido de tiempo real.

Los sistemas de tiempo real se clasifican por lo general en dos tipos dependiendo de lo serio de sus tiempos límite y de las consecuencias de omitir uno de ellos. Éstos son:

1. Sistemas de tiempo real suave.
2. Sistemas de tiempo real duro.

El **tiempo real suave** significa que no existe problema si se rebasa un tiempo límite. Por ejemplo, un conmutador telefónico bajo condiciones de sobrecarga podría perder o equivocar de ruta 10^5 llamadas y seguir cumpliendo sus especificaciones. Por el contrario, un tiempo límite no cumplido en un sistema de **tiempo real duro** es inaceptable, pues podría conducir a la pérdida de una vida o a una catástrofe ambiental. En la práctica, existen también sistemas intermedios en los que la omisión de un tiempo límite significa que falla toda la actividad actual, pero que la consecuencia no es fatal. Por ejemplo, si una botella de refresco ha pasado por la banda de conducción sin detenerse en la boquilla de alimentación, no hay necesidad de continuar vertiendo refresco en ella, pero los resultados no son fatales. Además, en algunos sistemas de tiempo real, algunos subsistemas son de tiempo real duro, mientras que otros son de tiempo real suave.

Los sistemas de tiempo real han estado por ahí durante décadas, de modo que existe mucha sabiduría popular acumulada sobre ellos, pero muchos de ellos son incorrectos. Stankovic (1988) ha señalado algunos de éstos, y aquí resumimos los peores.

Mito 1: Los sistemas de tiempo real tratan de la escritura de controladores de dispositivos en código ensamblador.

Tal vez esto era cierto en la década de 1970 para los sistemas de tiempo real que constaban de unos cuantos instrumentos conectados a una minicomputadora, pero los sistemas de

tiempo real de la actualidad son muy complicados para confiar en el lenguaje ensamblador y escribir los controladores de dispositivos es la menor de las preocupaciones de un diseñador de sistemas de tiempo real.

Mito 2: El cómputo de tiempo real es rápido.

No necesariamente. Un telescopio controlado por una computadora puede tener que rastrear las estrellas o galaxias en tiempo real, pero la aparente rotación del cielo es de tan sólo 15 grados de arco por hora de tiempo, que no es en particular rápido. Aquí lo que cuenta es la exactitud.

Mito 3: Las computadoras rápidas harán que el sistema de tiempo real sea obsoleto.

No. Sólo animan a las personas a construir sistemas de tiempo real que anteriormente estaban más allá de lo normal. Los cardiólogos estarían felices de tener un digitalizador MRI que muestre el latido de un corazón dentro de un paciente que realiza ejercicio en tiempo real. Cuando lo tengan, pedirán uno en tres dimensiones, a color, y con la posibilidad de realizar acercamientos o alejamientos. Además, para crear sistemas más rápidos mediante el uso de varios procesadores hay que resolver nuevos problemas de comunicación, sincronización y planificación.

4.6.2. Aspectos del diseño

Los sistemas distribuidos de tiempo real tienen ciertos aspectos de diseño únicos. En esta sección analizaremos algunos de los más importantes.

Sincronización del reloj

El primer aspecto es el mantenimiento de la propia hora. Con varias computadoras, cada una con su propio reloj local, la sincronización de los relojes es un aspecto fundamental. Examinamos este punto en el capítulo 3, de modo que no repetiremos ese análisis aquí.

Sistemas activados por eventos *vs.* sistemas activados por el tiempo

En un **sistema de tiempo real activado por eventos**, cuando ocurre un evento significativo en el mundo exterior, es detectado por algún sensor, lo que entonces provoca que el CPU conectado tenga una interrupción. Los sistemas activados por eventos están controlados entonces por las interrupciones. La mayor parte de los sistemas de tiempo real funcionan de esta manera. Para los sistemas de tiempo real suave con mucho poder de cómputo por compartir, este método es sencillo, funciona bien y sigue utilizándose con amplitud. Aún para sistemas más complejos, funciona bien si el compilador puede analizar el programa y conoce todo lo que hay que conocer acerca del comportamiento del sistema una vez que ocurra un evento, aunque no pueda decir el momento en que ocurrirá el evento.

El principal problema con los sistemas activados por eventos es que pueden fallar bajo condiciones de carga pesada, es decir, cuando muchos eventos ocurran a la vez. Por ejemplo,

consideremos lo que ocurre cuando un tubo se rompe en un reactor nuclear controlado por una computadora. Las alarmas de temperatura, las alarmas de presión, las alarmas de radioactividad y demás alarmas se activarán al mismo tiempo, lo que provoca una interrupción masiva. Esta **lluvia de eventos** puede sobrecargar al sistema de cómputo y hacer que falle, lo que potencialmente causaría problemas mucho más serios que la ruptura de un solo tubo.

Un diseño alternativo que sufre de este problema es el **sistema de tiempo real activado por el tiempo**. En este tipo de sistema, ocurre una interrupción del reloj cada ΔT milisegundos. En cada marca de reloj (ciertos) sensores se muestrean y (ciertos) actores se controlan. No hay más interrupciones que las marcas de reloj.

En el ejemplo anterior del tubo roto, el sistema sería consciente del problema en la primera marca del reloj posterior al evento, pero la carga de interrupciones no modificaría ni contaría para el problema, de modo que el sistema no se sobrecargaría. Operar normalmente en tiempos de crisis aumenta la posibilidad de tratar con éxito dicha crisis.

No hay que decir que ΔT debe elegirse con mucho cuidado. Si es muy pequeño, el sistema tendrá muchas interrupciones de reloj y desperdiciará mucho tiempo durante las revisiones. Si es muy grande, los eventos serios no serían notados hasta que fuese muy tarde. Además, la decisión acerca de los sensores que deben verificarse en cada marca de reloj, y cuáles verificar en otra marca de reloj, etc., son críticas. Por último, algunos eventos podrían ser más cortos que una marca de reloj, por lo que deberán guardarse para no ser omitidos. Se pueden preservar eléctricamente mediante ciertos circuitos o mediante microprocesadores integrados a los dispositivos externos.

Como ejemplo de la diferencia entre estos dos métodos, consideremos el diseño de un controlador de elevador en un edificio de 100 pisos. Supongamos que el elevador está en el piso 60, esperando clientes. Entonces alguien oprime el botón de llamada en el primer piso. Justo 100 milisegundos después, alguien más oprime el botón de llamada en el piso 100. En un sistema activado por eventos, la primera llamada genera una interrupción, lo que provoca el descenso del elevador. La segunda llamada llega después de tomar la decisión de bajar, por lo que se anota como referencia futura, pero el elevador continúa descendiendo.

Consideremos ahora un controlador de elevador, activado por el tiempo, que muestrea cada 500 milisegundos. Si ambas llamadas caen dentro de un periodo de muestreo, el controlador deberá tomar una decisión; por ejemplo, utilizar la regla de atender primero al cliente más cercano, en cuyo caso irá hacia arriba.

En resumen, los diseños activados por eventos dan respuesta rápida con carga baja, pero tienen mayor costo y probabilidad de fallar con carga alta. Los diseños activados por el tiempo tienen las propiedades opuestas y sólo son adecuados en un ambiente relativamente estático en donde se conozca mucho y de antemano acerca del comportamiento del sistema. Cuál de ellos será el mejor, dependerá de la aplicación. En todo caso, observamos que existe mucha controversia sobre este tema en los círculos del tiempo real.

Predictibilidad

Una de las propiedades más importantes de cualquier sistema de tiempo real es que su comportamiento sea predecible. De manera ideal, debe ser claro en el momento del diseño

que el sistema cumple con todos sus tiempos límite, incluso con carga pico. Los análisis estadísticos del comportamiento, suponiendo que los eventos son independientes, conducen con frecuencia a errores, pues existen correlaciones inesperadas entre los eventos, como entre las alarmas de temperatura, presión y radiactividad en el ejemplo anterior del entubamiento roto.

La mayoría de los sistemas distribuidos están acostumbrados a pensar en términos de usuarios independientes que tienen acceso aleatorio a archivos compartidos o de numerosos agentes de viajes que tienen acceso a una base de datos compartidos de una línea aérea en momentos impredecibles. Por fortuna, este tipo de comportamiento aleatorio rara vez ocurre en un sistema de tiempo real. Lo más frecuente es que, cuando se detecta el evento E, el proceso X se debe ejecutar, seguido por los procesos Y y Z, en ese orden o en paralelo. Además, con frecuencia se conoce (o debe conocerse) el comportamiento de estos procesos en el peor de los casos. Por ejemplo, si se sabe que X necesita 50 milisegundos, Y y Z necesitan 60 milisegundos cada uno, y el inicio del proceso tarda 5 milisegundos, entonces se puede garantizar de entrada que el sistema puede manejar sin fallar cinco eventos periódicos de tipo E por segundo en ausencia de cualquier otro trabajo. Este tipo de razonamiento y modelaje lleva a un sistema determinista, más que estocástico.

Tolerancia de fallas

Muchos sistemas de tiempo real controlan dispositivos donde la seguridad es crítica, en vehículos, hospitales y plantas de energía, por lo que la tolerancia de fallas es con frecuencia un aspecto a considerar. La réplica activa se utiliza en ciertas ocasiones, pero sólo si puede hacerse sin protocolos extensos (y que por lo tanto consumen tiempo) para que todos coincidan en todo, todo el tiempo. Los esquemas de respaldo primario son menos populares, puesto que los tiempos límite pueden omitirse durante la recuperación, después de que falle el primario. Un método híbrido consiste en seguir al líder, en donde una máquina toma todas las decisiones, pero las demás realizan su trabajo sin discusión, listas para tomar el mando en un momento dado.

En un sistema donde la seguridad es crítica, es de particular importancia que el sistema pueda controlar el peor de los escenarios. No basta decir que la probabilidad de que tres componentes fallen al mismo tiempo es tan baja que puede ignorarse. Las fallas no siempre son independientes. Por ejemplo, durante una súbita falla del suministro de energía eléctrica, todos tratan de hablar por teléfono, lo que puede provocar que el sistema telefónico se sobrecargue, aunque tenga su propio e independiente sistema de generación de energía. Además, la carga pico en el sistema ocurre con frecuencia precisamente en el momento en que ha fallado el máximo número de componentes, debido a que gran parte del tráfico se relaciona con el informe de las fallas. En consecuencia, los sistemas de tiempo real tolerantes de fallas deben poder enfrentar el número máximo de fallas y la carga máxima al mismo tiempo.

Algunos sistemas de tiempo real tienen la propiedad de que se pueden quedar congelados cuando ocurre una falla seria. Por ejemplo, cuando un sistema de señalización de

ferrocarriles se detiene, es probable que el sistema de control indique a todos los trenes que se detengan de inmediato. Si el diseño del sistema siempre da el espacio suficiente a los trenes y todos los trenes comienzan a frenar de manera simultánea, será posible evitar un desastre y que el sistema se recupere de manera gradual cuando regrese la energía. Un sistema que puede detener la operación como éste, sin peligro, es **seguro contra las fallas**.

Soporte del lenguaje

Aunque muchos sistemas y aplicaciones de tiempo real se programan en lenguajes de propósito general como C, los lenguajes especializados de tiempo real pueden ser de potencial ayuda. Por ejemplo, en tal lenguaje, debe ser fácil expresar el trabajo como colección de tareas pequeñas (es decir, procesos ligeros o hilos) que puedan planificarse de manera independiente, sujetos a la precedencia definida por el usuario y las restricciones de exclusión mutua.

El lenguaje debe diseñarse de modo que se pueda calcular el tiempo máximo de ejecución de cada tarea en el momento de su compilación. Este requisito significa que el lenguaje no puede soportar ciclos **while** generales. La iteración se realiza mediante ciclos **for** con parámetros constantes. La recursión tampoco se puede tolerar (empieza a parecer que FORTRAN tendrá un uso después de todo). Incluso estas restricciones podrían no ser suficientes para calcular el tiempo de ejecución de cada tarea de antemano, puesto que la falta de caché, el fallo de página y el robo de ciclos por los canales DMA afectan el desempeño, pero son un comienzo.

Los lenguajes de tiempo real necesitan trabajar con el tiempo mismo. Para comenzar, se debe disponer de una variable especial, *clock*, que contiene el tiempo actual en marcas. Sin embargo, hay que tener cuidado con la unidad en la que se expresa el tiempo. Mientras más fina sea la resolución, más rápido tendrá la variable *clock* un desbordamiento. Por ejemplo, si es un entero de 32 bits, el arreglo de las diversas resoluciones aparece en la figura 4-27. Lo ideal es que el reloj tenga un ancho de 64 bits y resolución de un nanosegundo.

Resolución del reloj	Arreglo
1 nanosegundo	4 segundos
1 μ-segundo	72 minutos
1 milisegundo	50 días
1 segundo	136 años

Figura 4-27. Rango de un reloj de 32 bits antes de un desbordamiento, para diversas resoluciones.

El lenguaje debe tener forma de expresar los retrasos mínimo y máximo. Por ejemplo, en Ada®, existe un enunciado de retraso que especifica un valor mínimo que debe suspen-

derse un proceso. Sin embargo, el retraso real podría ser una cantidad no acotada. No existe una forma de imponer una cota superior o un intervalo de tiempo en el que deba caer el retraso.

También debe haber una forma de expresar lo que deba hacerse si un evento inesperado no ocurre dentro de cierto intervalo. Por ejemplo, si un proceso se bloquea en un semáforo durante un tiempo mayor a uno dado, debe ser posible hacer que expire y libere sus recursos. De manera análoga, si se envía un mensaje pero la respuesta no llega lo bastante rápido, el emisor debe poder especificar lo que deba desbloquearse después de k milisegundos.

Por último, puesto que los eventos periódicos juegan un papel muy importante en los sistemas de tiempo real, sería útil tener un enunciado de la forma

```
every (25 msec) { ... }
```

que provoque que los enunciados dentro de las llaves se ejecuten cada 25 milisegundos. Mejor aún, si una tarea contiene varios de estos enunciados, el compilador podrá calcular el porcentaje de tiempo de CPU que necesitaría cada uno, y a partir de estos datos calcular el número máximo de máquinas necesarias para ejecutar todo el programa y la forma de asignar los procesos a dichas máquinas.

4.6.3. Comunicación en tiempo real

La comunicación en los sistemas distribuidos de tiempo real es diferente a la comunicación en los demás. Aunque un mejor desempeño siempre es bienvenido, la predictibilidad y el determinismo son las claves para el éxito. En esta sección analizaremos algunos aspectos de la comunicación de tiempo real, para LAN y WAN. Por último, examinaremos un ejemplo de sistema con cierto detalle para mostrar sus diferencias con los sistemas distribuidos convencionales (es decir, que no son de tiempo real). Otros enfoques se describen en (Malcolm y Zhao, 1994; y Ramanathan y Shin, 1992).

Lograr la predictibilidad en un sistema distribuido significa que la comunicación entre los procesadores también debe ser predecible. Los protocolos LAN que son inherentemente estocásticos, como Ethernet, son inaceptables, puesto que no proporcionan una cota superior conocida sobre el tiempo de transmisión. Una máquina que desea enviar un paquete en Ethernet puede chocar con una o más máquinas. Todas las máquinas esperan entonces un tiempo aleatorio y vuelven a probar, pero estas transmisiones también pueden chocar, etc. En consecuencia, no es posible dar una cota en el peor de los casos de transmisión de paquetes de antemano.

En contraste con Ethernet, consideremos una LAN con un anillo de fichas. Siempre que un procesador tenga un paquete por enviar, espera que le pasen la ficha en circulación, captura la ficha, envía su paquete y coloca la ficha de nuevo en el anillo, de modo que la siguiente máquina en el flujo tenga la oportunidad de recuperarla. Suponiendo que cada una de las k máquinas del anillo puede enviar a los más un paquete de n bytes por cada captura de fichas, se puede garantizar que un paquete urgente que llegue a cualquier parte del sistema siempre se puede transmitir en kn instantes. Éste es el tipo de cota superior que necesita un sistema distribuido de tiempo real.

Los anillos de fichas también pueden controlar el tráfico consistente en varias clases de prioridades. El objetivo en este caso es garantizar que si un paquete de alta prioridad espera ser transmitido, será enviado antes que todos los paquetes de menor prioridad de sus vecinos. Por ejemplo, es posible agregar un campo de reservación a cada paquete, que puede ser incrementado por cualquier procesador al pasar el paquete. Cuando el paquete haya dado toda la vuelta, el campo de reservación indica la clase de prioridad del siguiente paquete. Cuando el emisor actual termina su transmisión, regenera una ficha con esta clase de prioridad. Sólo los procesadores con un paquete pendiente de esta clase pueden obtener la ficha, y entonces sólo envían un paquete. Por supuesto, este esquema significa que la cota superior de *kn* veces sólo se aplica a los paquetes de la máxima clase de prioridad.

Una alternativa al anillo de fichas es el protocolo **TDMA** (Time division Múltiple Access [**acceso múltiple con división del tiempo**]) que se muestra en la figura 4-28. En este caso, el tráfico se organiza en marcos de tamaño fijo, cada uno de los cuales contiene *n* espacios. Cada espacio es asignado a un procesador, el que puede utilizarlo para transmitir un paquete cuando le llegue su momento. De esta forma, se evitan las colisiones, se acota el retraso, y cada procesador obtiene una fracción garantizada del ancho de banda, según la cantidad de espacios por marco que tenga asignados.

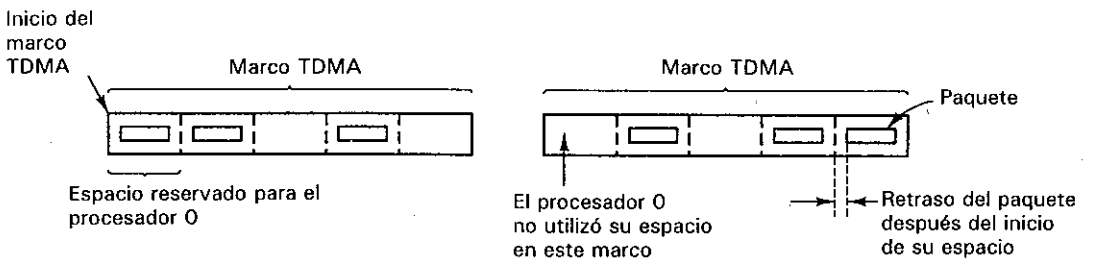


Figura 4-28. Marcos TDMA.

Los sistemas distribuidos que operan en redes de área amplia tienen la misma necesidad de predictibilidad que los confinados a un cuarto o edificio. La comunicación en estos sistemas está orientado, de manera invariable, hacia la conexión. Con frecuencia, existe la capacidad de establecer **conexiones de tiempo real** entre las máquinas distantes. Cuando se establece una conexión de este tipo, la calidad del servicio se negocia de antemano entre los usuarios de la red y el proveedor de la misma. Esta calidad puede implicar un retraso máximo garantizado, una inquietud máxima (varianza de los tiempos de entrega de un paquete), ancho de banda mínimo y otros parámetros. Para cumplir con esto, la red debe reservar buffers de memoria, entradas de tabla, ciclos de CPU, capacidad de enlace y otros recursos para la conexión durante la existencia de ésta. Es probable que el usuario reciba un cargo por estos recursos, ya sea que se utilicen o no, puesto que no están disponibles para otras conexiones.

Un problema potencial con los sistemas de tiempo real de área amplia es su tasa de pérdida de paquetes relativamente alta. Los protocolos estándar trabajan con la pérdida de paquetes

estableciendo un cronómetro cada vez que se transmite un paquete. Si el cronómetro expira antes de recibir el reconocimiento, el paquete se envía de nuevo. En los sistemas de tiempo real, este tipo de retraso no acotado en la transmisión rara vez es aceptable.

Una solución sencilla consiste en que el emisor *siempre* transmita cada paquete dos (o más) veces, de preferencia a través de conexiones independientes si esta opción está disponible. Aunque este esquema desperdicia al menos la mitad del ancho de banda, si se pierde un paquete de cada 10^5, digamos, una vez de cada 10^{10} se perderán ambas copias. Si un paquete tarda un milisegundo, esto implica una pérdida de un paquete cada cuatro meses. Con tres transmisiones, un paquete se pierde cada 30,000 años. El efecto neto de las transmisiones múltiples de cada paquete desde un principio es un retraso bajo y acotado todo el tiempo.

El protocolo activado por el tiempo

Como otra de las restricciones sobre los sistemas distribuidos de tiempo real, sus protocolos son con frecuencia poco usuales. En esta sección examinaremos uno de tales protocolos, **TTP (protocolo activado por el tiempo)** (Kopetz y Grunsteidl, 1994), que es tan diferente del protocolo de Ethernet como un cuarto de pinturas victorianas de una cantina del viejo oeste. TTP se utiliza en el sistema de tiempo real MARS (Kopetz *et al.*, 1989) y se entrelaza con él de varias formas, por lo que nos referiremos a las propiedades de MARS cuando sea necesario.

Un nodo en MARS consta de al menos un CPU, pero con frecuencia dos o tres trabajan juntos para presentar al mundo exterior la imagen de un nodo tolerante de fallas silentes. Los nodos en MARS se conectan mediante dos redes de transmisión TDMA confiables e independientes. Todos los paquetes se envían por ambas redes en paralelo. La tasa esperada de pérdidas es de un paquete cada 30 millones de años.

MARS es un sistema activado por el tiempo, por lo que la sincronización de los relojes es crítica. El tiempo es discreto, donde las marcas de reloj generalmente ocurren cada microsegundo. TTP supone que todos los relojes se sincronizan con una precisión del orden de decenas de microsegundos. Esta precisión es posible debido a que el propio protocolo proporciona una sincronización continua del reloj y ha sido diseñado para poder hacerlo en hardware con una precisión en extremo alta.

Todos los nodos en MARS son conscientes de los programas que se ejecutan en los demás nodos. En particular, todos los nodos conocen el momento en que un paquete se envía a otro nodo y pueden detectar su presencia o ausencia con facilidad. Puesto que no se supone que los paquetes se pierden (véase antes), la ausencia de un paquete en un momento dado significa que el nodo emisor ha fallado.

Por ejemplo, supongamos que se ha detectado cierto evento excepcional y que se transmite un paquete a todos informando de él. Se espera que el nodo 6 realice cierto cálculo y que después transmita una respuesta después de 2 milisegundos en el espacio 15 del marco TDMA. Si el mensaje no llega en el espacio esperado, los demás nodos suponen que el nodo 6 ha fallado y toman las medidas necesarias para recuperarse de su falla. Esta fuerte cota y el consenso instantáneo eliminan la necesidad de protocolos de acuerdo, que consumen tiempo, y permite que el sistema sea tolerante y opere en tiempo real.

Cada nodo mantiene el estado global del sistema. Estos estados deben ser idénticos en todos lados. Un error serio (y detectable) ocurre cuando alguien pierde el paso de los demás. El estado global consta de tres componentes:

1. El modo activo.
2. El tiempo global.
3. Un mapa de bits con la membresía actual del sistema.

El modo queda definido por la aplicación y tiene que ver con la fase en la que se encuentra el sistema. Por ejemplo, en una aplicación espacial, la cuenta regresiva, el lanzamiento, el vuelo y el aterrizaje podrían ser modos separados. Cada modo tiene su propio conjunto de procesos y el orden de su ejecución, la lista de nodos participantes, las asignaciones de espacios TDMA, los nombres y formato de los mensajes, y los modos que legalmente pueden sucederlo.

El segundo campo en el estado global es el tiempo global. Su granularidad queda definida por la aplicación, pero en todo caso debe ser lo bastante gruesa como para que todos los nodos coincidan en ella. El tercer campo lleva un registro de los nodos activos y los inactivos.

A diferencia de las series de protocolo OSI e Internet, el protocolo TTP consta de una capa que controla el transporte de datos, la sincronización de relojes y la administración de la membresía de extremo a extremo. Un formato de paquete típico aparece en la figura 4-29. Consta de un campo de inicio de paquete, un campo de control, un campo de datos y un campo CRC.

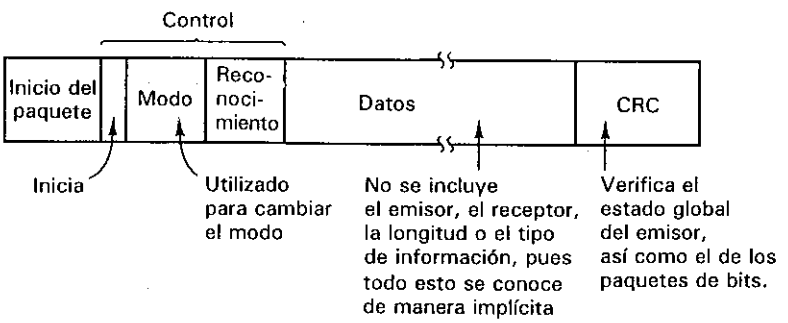


Figura 4-29. Un paquete TTP típico.

El campo de control contiene un bit utilizado para iniciar el sistema (hablaremos más de esto adelante), un subcampo para cambiar el modo activo, y un subcampo para reconocer los paquetes enviados por el nodo precedente (de acuerdo con la lista de membresía actual).

La finalidad de este campo es dejar que el nodo anterior sepa que funciona de manera correcta y que sus paquetes llegan a la red como deberían. Si falta un reconocimiento esperado, todos los nodos marcan al emisor esperado como fallido y lo sacan de los mapas de bits de membresía en su estado activo. Se espera que el nodo rechazado quede incomunicado sin protestar.

El campo de datos contiene todos los datos necesarios. El campo CRC es un tanto inusual, ya que no sólo proporciona una suma de verificación sobre el contenido del paquete, sino también sobre el estado global del emisor. Esto significa que si un emisor tiene un estado global incorrecto, el CRC de todos los paquetes que envíe no coincidirá con los valores que calculen los receptores utilizando sus estados. El siguiente emisor no reconocerá el paquete, y los demás nodos, incluyendo el que tiene el estado incorrecto, lo marcarán como fallido en sus mapas de bits de membresía.

De forma periódica, se transmite un paquete con el bit de iniciación. Este paquete también contiene el estado global activo. Cualquier nodo que se marque como no miembro, pero que se supone debe serlo, puede ahora unirse como miembro pasivo. Si un nodo debe ser un miembro, tiene asignado un espacio TDMA, por lo que no hay problema cuando responda (en su propio espacio TDMA). Una vez reconocido su paquete, los demás nodos lo marcan de nuevo como activo (operativo).

Un último aspecto interesante del protocolo es la forma en que controla la sincronización del reloj. Puesto que cada nodo conoce el tiempo en que se inician los marcos TDMA y la posición de su espacio dentro del marco, sabe con exactitud cuándo iniciar su paquete. Este esquema evita las colisiones. Sin embargo, también contiene información valiosa relativa al tiempo. Si un paquete inicia *n* microsegundos antes o después de lo debido, cada uno de los demás nodos puede detectar este problema y utilizarlo como estimación de la desviación entre su reloj y el del emisor. Al monitorear la posición inicial de cada paquete, un nodo podría aprender, por ejemplo, que los demás inician sus transmisiones 10 microsegundos tarde. En este caso, puede concluir de manera razonable que su reloj marcha 10 microsegundos adelante y realizar la corrección necesaria. Al mantener un promedio del retraso o adelanto de los demás paquetes, cada nodo puede ajustar su reloj de manera continua para mantenerlo sincronizado con los demás sin tener que ejecutar un protocolo especial para la administración del reloj.

En resumen, las propiedades inusuales del TTP son la detección de paquetes perdidos por los receptores, no los emisores, el protocolo automático de membresía y la forma de realizar la sincronización de los relojes.

4.6.4. Planificación de tiempo real

Los sistemas de tiempo real se programan con frecuencia como colección de pequeñas tareas (procesos o hilos), cada una con funciones bien definidas y un tiempo de ejecución con cota también bien definida. La respuesta a un estímulo dado puede requerir la ejecución de varias tareas, por lo general con restricciones acerca de su orden de ejecución. Además, hay que tomar una decisión en relación con las tareas por ejecutar en tal o cual procesador.

En esta sección trabajaremos con algunos de los aspectos relativos a la planificación de tareas en los sistemas de tiempo real.

Los algoritmos de planificación de tiempo real se pueden caracterizar mediante los siguientes parámetros:

1. Tiempo real duro *vs.* tiempo real suave.
2. Planificación con prioridad *vs.* sin prioridad.
3. Dinámico *vs.* estático.
4. Centralizado *vs.* descentralizado.

Los algoritmos de tiempo real duro deben garantizar que se cumple con todos los tiempos límite. Los algoritmos de tiempo real suave pueden vivir con un método del mejor esfuerzo. El caso más importante es el tiempo real duro.

La planificación con prioridad permite suspender una tarea de manera temporal al llegar una con mayor prioridad, y continuar su ejecución cuando no existen más tareas de mayor prioridad por ejecutar. La planificación sin prioridad ejecuta cada tarea hasta terminarla. Una vez que se inicia una tarea, continúa conservando a su procesador hasta terminar. Se utilizan ambos tipos de estrategias de planificación.

Los algoritmos dinámicos toman las decisiones de planificación durante su ejecución. Al detectar un evento, un algoritmo dinámico con prioridad decide en ese momento si ejecuta la (primera) tarea asociada con el evento o continúa ejecutando la tarea activa. Un algoritmo dinámico sin prioridad sólo observa que otra tarea se puede ejecutar. Cuando termina la tarea activa, elige entre las tareas listas para su ejecución.

Por el contrario, con los algoritmos estáticos, las decisiones de planificación, con prioridad o sin ella, se toman de antemano, antes de la ejecución. Cuando ocurre un evento, el planificador del tiempo de ejecución sólo observa una tabla para ver qué hacer.

Por último, la planificación puede ser centralizada, de modo que una máquina recoge toda la información y toma todas las decisiones, o puede ser descentralizada, de forma que cada procesador tome sus decisiones. En el caso centralizado, la asignación de tareas a los procesadores se puede realizar en el mismo momento. En el caso descentralizado, la asignación de tareas a los procesadores es distinta de la decisión del orden de ejecución de las tareas asignadas a un procesador dado.

Una cuestión fundamental que enfrentan todos los diseñadores de sistemas de tiempo real es si es posible o no cumplir con todas las restricciones. Si un sistema tiene un procesador y tiene 60 interrupciones/segundo, cada una de las cuales requiere 50 milisegundos de trabajo, los diseñadores tienen enorme problema en las manos.

Supongamos que un sistema distribuido de tiempo real periódico tiene m tareas y N procesadores para ejecutarlas. Sea C_i el tiempo de CPU necesario para la tarea i y sea P_i su periodo, es decir, el tiempo entre las interrupciones consecutivas. Para ser factible, el uso del sistema, μ, debe relacionarse con N mediante la ecuación

$$\mu = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq N$$

Por ejemplo, si una tarea se inicia cada 20 milisegundos y se ejecuta durante 10 milisegundos cada vez, utiliza 0.5 CPU. Cinco de tales tareas necesitarían tres CPU para realizar el trabajo. Un conjunto de tareas que cumple con el requisito anterior es **planeable**. Observe que la ecuación anterior da en realidad una cota inferior sobre la cantidad de CPU necesarias, puesto que ignora el tiempo de conmutación de tareas, el transporte de mensajes y otras fuentes de costos, y supone que es posible la planificación óptima.

En las siguientes dos secciones analizaremos la planificación dinámica y la estática, respectivamente, de los conjuntos de tareas periódicas. Para mayor información, véase (Ramamritham *et al.*, 1990; y Schwan y Zhou, 1992).

Planificación dinámica

Analizaremos primero algunos de los más conocidos algoritmos de planificación dinámica, algoritmos que deciden cuál tarea ejecutar a continuación durante la ejecución del programa. El algoritmo clásico es el **algoritmo monótono de tasa** (Liu y Layland, 1973). Fue diseñado para tareas de planificación con prioridad, sin restricciones de orden o exclusión mutua en un procesador. Funciona de la siguiente manera. De antemano, cada tarea tiene asignada una prioridad igual a su frecuencia de ejecución. Por ejemplo, una tarea que se ejecuta cada 20 milisegundos tiene asignada una prioridad de 50 y una tarea que se ejecuta cada 100 milisegundos tiene 10 de prioridad. Al tiempo de ejecución, el planificador siempre selecciona la tarea de máxima prioridad para su ejecución, priorizando sobre la tarea activa en caso necesario. Liu y Layland demostraron que este algoritmo es óptimo. También demostraron que cualquier conjunto de tareas que cumple la condición de uso

$$\mu = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

es planeable mediante el algoritmo monótono de tasa. El lado derecho converge a ln 2 (cerca de 0.693) cuando $m \rightarrow \infty$. En la práctica, este límite es muy pesimista; un conjunto de tareas con μ hasta 0.88 se puede planear por lo general.

Un segundo algoritmo dinámico con prioridad es el **algoritmo del primer límite en primer lugar**. Siempre que se detecta un evento, el planificador lo agrega a la lista de tareas en espera. Esta lista siempre se ordena por su tiempo límite, de modo que el límite más cercano esté en primer lugar. (Para una tarea periódica, el límite es la siguiente ocurrencia.) El planificador sólo elige entonces la primera tarea de la lista, la más cercana a su tiempo límite. Como el algoritmo monótono de tasa, produce resultados óptimos, aun para conjuntos de tareas con $\mu = 1$.

Un tercer algoritmo dinámico con prioridad calcula primero el tiempo total ocupado por cada tarea, llamado **laxitud** (carencia). Para una tarea que debe terminar en 200 mili-

segundos pero que puede ejecutarse en 150 milisegundos, tiene una laxitud de 50 milisegundos. Este algoritmo, llamado de **mínima laxitud**, elige la tarea con la mínima laxitud, es decir, la tarea con el menor espacio para respirar.

Ninguno de los algoritmos anteriores han demostrado ser óptimos en un sistema distribuido, pero se pueden utilizar como heurísticos. Además, ninguno de ellos toma en cuenta las restricciones de orden o de mútex, incluso en un uniprocesador, lo que los hace menos útiles en la práctica que en la teoría. En consecuencia, muchos sistemas prácticos utilizan la planificación estática cuando disponen de suficiente información. Los algoritmos estáticos no sólo toman en cuenta las restricciones colaterales, sino que tienen bajo costo al tiempo de ejecución.

Planificación estática

La planificación estática se realiza antes de que el sistema comience su operación. La entrada consta de una lista de todas las tareas que se deben ejecutar. El objetivo es determinar una asignación de tareas a cada procesador y una planificación estática con el orden de ejecución de las tareas. En teoría, el algoritmo de planificación puede realizar una búsqueda exhaustiva para determinar la solución óptima, pero el tiempo de búsqueda es exponencial con respecto del número de tareas (Ullman, 1976), de modo que se utiliza por lo general una heurística del tipo descrito antes. En vez de dar sólo más heurística, veremos un ejemplo con detalle, para mostrar la interrelación entre la planificación y la comunicación en un sistema distribuido de tiempo real con planificación estática sin prioridad (Kopetz *et al.*, 1989).

Supongamos que cada vez que se detecta cierto evento, la tarea 1 se inicia en el procesador A, como se muestra en la figura 4-30. Esta tarea, a su vez, inicia otras tareas, localmente y en un segundo procesador, B. Para simplificar la exposición, supondremos que la asignación de tareas a los procesadores queda determinada por consideraciones externas (cuál de las tareas necesita el acceso a cuál dispositivo de E/S) y que no es un parámetro en este caso. Todas las tareas ocupan una unidad de tiempo de CPU.

La tarea 1 inicia las tareas 2 y 3 en su máquina, así como la tarea 7 en el procesador B. Cada una de estas tres tareas inicia otra, etc., como se ilustra. Las flechas indican los mensajes enviados entre las tareas. En este sencillo ejemplo, tal vez sea más fácil pensar que $X \rightarrow Y$ significa que Y no puede iniciar hasta que llegue un mensaje de X. Algunas tareas, como 8, requieren dos mensajes antes de poder comenzar. El ciclo se completa cuando la tarea 10 se ha ejecutado y generado la respuesta esperada según el estímulo inicial.

Después de terminar la tarea 1, las tareas 2 y 3 son ejecutables. El planificador tiene que elegir cuál de ellas ejecutar. Supongamos que decide planear primero la ejecución de la tarea 2. Entonces tiene que elegir entre las tareas 3 y 4 como sucesores de la tarea 2. Si elige la tarea 3, entonces tendrá que elegir después entre las tareas 4 y 5. Sin embargo, si elige 4 en vez de 3, deberá elegir 3 después de 4, puesto que 6 no está activada todavía, y no lo estará hasta que 5 y 9 se ejecuten.

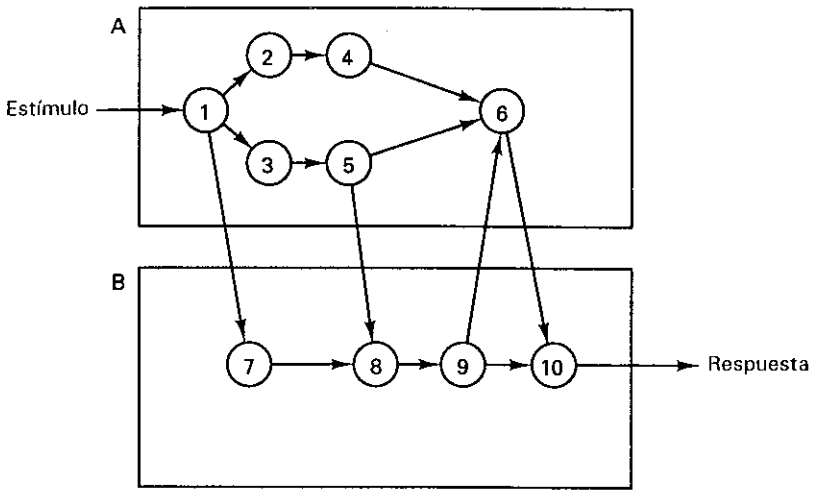


Figura 4-30. Diez tareas de tiempo real que deben ejecutarse en dos procesadores.

Mientras tanto, también ocurre una actividad en paralelo en el procesador *B*. Tan pronto como se ha iniciado la tarea 1, la 7 puede iniciarse en *B*, al mismo tiempo que 2 o 3. Cuando 5 y 7 han finalizado, la tarea 8 puede iniciar, etc. Observe que la tarea 6 requiere una entrada de 4, 5 y 9 para iniciar, y que produce una salida para 10.

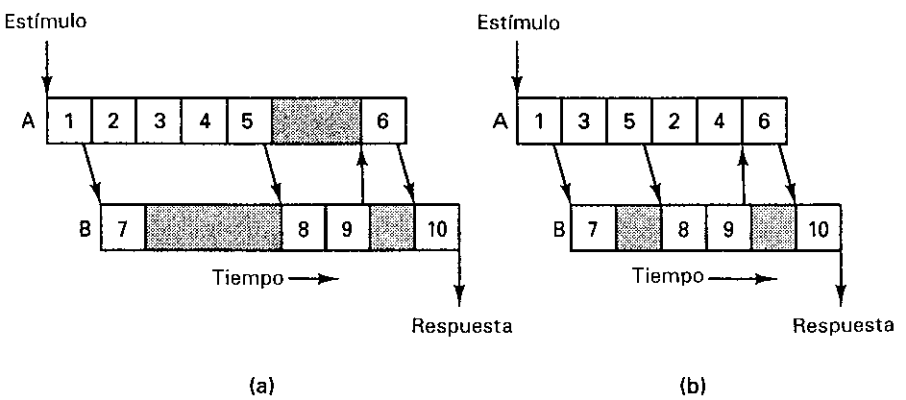


Figura 4-31. Dos planificaciones posibles para las tareas de la figura 4-30.

En la figura 4-31(a) y (b) se dan dos planificaciones posibles. Los mensajes entre las tareas en procesadores distintos se muestran aquí como flechas; los mensajes entre tareas en la misma máquina se manejan internamente y no se muestran. De las dos planificaciones mostradas, la de la figura 4-31(b) es una mejor opción, pues permite que la ejecución de la

tarea 5 sea temprana, lo que permite que la tarea 8 se ejecute antes. Si la tarea 5 se retrasa de manera significativa, como en la figura 4-31(a), entonces las tareas 8 y 9 se retrasan, lo que además significa que también se retrasa la tarea 6 y, de manera eventual, a 10.

Es importante darse cuenta de que con la planificación estática, la decisión de utilizar alguna de estas planificaciones, o alguna de varias alternativas se toma en el planificador *de antemano*, antes de que el sistema inicie su ejecución. Analiza la gráfica de la figura 4-30, utilizando también como entrada la información relativa a los tiempos de ejecución de todas las tareas y aplicando después cierta heurística para determinar una buena planificación. Una vez seleccionada la planificación, las elecciones hechas se incorporan en tablas de modo que al momento de ejecución, un servidor sencillo puede realizar la planificación a un bajo costo.

Consideremos ahora el problema de planificar las mismas tareas de nuevo, pero esta vez tomando en cuenta la comunicación. Utilizaremos la comunicación TDMA, con ocho espacios por cada marco TDMA. En este ejemplo, un espacio TDMA es igual a una cuarta parte del tiempo de ejecución de la tarea. Asignaremos de manera arbitraria el espacio 1 al procesador *A* y el espacio 5 al procesador *B*. La asignación de espacios TDMA a los procesadores es tarea del planificador estático y puede diferir entre las fases del programa.

En la figura 4-32 mostramos las dos planificaciones de la figura 4-31, pero tomando en cuenta ahora el uso de los espacios TDMA. Una tarea podría no enviar un mensaje hasta recibir un espacio de su procesador. Así, la tarea 5 podría no enviar un mensaje a la tarea 8 hasta que el primer espacio del siguiente marco TDMA aparezca en la rotación, lo que requiere un retraso en el inicio de la tarea 8 en la figura 4-32(a) que no estaba presente antes.

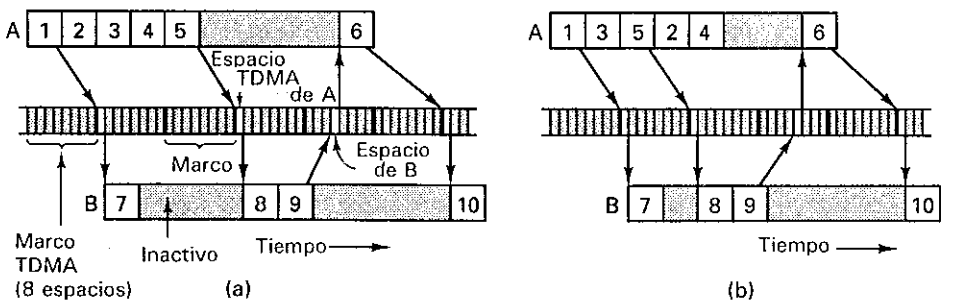


Figura 4-32. Dos planificaciones, incluyendo el procesamiento y la comunicación.

Lo importante de este ejemplo es que el comportamiento al tiempo de ejecución es por completo determinista, y se conoce antes de que el programa inicie su ejecución. Mientras no ocurran errores de comunicación o de los procesadores, el sistema siempre cumplirá con sus tiempos límites. Las fallas del procesador se pueden encubrir haciendo que cada nodo conste de dos o más CPU que se examinen de manera activa entre sí. Tendría que asignarse cierto tiempo de manera estática a cada intervalo de tarea para permitir la recu-

peración, en caso necesario. La pérdida o mezcla de paquetes podría controlarse si cada uno se envía dos veces al principio, ya sea por medio de redes ajenas o en una red, haciendo que los espacios TDMA tengan el doble de ancho.

Debe ser claro ahora que los sistemas de tiempo real no tratan de obtener la última gota de desempeño del hardware, sino que utilizan recursos adicionales para garantizar que las restricciones de tiempo real se cumplen bajo todas las condiciones. Sin embargo, el uso relativamente bajo del ancho de banda de comunicación en nuestro ejemplo no es lo usual. Es una consecuencia de este ejemplo, que sólo utiliza dos procesadores, con requisitos modestos de comunicación. Los sistemas prácticos de tiempo real tienen muchos procesadores y amplia comunicación.

Una comparación entre la planificación dinámica y la estática

La elección de la planificación dinámica o la estática es importante y tiene consecuencias de gran alcance para el sistema. La planificación estática está bien para un diseño activado por el tiempo y la planificación dinámica se ajusta bien a un diseño activado por los eventos. La planificación estática debe planearse con cuidado de antemano, y se utiliza un gran esfuerzo para elegir los diversos parámetros. La planificación dinámica no requiere mucho trabajo de antemano, puesto que las decisiones de planificación se realizan al vuelo, durante la ejecución.

La planificación dinámica puede hacer potencialmente un mejor uso de los recursos que la planificación estática. En esta última, el sistema debe exagerar con frecuencia las dimensiones para tener la capacidad suficiente y controlar los casos menos probables. Sin embargo, en un sistema de tiempo real duro, el desperdicio de recursos es con frecuencia el precio que debe pagarse para garantizar que se cumplen todos los límites.

Por otro lado, con el suficiente poder de cómputo, se puede obtener de antemano una planificación óptima o cercana a la óptima en un sistema estático. Para una aplicación como el control de un reactor, bien vale la pena una investigación de meses del tiempo de CPU para lograr la mejor planificación. Un sistema dinámico no se puede dar el lujo de un cálculo complejo de planificación durante su ejecución, así que para dar seguridad, también podría exagerar sus dimensiones, e incluso entonces, no existe garantía de que cumplirá con sus especificaciones. En vez de esto, se necesita un análisis amplio.

Como idea final, debe señalarse que nuestro análisis ha simplificado las cosas de manera considerable. Por ejemplo, las tareas podrían necesitar el acceso a variables compartidas, así que éstas deben reservarse de antemano. Con frecuencia existen restricciones de planificación, que hemos ignorado. Por último, algunos sistemas realizan una planificación de antemano durante su ejecución, lo que los convierte en híbridos entre lo estático y lo dinámico.

4.7. RESUMEN

Aunque los hilos de control no son una característica inherente de los sistemas operativos distribuidos, la mayoría de éstos tiene un paquete de hilos, por lo que los estudiamos

en este capítulo. Un hilo es un tipo de proceso ligero, que comparte el espacio de direcciones con uno o más hilos. Cada hilo tiene su propio contador de programa, su propia pila y se planifica de manera independiente de los demás hilos. Cuando un hilo hace una llamada al sistema con bloqueo, los otros hilos del mismo espacio de direcciones no se ven afectados. Los paquetes de hilos se pueden implantar en el espacio del usuario o en el espacio del núcleo, pero de cualquier forma hay que resolver algunos problemas. El uso de hilos ligeros también ha traído algunos interesantes resultados en la RPC ligera. Los hilos de aparición instantánea (pop-up threads) también son una técnica importante.

Se utilizan por lo común dos modelos de organización de los procesadores: el modelo de estación de trabajo y el de la pila de procesadores. En el primero, cada usuario tiene su propia estación de trabajo y a veces puede ejecutar procesos en las estaciones de trabajo inactivas. En el segundo, todas las instalaciones de cómputo son un recurso compartido. Los procesadores se asignan de manera dinámica a los usuarios conforme sea necesario y se regresan a la pila al terminar el trabajo. También son posibles los modelos híbridos.

Dada una colección de procesadores, se necesita un algoritmo para asignar los procesos a los procesadores. Tales algoritmos pueden ser deterministas o heurísticos, centralizados o distribuidos, óptimos o subóptimos, locales o globales, iniciados por el emisor o por el receptor.

Aunque los procesos se planifican por lo general de manera independiente, se puede mejorar el desempeño mediante la coplanificación, para garantizar que los procesos que deben comunicarse se ejecutan al mismo tiempo.

La tolerancia de fallas es importante en muchos sistemas distribuidos. Se puede lograr mediante la redundancia modular triple, la réplica activa o la réplica con respaldo primario. El problema de los dos ejércitos no se puede resolver en presencia de una comunicación no confiable, pero el problema de los generales bizantinos se puede resolver si más de las dos terceras partes de los procesadores no están fallando.

Por último, los sistemas distribuidos de tiempo real también son importantes. Vienen en dos tipos: tiempo real suave y tiempo real duro. Los sistemas activados por eventos son controlados por las interrupciones, mientras que los sistemas activados por el tiempo muestrean los dispositivos externos a intervalos fijos de tiempo. La comunicación de tiempo real debe utilizar protocolos predecibles, como los anillos de fichas o TDMA. Es posible la planificación dinámica y estática de las tareas. La planificación dinámica ocurre al tiempo de ejecución; la planificación estática ocurre de antemano.

PROBLEMAS

1. En este problema hay que comparar la lectura de un archivo mediante un servidor de archivos con un hilo o mediante un servidor con varios hilos. Una solicitud de trabajo tarda 15 milisegundos en llegar, ser despachada y hacer el resto del procesamiento necesario, suponiendo que los datos están dentro del bloque caché. Si se necesita una operación en disco, como ocurre la tercera parte del tiempo, se requieren 75 milise-

gundos más, durante los cuales el hilo duerme. ¿Cuántas solicitudes/segundo puede manejar el servidor si tiene un hilo? ¿Si tiene varios hilos?

2. En la figura 4-3, el conjunto de registros se enumera por hilo y no por proceso. ¿Por qué? Después de todo, la máquina sólo tiene un conjunto de registros.

3. En el texto describimos un servidor de archivos de varios hilos y mostramos por qué es mejor que un servidor con un hilo y que un servidor dado por una máquina de estado finito. ¿Existen circunstancias en las que el servidor de un hilo sea mejor? Dé un ejemplo.

4. En el análisis de las variables globales en los hilos, utilizamos un procedimiento *create_global* para asignar espacio de almacenamiento para un apuntador a la variable, en vez de la propia variable. ¿Es esto esencial, o también podrían funcionar los procedimientos con los propios valores?

5. Consideremos un sistema donde los hilos se implantan por completo dentro del espacio del usuario y el sistema de tiempo de ejecución logra una interrupción del reloj una vez por cada segundo. Supongamos que ocurre una interrupción del reloj mientras se ejecuta un hilo en el sistema de tiempo de ejecución. ¿Qué problemas se pueden presentar? ¿Puede usted sugerir una forma de resolverlos?

6. Supongamos que un sistema operativo no tiene algo parecido a la llamada SELECT para ver de antemano si es seguro leer de un archivo, entubamiento o dispositivo, pero que permite establecer alarmas del reloj para interrumpir las llamadas bloqueadas. Bajo estas condiciones, ¿es posible implantar un paquete de hilos en el espacio del usuario? Analice.

7. En cierto sistema basado en estaciones de trabajo, éstas tienen discos locales que contienen los binarios del sistema. Cuando surge una nueva versión de un binario, ésta se envía a cada estación. Sin embargo, ciertas estaciones pueden estar inactivas (o apagadas) cuando esto ocurra. Diseñe un algoritmo que permita una actualización automática, incluso aunque las máquinas estén inactivas.

8. ¿Puede usted pensar en otros tipos de archivos que se puedan guardar en forma segura en las estaciones de trabajo de los usuarios del tipo descrito en el ejercicio anterior?

9. ¿Funciona el esquema de Bershad *et al.*, (para hacer más rápida la RPC local) en un sistema con un hilo por cada proceso? ¿Qué hay acerca del método de Peregrine?

10. Cuando dos usuarios examinan en forma simultánea el registro de la figura 4-12, pueden elegir accidentalmente la misma estación inactiva. ¿Cómo se puede hacer una modificación sutil al algoritmo de modo que no ocurra esta competencia?

11. Imaginemos que un proceso se ejecuta en forma remota en una estación de trabajo previamente inactiva, la cual, como todas las demás estaciones, carece de discos. Para cada una de las siguientes llamadas al sistema de UNIX, indique si debe regresarse a la máquina de origen:
(a) READ (obtener datos de un archivo).

(b) IOCTL(cambiar el modo de la terminal controladora).
(c) GETPID(regresar el identificador del proceso).

12. Calcule las tasas de respuesta para la figura 4-15, con el procesador 1 como el procesador de referencia. ¿Cuál asignación minimiza la tasa de respuesta?

13. En el análisis de los algoritmos para la asignación de procesadores, señalamos que una opción era entre los centralizados y los distribuidos y que otra era entre los óptimos y los subóptimos. Diseñe dos algoritmos óptimos de localización, uno centralizado y otro descentralizado.

14. En la figura 4-17 vemos dos esquemas distintos de asignación, con distintas magnitudes del tráfico en la red. ¿Existen otras asignaciones que sean todavía mejores? Suponga que ninguna máquina puede ejecutar más de cuatro procesos.

15. El algoritmo arriba-abajo descrito en el texto es un algoritmo centralizado diseñado para asignar procesadores en forma justa. Invente un algoritmo centralizado cuyo objetivo sea no ser justo, sino que distribuya la carga de manera uniforme.

16. Cuando cierto sistema distribuido se sobrecarga, realiza m intentos por encontrar una estación de trabajo inactiva para descargarle algo de trabajo. La probabilidad de que una estación de trabajo tenga k tareas está dada por la fórmula de Poisson

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

donde λ es el promedio de tareas por estación de trabajo. ¿Cuál es la probabilidad de que una estación de trabajo sobrecargada encuentre una inactiva (es decir, una con $k = 0$) en m intentos? Evalúe su respuesta numéricamente con $m = 3$ y valores de λ de 1 a 4.

17. Con los datos de la figura 4-20, ¿cuál es el entubamiento de mayor longitud en UNIX que se puede coplanificar?

18. ¿Puede controlar el modelo de redundancia modular triple descrito en el texto las fallas bizantinas?

19. ¿Cuántos elementos con falla (dispositivos más votantes) puede controlar la figura 4-21? Dé un ejemplo del peor de los casos que puede encubrir.

20. ¿Se puede generalizar TMR a cinco elementos por grupo en vez de tres? En tal caso, ¿qué propiedades tiene?

21. Eloisa vive en el Hotel Plaza. Su actividad favorita es pararse en el recibidor, oprimiendo el botón del elevador. Ella puede hacer eso horas y horas. El hótel está instalando un nuevo sistema de elevador. Pueden elegir entre un sistema activado por el tiempo y un sistema activado por eventos. ¿Cuál deberían elegir? Analice.

22. Un sistema de tiempo real tiene procesos periódicos con los siguientes requisitos y periodos de cómputo:

P1: 20 milisegundos cada 40 milisegundos
P2: 60 milisegundos cada 500 milisegundos
P3: 5 milisegundos cada 20 milisegundos
P4: 15 milisegundos cada 100 milisegundos

¿Se puede planear este sistema en un CPU?

23. ¿Es posible determinar las prioridades que asignaría el algoritmo monótono de tasa a los procesos del problema anterior? En tal caso, ¿cuáles son? En caso contrario, ¿qué información falta?

24. Una red consta de dos cables paralelos: el enlace hacia adelante, en el que los paquetes viajan de izquierda a derecha, y el enlace inverso, en el que viajan de derecha a izquierda. Un generador en la punta de cada cable genera un flujo continuo de marcos de paquetes, cada uno de los cuales tiene un bit vacío/lleno (inicialmente vacío). Todas las computadoras están localizadas entre los dos cables, conectadas a ambos. Para enviar un paquete, una computadora determina el cable por utilizar (según si el destino está a la izquierda o la derecha de ella), espera un marco vacío, coloca un paquete en él y marca el marco como lleno. ¿Satisface esta red los requisitos para un sistema de tiempo real? Explique.

25. La asignación de procesos a espacios en la figura 4-32 es arbitraria. Son posibles otras asignaciones. Determine una asignación alternativa que mejore el desempeño del segundo ejemplo.

5

Sistemas distribuidos de archivos

Un componente fundamental de cualquier sistema distribuido es el sistema de archivos. Como en el caso de los sistemas con un procesador, la tarea del sistema de archivos en los sistemas distribuidos es almacenar los programas y los datos y tenerlos disponibles cuando sea necesario. Muchos de los aspectos de los sistemas distribuidos de archivos son similares a los de los sistemas convencionales, por lo que no repetiremos ese material. En vez de esto, nos concentraremos en aquellos aspectos de los sistemas distribuidos de archivos distintos al caso centralizado.

Para comenzar, en el caso de un sistema distribuido es importante distinguir entre los conceptos de servicio de archivos y el servidor de archivos. El **servicio de archivos** es la especificación de los servicios que el sistema de archivos ofrece a sus clientes. Describe las primitivas disponibles, los parámetros que utilizan y las acciones que llevan a cabo. Para los clientes, el servicio de archivos define con precisión el servicio con que pueden contar, pero no dice nada con respecto a su implantación. De hecho, el servicio de archivos especifica la interfaz del sistema de archivos con los clientes.

Por el contrario, un **servidor de archivos** es un proceso que se ejecuta en alguna máquina y ayuda a implantar el servicio de archivos. Un sistema puede tener uno o varios servidores de archivos, pero los clientes no deben conocer el número de servidores de archivos, su posición o función. Todo lo que saben es que al llamar los procedimientos especificados en el servicio de archivos, el trabajo necesario se lleva a cabo de alguna manera y se obtienen los resultados pedidos. De hecho, los clientes ni siquiera deben saber que el servicio de archivos es distribuido. Lo ideal es que se vea como un sistema de archivos normal de un procesador.

Puesto que un servidor de archivos es por lo general un proceso del usuario (o a veces un proceso del núcleo) que se ejecuta en una máquina, un sistema puede contener varios servidores de archivos, cada uno de los cuales ofrece un servicio de archivos distinto. Por

ejemplo, un sistema distribuido podría tener dos servidores que ofrezcan el servicio de archivos en UNIX y el servicio de archivos en MS-DOS, respectivamente, donde cada proceso usuario utilizaría el servidor apropiado. De esa forma, es posible que una terminal tenga varias ventanas y que en algunas de ellas se ejecuten programas en UNIX y en otras programas en MS-DOS, sin que esto provoque conflictos. Los diseñadores del sistema se encargan de que los servidores ofrezcan los servicios de archivo específicos, como UNIX o MS-DOS. El tipo y número de servicios de archivo disponibles puede cambiar con la evolución del sistema.

5.1. DISEÑO DE LOS SISTEMAS DISTRIBUIDOS DE ARCHIVOS

Por lo general, un sistema distribuido de archivos tiene dos componentes razonablemente distintos: el verdadero servicio de archivos y el servicio de directorios. El primero se encarga de las operaciones en los archivos individuales, como la lectura, escritura y adición, mientras que el segundo se encarga de crear y administrar directorios, añadir y eliminar archivos de los directorios, etc. En esta sección analizaremos la interfaz del verdadero servicio de archivos; en la siguiente analizaremos la interfaz del servicio de directorios.

5.1.1. La interfaz del servicio de archivos

El aspecto fundamental para cualquier servicio de archivos, ya sea para un procesador o un sistema distribuido, es la pregunta: "¿Qué es un archivo?" En muchos sistemas, como UNIX y MS-DOS, un archivo es una secuencia de bytes sin interpretación alguna. El significado y estructura de la información en los archivos queda a cargo de los programas de aplicación; esto no le interesa al sistema operativo.

Sin embargo, en los mainframes existen muchos tipos de archivos, cada uno con distintas propiedades. Por ejemplo, un archivo se puede estructurar como serie de registros, con llamadas al sistema operativo para leer o escribir un registro particular. Por lo general, se puede especificar el registro mediante su número (es decir, su posición dentro del archivo) o el valor de cierto campo. En el segundo caso, el sistema operativo mantiene al archivo como un árbol B o alguna otra estructura de datos adecuada, o bien utiliza tablas de dispersión para localizar con rapidez los registros. Puesto que la mayoría de los sistemas distribuidos están planeados para ambientes UNIX o MS-DOS, la mayoría de los servidores de archivos soporta el concepto de archivo como secuencia de bytes, en vez de una secuencia de registros con cierta clave.

Un archivo puede tener **atributos**, partes de información relativas a él pero que no son parte del archivo propiamente dicho. Los atributos típicos son el propietario, el tamaño, la fecha de creación y el permiso de acceso. Por lo general, el servicio de archivos proporciona primitivas para leer y escribir en alguno de los atributos. Por ejemplo, se pueden modificar los permisos de acceso pero no el tamaño (a menos que se agreguen datos al archivo). En

unos cuantos sistemas avanzados, se podrían crear y administrar atributos definidos por el usuario además de los usuales.

Otro aspecto importante del modelo de archivo es si los archivos se pueden modificar después de su creación. Lo normal es que sí se puedan modificar, pero en algunos sistemas distribuidos, las únicas operaciones de archivo son CREATE y READ. Una vez creado un archivo, no puede ser modificado. Se dice que tal archivo es **inmutable**. El hecho de contar con archivos inmutables facilita el soporte del ocultamiento y duplicación de archivos, puesto que esto elimina todos los problemas asociados con la actualización de todas las copias de un archivo cada vez que éste se modifique.

La protección en los sistemas distribuidos utiliza en esencia las mismas técnicas de los sistemas con un procesador: posibilidades y listas para control de acceso. En el caso de las posibilidades, cada usuario tiene cierto tipo de boleto, llamado **posibilidad**, para cada objeto al que tiene acceso. La posibilidad determina los tipos de acceso permitidos (por ejemplo, se permite la lectura pero no la escritura).

Todos los esquemas de **lista para control de acceso** asocian a cada archivo una lista implícita o explícita de los usuarios que pueden tener acceso al archivo y la forma de dicho acceso. El esquema de UNIX es una lista para control de acceso simplificada, con bits que controlan la lectura, escritura y ejecución de cada archivo, en forma independiente para el propietario, el grupo del propietario y todas las demás personas.

Los servicios de archivos se pueden dividir en dos tipos, según si soportan un modelo carga/descarga o un modelo de acceso remoto. En el **modelo carga/descarga**, que se muestra en la figura 5-1(a), el servicio de archivos sólo proporciona dos operaciones principales: la lectura de un archivo y la escritura del mismo. La primera operación transfiere todo un archivo de uno de los servidores de archivos al cliente solicitante. La segunda operación transfiere todo un archivo en sentido contrario, del cliente al servidor. Así, el modelo conceptual es el traslado de archivos completos en alguna de las direcciones. Los archivos se pueden almacenar en memoria o en un disco local, como sea necesario.

La ventaja del modelo carga/descarga es la sencillez del concepto. Los programas de aplicación buscan los archivos que necesitan y después los utilizan de manera local. Los

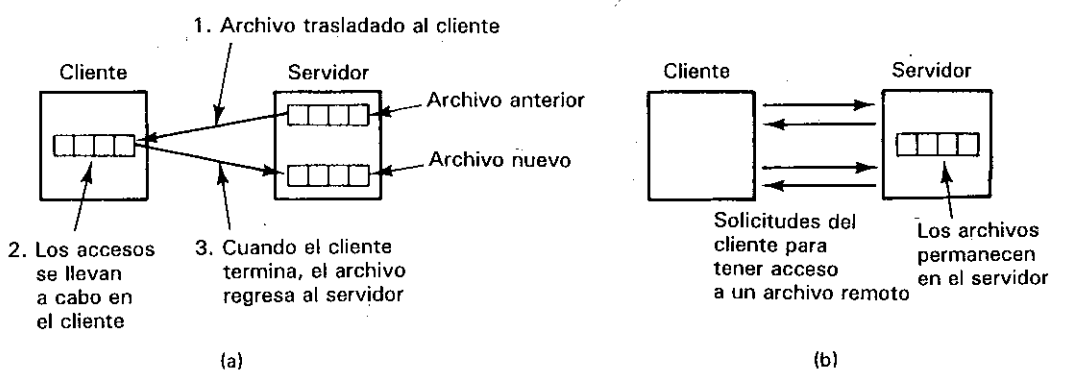


Figura 5-1. (a) El modelo carga/descarga. (b) El modelo de acceso remoto.

archivos modificados o nuevos se escriben de regreso al terminar el programa. No hay que administrar una complicada interfaz del servicio de archivos para utilizar este modelo. Además, la transferencia de archivos completos es muy eficiente. Sin embargo, el cliente debe disponer de un espacio suficiente de almacenamiento para todos los archivos necesarios. Además, si sólo se necesita una pequeña fracción de un archivo, el traslado del archivo completo es un desperdicio.

El otro tipo de servicio de archivos es el **modelo de acceso remoto**, que se muestra en la figura 5-1(b). En este modelo, el servicio de archivos proporciona gran número de operaciones para abrir y cerrar archivos, leer y escribir partes de archivos, moverse a través de un archivo (LSEEK), examinar y modificar los atributos de archivo, etc. Mientras en el modelo carga/descarga el servicio de archivos sólo proporciona el almacenamiento físico y la transferencia, en este caso el sistema de archivos se ejecuta en los servidores y no en los clientes. Su ventaja es que no necesita mucho espacio por parte de los clientes, a la vez que elimina la necesidad de transferir archivos completos cuando sólo se necesita una pequeña parte de ellos.

5.1.2. La interfaz del servidor de directorios

La otra parte del servicio de archivos es el servicio de directorios, el cual proporciona las operaciones para crear y eliminar directorios, nombrar o cambiar el nombre de archivos y mover éstos de un directorio a otro. La naturaleza del servicio de directorios no depende del hecho de que los archivos individuales se transfieran en su totalidad o que se tenga un acceso remoto a ellos.

El servicio de directorios define un alfabeto y una sintaxis para formar los nombres de archivos (y directorios). Lo usual es que los nombres de archivos tengan de 1 hasta un cierto número máximo de letras, números y ciertos caracteres especiales. Algunos sistemas dividen los nombres de archivo en dos partes, a menudo separadas mediante un punto, como *prog.c* para un programa en C o *man.txt* para un archivo de texto. La segunda parte del nombre, llamada la **extensión de archivo**, identifica el tipo de éste. Otros sistemas utilizan un atributo explícito para este fin, en vez de utilizar una extensión dentro del nombre.

Todos los sistemas distribuidos permiten que los directorios contengan subdirectorios, para que los usuarios puedan agrupar los archivos relacionados entre sí. De acuerdo con esto, se dispone de operaciones para la creación y eliminación de directorios, así como para introducir, eliminar y buscar archivos en ellos. Por lo general, cada subdirectorio contiene todos los archivos de un proyecto, como un programa o documento de gran tamaño (por ejemplo, un libro). Cuando se despliega el (sub)directorio, sólo se muestran los archivos relevantes; los archivos no relacionados están en otros (sub)directorios y no agrandan la lista. Los subdirectorios pueden contener sus propios subdirectorios y así en lo sucesivo, lo que conduce a un árbol de directorios, el cual se conoce como **sistema jerárquico de archivos**. La figura 5-2(a) muestra un árbol con cinco directorios.

En ciertos sistemas, es posible crear enlaces o apuntadores a un directorio arbitrario. Éstos se pueden colocar en cualquier directorio, lo que permite construir no sólo árboles,

sino gráficas arbitrarias de directorios, que son más poderosas. La distinción entre árboles y gráficas es de particular importancia en un sistema distribuido.

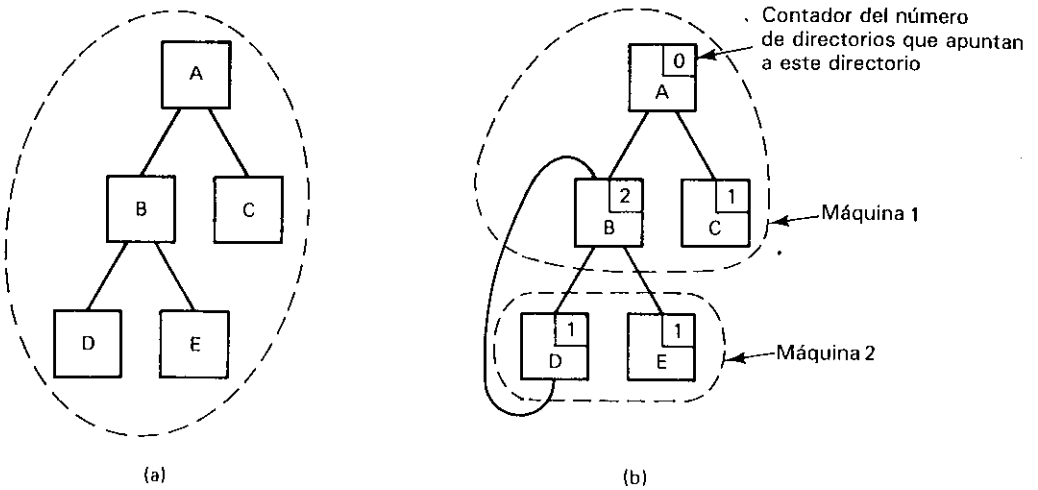


Figura 5-2. (a) Un árbol de directorios contenido en una máquina. (b) Una gráfica de directorios en dos máquinas.

La naturaleza de la dificultad se puede ver en la gráfica de directorios de la figura 5-2(b). En ésta, el directorio *D* tiene un enlace con el directorio *B*. El problema aparece cuando se elimina el enlace de *A* a *B*. En una jerarquía con estructura de árbol, sólo se puede eliminar un enlace con un directorio si el directorio al cual se apunta es vacío. En una gráfica, se permite la eliminación de un enlace mientras exista al menos otro enlace. Mediante un contador de referencias, el cual se muestra en la esquina superior derecha de cada directorio de la figura 5-2(b), se puede determinar si el enlace por eliminar es el último.

Después de eliminar el enlace de *A* a *B*, el contador de referencias de *B* se reduce de 2 a 1, lo cual está bien en el papel. Sin embargo, ahora no es posible llegar a *B* desde la raíz del sistema de archivos (*A*). Los tres directorios *B*, *D* y *E* y todos sus archivos se convierten en huérfanos.

Este problema también existe en los sistemas centralizados, pero es más serio en los distribuidos. Si todo está en una máquina, es posible, aunque costoso, descubrir los directorios huérfanos, puesto que toda la información está en un lugar. Se puede detener toda la actividad de los archivos y recorrer la gráfica desde la raíz, para señalar todos los directorios alcanzables. Al final de este proceso, se sabe que todos los directorios no marcados son inalcanzables. En un sistema distribuido existen varias máquinas y no se puede detener toda la actividad, por lo que es difícil, sino es que imposible, tomar una foto "instantánea".

Un aspecto fundamental en el diseño de cualquier sistema distribuido de archivos es si todas las máquinas (y procesos) deben tener con exactitud la misma visión de la jerarquía de los directorios. Como ejemplo de lo que queremos decir en esta observación, consideremos la figura 5-3. En la figura 5-3(a) mostramos dos servidores de archivos, cada uno de

los cuales tiene tres directorios y algunos archivos. En la figura 5-3(b) tenemos un sistema donde todos los clientes (y otras máquinas) tienen la misma visión del sistema distribuido de archivos. Si la trayectoria */D/E/x* es válida en una máquina, entonces es válida en todas ellas.

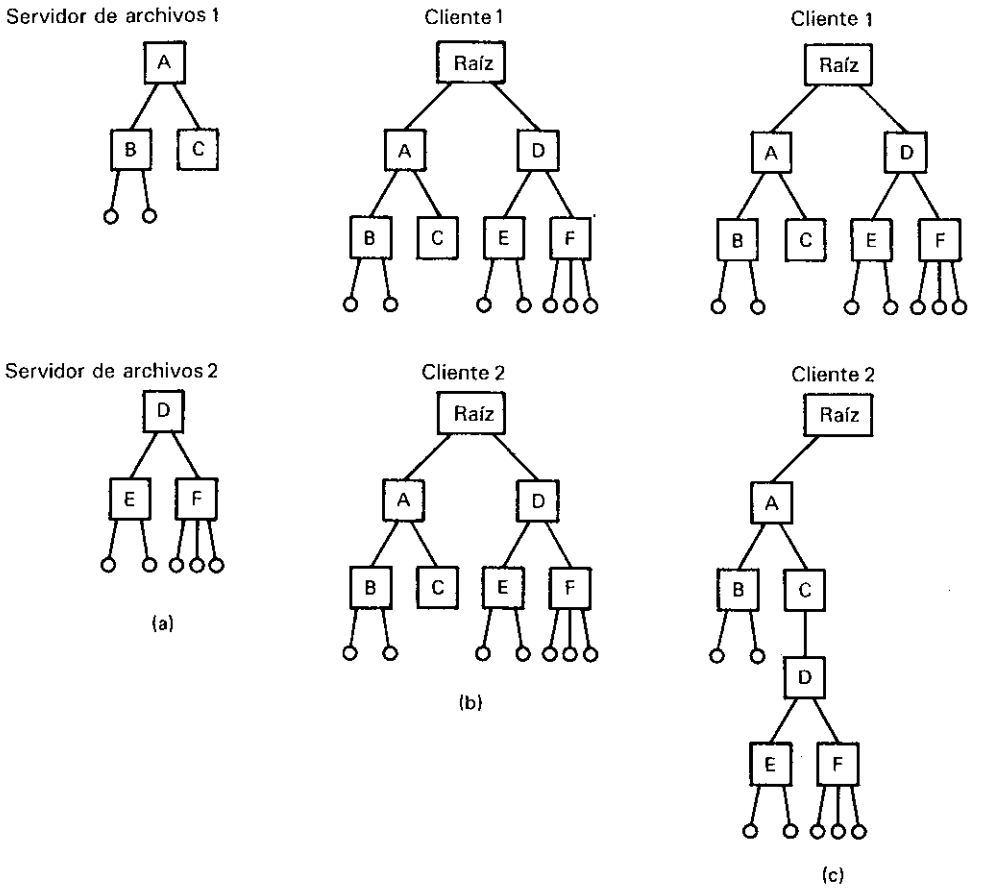


Figura 5-3. (a) Dos servidores de archivos. Los cuadrados son directorios y los círculos son archivos. (b) Un sistema en el cual todos los clientes tienen la misma visión del sistema de archivos. (c) Un sistema en el cual los diversos clientes tienen diferente visión del sistema de archivos.

Por el contrario, en la figura 5-3(c), las diferentes máquinas pueden tener visiones diferentes del sistema de archivos. Para repetir el ejemplo anterior, la trayectoria */D/E/x* podría ser válida en el cliente 1 pero no en el cliente 2. En los sistemas que administran varios servidores de archivos mediante el montaje remoto, la norma es la figura 5-3(c). Es flexible y tiene implantación directa, pero tiene la desventaja de que el sistema no se comporta como un sistema de tiempo compartido tradicional. En un sistema de tiempo compartido, el

sistema de archivos se ve igual para todos los procesos [es decir, el modelo de la figura 5-3(b)]. Esta propiedad hace que un sistema sea fácil de programar y comprender.

Una cuestión muy relacionada con esto es si existe un directorio raíz global, al que todas las máquinas reconozcan como la raíz. Una vía para tener un directorio raíz global es que dicha raíz sólo contenga una entrada por cada servidor. En estas circunstancias, las trayectorias toman la forma */servidor/ruta*, que tiene sus propias desventajas, pero al menos es la misma en todas las partes del sistema.

Transparencia de los nombres

El principal problema de esta forma de los nombres es que no es por completo transparente. En este contexto, son relevantes dos formas de transparencia y es importante distinguirlas. La primera, la **transparencia con respecto a la posición**, significa que el nombre de la ruta de acceso no sugiere la posición del archivo (o de algún otro objeto). Una ruta como */servidor1/dir1/dir2/x* indica que *x* está localizado en el servidor 1, pero no indica la posición del servidor. Éste es libre de moverse dentro de la red, sin que el nombre de la ruta de acceso deba ser modificada. Así, este sistema es transparente con respecto a la posición.

Sin embargo, supongamos que el archivo *x* es muy grande y que hay poco espacio en el servidor 1. Además, supongamos que hay mucho espacio en el servidor 2. El sistema podría desplazar de forma automática *x* al servidor 2. Por desgracia, si el primer componente de todas las rutas de acceso es el servidor, el sistema no puede desplazar el archivo al otro servidor en forma automática, aunque *dir1* y *dir2* existieran en ambos servidores. El problema es que el desplazamiento automático del archivo cambia el nombre de su ruta de acceso, de */servidor1/dir1/dir2/x* a */servidor2/dir1/dir2/x*. Los programas que tienen integrada la primera cadena no podrán funcionar si la ruta de acceso se modifica. Un sistema donde los archivos se pueden desplazar sin que cambien sus nombres tiene **independencia con respecto a la posición**. Un sistema distribuido que incluya los nombres de la máquina o el servidor en los nombres de las rutas de acceso no es independiente con respecto a la posición. Tampoco lo es uno basado en el montaje remoto, puesto que no es posible desplazar un archivo de un grupo de archivos (la unidad de montaje) a otro y conservar el antiguo nombre de la ruta de acceso. La independencia con respecto a la posición no es fácil de lograr, pero es una propiedad deseable en un sistema distribuido.

Para resumir lo anterior, existen tres métodos usuales para nombrar los archivos y directorios en un sistema distribuido:

1. Nombre máquina + ruta de acceso, como */máquina/ruta* o *máquina:ruta*.
2. Montaje de sistemas de archivos remotos en la jerarquía local de archivos.
3. Un espacio de nombres que tenga la misma apariencia en todas las máquinas.

Los primeros dos son fáciles de implantar, en particular como una forma de conectar sistemas ya existentes que no estaban diseñados para su uso distribuido. El tercer método es

difícil y requiere de un diseño cuidadoso, pero es necesario si se quiere lograr el objetivo de que el sistema distribuido actúe como una computadora.

Nombres de dos niveles

La mayoría de los sistemas distribuidos utilizan cierta forma de nombres con dos niveles. Los archivos (y otros objetos) tienen **nombres simbólicos**, como *prog.c*, para uso de las personas, pero también pueden tener **nombres binarios** internos, para uso del propio sistema. Lo que los directorios hacen en realidad es proporcionar una asociación entre estos dos nombres. Para las personas y los programas, es conveniente utilizar nombres simbólicos (ASCII), pero para el uso dentro del propio sistema, estos nombres son muy grandes y difíciles. Así, cuando un usuario abre un archivo o hace referencia a un nombre simbólico, el sistema busca de inmediato el nombre simbólico en el directorio apropiado para obtener el nombre binario, el cual utilizará para localizar en realidad al archivo. A veces, los nombres binarios son visibles a los usuarios y a veces no.

La naturaleza de los nombres binarios varía mucho de un sistema a otro. En un sistema con varios servidores de archivos, cada uno de los cuales esté autocontenido (es decir, no tenga referencias a directorios o archivos en otros servidores), el nombre binario puede ser sólo un número de un nodo-i local, como en UNIX.

Un esquema más general para los nombres es que el nombre binario indique el servidor y un archivo específico en ese servidor. Este método permite que un directorio en un servidor contenga un archivo en un servidor distinto. Otra alternativa, que a veces es preferible, es utilizar un **enlace simbólico**. Un enlace simbólico es una entrada de directorio asociada a una cadena (servidor, nombre de archivo), la cual se puede buscar en el servidor correspondiente para encontrar el nombre binario. El propio enlace simbólico es sólo el nombre de una ruta de acceso.

Otra idea más es utilizar las posibilidades como los nombres binarios. En este método, la búsqueda de un nombre en ASCII produce una posibilidad, la cual puede tomar una de varias formas. Por ejemplo, puede contener un número físico o lógico de una máquina o la dirección en la red del servidor apropiado, así como un número que indique el archivo específico necesario. Se puede utilizar una dirección física para enviar un mensaje al servidor sin mayor interpretación. Una dirección lógica se puede localizar mediante una transmisión o mediante una búsqueda en un servidor de nombres.

Un último giro que a veces está presente en un sistema distribuido, pero casi nunca en uno centralizado, es la posibilidad de buscar un nombre en ASCII y obtener no *uno*, sino *varios* nombres binarios (nodos-i, posibilidades, o alguna otra cosa). Por lo general, éstos representan al archivo original y todos sus respaldos. Con varios nombres binarios, es posible entonces intentar la localización de uno de los archivos correspondientes; si éste no está disponible por alguna razón, se intenta con los otros. Este método proporciona cierto grado de tolerancia de fallas por medio de la redundancia.

5.1.3. Semántica de los archivos compartidos

Si dos o más usuarios comparten el mismo archivo, es necesario definir con precisión la semántica de la lectura y escritura para evitar problemas. En los sistemas con un procesador que permiten a los procesos compartir archivos, como UNIX, la semántica establece por lo general que si una operación READ sigue después de una operación WRITE, READ regresa el valor recién escrito, como se muestra en la figura 5-4(a). De manera análoga, cuando dos WRITE se realizan en serie y después se ejecuta un READ, el valor que se lee es el almacenado en la última escritura. De hecho, el sistema impone en todas las operaciones un orden absoluto con respecto del tiempo y siempre regresa el valor más reciente. Nos referiremos a este modelo como la **semántica de UNIX.** Este modelo es fácil de comprender y tiene una implantación directa.

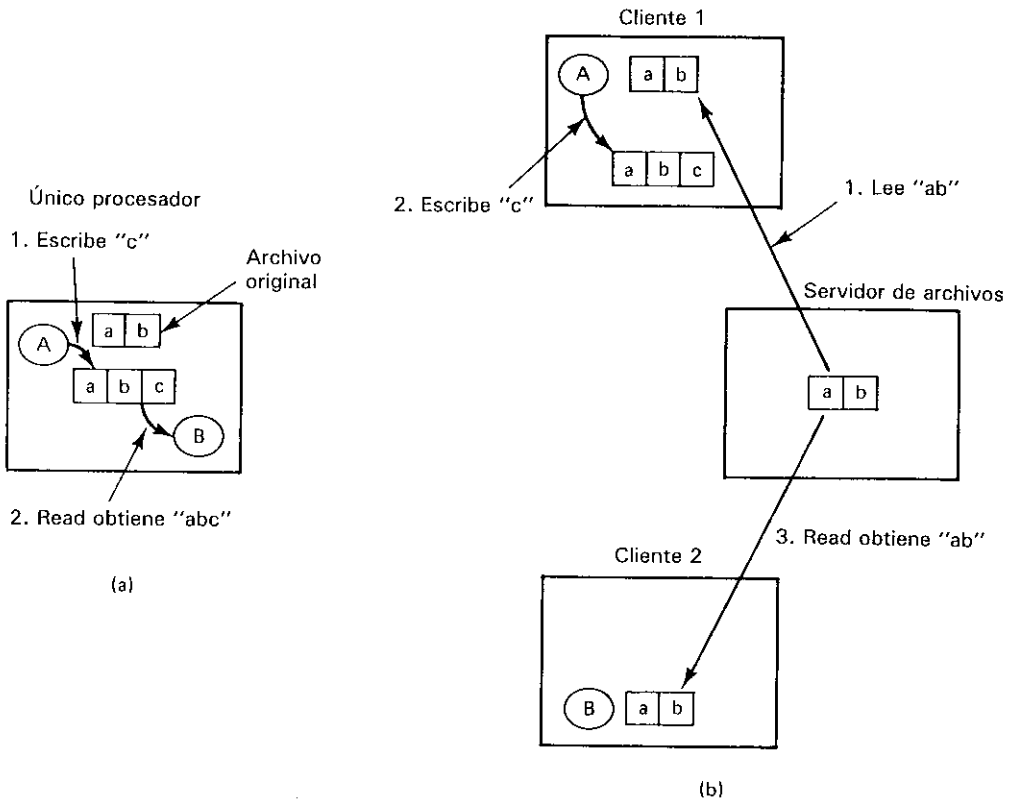


Figura 5-4. (a) En un procesador, cuando un READ va después de un WRITE, el valor regresado por READ es el valor recién escrito. (b) En un sistema distribuido con ocultamiento, el valor regresado puede ser obsoleto.

En un sistema distribuido, la semántica de UNIX se puede lograr fácilmente, mientras sólo exista un servidor de archivos y los clientes no oculten los archivos. Todas las instrucciones READ y WRITE pasan en forma directa al servidor de archivos, que los procesa en

forma secuencial. Este método proporciona la semántica de UNIX (excepto por un problema menor: los retrasos en la red pueden hacer que un READ ocurrido un microsegundo después de un WRITE llegue primero al servidor y que obtenga el valor anterior).

Sin embargo, en la práctica, el desempeño de un sistema distribuido donde todas las solicitudes de archivos deban pasar a un servidor con frecuencia es pobre. Este problema se puede resolver si se permite a los clientes que mantengan copias locales de los archivos de uso frecuente en sus cachés particulares. Aunque analizaremos más adelante los detalles del ocultamiento de archivos, por el momento basta señalar que si un cliente modifica en forma local un archivo en caché y poco después otro cliente lee el archivo del servidor, el segundo cliente obtendrá un archivo obsoleto, como se muestra en la figura 5-4(b).

Una forma de salir de esta dificultad es propagar de manera inmediata todas las modificaciones de los archivos en caché de regreso al servidor. Aunque esto es sencillo desde el punto de vista conceptual, el método es ineficiente. Otra solución consiste en relajar la semántica de los archivos compartidos. En vez de pedir que un READ vea los efectos de todos los WRITE anteriores, uno puede tener una nueva regla que diga: "los cambios a un archivo abierto sólo pueden ser vistos en un principio por el proceso (o tal vez la máquina) que modificó el archivo. Los cambios serán visibles a los demás procesos (o máquinas) sólo cuando se cierre el archivo". La adopción de esta regla no modifica lo que ocurre en la figura 5-4(b), pero redefine el comportamiento (*B* obtiene el valor original del archivo) como el correcto. Cuando *A* cierra el archivo, envía una copia al servidor, de modo que los siguientes READ obtienen el nuevo valor, como se pide. Esta regla es de uso común y se conoce como **semántica de sesión**.

El uso de la semántica de sesión hace surgir la pregunta de lo que ocurre si dos o más clientes ocultan y modifican el mismo archivo en forma simultánea. Una solución consiste en decir que, al cerrarse cada archivo, su valor se envía de regreso al servidor, de modo que el resultado final depende de quién lo cierre más rápido. Una alternativa menos agradable, pero más fácil de implantar, es decir, que el resultado final es uno de los candidatos, pero no se especifica la elección de uno de ellos.

La dificultad final con el uso de cachés y la semántica de sesión es que viola otro aspecto de la semántica de UNIX además del hecho de que no todos los READ regresen el valor de escritura más reciente. En UNIX, a cada archivo abierto se le asocia un apuntador que indica la posición actual en el archivo. Una instrucción READ toma los datos a partir de esa posición y WRITE deposita los datos ahí. Este apuntador es compartido por los procesos que abrieron el archivo y todos sus hijos. Con la semántica de sesión, cuando los hijos se ejecutan en máquinas distintas, no se puede lograr compartir el archivo.

Para ver las consecuencias del hecho de abandonar los apuntadores de archivo compartidos, consideremos un comando como

```
run >out
```

en donde *run* es un guión del shell que ejecuta dos programas, *a* y *b*, uno después del otro. Si ambos programas producen una salida, se espera que la salida producida por *b* continúe directamente después de la salida de *a* dentro de *out*. La forma de lograr esto es que al

iniciar *b*, éste herede el apuntador de archivo de *a*, el cual es compartido por el shell y ambos procesos. De esta forma, el primer byte donde escriba *b* será el inmediato posterior al último byte escrito por *a*. Con la semántica de sesión y sin apuntadores compartidos, se necesita un mecanismo por completo diferente para que funcionen los guiones del shell y otras construcciones similares que utilizan los apuntadores a archivos compartidos. Puesto que no se conoce una solución de propósito general para este problema, cada sistema lo debe enfrentar de una manera *ad hoc*.

Un método por completo distinto a la semántica de los archivos compartidos en un sistema distribuido es que todos los archivos sean inmutables. Así, no existe forma de abrir un archivo para escribir en él. En efecto, las únicas operaciones en los archivos son CREATE y READ.

Lo que es posible es crear un archivo por completo nuevo e introducirlo en el sistema de directorios, con el nombre de un archivo ya existente, el cual se vuelve inaccesible (al menos con ese nombre). Así, aunque se vuelve imposible modificar el archivo *x*, es posible remplazarlo (en forma atómica) por un archivo nuevo. En otras palabras, aunque los *archivos* no se pueden actualizar, los *directorios* sí. Una vez que hemos decidido que los archivos no se pueden modificar, desaparece el problema de enfrentarse a dos procesos, uno de los cuales escribe en un archivo y el otro lo lee.

Sigue presente el problema de qué hacer si dos procesos intentan remplazar el mismo archivo a la vez. Como en el caso de la semántica de sesión, parece que la mejor solución es permitir que uno de los nuevos archivos remplace al anterior, ya sea el último u otro dado de manera no determinista.

Un problema más molesto consiste en qué hacer si un archivo se remplaza mientras otro proceso está ocupado leyéndolo. Una solución es arreglárselas de tal forma que el lector utilice el archivo anterior, aunque éste ya no exista en directorio alguno, en forma análoga al hecho de que UNIX permite que un proceso con un archivo abierto continúe utilizándolo, aun cuando éste haya sido eliminado de todos los directorios. Otra solución consiste en detectar la modificación del archivo y hacer que fallen los intentos posteriores por leerlo.

Una cuarta vía para enfrentar el uso de los archivos compartidos en un sistema distribuido es usar las transacciones atómicas, analizadas con detalle en el capítulo 3, pero que aquí resumiremos. Para tener acceso a un archivo o grupo de archivos, un proceso ejecuta en primer lugar cierto tipo de primitiva BEGINTRANSACTION para señalar que lo que sigue debe ejecutarse de manera indivisible. Después vienen las llamadas al sistema para leer o escribir en uno o más archivos. Al terminar el trabajo, se ejecuta una primitiva END TRANSACTION. La propiedad fundamental de este método es que el sistema garantiza que todas las llamadas contenidas dentro de la transacción se llevarán a cabo en orden, sin interferencias de otras transacciones concurrentes. Si dos o más transacciones se realizan al mismo tiempo, el sistema garantiza que el resultado final es el mismo que si se ejecutasen en cierto orden secuencial (indeterminado).

El ejemplo clásico donde las transacciones facilitan la programación es un sistema bancario. Imaginemos que cierta cuenta bancaria contiene 100 dólares y que dos procesos intentan añadirle 50 dólares. En un sistema sin restricciones, cada proceso puede leer de

manera simultánea el archivo que contiene el balance actual (100), calcular en forma individual el nuevo balance (150) y escribir en el archivo el nuevo valor. El resultado final podría ser 150 o 200, según la sincronización de la lectura y la escritura. Al agrupar todas las operaciones en una transacción, los procesos no se pueden intercalar y el resultado final siempre será 200.

En la figura 5-5 resumimos los cuatro métodos analizados para utilizar los archivos compartidos en un sistema distribuido.

Método	Comentarios
Semántica de UNIX	Cada operación en un archivo es visible a todos los procesos de manera instantánea
Semántica de sesión	Ningún cambio es visible a otros procesos hasta que el archivo se cierra
Archivos inmutables	No existen actualizaciones; es más fácil compartir y replicar
Transacciones	Todos los cambios tienen la propiedad del todo o nada

Figura 5-5. Cuatro maneras de compartir archivos en un sistema distribuido.

5.2. IMPLANTACIÓN DE UN SISTEMA DISTRIBUIDO DE ARCHIVOS

En la sección anterior describimos varios aspectos de los sistemas distribuidos de archivos, desde el punto de vista del usuario; es decir, cómo se ven ante el usuario. En esta sección veremos la forma en que se implantan dichos sistemas. Comenzaremos con la presentación de cierta información experimental acerca del uso de los archivos. Después revisaremos la estructura del sistema, la implantación del ocultamiento, la réplica y el control de la concurrencia. Por último, concluiremos con un breve análisis de algunas lecciones que nos ha dado la experiencia.

5.2.1. Uso de archivos

Antes de implantar cualquier sistema, distribuido o no, es útil tener una buena idea de su posible uso, para garantizar la eficiencia de las operaciones de ejecución frecuente. Con este fin, Satyanarayanan (1981) hizo un amplio estudio de los patrones de uso de los archivos. Más adelante presentaremos sus principales conclusiones.

Sin embargo, en primer lugar debemos hacer unas advertencias acerca de éstas y otras mediciones. Algunas de las mediciones son estáticas, lo que quiere decir que representan una toma instantánea del sistema en cierto momento. Las mediciones estáticas se realizan al examinar el disco y ver lo que hay en él. Entre ellas se encuentran la distribución de tamaños de los archivos, la distribución de tipos de archivos y la cantidad de espacio que ocupan los archivos de varios tamaños y tipos. Otras mediciones son dinámicas, se llevan

a cabo al modificar el sistema de archivos, de modo que registre todas las operaciones en una bitácora para un análisis posterior. Estos datos proporcionan información con respecto a la frecuencia relativa de varias operaciones, el número de archivos abiertos en un momento dado y la cantidad de hechos compartidos. Al combinar las medidas estáticas y dinámicas, aunque sean diferentes en lo fundamental, obtenemos una mejor idea de la forma de uso del sistema.

Un problema siempre presente en las mediciones de cualquier sistema existente es saber qué tan típica es la población observada. Las mediciones de Satyanarayanan fueron llevadas a cabo en una universidad. ¿También se pueden aplicar a laboratorios de investigación industrial? ¿A proyectos de automatización en oficinas? ¿A sistemas bancarios? Nadie lo sabe, hasta que estos sistemas se instrumenten y se realicen las mediciones en ellos.

Otro problema inherente en las mediciones es que se debe tener cuidado con las características que se miden en el sistema. Como ejemplo sencillo, al analizar la distribución de los nombres de archivo en un sistema MS-DOS, uno podría concluir con rapidez que los nombres de archivo nunca tienen más de ocho caracteres (más una extensión opcional de tres caracteres). Sin embargo, sería un error concluir de ahí que son suficientes ocho caracteres, puesto que nadie utiliza más de ocho. Puesto que MS-DOS no permite más de ocho caracteres en un nombre de archivo, es imposible decir lo que harían los usuarios si no tuvieran tal restricción en la longitud.

Por último, las mediciones de Satyanarayanan se llevaron a cabo en sistemas UNIX más o menos tradicionales. No se sabe con certeza si se pueden trasladar o extrapolar a los sistemas distribuidos.

Dicho esto, las conclusiones más importantes se muestran en la figura 5-6. A partir de estas observaciones, uno puede extraer ciertas conclusiones. Para comenzar, la mayoría de los archivos están por debajo de los 10K, lo que coincide con los resultados de Mullender y Tanenbaum (1984) obtenidos bajo circunstancias diferentes. Esta observación sugiere la factibilidad de la transferencia de archivos completos en vez de bloques de disco entre el servidor y el cliente. Puesto que la transferencia de archivos completos es por lo general más sencilla y eficiente, esta idea debe tomarse en cuenta. Por supuesto, algunos archivos son de gran tamaño, por lo que también se deben tomar medidas preventivas con respecto a éstos. Aún así, una buena norma consiste en optimizar el caso normal y tratar de modo especial el caso anormal.

Una observación interesante es que la mayoría de los archivos tienen tiempos de vida cortos. En otras palabras, un patrón común es crear un archivo, leerlo (una vez) y después eliminarlo. Un ejemplo común podría ser el de un compilador que crea archivos temporales para la transmisión de información entre sus distintas fases. Esto implica que sería una buena idea crear el archivo en el cliente y mantenerlo ahí hasta su eliminación. Esto elimina una cantidad importante de tráfico entre el cliente y el servidor.

El hecho de que unos cuantos archivos se compartan argumenta en favor del ocultamiento por parte del cliente. Como hemos visto, el ocultamiento complica la semántica,

La mayoría de los archivos son pequeños (menos de 10 K)
La lectura es más común que la escritura
La lectura y la escritura son secuenciales; es raro el acceso aleatorio
La mayoría de los archivos tienen una vida corta
Es poco usual compartir archivos
Los procesos promedio utilizan sólo unos cuantos archivos
Existen distintas clases de archivos con propiedades diferentes

Figura 5-6. Propiedades observadas de los sistemas de archivos.

pero si es raro el uso de los archivos compartidos, podría ser mejor el ocultamiento por parte del cliente y aceptar las consecuencias de la semántica de sesión en favor de un mejor desempeño.

Por último, la clara existencia de distintas clases de archivos sugiere que tal vez se deberían utilizar mecanismos diferentes para el manejo de las distintas clases. Los binarios del sistema necesitan estar presentes en diversas partes, pero es raro que se modifiquen, por lo que tal vez se podrían duplicar en varias partes, aunque esto implique una actualización ocasional compleja. Los compiladores y los archivos temporales son cortos, no compartidos y desaparecen con rapidez, por lo que deben mantener su carácter local mientras sea posible. Los buzones electrónicos se actualizan con frecuencia, pero es raro que se compartan, por lo que su réplica no sirve de mucho. Es posible compartir los archivos ordinarios de datos, por lo que éstos requieren de otro tipo de manejo.

5.2.2. Estructura del sistema

En esta sección analizaremos algunas de las formas de organización interna de los servidores de archivos y directorios, con atención especial a los métodos alternativos. Comenzaremos con una pregunta sencilla: "¿Son distintos los clientes y servidores?" Es sorprendente el hecho de que no exista acuerdo en este tema.

En ciertos sistemas no existe distinción alguna entre un cliente y un servidor. Todas las máquinas ejecutan el mismo software básico, de modo que una máquina que desee dar servicio de archivos al público en general es libre de hacerlo. Este ofrecimiento del servicio de archivos consiste sólo en exportar los nombres de los directorios seleccionados, de modo que otras máquinas puedan tener acceso a ellos.

En otros sistemas, el servidor de archivos y el de directorios son sólo programas del usuario, por lo que se puede configurar un sistema para que ejecute o no el software de

cliente o servidor en la misma máquina, como se desee. Por último, en el otro extremo están los sistemas donde los clientes y los servidores son máquinas esencialmente distintas, ya sea en términos de hardware o software. Los servidores y clientes pueden ejecutar incluso versiones distintas del sistema operativo. Aunque la separación de funciones es un poco más transparente, no existe razón fundamental para preferir un método por encima de los demás.

Un segundo aspecto de la implantación donde difieren los sistemas es la forma de estructurar el servicio a archivos y directorios. Una forma de organización consiste en combinar ambos en un servidor, que maneje todas las llamadas a directorios y archivos. Sin embargo, otra posibilidad es separarlos. En este caso, la apertura de un archivo exige ir hasta el servidor de directorios para asociar su nombre simbólico con el nombre binario (por ejemplo, máquina + nodo-i) y después ir hasta el servidor de archivos con el nombre en binario para llevar a cabo la lectura o escritura real del archivo.

El argumento en favor de la separación es que las dos funciones no tienen relación real entre sí, por lo que es más flexible mantenerlas separadas. Por ejemplo, se puede implantar un servidor de directorios en MS-DOS y otro servidor de directorios en UNIX donde ambos utilicen el mismo servidor de archivos para el almacenamiento físico. También es probable que la separación de funciones produzca un software más sencillo. Un contraargumento es que el hecho de contar con dos servidores requiere de mayor comunicación.

Por el momento, consideremos el caso de servidores de archivos y directorios independientes. En el caso normal, el cliente envía un nombre simbólico al servidor de directorios, que a su vez regresa el nombre en binario que comprende el servidor de archivos. Sin embargo, es posible que una jerarquía de directorios se reparta entre varios servidores, como se muestra en la figura 5-7. Por ejemplo, supongamos que tenemos un sistema en el que el directorio de trabajo, en el servidor 1, contiene una entrada *a* para otro directorio en el servidor 2. De manera similar, este directorio contiene una entrada *b* para un directorio en el servidor 3. Este tercer directorio contiene una entrada para un archivo *c*, junto con su nombre binario.

Para buscar */a/b/c*, el cliente envía un mensaje al servidor 1, que controla su directorio de trabajo. El servidor envía *a*, pero ve que el nombre binario se refiere a otro servidor. Ahora tiene dos alternativas. Puede indicar al cliente el servidor que contiene a *b* y que el cliente busque por su cuenta *b/c*, como se muestra en la figura 5-7(a), o puede enviar el resto de la solicitud al servidor 2 y no responder, como se muestra en la figura 5-7(b). El primer esquema requiere que el cliente sea consciente de cuál servidor contiene cuál directorio y necesita más mensajes. El segundo método es más eficiente, pero no se puede administrar mediante la RPC normal, puesto que el proceso adonde envía el cliente el mensaje no es el que envía la respuesta.

La búsqueda de nombres de rutas de acceso todo el tiempo, en particular si se utilizan varios servidores de directorios, puede ser cara. Algunos sistemas intentan mejorar su desempeño al mantener un caché de indicadores, es decir, de nombres buscados de manera reciente, así como los resultados de esas búsquedas. Al abrir un archivo, se verifica si el

caché contiene esa ruta de acceso. En caso afirmativo, se omite la búsqueda directorio por directorio y la dirección del binario se obtiene del caché. En caso contrario, se busca.

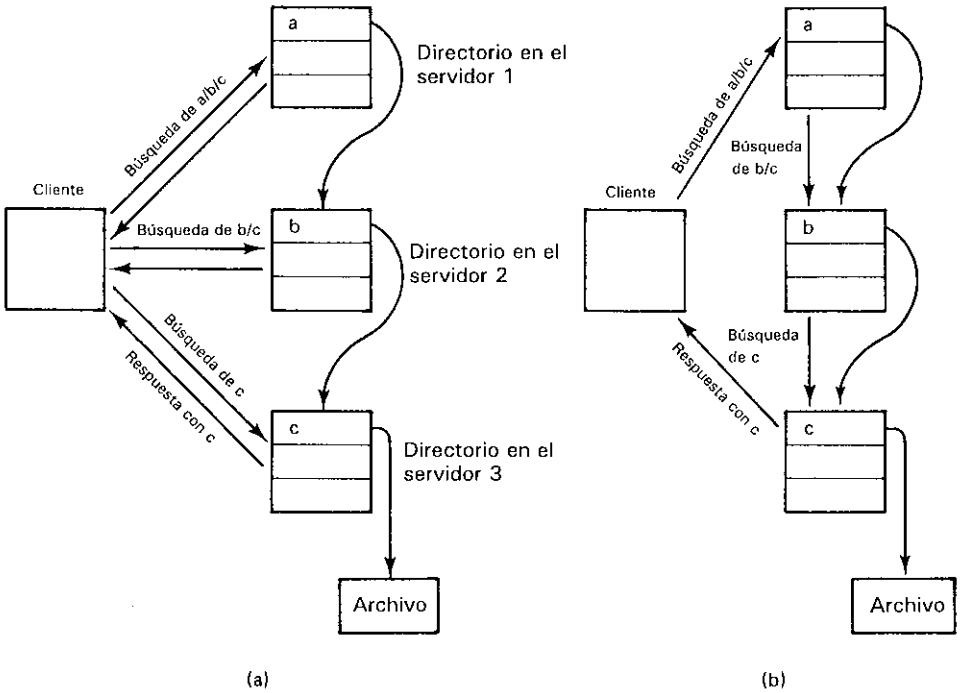


Figura 5-7. (a) Búsqueda iterativa de *a/b/c*. (b) Búsqueda automática.

Para que funcione el ocultamiento de los nombres, es esencial que cuando se utilice de manera inadvertida un nombre binario obsoleto, se le informe de esto al cliente de alguna manera, para que pueda recurrir a la búsqueda directorio por directorio para encontrar el archivo y poder actualizar el caché. Además, para que tenga algún beneficio el ocultamiento de los indicadores, éstos deben ser correctos la mayor parte del tiempo. Si se cumplen estas condiciones, el ocultamiento de indicadores puede ser una poderosa técnica aplicable a muchas áreas de los sistemas operativos distribuidos.

El aspecto estructural final a considerar es si los servidores de archivos, directorios o de otro tipo deben contener la información de estado de los clientes. Este aspecto tiene una controversia moderada, donde existen dos escuelas de pensamiento en competencia.

Una escuela piensa que los servidores no deben contener los estados, es decir, ser **sin estado**. En otras palabras, cuando un cliente envía una solicitud a un servidor, éste la lleva a cabo, envía la respuesta y elimina de sus tablas internas toda la información relativa a dicha solicitud. El servidor no guarda información alguna relativa a los clientes entre las solicitudes. La otra escuela de pensamiento sostiene que es correcto que los servidores

conserven información de estado de los clientes entre las solicitudes. Después de todo, los sistemas operativos centralizados mantienen la información de estado de los procesos activos, así que ¿Por qué se convierte de pronto en inaceptable este comportamiento tradicional?

Para comprender mejor la diferencia, consideremos un servidor de archivos con comandos para abrir, leer, escribir y cerrar archivos. Después de abrir un archivo, el servidor debe mantener la información que relacione los clientes con los archivos abiertos por éstos. Por lo general, al abrir un archivo, el cliente recibe un descriptor de archivo o algún otro número que se utiliza en las llamadas posteriores para identificación del archivo. Al recibir una solicitud, el servidor utiliza el descriptor de archivo para determinar el archivo necesario. La tabla que asocia los descriptores de archivo con los archivos propiamente dichos es información de estado.

En el caso de un servidor sin estado, cada solicitud debe estar autocontenida. Debe contener todo el nombre del archivo y el ajuste dentro de éste, para que el servidor pueda realizar el trabajo. Esta información aumenta la longitud del mensaje.

Otra forma de ver la información de estado es considerar lo que ocurre si un servidor falla y todas sus tablas se pierden de manera irremediable. Al volver a arrancar el servidor, éste ya no tiene idea de la relación entre los clientes y los archivos abiertos por éstos. Fracasarán entonces los intentos posteriores por leer y escribir en archivos abiertos y la recuperación, de ser posible, quedará por completo a cargo de los clientes. En consecuencia, los servidores sin estado tienden a ser más tolerantes de las fallas que los que mantienen los estados, lo cual es un argumento a favor de los primeros.

Ventajas de los servidores sin estado	Ventajas de los servidores con estado
Tolerancia de fallas	Mensajes de solicitud más cortos
No necesita llamadas OPEN/CLOSE	Mejor desempeño
No se desperdicia el espacio del servidor en tablas	Es posible la lectura adelantada
No existe límite para el número de archivos abiertos	Es más fácil la idempotencia
No hay problemas si un cliente falla	Es posible la cerradura de archivos

Figura 5-8. Una comparación entre los servidores con estado y sin estado.

Los argumentos de ambos casos se resumen en la figura 5-8. Los servidores sin estado son tolerantes de fallas de manera inherente, como hemos mencionado. Las llamadas OPEN y CLOSE no son necesarias, lo cual reduce el número de mensajes, en particular en el frecuente caso de que todo el archivo se lea en una acción. No se desperdicia el espacio del servidor en tablas. Si se utilizan tablas y demasiados clientes tienen muchos archivos abiertos al mismo tiempo, las tablas se pueden colmar y no se podrían abrir más archivos. Por último, con un servidor sin estado, si un cliente falla después de abrir un archivo, el servidor está frente a un dilema. Si no realiza acción alguna, sus tablas se llenarán con basura en

cierto momento. Si elimina los archivos abiertos inactivos, entonces un cliente que espere mucho tiempo entre sus solicitudes no obtendrá servicio alguno y algunos programas correctos no podrán funcionar de forma correcta. Los servidores sin estado eliminan estos problemas.

Los servidores con estado también tienen puntos a su favor. Puesto que los mensajes READ y WRITE no tienen que contener los nombres de archivos, pueden ser más cortos, con lo cual se utiliza menos ancho de banda en la red. Con frecuencia, se puede lograr mejor desempeño, puesto que la información relativa a los archivos abiertos (en términos de UNIX, los nodos-i) pueden permanecer dentro de la memoria principal hasta que los archivos se cierren. Los bloques se pueden leer de antemano para eliminar los retrasos, puesto que la mayoría de los archivos se leen en forma secuencial. Si el tiempo de espera de un cliente se termina y envía la misma solicitud dos veces (por ejemplo, APPEND), es más fácil detectar esto con los estados (mediante un número secuencial en cada mensaje). El objetivo de idempotencia es más difícil de lograr en presencia de una comunicación no confiable y una operación sin estados. Por último, la cerradura de archivos es imposible en un verdadero sistema sin estados, puesto que el único efecto que tienen las cerraduras es introducir el estado dentro del sistema. En los sistemas sin estados, un servidor especial de cerraduras realiza la cerradura de archivos.

5.2.3. Ocultamiento

En un sistema cliente-servidor, cada uno con su memoria principal y un disco, existen cuatro lugares donde se pueden almacenar los archivos o partes de archivos: el disco del servidor, la memoria principal del servidor, el disco del cliente (si éste existe) o la memoria principal del cliente, como se muestra en la figura 5-9. Estos lugares de almacenamiento tienen distintas propiedades, como veremos en breve.

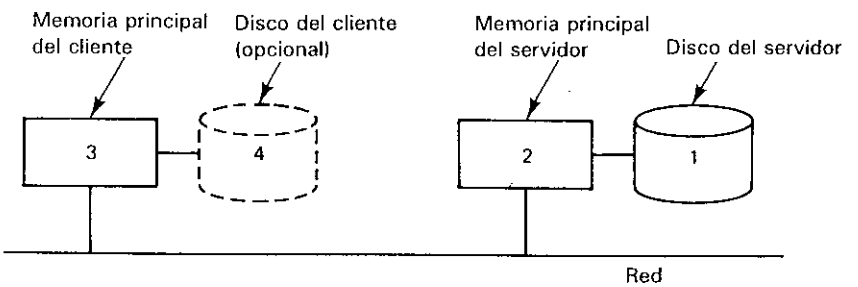


Figura 5-9. Cuatro lugares para guardar archivos o partes de ellos.

El lugar más directo para almacenar todos los archivos es el disco del servidor. Ahí existe mucho espacio y los archivos serían accesibles a todos los clientes. Además, con sólo una copia de cada archivo, no surgen problemas de consistencia.

El problema con el uso del disco del servidor es el desempeño. Antes de que un cliente pueda leer un archivo, éste debe ser transferido primero del disco del servidor a la memoria

principal del servidor y luego, a través de la red, a la memoria principal del cliente. Ambas transferencias tardan cierto tiempo.

Se puede lograr un desempeño mucho mejor si se **ocultan** (es decir, se conservan) los archivos de más reciente uso en la memoria principal del servidor. Un cliente que lea un archivo ya presente en el caché del servidor elimina la transferencia del disco, aunque se deba realizar la transferencia a la red. Puesto que la memoria principal siempre es menor que el disco, se necesita un algoritmo para determinar los archivos o partes de archivos que deben permanecer en el caché.

Este algoritmo debe resolver dos problemas. El primero es el tamaño de la unidad que administra el caché. Puede administrar archivos completos o bloques del disco. Si se ocultan los archivos completos, éstos se pueden almacenar en forma adyacente en el disco (o al menos en pedazos muy grandes), lo cual permite transferencias a alta velocidad entre la memoria y el disco, así como un buen desempeño en general. Sin embargo, el ocultamiento de bloques de disco utiliza el caché y el espacio en disco en forma más eficiente.

El segundo es que el algoritmo debe decidir qué hacer si se utiliza toda la capacidad del caché y hay que eliminar a alguien. Aquí se puede utilizar cualquiera de los algoritmos comunes de ocultamiento, pero como las referencias al caché son poco frecuentes comparadas con las referencias a memoria, por lo general es factible una implantación exacta de LRU mediante listas ligadas. Cuando hay que eliminar a alguien de la memoria, se elige al más antiguo. Si existe una copia actualizada en el disco, sólo se descarta la copia del caché. En caso contrario, primero se actualiza el disco.

El mantenimiento de un caché en la memoria principal del servidor es fácil de lograr y es por completo transparente para los clientes. Puesto que el servidor puede mantener sincronizadas sus copias en memoria y en disco, desde el punto de vista de los clientes sólo existe una copia de cada archivo, por lo que no hay problemas de consistencia.

Aunque el uso del caché en el servidor elimina una transferencia de disco en cada acceso, tiene aún un acceso a la red. La única forma de deshacerse del acceso a la red es hacer el ocultamiento en el lado del cliente, que es donde aparecen todos los problemas. El uso de la memoria principal del cliente o su disco es un problema de espacio *vs.* desempeño. El disco puede contener más información, pero es más lento. Al considerar las opciones de tener un caché en la memoria principal del servidor o en el disco del cliente, la primera es en general más rápida y siempre más sencilla. Por supuesto, si se utilizan grandes cantidades de datos, podría preferirse un caché en el disco del cliente. En todo caso, la mayoría de los sistemas que realizan un ocultamiento en el lado del cliente lo llevan a cabo en la memoria principal, por lo que nos concentraremos en este caso.

Si los diseñadores deciden colocar el caché en la memoria principal del cliente, existen tres opciones para su posición precisa. La más sencilla consiste en ocultar los archivos en forma directa, dentro del propio espacio de direcciones de un proceso usuario, como se muestra en la figura 5-10(b). Lo usual es que el caché sea administrado por la biblioteca de llamadas al sistema. Cuando los archivos se abren, cierran, leen o escriben, la biblioteca mantiene los archivos de más uso en algún sitio, de modo que estén disponibles cuando se vuelvan a utilizar. Cuando el proceso hace su salida, todos los archivos modificados se escriben de

nuevo en el servidor. Aunque este esquema tiene costo mínimo, sólo es eficaz si los procesos individuales abren y cierran los archivos varias veces. Un proceso administrador de una base de datos se podría ajustar a esta descripción, pero en el ambiente usual para el desarrollo de programas, la mayoría de los procesos sólo leen una vez cada archivo, por lo que no se gana nada con el ocultamiento dentro de la biblioteca.

El segundo sitio donde se puede colocar el caché es el núcleo, como se muestra en la figura 5-10(c). La desventaja en este caso es que siempre hay que llamar al núcleo, incluso en aquellos casos en que los archivos estén dentro del caché, pero el que el caché sobreviva al proceso compensa en mucho este hecho. Por ejemplo, supongamos que un compilador de dos etapas se ejecuta como dos procesos. La primera etapa escribe un archivo intermedio, el cual es leído por la segunda etapa. En la figura 5-10(c), después de terminar el proceso de la etapa 1, es probable que el archivo intermedio esté dentro del caché, por lo que no hay que llamar al servidor cuando el proceso de la segunda etapa lea dicho archivo.

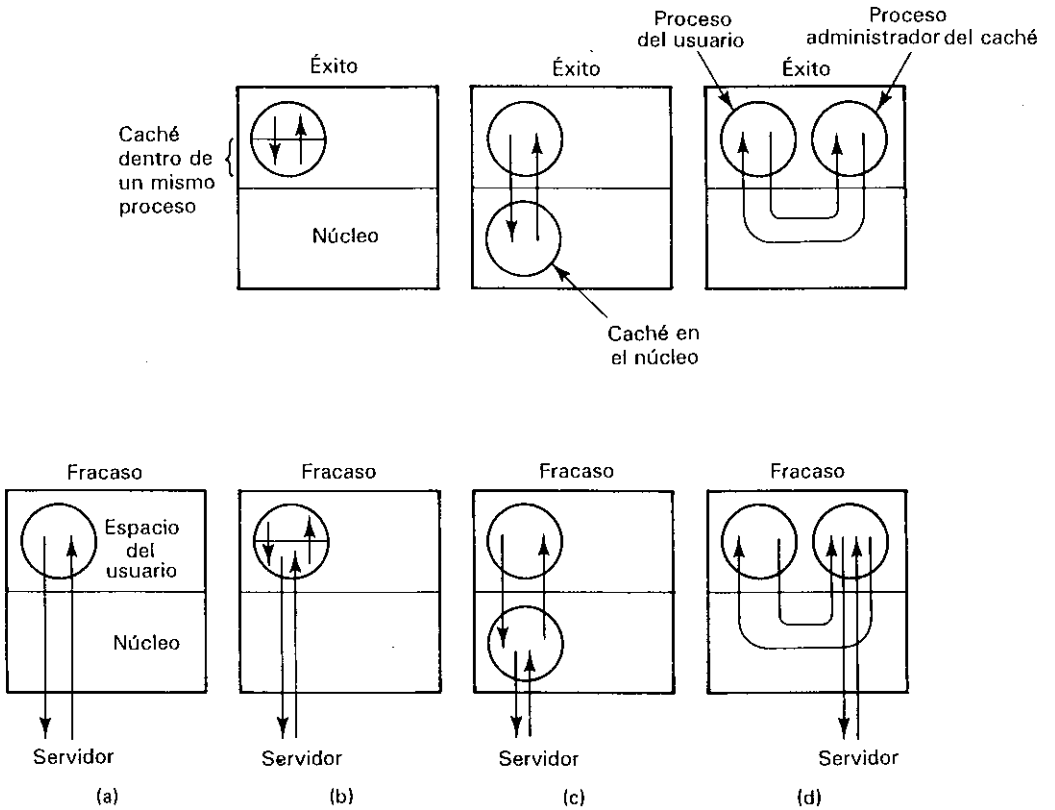


Figura 5-10. Varias maneras de realizar un ocultamiento en la memoria del cliente. (a) Sin ocultamiento. (b) Ocultamiento dentro de cada proceso. (c) Ocultamiento en el núcleo. (d) El administrador del caché como un proceso del usuario.

El tercer lugar donde se puede colocar el caché es en un proceso administrador del caché, independiente y a nivel usuario, como se muestra en la figura 5-10(d). La ventaja de un admi-

nistrador del caché a nivel usuario es que libera al (micro)núcleo del código del sistema de archivos, es más fácil de programar (puesto que está por completo aislado) y es más flexible.

Por otro lado, si el núcleo administra el caché, puede decidir en forma dinámica la cantidad de memoria que debe reservar para los programas y la cantidad para el caché. Si un administrador del caché a nivel usuario se ejecuta en una máquina con memoria virtual, es concebible que el núcleo podría decidir si transfiere parte o todo el caché al disco, de modo que un caso donde el archivo esté dentro del caché ("éxito") requeriría la recuperación de una o más páginas. No es necesario decir que esto va por completo en contra de la idea de ocultamiento del cliente. Sin embargo, si el administrador del caché puede asignar y bloquear en memoria cierto número de páginas, se puede evitar esta situación tan irónica.

Al evaluar si el ocultamiento vale la pena, es importante observar en la figura 5-10(a) que sólo se utiliza una RPC para hacer una solicitud de archivo, sin importar lo demás. Tanto en la figura 5-10(c) como en la figura 5-10(d) se utilizan uno o dos, según si la solicitud se debe satisfacer dentro del caché o no. Así, el promedio de RPC siempre es menor si se utiliza el ocultamiento. En caso de que las RPC sean rápidas y las transferencias en la red sean lentas (CPU rápidos, redes lentas), el ocultamiento puede ser un factor importante en el desempeño. Sin embargo, si las transferencias en la red son muy rápidas (por ejemplo, redes con fibras ópticas de alta velocidad), el tiempo de transferencia a través de la red no sería tan importante, por lo que la RPC adicional podría consumir una fracción sustancial de la ganancia. Así, el mejoramiento en el desempeño mediante ocultamiento depende en cierta medida de la tecnología disponible para los CPU y las redes, así como de las aplicaciones, por supuesto.

Consistencia del caché

Como es usual en la ciencia de la computación, no se obtiene algo a cambio de nada. El ocultamiento por parte del cliente introduce inconsistencia en el sistema. Si dos clientes leen un mismo archivo en forma simultánea y después lo modifican, aparecen varios problemas. Uno de ellos es que, cuando un tercer proceso lee el archivo del servidor, obtendrá la versión original y no alguna de las dos nuevas. Este problema se puede evitar mediante la semántica de sesión (donde se establezca de manera oficial que los efectos de modificación de un archivo no deben ser visibles en forma global hasta cerrar el archivo). En otras palabras, este comportamiento "incorrecto" se declara sólo como comportamiento "correcto". Por supuesto, este truco no funciona si el usuario espera la semántica de UNIX.

Por desgracia, existe otro problema que no se puede eliminar mediante definición alguna: cuando dos archivos se escriben de nuevo al servidor, el último de ellos se escribirá sobre el otro. La moraleja de esta historia es que el ocultamiento por parte del cliente se debe pensar con mucho cuidado. Más adelante analizaremos algunos de los problemas y las soluciones propuestas.

Una forma de resolver el problema de inconsistencia es utilizar el algoritmo de **escritura a través del caché**. Cuando se modifica una entrada del caché (archivo o bloque), el nuevo valor se mantiene dentro de él, pero también se envía de inmediato al servidor. Como consecuencia, cuando otro proceso lee el archivo, obtiene el valor más reciente.

Sin embargo, surge el siguiente problema. Supongamos que un proceso cliente en la máquina *A* lee un archivo *f*, pero que la máquina mantiene *f* en su caché. Más tarde, un cliente en la máquina *B* lee el mismo archivo, lo modifica y lo escribe en el servidor. Por último, un nuevo proceso cliente inicia en la máquina *A*. Lo primero que hace es abrir y leer *f*, el cual se toma del caché. Por desgracia, su valor es obsoleto.

Una posible salida es exigir al administrador del caché que verifique al servidor antes de proporcionar al cliente un archivo del caché. Esta verificación se puede hacer mediante una comparación de la hora de la última modificación de la versión en el caché con la última versión del servidor. Si son iguales, el caché está actualizado. Si no, la versión actual se debe buscar en el servidor. En lugar de utilizar fechas, se pueden usar también números de versión o sumas de verificación. Aunque esta verificación de fechas, números de versión o sumas de verificación con el servidor ocupa una RPC, la cantidad de información intercambiada es muy pequeña. Aún así, tarda cierto tiempo.

Otro problema con este algoritmo de escritura a través del caché es que, aunque ayuda en las lecturas, el tráfico en la red en el caso de las escrituras es igual que en el caso de no ocultamiento. Muchos diseñadores de sistemas no aceptan esto y hacen el siguiente truco: en vez de ir hacia el servidor en el instante en que se realiza la escritura, el cliente hace una nota donde indica que ha actualizado un archivo. Una vez cada 30 segundos, o un intervalo similar, todas las actualizaciones se recolectan y envían al servidor al mismo tiempo. Un bloque es más eficiente que muchos pequeños.

Además, muchos programas crean archivos de tipo borrador, los escriben, los leen de nuevo y después los eliminan, todo lo cual se realiza con rapidez. En caso de que todo esto ocurra antes de que sea hora de enviar todos los archivos modificados al servidor, los archivos ya eliminados no tienen que escribirse de nuevo. El hecho de no tener que utilizar el servidor de archivos para los archivos temporales puede permitir un mejor desempeño.

Por supuesto, el retraso de la escritura oscurece la semántica, puesto que si otro proceso lee el archivo, lo que obtenga dependerá de la sincronización de los eventos. Así, la decisión de posponer la escritura depende de si se desea un mejor desempeño o una semántica más limpia (la cual se traduce en una programación más sencilla).

El siguiente paso en esta dirección es adoptar la semántica de sesión y sólo escribir un archivo de nuevo en el servidor hasta que éste se cierre. Este algoritmo se llama **de escritura al cierre**. Es mejor aún esperar 30 segundos después del cierre para ver si el archivo es eliminado en ese lapso. Como hemos visto, el uso de este método implica que si dos archivos en el caché se escriben al servidor, el segundo se escribe sobre el primero. La única solución a este problema consiste en observar que esto no es tan malo como parece. En un sistema con un CPU, es posible què dos procesos abran y lean un archivo, lo modifiquen dentro de sus respectivos espacios de direcciones y que lo escriban de nuevo al servidor. En consecuencia, el algoritmo de escritura al cierre con semántica de sesión no es peor de lo que puede ocurrir en un sistema con un CPU.

Un método por completo distinto a la consistencia es utilizar un algoritmo de control centralizado. Al abrir un archivo, la máquina que lo abre envía un mensaje al servidor para anunciar este hecho. El servidor de archivos tiene un registro de los archivos abiertos, sus

poseedores y si están abiertos para la lectura, escritura o para ambos procesos. Si se abre un archivo para la lectura, no hay problema en dejar que otros procesos lo abran, pero hay que evitar que lo abran para una escritura. En forma análoga, si algún proceso tiene abierto un archivo para escritura, hay que evitar que otros procesos lo abran. Al cerrar un archivo, hay que informar de ello, de modo que el servidor pueda actualizar sus tablas para indicar los archivos abiertos y sus poseedores. El archivo modificado también se puede enviar al servidor en ese momento.

Cuando un cliente intenta abrir un archivo y éste ya está abierto en otra parte del sistema, la nueva solicitud se puede otorgar o negar. Otra alternativa es que el servidor envíe un **mensaje no solicitado** a todos los clientes que tengan abierto el archivo para indicarles que eliminen ese archivo de sus cachés y desactiven el ocultamiento de ese archivo en particular. De este modo, se pueden ejecutar en forma simultánea varios lectores y escritores y los resultados no son ni mejores ni peores que los que se pueden obtener en un sistema con un CPU.

Aunque es claro que se pueden enviar mensajes no solicitados, esto no es elegante, puesto que invierte el papel del cliente y el servidor. Por lo general, los servidores no envían mensajes en forma espontánea a los clientes o inician RPC con ellos. Si los clientes tienen varios hilos, uno de éstos se puede asignar de manera permanente para esperar las solicitudes del servidor; sin embargo, si ése no es el caso, el mensaje no solicitado puede provocar una interrupción.

Aun con estas precauciones, hay que ser cuidadosos. En particular, si una máquina abre un archivo, lo guarda en un caché y después lo cierra, el administrador del caché debe verificar si el archivo que contiene es válido cuando éste se abra de nuevo. Después de todo, otros procesos pueden abrirlo, modificarlo y cerrarlo. Existen muchas variantes de este algoritmo de control centralizado, con distintas semánticas. Por ejemplo, los servidores pueden tener un registro de los archivos en los cachés, en vez de los archivos abiertos. Todos estos métodos tienen un punto de falla y ninguno se escala bien a los sistemas de gran tamaño.

Método	Comentarios
Escritura a través del caché	Funciona, pero no afecta el tráfico de escritura
Escritura retrasada	Mejor desempeño pero es posible que la semántica sea ambigua
Escritura al cierre	Concuerda con la semántica de la sesión
Control centralizado	Semántica de UNIX, pero no es robusto y es poco escalable

Figura 5-11. Cuatro algoritmos para administrar el caché de archivos del cliente.

Los cuatro algoritmos para el control del caché analizados arriba se resumen en la figura 5-11. Para resumir el tema del ocultamiento, diremos que el ocultamiento por parte del servidor es fácil y casi siempre vale la pena, sin importar si está presente o no el ocultamiento

por parte del cliente. El ocultamiento en el servidor no tiene efectos en la semántica del sistema de archivos, desde el punto de vista de los clientes. Por el contrario, el ocultamiento en el cliente ofrece mejor desempeño a costa de mayor complejidad y es posible una semántica más difusa. Para hacer este ocultamiento, los diseñadores deben sopesar los factores de desempeño, complejidad y facilidad de programación.

En una sección anterior de este capítulo, cuando analizamos la semántica de los sistemas distribuidos de archivos, señalamos que una de las opciones de diseño son los archivos inmutables. Uno de las grandes atractivos de un archivo inmutable es la posibilidad de tenerlo en un caché dentro de una máquina *A* sin tener que preocuparse por el hecho de que una máquina *B* lo pueda modificar. No se permiten las modificaciones. Por supuesto, se puede crear un nuevo archivo y darle el mismo nombre simbólico del archivo en el caché, pero hay que verificar esto cada vez que se vuelva a abrir un archivo en el caché. Este modelo tiene el mismo costo en RPC ya analizado, pero la semántica es menos difusa.

5.2.4. Réplica

Con frecuencia, los sistemas distribuidos de archivos proporcionan la réplica de archivos como servicio a sus clientes. En otras palabras, se dispone de varias copias de algunos archivos, donde cada copia está en un servidor de archivos independiente. Las razones para la existencia de tal servicio varían, pero algunas de las razones principales son:

1. Aumentar la confiabilidad al disponer de respaldos independientes de cada archivo. Si un servidor falla o se pierde en forma permanente, no se pierden los datos. Esta propiedad es deseable en extremo para gran número de aplicaciones.

2. Permitir el acceso al archivo aunque falle un servidor de archivos. El lema aquí es: El espectáculo debe continuar. La falla de un servidor no debe hacer que todo el sistema se detenga hasta que se pueda volver a arrancar.

3. Repartir la carga de trabajo entre varios servidores. Al crecer el tamaño del sistema, el hecho de tener todos los archivos en un procesador se puede convertir en un cuello de botella. Con varios archivos duplicados en dos o más servidores, se puede utilizar el que tenga menor carga.

Las dos primeras se relacionan con el mejoramiento de la confiabilidad y la disponibilidad; la tercera se refiere al desempeño. Todas son importantes.

Un aspecto clave en la réplica es la transparencia (como de costumbre). ¿En qué medida tienen conciencia los clientes de que los archivos se duplican? ¿Juegan un papel en el proceso de réplica, o éste se administra en forma automática? En un caso extremo, los usuarios pueden tener total conciencia del proceso de réplica e incluso lo pueden controlar. En el otro extremo, el sistema hace todo a espaldas de los usuarios. En el segundo caso, decimos que el sistema es **transparente con respecto a la réplica**.

La figura 5-12 muestra tres formas de llevar a cabo la réplica. La primera forma, que se muestra en la figura 5-12(a), consiste en que el programador controle todo el proceso. Cuando un proceso crea un archivo, lo hace en un servidor específico. Entonces puede crear

copias adicionales en otros servidores, si así lo desea. Si el servidor de directorios permite varias copias de un archivo, las direcciones en la red de todas las copias se pueden asociar con el nombre del archivo, como se muestra en la parte inferior de la figura 5-12(a), de modo que cuando se busque el nombre, se encuentren todas las copias. Cuando el archivo se abre de nuevo, se pueden buscar las copias de manera secuencial en cierto orden, hasta encontrar una disponible.

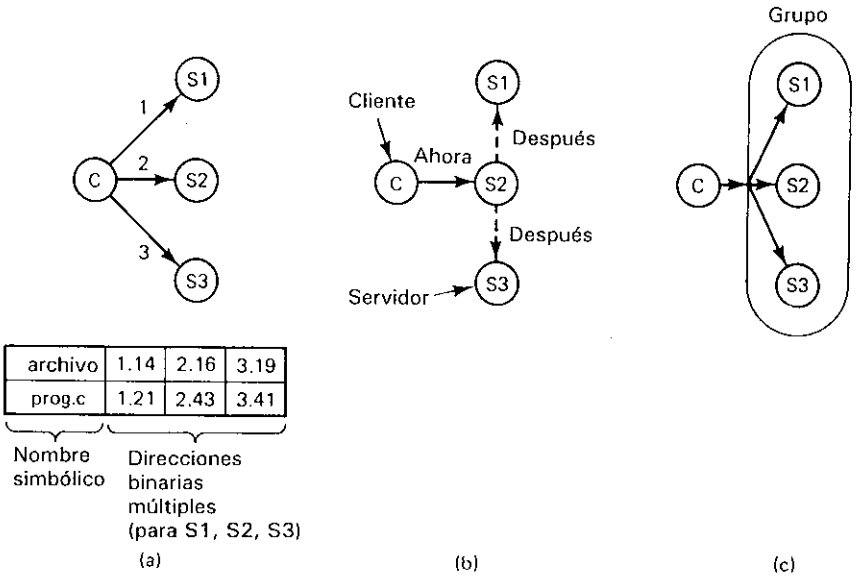


Figura 5-12. (a) Réplica explícita de archivos. (b) Réplica retrasada de archivos. (c) Réplica de archivos mediante un grupo.

Para familiarizarnos con el concepto de réplica explícita, consideremos la forma de llevarla a cabo en un sistema basado en el montaje remoto en UNIX. Supongamos que el directorio de origen de un programador es */máquina1/usr/ast*. Después de crear un archivo, digamos, el archivo */máquina1/usr/ast/xyz*, el programador, proceso o biblioteca puede utilizar el comando *cp* (o su equivalente) para hacer copias de él en */máquina2/usr/ast/xyz* y */máquina3/usr/ast/xyz*. Se pueden escribir programas que acepten cadenas del tipo */usr/ast/xyz* como argumentos y que intenten abrir las copias hasta lograr abrir una. Aunque se puede lograr que este esquema funcione, tiene muchos problemas. Por esta razón es mejor un sistema distribuido.

En la figura 5-12(b) vemos un método alternativo, la **réplica retrasada**. En este caso, sólo se crea una copia de cada archivo en un servidor. Más tarde, el propio servidor crea réplicas en otros servidores en forma automática, sin el conocimiento del programador. El sistema debe ser lo bastante hábil como para recuperar alguna de estas copias en caso necesario. Al crear copias secundarias de esta manera, es importante poner atención en el hecho de que el archivo podría cambiar antes de que se hagan las copias.

Nuestro último método es el uso de la comunicación en grupo, como se muestra en la figura 5-13(c). En este esquema, todas las llamadas WRITE al sistema se transmiten en forma simultánea a todos los servidores a la vez, por lo que las copias adicionales se hacen al mismo tiempo que el original. Existen dos diferencias fundamentales entre la réplica retrasada y el uso de un grupo. En primer lugar, con la réplica retrasada, se direcciona un servidor y no un grupo. En segundo lugar, la réplica retrasada ocurre en un plano secundario, cuando el servidor tiene cierto tiempo libre, mientras que con el uso de la comunicación en grupo, todas las copias se crean al mismo tiempo.

Protocolos de actualización

Anteriormente analizamos el problema de la creación de réplicas de archivos. Ahora veremos cómo se pueden modificar los ya existentes. No es una buena idea enviar un simple mensaje de actualización a cada copia en serie, puesto que si el proceso que realiza la actualización se detiene a la mitad del camino, algunas copias estarán modificadas y otras no. Como resultado de esto, algunas de las próximas lecturas obtendrán el valor anterior y otras el nuevo, lo cual es poco deseable. Analizaremos dos algoritmos bien conocidos para la solución de este problema.

El primero es el de **réplica de la copia primaria**. En este caso, uno de los servidores se denomina como primario. Todos los demás son secundarios. Si hay que actualizar un archivo duplicado, el cambio se envía al servidor primario, que realiza los cambios en forma local y después envía comandos a los secundarios para ordenarles la misma modificación. Las lecturas se pueden hacer de cualquier copia, primaria o secundaria.

Para protegerse de la situación en que falle el primario antes de que pueda dar instrucciones a todos los secundarios, la actualización debe escribirse en un espacio estable de almacenamiento antes de modificar la copia primaria. De este modo, cuando un servidor vuelve a arrancar después de una falla, se debe verificar si existían actualizaciones al momento de la falla. En caso afirmativo, se pueden llevar a cabo. En algún momento, todos los secundarios se actualizarán.

Aunque el método es directo, tiene la desventaja de que si falla el primario, no se pueden llevar a cabo las actualizaciones. Para deshacerse de esta asimetría, Gifford (1979) propuso un método más robusto, el del **voto**. La idea fundamental es exigir a los clientes que soliciten y adquieran el permiso de varios servidores antes de leer o escribir un archivo replicado.

Como ejemplo sencillo del funcionamiento de este algoritmo, supongamos que un archivo se replica en N servidores. Podemos establecer la regla de que para actualizar un archivo, un cliente debe establecer contacto con al menos la mitad de los servidores, más 1 (una mayoría) y ponerlos de acuerdo para llevar a cabo la actualización. Una vez de acuerdo, el archivo se modifica y se asocia un nuevo número de versión al nuevo archivo. El número de versión se utiliza para identificar la versión del archivo y es la misma para todos los archivos recién actualizados.

Para leer un archivo replicado, un cliente debe establecer también contacto con la mitad de los servidores, más 1, y pedirles que envíen el número de versión asociado a cada archivo.

Si coinciden todos los números de versión, ésta debe ser la más reciente, puesto que un intento por actualizar sólo a los servidores restantes fracasará, porque no son un número suficiente.

Por ejemplo, si existen cinco servidores y un cliente determina que tres de ellos tienen la versión 8, es imposible que los otros dos tengan la versión 9. Después de todo, cualquier actualización exitosa de la versión 8 a la versión 9 necesita obtener tres servidores que estén de acuerdo y no sólo dos.

El esquema de Gifford es en realidad un poco más general que esto. En el esquema, para leer un archivo del que existen N réplicas, un cliente necesita conformar un **quórum de lectura**, una colección arbitraria de N_r servidores o más. En forma análoga, para modificar un archivo, se necesita un **quórum de escritura** de al menos N_w servidores. Los valores de N_r y N_w están sujetos a la restricción $N_r + N_w > N$. Sólo después de que el número adecuado de servidores ha acordado participar, se puede leer o escribir en un archivo.

Para ver el funcionamiento de este algoritmo, consideremos la figura 5-13(a), la cual tiene $N_r = 3$ y $N_w = 10$. Imaginemos que el más reciente quórum de escritura constaba de 10 servidores, C hasta L. Todos ellos tienen la nueva versión y el nuevo número de versión. Cualquier quórum de lectura posterior con tres servidores debe contener al menos un miembro de este conjunto. Cuando el cliente analiza los números de versión, sabrá cuál es el más reciente de ellos y lo tomará.

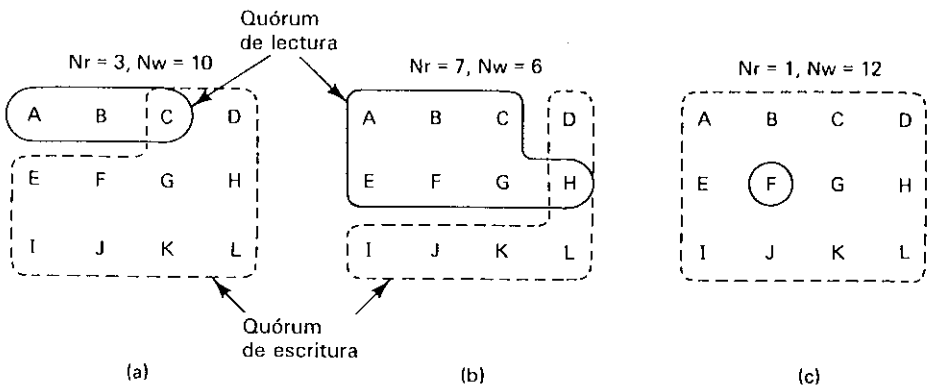


Figura 5-13. Tres ejemplos del algoritmo del voto.

En las figuras 5-13(b) y (c) vemos dos ejemplos más. El último es en particular interesante, puesto que hace $N_r = 1$, lo que permite leer un archivo replicado al encontrar cualquiera de sus copias y utilizar ésta. Sin embargo, el precio que se paga es que las actualizaciones de escritura necesitan adquirir todas las copias.

Una variante interesante del voto es el **voto con fantasmas** (Van Renesse y Tanenbaum, 1988). En la mayoría de las aplicaciones, las lecturas son más frecuentes que las escrituras, por lo que N_r es por lo general pequeño y N_w es muy cercano a N. Esta opción quiere decir que si fallan unos cuantos servidores, podría ser imposible obtener un quórum de escritura.

El voto con fantasmas elimina este problema al crear un servidor fantasma, sin espacio de almacenamiento, para cada servidor real que haya fallado. No se permite un fantasma en un quórum de lectura (de hecho, no tiene archivo alguno) pero se puede unir a un quórum de escritura, en cuyo caso sólo desecha el archivo escrito en él. La escritura sólo tiene éxito si al menos uno de los servidores es real.

Al arrancar de nuevo un servidor fallido, debe obtener un quórum de lectura para localizar la versión más reciente, la cual copia en su espacio antes de iniciar su operación normal. El algoritmo funciona debido a que tiene la misma propiedad del esquema básico de votación; es decir, N_r y N_w se eligen de modo que sea imposible obtener un quórum de lectura y otro de escritura al mismo tiempo. La única diferencia aquí es que se permiten las máquinas muertas en el quórum de escritura, con la condición de que cuando regresen obtengan de manera inmediata la versión actual antes de dar un servicio.

En (Bernstein y Goodman, 1984; Brereton, 1986; Pu *et al.*, 1986; Purdin *et al.*, 1987) se describen otros algoritmos de réplica.

5.2.5. Un ejemplo: el sistema de archivos de red (NFS) de Sun

En esta sección analizaremos un ejemplo de sistema de archivo de red, el **sistema de archivo de red** de Sun Microsystems, conocido universalmente como **NFS** que fue diseñado e implantado por Sun en un principio para su uso en las estaciones de trabajo con base en UNIX. Ahora, otros fabricantes lo soportan, tanto para UNIX como para otros sistemas operativos (incluido MS-DOS). NFS soporta sistemas heterogéneos; por ejemplo, clientes de MS-DOS que utilizan servidores UNIX. Ni siquiera se pide que todas las máquinas utilicen el mismo hardware. Es común encontrar clientes de MS-DOS que utilizan CPU con Intel 386 y obtienen algún servicio de los servidores de archivos de UNIX que se ejecutan en Motorola 68 030 o CPU SPARC de Sun.

Tres aspectos de NFS son de interés: la arquitectura, el protocolo y la implantación. Analizaremos cada una en su turno.

Arquitectura de NFS

La idea básica detrás de NFS es permitir que una colección arbitraria de clientes y servidores compartan un sistema de archivos común. En la mayoría de los casos, todos los clientes y servidores están en la misma LAN, pero esto no es necesario. Es posible ejecutar NFS en una red de área amplia. Para simplificar nuestra exposición hablaremos de los clientes y servidores como si estuvieran en máquinas diferentes, pero de hecho, NFS permite que cada máquina sea cliente y servidor al mismo tiempo.

Cada servidor NFS exporta uno o más de sus directorios para su acceso por parte de los clientes remotos. Cuando se dispone de un directorio, también de todos sus subdirectorios, así que, de hecho, los árboles de directorios se exportan como una unidad. La lista de directorios que puede exportar un servidor se conserva en el archivo */etc/exports*, de modo que estos directorios se puedan exportar de manera automática al arrancar el servidor.

Los clientes tienen acceso a los directorios exportados al montarlos. Cuando un cliente monta un directorio (remoto), éste se vuelve parte de su jerarquía de directorios, como se muestra en la figura 5-13. Muchas estaciones de trabajo Sun no tienen disco. Si así lo desea, un cliente sin disco puede montar un sistema de archivos remoto en su directorio raíz, lo que produce un sistema de archivos por completo soportado en un servidor remoto. Las estaciones de trabajo que sí tienen discos locales pueden montar directorios remotos en el sitio que deseen de su jerarquía local de directorios, lo que produce un sistema de archivos parcialmente local y parcialmente remoto. Para que los programas se ejecuten en la máquina cliente (casi) no existe diferencia entre un archivo localizado en un servidor de archivos remoto y un archivo localizado en el disco local.

Así, la característica básica de la arquitectura de NFS es que los servidores exportan directorios y los clientes los montan de manera remota. Si dos o más clientes montan el mismo directorio al mismo tiempo, se pueden comunicar compartiendo archivos en sus directorios comunes. Un programa en un cliente puede crear un archivo, y un programa en otro cliente puede leer el archivo. Una vez realizados los montajes, no hay que hacer nada especial para lograr compartir los archivos. Los archivos compartidos sólo están ahí, en la jerarquía de directorios de varias máquinas y pueden leerse y escribir en ellos de la manera usual. Esta sencillez es una de las grandes atracciones de NFS.

Protocolos de NFS

Puesto que uno de los objetivos de NFS es soportar un sistema heterogéneo, con clientes y servidores que tal vez ejecuten diferentes sistemas operativos con un hardware distinto, es esencial que la interfaz entre los clientes y los servidores esté bien definida. Sólo entonces es posible que cualquiera pueda escribir una nueva implantación cliente y espere que funcione de manera correcta con los servidores existentes y viceversa.

NFS logra este objetivo al definir dos protocolos cliente-servidor. Un **protocolo** es un conjunto de solicitudes enviadas por los clientes a los servidores, junto con las respuestas enviadas de regreso de los servidores a los clientes. (Los protocolos son un tema importante en los sistemas distribuidos; regresaremos a ellos posteriormente con más detalle.) Mientras un servidor reconozca y pueda controlar todas las solicitudes en los protocolos, no necesita saber nada de sus clientes. De manera análoga, los clientes pueden tratar a los servidores como "cajas negras" que aceptan y procesan un conjunto específico de solicitudes. La forma en que lo hacen es asunto de ellos.

El primer protocolo NFS controla el montaje. Un cliente puede enviar un nombre de ruta de acceso a un servidor y solicitar que monte ese directorio en alguna parte de su jerarquía de directorios. El lugar donde se montará no está contenido en el mensaje, ya que el servidor no se preocupa por dicho lugar. Si la ruta es válida y el directorio especificado ha sido exportado, el servidor regresa un **asa de archivo** al cliente. El asa de archivo contiene campos que identifican de manera única al tipo de sistema de archivo, el disco, el número de nodo-i del directorio, e información de seguridad. Las llamadas posteriores para la lectura y escritura de archivos en el directorio montado utilizan el asa del archivo.

Muchos clientes están configurados de modo que monten ciertos directorios remotos sin intervención manual. Por lo general, estos clientes contienen un archivo llamado */etc/rc*, que es un guión de shell que contiene las instrucciones para el montaje remoto. Este guión se ejecuta de manera automática cuando el cliente se arranca.

Otra alternativa a la versión de UNIX de Sun soporta también el **automontaje**. Esta característica permite asociar un conjunto de directorios con un directorio local. Ninguno de los directorios remotos se monta (ni se realiza el contacto con sus servidores) cuando arranca el cliente. En vez de esto, la primera vez que se abre un archivo remoto, el sistema operativo envía un mensaje a cada uno de los servidores. El primero en responder gana, y se monta su directorio.

El automontaje tiene dos ventajas principales sobre el montaje estático por medio del archivo */etc/rc*. La primera es que si uno de los servidores NFS llamados en */etc/rc* no sirve, es imposible despertar al cliente, al menos no sin cierta dificultad, retraso y unos cuantos mensajes de error. Si el usuario ni siquiera necesita ese servidor por el momento, todo ese trabajo se desperdicia. En segundo lugar, al permitir que el cliente intente comunicarse con un conjunto de servidores en paralelo, se puede lograr cierto grado de tolerancia de fallas (puesto que sólo se necesita que uno de ellos esté activo) y se puede mejorar el desempeño (al elegir el primero que responda, que supuestamente tiene la menor carga).

Por otro lado, se supone de manera implícita que todos los sistemas de archivos especificados como alternativas para el automontaje son idénticos. Puesto que NFS no da soporte para la réplica de archivos o directorios, el usuario debe lograr que todos los sistemas de archivos sean iguales. En consecuencia, el automontaje se utiliza con más frecuencia para los sistemas de archivos exclusivos para lectura que contienen binarios del sistema y para otros archivos que rara vez cambian.

El segundo protocolo NFS es para el acceso a directorios y archivos. Los clientes pueden enviar mensajes a los servidores para que manejen los directorios y lean o escriban en archivos. Además, también pueden tener acceso a los atributos de un archivo, como el modo, tamaño y tiempo de su última modificación. La mayor parte de las llamadas al sistema UNIX son soportadas por NFS, con la probable sorpresa de OPEN y CLOSE.

La omisión de OPEN y CLOSE no es un accidente. Es por completo intencional. No es necesario abrir un archivo antes de leerlo, no cerrarlo al terminar. En vez de esto, para leer un archivo, un cliente envía al servidor un mensaje con el nombre del archivo, una solicitud para buscarlo y regresar un asa de archivo, que es una estructura de identificación del archivo. A diferencia de una llamada OPEN, esta operación LOOKUP no copia información a las tablas internas del sistema. La llamada READ contiene al asa de archivo que se desea leer, el ajuste para determinar el punto de inicio de la lectura y el número de bytes deseados. Cada uno de estos mensajes está autocontenido. La ventaja de este esquema es que el servidor no tiene que recordar lo relativo a las conexiones abiertas entre las llamadas a él. Así, si un servidor falla y después se recupera, no se pierde información acerca de los archivos abiertos, puesto que no hay ninguno. Un servidor como éste, que no conserva información del estado de los archivos abiertos es **sin estado**.

Por el contrario, en el sistema V de UNIX, el **sistema de archivos remotos (RFS)** requiere abrir un archivo antes de leerlo o escribir en él. El servidor crea entonces una entrada de tabla con un registro del hecho de que el archivo está abierto y la posición actual del lector, de modo que cada solicitud no necesita un ajuste. La desventaja de este esquema es que si un servidor falla y vuelve a arrancar rápidamente, se pierden todas las conexiones abiertas, y fallan los programas cliente. NFS no tiene esta propiedad.

Por desgracia, el método NFS dificulta el hecho de lograr la semántica de archivo propia de UNIX. Por ejemplo, en UNIX un archivo se puede abrir y bloquear para que otros procesos no tengan acceso a él. Al cerrar el archivo, se liberan las cerraduras de bloqueo. En un servidor sin estado como NFS, las cerraduras no se pueden asociar con los archivos abiertos, puesto que el servidor no sabe cuáles archivos están abiertos. Por lo tanto, NFS necesita un mecanismo independiente adicional para controlar la cerradura.

NFS utiliza el mecanismo de protección de UNIX, con los bits *rwx* para el propietario, grupo y demás personas. En un principio, cada mensaje de solicitud sólo contenía los identificadores del usuario y del grupo de quien realizó la llamada, lo que utilizaba el servidor NFS para validar el acceso. De hecho, confiaba en que los clientes no mintieran. Varios años de experiencia han demostrado ampliamente que tal hipótesis era (¿Cómo decirlo?) ingenua. Actualmente, se puede utilizar la criptografía de claves públicas para establecer una clave segura y validar al cliente y al servidor en cada solicitud y respuesta. Cuando esta opción se activa, un cliente malicioso no puede personificar a otro cliente, pues no conoce la clave secreta del mismo. Por cierto, la criptografía sólo se utiliza para autenticar a las partes. Los propios datos nunca se cifran.

Todas las claves utilizadas para la autenticación, así como la demás información, son mantenidas por el **NIS (servicio de información de la red)**. NIS se conocía antes como el **directorio amarillo (yellow pages)**. Su función es la de guardar parejas (clave, valor). Cuando se proporciona una clave, regresa el valor correspondiente. No sólo controla las claves de cifrado, sino también la asociación de los nombres de usuario con las contraseñas (cifradas), así como la asociación de los nombres de las máquinas con las direcciones de la red, y otros elementos.

Los servidores de información de la red se duplican mediante un orden maestro/esclavo. Para leer sus datos, un proceso puede utilizar al maestro o cualquiera de sus copias (esclavos). Sin embargo, todas las modificaciones deben ser realizadas únicamente en el maestro, que entonces las propaga a los esclavos. Existe un breve intervalo después de una actualización en el que la base de datos es inconsistente.

Implantación de NFS

Aunque la implantación del código del cliente y el servidor es independiente de los protocolos NFS, es interesante echar un vistazo a la implantación de NFS. Consta de tres capas, como se muestra en la figura 5-14. La capa superior es la capa de llamadas al sistema, la cual controla las llamadas como OPEN, READ y CLOSE. Después de analizar la llamada y verificar sus parámetros, llama a la segunda capa, la capa del sistema virtual de archivos (VFS).

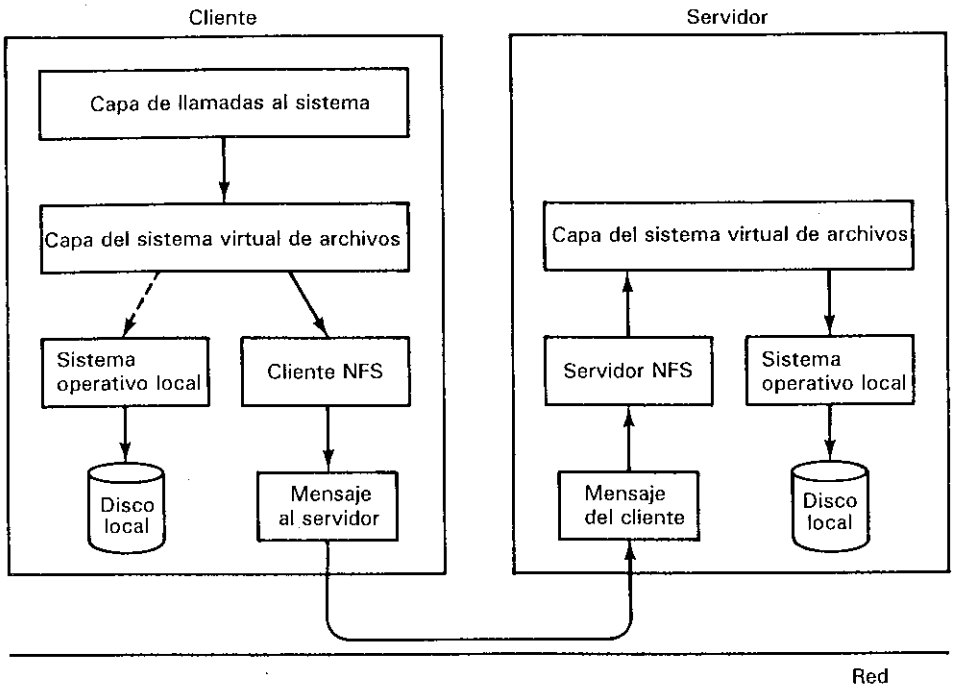


Figura 5.14. Estructura de capas de NFS.

La tarea de la capa VFS es mantener una tabla con una entrada por cada archivo abierto, análoga a la tabla de nodos-i para los archivos abiertos en UNIX. En el UNIX ordinario, un nodo-i queda indicado de manera única mediante una pareja (dispositivo, nodo_i). En vez de esto, la capa VFS tiene una entrada, llamada **nodo-v (nodo-i virtual)**, para cada archivo abierto. Los nodos-V se utilizan para indicar si un archivo es local o remoto. Para los archivos remotos, se dispone de suficiente información para tener acceso a ellos.

Para ver la forma de utilizar los nodos-v, sigamos una secuencia de llamadas al sistema MOUNT, OPEN y READ. Para montar un sistema remoto de archivos, el administrador del sistema llama al programa *mount* con la información del directorio remoto, el directorio local donde será montado y algunos otros datos adicionales. El programa *mount* analiza el nombre del directorio remoto por montar y descubre el nombre de la máquina donde se localiza dicho directorio. Entonces, entra en contacto con la máquina en la que se localiza el directorio remoto. Si el directorio existe y está disponible para su montaje remoto, el servidor regresa entonces un asa de archivo para el directorio. Por último, llama a MOUNT para transferir el asa del archivo al núcleo.

El núcleo construye entonces un nodo-v para el directorio remoto y pide el código del cliente NFS en la figura 5-14 para crear un **nodo-r (nodo-i remoto)** en sus tablas internas, con el fin de mantener el asa del archivo. El nodo-v apunta al nodo-r. Así, cada nodo-v de

la capa VFS contendrá en última instancia un apuntador a un nodo-r en el código del cliente NFS o un apuntador a un nodo-i en el sistema operativo local (véase la figura 5-14). Así, es posible ver desde el nodo-v si un archivo o directorio es local o remoto y, si es remoto, encontrar su asa de archivo.

Al abrir un archivo remoto, en cierto momento durante el análisis del nombre de la ruta de acceso, el núcleo alcanza el directorio donde se desea montar el sistema de archivos remoto. Ve que este directorio es remoto y en el nodo-v del directorio encuentra el apuntador al nodo-r. Le pide entonces al código del cliente NFS que abra el archivo. El código del cliente NFS busca en la parte restante del nombre de la ruta de acceso en el servidor remoto asociado con el directorio montado y regresa un asa de archivo para él. Crea en sus tablas un nodo-r para el archivo remoto y regresa a la capa VFS, la cual coloca en sus tablas un nodo-v para el archivo que apunta al nodo-r. De nuevo, vemos aquí que todo archivo o directorio abierto tiene un nodo-v que apunta a un nodo-r o a un nodo-i.

Quien hizo la llamada recibe un descriptor de archivo para el archivo remoto. Este descriptor de archivo se asocia con el nodo-v mediante las tablas en la capa VFS. Observe que no se crean entradas en las tablas del lado del servidor. Aunque el servidor está listo para proporcionar las asas de archivo que le soliciten, no mantiene un registro de los archivos que tienen asas activas y los que no. Cuando se le envía un asa de archivo para el acceso a un archivo, verifica el asa y, si ésta es válida, la utiliza. El proceso de validación puede incluir una clave de autenticación contenida en los encabezados RPC, si la seguridad está activada.

Cuando el descriptor de archivo se utiliza en una llamada posterior al sistema, por ejemplo, READ, la capa VFS localiza el nodo-v correspondiente y por medio de él determina si es local o remoto y el nodo-i o nodo-r que lo describe.

Por razones de eficiencia, las transferencias entre el cliente y el servidor se realizan en bloques grandes, por lo general de 8 192 bytes, aunque se soliciten menos. Después de que la capa VFS del cliente ha obtenido el bloque de 8K que necesitaba, emite en forma automática una solicitud del siguiente bloque, por lo que lo recibirá rápidamente. Esta característica se conoce como **lectura adelantada** y mejora en forma considerable el desempeño.

Se sigue una política análoga para la escritura. Si una llamada WRITE proporciona menos de 8 192 bytes de datos, los datos se acumulan en forma local. Sólo cuando el último pedazo de 8K está completo, se envía al servidor. Sin embargo, al cerrar un archivo, todos sus datos se envían al servidor de manera inmediata.

Otra de las técnicas que se utilizan para mejorar el desempeño es el ocultamiento, como en el UNIX ordinario. Los servidores ocultan los datos para evitar el acceso al disco, pero esto es invisible para los clientes. Los clientes mantienen dos cachés, uno para los atributos de archivo (nodos-i) y otro para los datos del archivo. Cuando se necesita un nodo-i o un bloque del archivo, primero hay que verificar si esta solicitud se puede satisfacer mediante el caché del cliente. En este caso, se evita el tráfico en la red.

Aunque el ocultamiento por parte del cliente ayuda en mucho al desempeño, también presenta algunos problemas molestos. Supongamos que dos clientes ocultan el mismo bloque del archivo y que uno de ellos lo modifica. Cuando el otro lee el bloque, obtiene el

valor antiguo. El caché no es coherente. Hemos visto el mismo problema con los multiprocesadores. Sin embargo, en ese caso se resolvió el problema al hacer que los cachés realizaran un monitoreo en el bus para detectar todas las escrituras e invalidar o actualizar las entradas del caché según lo correspondiente. Con un caché de archivos, esto no es posible, puesto que la escritura a un archivo que provoque un encuentro con el caché de un cliente no genera tráfico en la red. Aunque lo hiciera, el monitoreo en la red es casi imposible con el hardware actual.

Debido a la severidad potencial de este problema, la implantación de NFS hace varias cosas para mitigarlo. Una de ellas es que a cada bloque caché se le asocia un cronómetro. Cuando éste expira, la entrada se descarta. Por lo general, el tiempo es de 3 segundos para los bloques de datos y de 30 segundos para los bloques de directorio. Esto reduce un poco el riesgo. Además, al abrir un archivo con caché, se envía un mensaje al servidor para revisar la hora de la última modificación. Si la última modificación ocurrió antes de capturar en el caché la copia local, se descarta la copia del caché y se utiliza la nueva copia del servidor. Por último, el cronómetro del caché expira cada 30 segundos y todos los bloques sucios (es decir, modificados) en el caché se envían al servidor.

Aún así, NFS ha recibido amplias críticas por no implantar de manera adecuada la semántica apropiada de UNIX. Una escritura a un archivo de un cliente podría o no ser vista cuando otro cliente lea el archivo, según la sincronización. Además, al crear un archivo, esta acción podría no ser visible para el mundo exterior durante un periodo de 30 segundos. Existen otros problemas similares.

Por medio de este ejemplo, vemos que aunque NFS tiene un sistema compartido de archivos, como el sistema resultante es una especie de UNIX parchado, la semántica del acceso a los archivos no está por completo bien definida y la ejecución de un conjunto de programas que cooperen entre sí podría producir diversos resultados, según la sincronización. Además, lo único con lo que trata NFS es con el sistema de archivos. No hace referencia a otros aspectos, como la ejecución de un proceso. A pesar de todo, NFS es popular y tiene un uso amplio.

5.2.6. Lecciones aprendidas

Con base en esta experiencia con varios sistemas distribuidos de archivos, Satyanarayanan (1990b) estableció algunos principios generales que piensa deben seguir los diseñadores de sistemas distribuidos de archivos. Hemos resumido estos principios en la figura 5-15. El primer principio indica que las estaciones de trabajo tienen el poder suficiente de CPU, por lo que es inteligente utilizarlo siempre que sea posible. En particular, si se tiene la opción de hacer algo en una estación de trabajo o en un servidor, hay que elegir la estación de trabajo, puesto que los ciclos del servidor son costosos y los ciclos de la estación de trabajo no lo son.

El segundo principio dice que deben utilizarse los cachés. Con frecuencia ahorran gran cantidad de tiempo de cómputo y ancho de banda de la red.

El tercer principio dice que hay que explotar las propiedades del uso. Por ejemplo, en un sistema UNIX típico, cerca de un tercio de todas las referencias a archivos son a archivos temporales, los cuales tienen tiempos de vida cortos y nunca son compartidos. Si se tratan estos archivos de manera especial se puede mejorar en mucho el desempeño. Para ser justos, existe otra escuela de pensamiento que opina: "Establece un mecanismo y apégate a él. No tengas cinco vías para hacer lo mismo". El punto de vista que uno adopte depende de si se prefiere la eficiencia o la sencillez.

Las estaciones de trabajo tienen ciclos que hay que utilizar
Utilizar el caché el máximo posible
Explotar las propiedades de uso
Minimizar el conocimiento y modificación a lo largo del sistema
Confiar en el menor número posible de entidades
Crear lotes de trabajo mientras sea posible

Figura 5-15. Principios de diseño de un sistema distribuido de archivos.

La minimización del conocimiento y modificación a lo largo de todo el sistema es importante si queremos que el sistema tenga escalabilidad. Los diseños jerárquicos ayudan a este respecto.

Un principio ya establecido en el mundo de la seguridad es el hecho de confiar en las menos entidades posibles. Si el funcionamiento correcto del sistema depende de que 10 000 estaciones de trabajo realicen lo que debieran, el sistema tiene gran problema.

Por último, el uso del procesamiento por lotes puede contribuir a un mejor desempeño. La transmisión de un archivo de 50K de una sola vez es mucho más eficiente que su envío en forma de 50 bloques de 1K.

5.3. TENDENCIAS EN LOS SISTEMAS DISTRIBUIDOS DE ARCHIVOS

Aunque los cambios rápidos son parte de la industria de la computación desde su inicio, en los últimos años los nuevos desarrollos parecen ser mucho más rápidos, tanto en el hardware como en el software. Es probable que los cambios en el hardware tengan un efecto muy importante en los sistemas distribuidos de archivos del futuro. También es probable que tengan un efecto importante los cambios en las expectativas y aplicaciones del usuario. En esta sección daremos un panorama de los cambios esperados en un futuro próximo y analizaremos algunas de las implicaciones que pueden tener estos cambios en los sistemas de archivos. Esta sección mostrará más preguntas que respuestas, pero sugerirá algunas direcciones interesantes para próximas investigaciones.

5.3.1. Hardware reciente

Antes de analizar el nuevo hardware, debemos fijarnos en el hardware antiguo con nuevos precios. Mientras la memoria se siga abaratando, podríamos ver una revolución en la forma de organización de los servidores de archivos. En la actualidad, todos los servidores de archivos utilizan discos magnéticos para el almacenamiento. La memoria principal se utiliza con frecuencia para el ocultamiento de los servidores, pero esto es tan sólo una optimización para un mejor desempeño. No es esencial.

Dentro de unos cuantos años, la memoria se puede volver tan barata que incluso las organizaciones pequeñas podrán equipar todos sus servidores de archivos con gigabytes de memoria física. Como consecuencia, el sistema de archivos puede estar presente en la memoria en forma permanente y no existirá la necesidad de los discos. Tal paso mejoraría en mucho el desempeño y haría mucho más sencilla la estructura del sistema de archivos.

La mayoría de los sistemas de archivos de la actualidad organizan los archivos como una colección de bloques, ya sea como un árbol (por ejemplo, UNIX) o como una lista ligada (por ejemplo, MS-DOS). Con un sistema de archivos en el núcleo, podría ser más sencillo un almacenamiento del archivo en forma adyacente en la memoria, en vez de separarlo en bloques. Es más fácil llevar un registro de los archivos almacenados en forma adyacente, además de que se pueden transmitir más rápido en la red. La razón de que los archivos adyacentes no se utilicen en los discos es que, si un archivo crece, su desplazamiento hacia un área del disco con más espacio es una operación cara. Por el contrario, el desplazamiento de un archivo a otra área de la memoria es una operación factible.

Sin embargo, los servidores de archivos en la memoria principal presentan un serio problema. Si se interrumpe la energía eléctrica, se pierden todos los archivos. A diferencia de los discos, que no pierden la información por una falla en la energía, la memoria principal se borra al eliminar la electricidad. La solución sería hacer respaldos continuos o por incrementos en cinta de video. Con la tecnología actual, es posible almacenar cerca de cinco gigabytes en una sola cinta de video de 8 mm, con un costo menor de 10 dólares. Aunque el tiempo de acceso es largo, si sólo se necesita tener acceso una o dos veces por año para recuperarse de las fallas en la energía, este sistema podría ser irresistible.

Un desarrollo en hardware que puede afectar a los sistemas de archivos es el disco óptico. En un principio, estos dispositivos tenían la propiedad de que sólo se podía escribir en ellos una vez (haciendo marcas en la superficie mediante un láser), pero no podían modificarse. A veces se les conocía como dispositivos **WORM (write once, read many**, una escritura y muchas lecturas). Algunos de los actuales discos ópticos utilizan láser que afectan la estructura del cristal del disco, pero no lo dañan, por lo que se pueden borrar.

Los discos ópticos tienen tres propiedades importantes:

1. Son lentos.
2. Tienen un enorme espacio de almacenamiento.
3. Tienen acceso aleatorio.

También son relativamente baratos, aunque más caros que las cintas de video. Las primeras dos propiedades son iguales a las de las cintas de video, pero la tercera abre la siguiente posibilidad. Imaginemos un servidor de archivos con un sistema de archivos de *n* gigabytes en la memoria principal y un disco óptico de *n* gigabytes como respaldo. Cuando se crea un archivo, se guarda en la memoria principal y se señala que aún no tiene un respaldo. Todos los accesos son a través de la memoria principal. Cuando la carga de trabajo es baja, los archivos que no hayan sido respaldados todavía se transfieren al disco óptico de manera secundaria, de manera que el byte *k* de la memoria vaya a dar al byte *k* del disco. Como el primer esquema, lo que tenemos aquí es un servidor de archivos en la memoria principal, pero con un dispositivo de respaldo conveniente y una asociación uno a uno con la memoria.

Otro desarrollo interesante en hardware son las redes de fibras ópticas de alta velocidad. Como hemos analizado, la razón para el uso de cachés por el cliente, con todas sus complicaciones inherentes, es evitar la lenta transferencia del servidor al cliente. Pero supongamos que podemos equipar al sistema con un servidor de archivos en la memoria principal y una red de fibras ópticas de alta velocidad. Podría ser factible deshacerse del caché del cliente y del disco del servidor, para operar con la memoria de éste último, con respaldos en el disco óptico. Esto simplificaría en mucho el software.

Al estudiar el uso de cachés del cliente, vimos que gran parte del problema es provocada por el hecho de que si dos clientes ocultan el mismo archivo y uno de ellos lo modifica, el otro no descubre esto, lo cual conduce a ciertas inconsistencias. Un poco de reflexión en torno a este tema revelará que la situación es análoga a los cachés de memoria en un multiprocesador. Sólo que en este caso, cuando un procesador modifica una palabra compartida, se envía una señal de hardware a través del bus de la memoria a los demás cachés, con el fin de permitirles que invaliden o actualicen dicha palabra. Esto no se hace en los sistemas distribuidos de archivos.

¿Por qué no se hace esto? La razón es que las interfaces actuales en la red no soportan tales señales. Sin embargo, podría ser posible construir interfaces de red que lo hicieran. Como ejemplo sencillo, consideremos el sistema de la figura 5-16, donde cada interfaz de red tiene un mapa de bits, un bit por cada archivo en el caché. Para modificar un archivo, un procesador activa el bit correspondiente en la interfaz, el cual es 0 si ningún procesador ha actualizado recientemente el archivo. La activación de un bit hace que la interfaz cree y envíe un paquete a través del anillo que verifique y active el bit correspondiente en las demás interfaces. Si el paquete recorre todo el camino sin encontrar otras máquinas que intenten utilizar el archivo, algún otro registro en la interfaz toma también el valor 1. En caso contrario, toma el valor 0. De hecho, este mecanismo proporciona una forma para cerrar el archivo en forma global en todas las máquinas, en unos cuantos microsegundos.

Después de establecer la cerradura, el procesador actualiza el archivo. Se anota cada uno de los bloques modificados del archivo (por ejemplo, mediante el uso de bits en la tabla de páginas). Al terminar la actualización, el procesador limpia el bit del mapa de bits, lo cual hace que la interfaz de la red localice el archivo mediante una tabla en memoria y que deposite en forma automática todos los bloques modificados en su posición correcta en las

demás máquinas. Cuando el archivo queda actualizado en todas partes, el bit del mapa de bits se limpia en todas las máquinas.

Es claro que esta sencilla solución se puede mejorar de varias formas, pero muestra la forma en que un hardware bien diseñado puede resolver problemas difíciles de administrar a nivel de software. Es probable que los futuros sistemas distribuidos sean apoyados por hardware especializado de varios tipos.

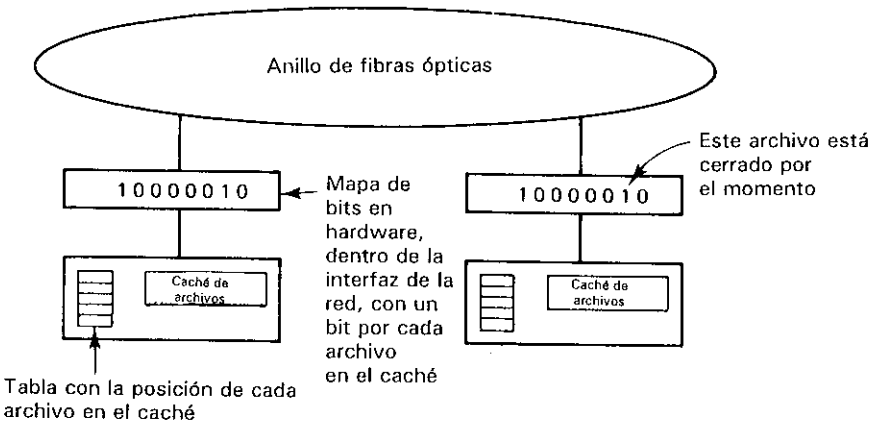


Figura 5-16. Un esquema en hardware para actualizar archivos compartidos.

5.3.2. Escalabilidad

Una tendencia definida en los sistemas distribuidos es hacia los sistemas cada vez más grandes. Esta observación tiene implicaciones para el diseño de los sistemas distribuidos de archivos. Los algoritmos que funcionan bien para los sistemas con 100 máquinas pueden trabajar un poco mal para los sistemas con 1 000 máquinas y no funcionar para sistemas con 10 000 máquinas. Para los principiantes, los algoritmos centralizados no se escalan bien. Si la apertura de un archivo necesita el contacto con un servidor centralizado para registrar el hecho de que el archivo está abierto, ese servidor se convertirá en un cuello de botella en cierto momento de crecimiento del sistema.

La forma general de enfrentar este problema es separar el sistema en unidades más pequeñas e intentar que cada una de ellas sea relativamente independiente de las demás. Si se tiene un servidor por cada unidad de asignación, esto se escala mucho mejor que un servidor. El hecho de que todos los servidores registren todas las aperturas podría ser aceptable bajo estas circunstancias.

Las transmisiones son otra área problemática. Si cada máquina realiza una transmisión por cada segundo, con n máquinas, hay un total de n transmisiones en la red por cada segundo, lo cual genera n^2 interrupciones. Es claro que si n crece, esto se puede convertir en un problema.

Los recursos y algoritmos no deben ser lineales con respecto al número de usuarios. Por ejemplo, no es una buena idea tener un servidor con una lista lineal de usuarios para protección u otros efectos. Por el contrario, las tablas de dispersión son adecuadas por lo general, puesto que el tiempo de acceso es más o menos constante y es casi independiente del número de entradas.

En general, las semánticas estrictas, como la semántica de UNIX, son más difíciles de implantar al crecer los sistemas. Son más fáciles de implantar las garantías débiles. Es claro que en este caso hay que tomar una decisión, puesto que los programadores prefieren una semántica que sea bien definida con facilidad, pero éstas son precisamente las que no se escalan bien.

En un sistema muy grande, tal vez sea necesario volver a examinar el concepto de un árbol de archivos, a la manera de UNIX. Es inevitable que al crecer el sistema, crezca también la longitud de los nombres de las rutas de acceso, lo cual tiene un costo adicional. En cierto momento, tal vez sea necesario partir el árbol en árboles más pequeños.

5.3.3. Redes de área amplia

La mayoría del trabajo actual relativo a los sistemas distribuidos se centra en los sistemas basados en LAN. En el futuro, muchos sistemas distribuidos basados en LAN serán conectados entre sí, para formar sistemas distribuidos transparentes a través de países y continentes. Por ejemplo, la PTT de Francia está colocando una pequeña computadora en cada departamento y casa de ese país. Aunque el objetivo inicial es eliminar la necesidad de las operadoras de información y los directorios telefónicos, en algún momento alguien preguntará si es posible conectar diez millones o más computadoras distribuidas en toda Francia mediante un sistema transparente, para aplicaciones jamás pensadas. ¿Qué tipo de sistemas de archivos será necesario para servir a toda Francia? ¿A toda Europa? ¿A todo el mundo? Nadie lo sabe en estos momentos.

Aunque las máquinas francesas son idénticas, en la mayoría de las redes de área amplia existe gran variedad de equipo. Esta diversidad es inevitable si existen varios compradores con presupuestos y objetivos distintos y la adquisición de equipo se realiza durante varios años en una era de rápidos cambios tecnológicos. Así, un sistema distribuido de área amplia necesita enfrentarse a la heterogeneidad. Esto hace que surjan preguntas tales como la forma de almacenar un archivo de caracteres si no todos utilizan ASCII, o el formato que se debe utilizar para los archivos que contienen números de punto flotante, si existen varias representaciones de éstos.

También es importante el cambio esperado en las aplicaciones. La mayoría de los sistemas distribuidos experimentales que se construyen en las universidades se centran en la programación en un ambiente parecido a UNIX como aplicación canónica, puesto que eso es lo que hacen los propios investigadores todos los días (al menos mientras no se encuentran en reuniones de comités o escribiendo propuestas para apoyos económicos). Los datos iniciales sugieren que no todos los cincuenta millones de ciudadanos franceses mencionarán la programación en C como su actividad principal. Al difundirse cada vez más los sistemas

distribuidos, es probable que veamos un corrimiento hacia el correo electrónico, la banca electrónica, el acceso a las bases de datos y actividades recreativas, todo lo cual cambiará el uso de los archivos, los patrones de acceso y muchas otras cosas que no sabemos en este momento.

Un problema inherente en los sistemas distribuidos masivos es que el ancho de banda de la red es muy bajo. Si la línea telefónica es la conexión principal, parece poco probable que se obtengan de ella más de 64 Kb por segundo. Tardará décadas el llevar las fibras ópticas a todas las casas, además de que costará muchos millones. Por otro lado, se pueden almacenar grandes cantidades de datos en forma barata en los discos compactos y las cintas de video. En vez de conectarse a la computadora de la compañía telefónica para buscar cierto número, podría ser más barato enviar a cada persona un disco o cinta con toda la base de datos. Tal vez habría que desarrollar sistemas de archivos donde se distinga entre la información estática y exclusiva para lectura (por ejemplo, el directorio telefónico) y la información dinámica (por ejemplo, el correo electrónico). Esta distinción podría ser la base de todo el sistema de archivos.

5.3.4. Usuarios móviles

Las computadoras portátiles son el segmento de mayor crecimiento en la industria de la computación. Las computadoras laptop, notebook y de bolsillo se pueden encontrar en todas partes y se multiplican como conejos. Aunque es difícil trabajar con la computadora mientras se maneja, no es difícil mientras se vuela. Los teléfonos en los aviones son comunes hoy en día, por lo que no debe extrañarnos tener FAX y módems móviles. Sin embargo, el ancho de banda disponible de un avión hacia la tierra es muy bajo y muchos de los lugares a los que desean ir los usuarios no tienen una conexión en línea.

La conclusión inevitable es que gran parte del tiempo, el usuario estará fuera de la línea, desconectado del sistema de archivos. Pocos sistemas actuales se diseñaron para tal uso, aunque Satyanarayanan (1990) ha informado de cierto trabajo inicial en esa dirección.

Es probable que cualquier solución tenga que basarse en el ocultamiento. Mientras esté conectado, el usuario cargará en la computadora portátil los archivos que piensa necesitar después. Éstos se utilizan mientras está desconectado. Al reconectarse, los archivos en el caché deben fusionarse con los existentes en el árbol de directorios. Puesto que la desconexión puede durar horas e incluso días, los problemas de mantenimiento de la consistencia del caché son mucho más severos que en los sistemas en línea.

Otro problema es que al ocurrir la reconexión, el usuario se puede encontrar en una ciudad lejana de su origen. Una llamada telefónica a la máquina de origen es una forma de volverse a sincronizar, pero el ancho de banda del teléfono es bajo. Además, en un sistema distribuido verdadero, bastaría contactar con el servidor local de archivos. El diseño de un sistema distribuido por completo transparente para su uso simultáneo por parte de millones de usuarios móviles que frecuentemente se desconecten se deja como ejercicio para el lector.

5.3.5. Tolerancia de fallas

Los sistemas de cómputo de la actualidad, excepto por algunos muy especializados, como los que se utilizan para el control del tráfico aéreo, no son tolerantes de fallas. Cuando

la computadora falla, se espera que los usuarios acepten esto como un hecho de la vida. Por desgracia, la población en general espera que las cosas funcionen. Si un canal de televisión, el sistema telefónico o la compañía de luz eléctrica fallan durante media hora, al otro día existen muchas personas descontentas. Con la difusión de los sistemas distribuidos, crecerá la demanda de sistemas que esencialmente nunca fallen. Los sistemas actuales no pueden cumplir ese requisito.

Es claro que tales sistemas necesitarán una considerable redundancia en el hardware y la infraestructura de comunicación, pero también la necesitarán en el software y particularmente en los datos. La réplica de archivos, que a menudo es una idea tardía en los sistemas distribuidos actuales, será un requisito esencial en los sistemas futuros. También se tendrán que diseñar los sistemas de modo que puedan funcionar cuando sólo se disponga de una parte de los datos, puesto que la insistencia en la disponibilidad de todos los datos no conduce a la tolerancia de fallas. Los tiempos de falla que ahora consideran aceptables los programadores y otros usuarios complejos, lo serán cada vez menos, al difundirse el uso de las computadoras entre las personas no especializadas.

5.3.6. Multimedia

Las nuevas aplicaciones, en particular las que implican el video en tiempo real o multimedia, tendrán un efecto enorme en los sistemas distribuidos de archivos del futuro. Los archivos de texto rara vez tienen más de unos cuantos megabytes de longitud, pero los archivos de video pueden exceder con facilidad un gigabyte. Para controlar las aplicaciones como el video por solicitud, se necesitarán sistemas de archivos por completo diferentes.

5.4. RESUMEN

El corazón de cualquier sistema distribuido es el sistema distribuido de archivos. El diseño de dicho sistema comienza con la interfaz: ¿Cuál es el modelo de archivo y qué funcionalidad debe proporcionar? Por regla, la naturaleza de un archivo no debe ser diferente en el caso distribuido y en el caso de un procesador. Como es usual, una parte importante de la interfaz la forman los nombres de los archivos y el sistema de directorios. El problema de los nombres trae consigo el aspecto de la transparencia. ¿En qué medida se relaciona el nombre de un archivo con su posición? ¿Puede el sistema desplazar por su cuenta un archivo sin que cambie su nombre? Los diversos sistemas pueden tener distintas respuestas para estas preguntas.

El uso de los archivos compartidos en un sistema distribuido es un tema complejo pero importante. Se han propuesto varios modelos de semántica, como la semántica de UNIX, la semántica de sesión, los archivos inmutables y la semántica de transacción. Cada una tiene su fortaleza y debilidad. La semántica de UNIX es intuitiva y familiar para la mayoría de los programadores (incluso para los programadores que no utilizan UNIX), pero su implantación es cara. La semántica de sesión es menos determinista, pero más eficiente. Los

archivos inmutables no son familiares para la mayoría de las personas y dificultan la actualización de los archivos. Las transacciones son con frecuencia redundantes.

La implantación de un sistema distribuido de archivos implica la toma de varias decisiones: ver si el sistema es con estado o sin estado, si se debe hacer el ocultamiento y la forma de administrar la réplica de archivos. Cada una de estas decisiones tiene consecuencias de gran alcance para los diseñadores y los usuarios. NFS ilustra una forma de construir un sistema distribuido de archivos.

Los futuros sistemas distribuidos de archivos probablemente tengan que enfrentar los cambios en la tecnología del hardware, la escalabilidad, los sistemas de área amplia, los usuarios móviles y la tolerancia de fallas, así como la introducción de multimedia. Muchos excitantes retos nos esperan.

PROBLEMAS

1. ¿Cuál es la diferencia entre un servicio de archivo que utiliza el modelo de carga/descarga y uno que utiliza el modelo de acceso remoto?
2. Un sistema de archivos permite enlaces entre los directorios. De esta forma, un directorio puede "incluir" un subdirectorio. En este contexto, ¿cuál es el criterio esencial para distinguir un sistema de directorios con estructura de árbol de un sistema con una estructura general de gráfica?
3. En el texto se señaló que los apuntadores a archivos compartidos no se pueden implantar de manera razonable con la semántica de sesión. ¿Se puede implantar si existe un servidor de archivos que proporcione la semántica de UNIX?
4. Mencione dos propiedades útiles de los archivos inmutables.
5. ¿Por qué ciertos sistemas distribuidos utilizan nombres de dos niveles?
6. ¿Por qué incluyen los servidores sin estado un ajuste de archivo en cada solicitud? ¿Es esto necesario para los servidores con estado?
7. Uno de los argumentos del texto en favor de los servidores de archivos con estado es que los nodos-i de los archivos abiertos se pueden mantener dentro de la memoria, lo que reduce el número de operaciones en disco. Proponga una implantación de un servidor sin estado que logre casi la misma mejora en el desempeño. ¿En cuáles sentidos, si existe alguno, es su propuesta mejor o peor que la propuesta de servidor con estado?
8. Al utilizar la semántica de sesión, siempre ocurre que los cambios a un archivo son visibles en forma inmediata a los procesos que realizan el cambio y nunca son visibles a los procesos de otras máquinas. Sin embargo, está abierto el problema de si deben o no ser visibles en forma inmediata a los demás procesos de la misma máquina. Dé un argumento en cualquiera de los sentidos.

9. ¿Por qué pueden utilizar los cachés de archivos el algoritmo LRU y los algoritmos de paginación de la memoria virtual no? Respalde sus argumentos con cifras aproximadas.
10. En la sección de consistencia del caché analizamos el problema de la forma en que un administrador del caché del cliente sabe que un archivo del caché está actualizado. El método sugerido consiste en hacer contacto con el servidor para que éste compare los tiempos del cliente y el servidor. ¿Fracasa este método si los relojes del cliente y el servidor son distintos?
11. Consideremos un sistema donde el ocultamiento del cliente se realiza mediante el algoritmo de escritura a través del caché. Se guardan en el caché los bloques individuales, en vez de archivos completos. Supongamos que un cliente está a punto de leer un archivo en forma secuencial y que algunos de los bloques están en el caché y otros no. ¿Qué problema puede surgir y qué se puede hacer con él?
12. Imaginemos que un sistema distribuido de archivos utiliza el ocultamiento del cliente con una política de escritura retrasada al servidor. Una máquina abre, modifica y cierra un archivo. Un minuto después, otra máquina lee el archivo del servidor. ¿Cuál versión obtiene?
13. Algunos sistemas distribuidos de archivos utilizan el ocultamiento del cliente con una escritura retrasada al servidor o una escritura al cierre. Además de los problemas con la semántica, estos sistemas presentan otro problema. ¿Cuál? (Sugerencia: Piense en la confiabilidad.)
14. Las mediciones han mostrado que muchos archivos tienen una vida muy breve. ¿Qué implicaciones tiene esto para la política de ocultamiento del cliente?
15. Algunos sistemas distribuidos de archivos utilizan nombres de dos niveles, ASCII y binario, como hemos analizado en el capítulo; otros no utilizan este esquema y sólo usan nombres en ASCII. De manera análoga, algunos servidores de archivos son con estado y otros sin estado, lo que produce cuatro combinaciones de estas características. Una de estas combinaciones es un poco menos recomendable que las demás. ¿Cuál es? ¿Por qué?
16. Cuando los sistemas de archivos duplican a éstos, por lo general no duplican a todos. Dé un ejemplo de un archivo que no vale la pena duplicar.
17. Un archivo se reproduce en 10 servidores. Enumere todas las combinaciones de quórum de lectura y quórum de escritura permitidas por el algoritmo del voto.
18. En el caso de un servidor de archivos en la memoria principal que guarda los archivos de forma adyacente, cuando un archivo crece más allá de su unidad de asignación actual, éste debe copiarse. Supongamos que el tamaño promedio de los archivos es de 20K bytes y que se necesitan 200 nanosegundos para copiar una palabra de 32 bits. ¿Cuántos archivos se pueden copiar cada segundo? ¿Puede usted sugerir una forma de copiado que no ocupe el CPU del servidor de archivos todo el tiempo?
19. En NFS, cuando se abre un archivo, se regresa un asa de archivo, de manera análoga a cuando se regresa un descriptor de archivo en UNIX. Supongamos que un servidor en NFS falla después de dar un asa de archivo a un usuario. Cuando el servidor vuelve a

arrancar, ¿seguirá siendo válida el asa de archivo? En tal caso, ¿cómo funciona? En caso contrario, ¿viola esto el principio de no estado?

20. En el esquema del mapa de bits de la figura 5-16, ¿es necesario que todas las máquinas que ocultan un archivo dado utilicen la misma entrada de la tabla para él? En tal caso, ¿cómo se puede arreglar esto?

6

Memoria compartida distribuida

En el capítulo 1 vimos que existen dos tipos de sistemas con varios procesadores: multiprocesadores y multicomputadoras. En un multiprocesador, dos o más CPU comparten una memoria principal común. Cualquier proceso, en cualquier procesador, puede leer o escribir cualquier palabra en la memoria compartida, sólo moviendo datos desde o hacia la localidad deseada. Por el contrario, en una multicomputadora, cada CPU tiene su memoria particular. Nada se comparte.

Para hacer una analogía en agricultura, un multiprocesador es un sistema como una manada de cerdos (procesos) comiendo de un solo plato (memoria compartida). Una multicomputadora es un diseño mediante el cual cada cerdo tiene su plato para comer. Haciendo una analogía educacional, un multiprocesador es un pizarrón al frente del salón que todos los estudiantes pueden observar, mientras que una multicomputadora es cada estudiante viendo al propio cuaderno. Aunque esta diferencia parecería menor, tiene consecuencias de gran alcance.

Las consecuencias afectan tanto al hardware como al software. Primero veamos las implicaciones para el hardware. Diseñar una máquina en donde muchos procesadores utilizan la misma memoria de forma simultánea es muy difícil. Los multiprocesadores basados en buses, descritos en la sección 1.3.1, no pueden utilizarse con más de una docena de procesadores, ya que el bus puede convertirse en un cuello de botella. Los multiprocesadores con conmutador, descritos en la sección 1.3.2, pueden escalarse a grandes sistemas, pero son relativamente caros, lentos, complejos y difíciles de mantener.

En contraste, las grandes multicomputadoras son fáciles de construir. Se puede tomar un número casi ilimitado de computadoras sencillas, cada una con un CPU, una memoria y un interfaz de red y conectarlas entre sí. Las multicomputadoras con miles de procesadores están disponibles en los comercios con diversos fabricantes. (Observe por favor, que durante

este capítulo utilizaremos los términos "CPU" y "procesador" de forma indistinta). Desde la perspectiva del diseñador de hardware, por lo general las multicomputadoras son preferibles a los multiprocesadores.

Ahora consideremos el software. Se conocen muchas técnicas de programación de multiprocesadores. Para la comunicación, un proceso sólo escribe datos a memoria, para que sean leídos por los demás. Para la sincronización, se pueden utilizar las regiones críticas, con semáforos o monitores que proporcionen la exclusión mutua necesaria. Existe muchísima gran cantidad de bibliografía disponible acerca de la comunicación entre procesos y la sincronización en máquinas de memoria compartida. Cada libro de texto de sistemas operativos escrito en los últimos veinte años dedica uno o más capítulos al tema. En resumen, existe gran cantidad de conocimiento teórico y práctico acerca de la forma de programar un multiprocesador.

Con las multicomputadoras, ocurre lo inverso. Por lo general, la comunicación tiene que utilizar la transferencia de mensajes, haciendo que la entrada/salida sea la abstracción central. La transferencia de mensajes trae consigo varios aspectos delicados; entre ellos, el flujo de control, la pérdida de mensajes, el uso de buffer y el bloqueo. Aunque se han propuesto varias soluciones, la programación con transferencia de mensajes todavía es difícil.

Para ocultar algunas dificultades asociadas con la transferencia de mensajes, Birrell y Nelson (1984) propusieron utilizar las llamadas a procesamientos remotos. En su esquema, que ahora tiene amplio uso, la comunicación real se esconde bajo procedimientos de biblioteca. Para utilizar un servicio remoto, un proceso solo llama al procedimiento de biblioteca adecuado, el cual empaqueta el código de operación y los parámetros en un mensaje, enviándolo por la red y esperando la respuesta. Aunque esto funciona con frecuencia, no puede utilizarse con facilidad para la transferencia de gráficas y otras estructuras de datos complejas con apuntadores. También falla en los programas que utilizan variables globales, y hace cara a la transferencia de grandes arreglos, ya que estos deben pasarse por valor y no por referencia.

En resumen, desde la perspectiva de un diseñador de software, los multiprocesadores son en definitivo preferibles a las multicomputadoras. He ahí el dilema. Las multicomputadoras son más fáciles de construir pero más difíciles de programar y los multiprocesadores al contrario: más difíciles de construir pero más fáciles de programar. Lo que necesitamos son sistemas que sean fáciles de construir y de programar. Intentar construir estos sistemas es el tema de este capítulo.

6.1. INTRODUCCIÓN

En los primeros días de la computación distribuida, todos suponían de manera implícita que los programas en las máquinas sin memoria compartida físicamente (es decir, multicomputadoras) se ejecutaban obviamente en diferentes espacios de direcciones. Con este punto de vista, la comunicación se pensaba de manera natural en términos de la transferencia

de mensajes entre espacios de direcciones ajenos, como hemos descrito. En 1986, Li propuso un esquema diferente, conocido como **memoria compartida distribuida (DSM)** (Li, 1986; y Li y Hudak, 1989). En resumen, Li y Hudak propusieron tener una colección de estaciones de trabajo conectadas por una LAN compartiendo un solo espacio de direcciones virtuales con páginas. En la variante más simple, cada página está presente en una máquina. En el hardware se hace una llamada a las páginas *locales*, con toda la velocidad de la memoria. Un intento por llamar a una página en una máquina diferente causa un fallo de página en hardware, el cual realiza un señalamiento al sistema operativo. Entonces, el sistema operativo envía un mensaje a la máquina remota, quien encuentra la página necesaria y la envía al procesador solicitante. Entonces se reinicia la instrucción detenida y se puede concluir.

En esencia, este diseño es similar a los sistemas de memoria virtual tradicionales: cuando un proceso toca una página no residente, ocurre un señalamiento y el sistema operativo busca la página y la asocia con el proceso. La diferencia aquí es que en vez de obtener la página del disco, el sistema operativo la obtiene de otro procesador en la red. Sin embargo, para el usuario de los procesos, el sistema se parece mucho a un multiprocesador tradicional, con varios procesadores libres para leer y escribir en la memoria compartida. Toda la comunicación y la sincronización se puede hacer por medio de la memoria, sin que la comunicación sea visible para el usuario de los procesos. De hecho, Li y Hudak diseñaron un sistema que es fácil de programar (memoria compartida lógicamente) y fácil de construir (sin memoria compartida físicamente).

Por desgracia, no existe algo gratuito en la vida. Aunque de hecho este sistema es fácil de programar y de construir, para muchas aplicaciones exhibe un desempeño pobre, ya que las páginas andan de un lado al otro de la red. Este comportamiento es análogo a examinar los sistemas de memoria virtual de un procesador. En los últimos años, un área de investigación intensa ha sido la de hacer que estos sistemas de memoria compartida distribuidas sean más eficientes, de modo que todavía hay numerosas técnicas por descubrir. Todas éstas tienen el objetivo de minimizar el tráfico de la red y reducir la latencia entre el momento de una solicitud a memoria y el momento que se satisface ésta.

Un método consiste en no compartir todo el espacio de direcciones, sino sólo una porción seleccionada de éste, a saber, aquellas variables o estructuras de datos que se necesitan utilizar en más de un proceso. En este modelo, uno no piensa que cada máquina tiene acceso directo a una memoria ordinaria sino a una colección de variables compartidas, lo cual produce un alto nivel de abstracción. Esta estrategia no sólo reduce en gran medida la cantidad de datos por compartir, sino que, en la mayoría de los casos, se dispone de información considerable acerca de los datos compartidos disponibles, como su tipo, lo que puede ayudar a optimizar la implantación.

Una posible optimización consiste en repetir las variables compartidas en varias máquinas. Al compartir las variables duplicadas en vez de páginas completas, el problema de simular un multiprocesador ha sido reducido al de conservar de manera consistente varias copias de un conjunto de estructuras de datos tipificados. En potencia, las lecturas pueden hacerse de manera local, sin ningún tráfico en la red, y las escrituras mediante un protocolo

de actualización con varias copias. Estos protocolos tienen amplio uso en los sistemas distribuidos de bases de datos, por lo que algunas ideas en ese campo podrán ser de utilidad.

Si vamos más lejos en la dirección de estructurar el espacio de direcciones, en vez de sólo compartir las variables podemos compartir los tipos de datos encapsulados, con frecuencia llamados **objetos**. Estos difieren de las variables compartidas en que cada objeto no sólo tiene algunos datos, sino también procedimientos, llamados **métodos**, que actúan sobre los datos. Los programas sólo manipulan un dato del objeto mediante sus métodos. El acceso directo a los datos no están permitidos. Al restringir el acceso de esta forma, se pueden obtener diversas optimizaciones nuevas.

El hacer todo en software tiene un conjunto diferente de ventajas y desventajas al hecho de utilizar el hardware de paginación. En general, esto tiende a establecer más restricciones para los programadores pero se puede obtener un mejor desempeño. Muchas de estas restricciones (por ejemplo, el trabajo con objetos) se considera una buena práctica en ingeniería de software y son recomendables. Regresaremos a este tema más tarde.

Antes de entrar con más detalle a la memoria compartida distribuida, debemos retroceder algunos pasos para ver lo que en realidad es la memoria compartida y cómo trabajan los multiprocesadores con dicha memoria. Después de esto, examinaremos la semántica del hecho de compartir, que será muy sutil. Por último, regresaremos al diseño de sistemas de memoria compartida distribuida. Puesto que la memoria compartida distribuida puede relacionarse de forma íntima con la arquitectura de la computadora, los sistemas operativos, los de tiempo de ejecución e incluso con los lenguajes de programación, todos estos temas jugarán cierto papel en este capítulo.

6.2. ¿QUÉ ES LA MEMORIA COMPARTIDA?

En esta sección examinaremos varios tipos de multiprocesadores de memoria compartida, desde las más sencillos, que operan sobre un bus, hasta los más avanzados, con esquemas de ocultamiento muy complejos. Estas máquinas son importantes para comprender la memoria compartida distribuida, ya que gran parte del trabajo de la DSM ha sido inspirado por los avances en la arquitectura de multiprocesadores. Además, muchos de los algoritmos son tan similares que a veces es difícil decir si una máquina avanzada es un multiprocesador o una multicomputadora que utiliza una implantación en hardware de la memoria compartida distribuida. Concluiremos con una comparación de las diversas arquitecturas de multiprocesadores con algunos sistemas de memoria compartida distribuida y descubriremos que existe un espectro de diseños posibles, desde aquellos por completo contenidos en hardware hasta aquellos contenidos en software. Al examinar todo el espectro, podemos tener mejor idea de dónde encaja DSM.

6.2.1. Memoria en circuitos

Aunque la mayoría de las computadoras tienen memoria externa, existen también circuitos independientes con un CPU y con toda la memoria. Estos circuitos se producen por millones y se utilizan con amplitud en los automóviles, en aparatos e incluso en juguetes. En este diseño, la porción de CPU del circuito tiene direcciones y líneas de datos que se conectan en directo a la porción de memoria. La figura 6-1(a) muestra un diagrama simplificado de este circuito.

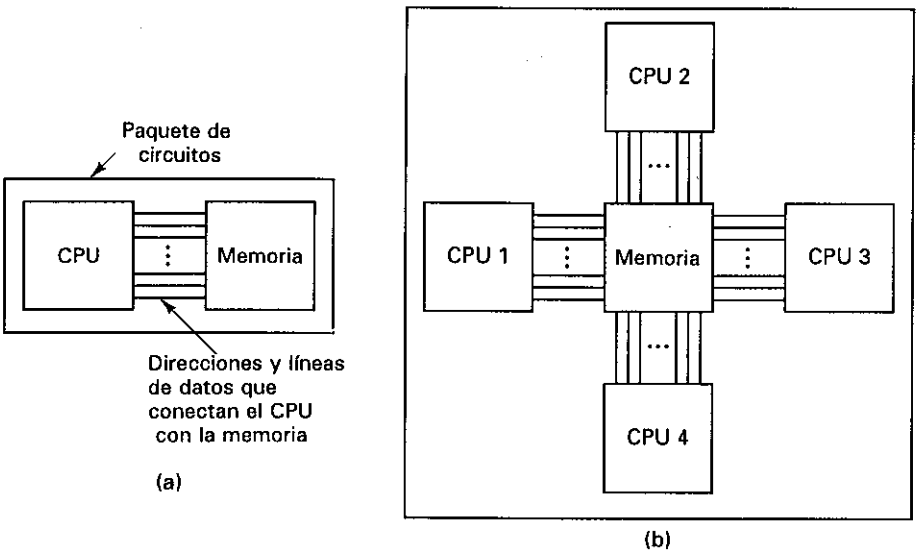


Figura 6-1.(a) Una computadora de un solo circuito. (b) Un multiprocesador de memoria compartida hipotético.

Uno puede imaginar que una extensión simple de este circuito tendría varios CPU compartiendo de forma directa la misma memoria, como se muestra en la figura 6-1(b). Aunque es posible construir un circuito como éste, sería complicado, caro y poco usual. Un intento por construir un multiprocesador de un circuito de esta forma, digamos con 100 CPU que tengan un acceso directo a la misma memoria sería imposible por razones de ingeniería. Se necesita un método diferente para compartir la memoria.

6.2.2. Multiprocesadores basados en un bus

Si analizamos de cerca la figura 6-1(a), vemos que la conexión entre el CPU y la memoria es una colección de cables paralelos, algunos con la dirección a la que desea leer o escribir el CPU, algunos para enviar o recibir datos y el resto para controlar las transferencias. A esta colección de cables se le llama un **bus**. Este bus está integrado en un circuito, pero en muchos sistemas, los buses son externos y se utilizan para conectar tarjetas de

circuitos impresos con CPU, memorias y controladores de E/S. En una computadora de escritorio, el bus está grabado por lo general en la tarjeta principal (la tarjeta madre), que contiene al CPU y parte de la memoria, donde se conectan las tarjetas de E/S. En las minicomputadoras, el bus es a veces un cable plano tendido entre los procesadores, las memorias y los controladores de E/S.

La forma simple pero práctica de construir un multiprocesador es basarlo en un bus al que se conecte más de un CPU. La figura 6-2(a) ilustra un sistema con tres CPU y una memoria compartida entre ellos. Cuando cualquiera de los CPU desea leer una palabra de la memoria, coloca la dirección de la palabra deseada en el bus y tiende (coloca una señal en) una línea de control de bus para indicar que desea realizar una lectura. Cuando la memoria ha encontrado la palabra requerida, coloca ésta en el bus y tiende otra línea de control para anunciar que está lista. Entonces, el CPU lee la palabra. La escritura se realiza de manera análoga.

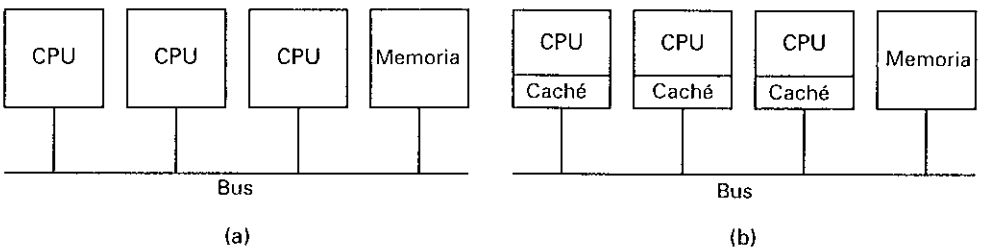


Figura 6-2.(a) Un multiprocesador. (b) Un multiprocesador con ocultamiento.

Para evitar que dos o más CPU intenten el acceso a la memoria al mismo tiempo, se necesita algún tipo de arbitraje del bus. Se pueden utilizar varios esquemas. Por ejemplo, para conseguir el bus, un CPU primero debe solicitarlo, al tender una línea de solicitud especial. Después de recibir el permiso podrá utilizar el bus. La concesión de este permiso puede hacerse de forma centralizada, utilizando un dispositivo de arbitraje de bus, o de forma descentralizada, donde el primer CPU que realice una solicitud en el bus ganará cualquier conflicto.

La desventaja de tener un solo bus es que, con tres o cuatro CPU, éste de seguro se sobrecargará. El método usual para reducir la carga del bus es equipar a cada CPU con un **caché husmeador**, llamado así debido a que "husmea" en el bus. Los cachés se muestran en la figura 6-2(b). Han sido el tema de gran cantidad de investigación durante años (Agarwal *et al.*, 1988; Agarwal y Cherian, 1989; Archibald y Baer, 1986; Cheong y Veidenbaum, 1988; Dahlgren *et al.*, 1994; Eggers y Katz, 1989a, 1989b; Nayfeh y Olukotun, 1994; Przybylski *et al.*, 1988; Scheurich y Dubois, 1987; Thekkath y Eggers, 1994; Vernon *et al.*, 1988 y Weber y Gupta, 1989). Todos estos artículos presentan ligeras variantes de **protocolos de consistencia del caché**, es decir, las reglas para asegurarse de que los diferentes cachés no contengan valores diferentes para la misma localidad de memoria.

Un protocolo en particular común es el de **escritura a través del caché**. Cuando un CPU lee por primera vez una palabra de memoria, esa palabra es llevada por el bus y guardada en el caché del CPU solicitante. Si más tarde se necesita esa palabra, el CPU la toma del caché sin necesidad de hacer una solicitud a memoria, reduciendo así el tráfico del bus. Estos dos casos, el fracaso de la lectura (la palabra no aparece en el caché) y el éxito de la lectura (la palabra sí está en el caché) se muestran en la figura 6-3 como las dos primeras líneas de la tabla. En los sistemas simples, sólo se guarda en el caché la palabra solicitada, pero en la mayoría de los sistemas, un bloque de palabras (digamos, 16 o 32 palabras) se transfiere y guarda en el caché en el acceso inicial y se conserva ahí para un posible uso posterior.

Evento	Acción realizada por un caché en respuesta a la operación de su propia CPU	Acción realizada por un caché en respuesta a una operación de un CPU remoto
Fracaso de lectura	Buscar datos en memoria y guardar en el caché	(No hay acción)
Éxito de lectura	Buscar datos del caché local	(No hay acción)
Fracaso de escritura	Actualizar los datos en memoria y guardar en caché	(No hay acción)
Éxito de escritura	Actualizar memoria y caché	Invalida la entrada del caché

Figura 6-3. El protocolo de consistencia de *escritura a través del caché*. Las entradas para el *éxito* en la tercera columna significan que el CPU husmeador tiene la palabra en su caché, y no que el CPU solicitante la tiene.

Cada CPU realiza su ocultamiento en forma independiente de los demás. En consecuencia, es posible que una palabra en particular se oculte en dos o más CPU al mismo tiempo. Ahora, consideremos lo que sucede cuando se realiza una escritura. Si ningún CPU tiene la palabra escrita en su caché, la memoria sólo se actualiza, como si el ocultamiento no hubiera sido utilizado. Esta operación requiere un ciclo de bus normal. Si el CPU que realiza la escritura tiene la única copia de la palabra, se actualiza su caché y también la memoria mediante el bus.

Hasta aquí todo va bien. El problema surge cuando un CPU desea escribir una palabra que se encuentra en los cachés de dos o más CPU. Si la palabra se encuentra en la actualidad en el caché del CPU que realiza la escritura, la entrada de caché se actualiza. Esté o no, también se escribe en el bus para actualizar la memoria. Todos los demás cachés ven la escritura (ya que se encuentran husmeando el bus) y verifican si contienen también la palabra por modificar. Si es así, invalidan sus entradas de caché, de modo que después de que completan la escritura, la memoria está actualizada y sólo una máquina tiene la palabra en su caché.

Una alternativa para invalidar otras entradas de caché es actualizarlas todas. Sin embargo, en la mayoría de los casos, la actualización es más lenta que la invalidación, ya que la última sólo requiere proporcionar la dirección por invalidar, mientras que la actualización necesita además proporcionar la nueva entrada para el caché. Si estos dos elementos deben estar presentes en el bus de manera consecutiva, se requerirán ciclos adicionales. Aunque

sea posible colocar una dirección y una palabra dato en el bus de manera simultánea, si el tamaño de bloque del caché es mayor de una palabra, se necesitarán varios ciclos de bus para actualizar todo el bloque. El dilema invalidación *vs.* actualización aparece en todos los protocolos de caché y también en los sistemas DSM.

El protocolo completo se resume en la figura 6-3. La primera columna enumera los cuatro eventos básicos que pueden suceder. La segunda dice lo que hace un caché en respuesta a las acciones del *propio* CPU. La tercera nos dice lo que sucede cuando un caché ve (al husmear) que un CPU *diferente* tiene éxito o fracaso. El único momento en que el caché *S* (el husmeador) hace algo es cuando ve que otro CPU ha escrito una palabra que *S* tiene oculto (un éxito de escritura desde el punto de vista de *S*). La acción de *S* consiste en eliminar la palabra de su caché.

El protocolo de *escritura a través del caché* es fácil de entender e implantar, pero tiene la seria desventaja de que todas las escrituras utilizan el bus. Aunque es cierto que el protocolo reduce el tráfico del bus en cierta medida, el número de CPU que se conecta a un bus es aún muy pequeño como para permitir la construcción de multiprocesadores a larga escala que lo utilicen.

Por fortuna, para muchos programas actuales, una vez que un CPU ha escrito una palabra, es probable que ese CPU necesite otra vez la palabra, y es poco probable que otro CPU utilice la palabra con rapidez. Esta situación sugiere que si el CPU que utiliza la palabra puede de alguna manera tener una "membresía" temporal, podría evitar tener que actualizar la memoria en escrituras posteriores hasta que un CPU diferente exhiba su interés por la palabra. Tales protocolos de caché existen. Goodman (1983) diseñó el primero, llamado **de una escritura**. Sin embargo, este protocolo fue diseñado para trabajar con un bus existente y por lo tanto fue más complicado que lo estrictamente necesario. Más adelante describiremos una versión simplificada de él, típica de todos los protocolos de membresía. Archibald y Baer (1986) describen y comparan otros protocolos.

Nuestro protocolo maneja bloques de cachés, cada uno de los cuales puede estar en uno de los siguientes tres estados:

1. INVÁLIDO: Este bloque de caché no contiene datos válidos.
2. LIMPIO: La memoria está actualizada; el bloque puede estar en otros cachés.
3. SUCIO: La memoria es incorrecta; ningún otro caché puede contener al bloque.

La idea básica es que una palabra leída por varios CPU esté presente en todos sus cachés. Una palabra en la que una máquina escribe repetidamente se guarda en su caché y no se vuelve a escribir en memoria después de cada escritura, para reducir el tráfico del bus.

La operación del protocolo puede ilustrarse mejor con un ejemplo. Para simplificar este ejemplo, supondremos que cada bloque de caché consta de una palabra. En un principio, *B* tiene una copia oculta de la palabra en la dirección *W*, como se ilustra en la figura 6-4(a). El valor es W_1. La memoria también tiene una copia válida. En la figura 6-4(b), *A* solicita y obtiene una copia de *W* desde la memoria. Aunque *B* ve pasar la solicitud de lectura, no responde.

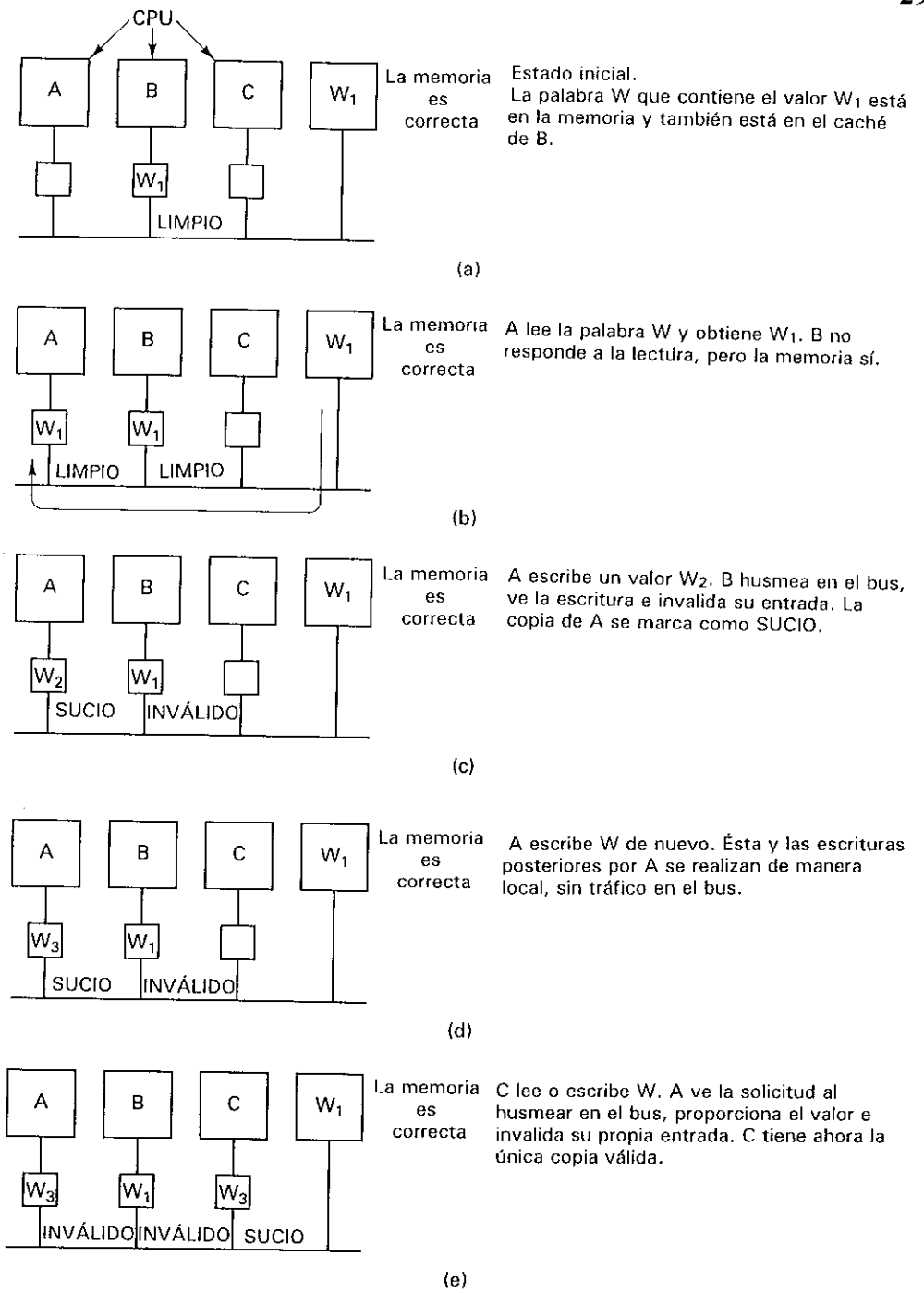


Figura 6-4. Un ejemplo de cómo funciona un protocolo de membresía de caché.

Ahora *A* escribe un nuevo valor, W_2, a *W*. *B* ve la solicitud de escritura y responde invalidando su entrada de caché. El estado de *A* cambia a SUCIO, como se muestra en la figura 6-4(c). El estado SUCIO significa que *A* tiene la única copia oculta de *W* y que la memoria ya no está actualizada para *W*.

En este punto, *A* escribe la palabra otra vez, como se muestra en la figura 6-4(d). La escritura se hace localmente, en el caché, sin tráfico de bus. Las escrituras posteriores también evitan la actualización de la memoria.

En algún momento, algún otro CPU, *C* en la figura 6-4(c), tiene acceso a la palabra. *A* ve la solicitud en el bus y da una señal que inhibe la respuesta de la memoria. En vez de eso, *A* proporciona la palabra necesaria e invalida su entrada. *C* ve que la palabra proviene de otro caché y no de la memoria, y que se encuentra en el estado SUCIO, de modo que marca la entrada como corresponde. *C* es ahora el propietario, lo que significa que puede leer y escribir la palabra sin hacer solicitud al bus. Sin embargo, también tiene la responsabilidad de observar si otro CPU solicita la palabra, y hacer el servicio él mismo. La palabra permanece en estado SUCIO hasta que se elimine del caché donde se encuentra en la actualidad por razones de espacio. En ese momento, desaparece de todos los cachés y se escribe en la memoria.

Muchos multiprocesadores pequeños utilizan un protocolo de consistencia de caché similar a éste, por lo general con pequeñas variaciones. Tiene tres propiedades importantes:

1. La consistencia se logra haciendo que todos los cachés husmeen el bus.
2. El protocolo se integra dentro de la unidad de administración de memoria.
3. Todo el algoritmo se realiza en un ciclo de memoria.

Como veremos posteriormente, una parte de esto no funciona para multiprocesadores (con conmutador) de mayor tamaño y nada es válido para la memoria compartida distribuida.

6.2.3. Multiprocesadores basados en un anillo

El siguiente paso en la ruta hacia los sistemas de memoria compartida distribuida son los multiprocesadores basados en un anillo, cuyo ejemplo es **Memnet** (Delp, 1988; Delp *et al.*, 1991 y Tam *et al.*, 1990). En Memnet, un espacio de direcciones se divide en una parte privada y una compartida. La parte privada se divide en regiones, de modo que cada máquina tenga un pedazo para su pila y otros datos y códigos no compartidos. La parte compartida es común para todas las máquinas (y distribuida entre ellas) y se guarda de manera consistente mediante un protocolo de hardware parecido a los utilizados en los multiprocesadores basados en bus. La memoria compartida se divide en bloques de 32 bytes, que es la unidad mediante la cual se realizan las transferencias entre las máquinas.

Todas las máquinas en Memnet están conectadas mediante un anillo de fichas modificado. El anillo consta de 20 cables paralelos, que juntos permiten enviar 16 bits de datos y 4 bits de control cada 100 nanosegundos, para una velocidad de datos de 160 Mb/segundo.

El anillo se ilustra en la figura 6-5(a). La interfaz de anillo, MMU (Unidad de Administración de Memoria), el caché y una parte de la memoria se integran en el **dispositivo Memnet**, que se muestra en el tercio superior de la figura 6-5(b).

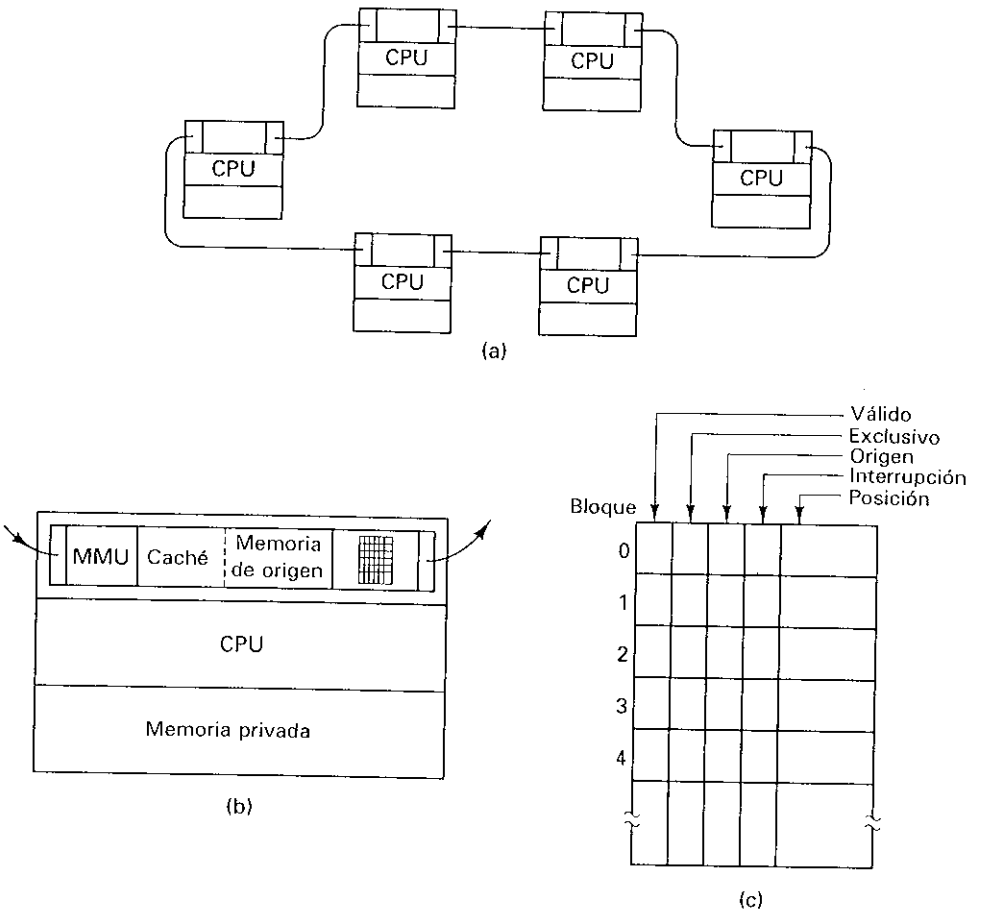


Figura 6-5.(a) El anillo Memnet. (b) Una máquina. (c) la tabla del bloque.

A diferencia de los multiprocesadores basados en bus de la figura 6-2, en Memnet no existe memoria global centralizada. En vez de esto, cada bloque de 32 bytes en el espacio compartido de direcciones tiene una máquina de origen donde la memoria física está siempre reservada para él, en el campo de *Memoria de origen* de la figura 6-5(b). Un bloque puede ocultarse en una máquina distinta a su máquina de origen. (Las áreas del caché y de la memoria de origen comparten la misma pila de buffers, pero como se utilizan de manera diferente, aquí los consideramos como entes separados.) Un bloque exclusivo de lectura puede estar presente en varias máquinas; un bloque de escritura-lectura sólo puede estar

presente en una máquina. En ambos casos, un bloque no tiene que estar presente en su máquina de origen. Todo lo que hace la máquina de origen es proporcionar un lugar seguro para guardar el bloque si ninguna otra máquina desea ocultarlo. Esta característica es necesaria ya que no existe la memoria global. De hecho, la memoria global ha estado extendida sobre todas las máquinas.

El dispositivo Memnet en cada máquina contiene una tabla, que se muestra en la figura 6-5(c), la cual contiene una entrada por cada bloque en el espacio compartido de direcciones, indizada por el número de bloque. Cada entrada contiene un bit *Válido* que indica si el bloque está presente en el caché y actualizado, un bit *Exclusivo*, especificando si la copia local, si existe, es la única, un bit *Origen*, que se activa sólo si ésta es la máquina de origen del bloque, un bit de *Interrupción*, utilizado para forzar las interrupciones y un campo de *Posición* que indica la localización del bloque en el caché si está presente y es válido.

Después de ver la arquitectura de Memnet, examinaremos el protocolo que utiliza. Cuando el CPU desea leer una palabra de la memoria compartida, la dirección de memoria por leer se transfiere al dispositivo Memnet, el cual verifica la tabla del bloque para ver si está presente. De ser así, la solicitud es satisfecha de inmediato. En caso contrario, el dispositivo Memnet espera hasta capturar la ficha que circula; después, coloca un paquete de solicitud en el anillo y suspende el CPU. El paquete de solicitud contiene la dirección deseada y un campo vacío de 32 bytes.

Mientras el paquete pasa por el anillo, cada dispositivo Memnet del camino verifica si tiene el bloque necesario. De ser así, coloca el bloque en el campo vacío y modifica el encabezado del paquete para inhibir la acción de las máquinas posteriores. Si el bit *Exclusivo* del bloque está activo, se limpia. Como el bloque tiene que encontrarse en algún lado, cuando el paquete regresa al emisor, se garantiza que contiene al bloque solicitado. El CPU que envía la solicitud guarda entonces el bloque, satisface la solicitud y libera al CPU.

Surge un problema si la máquina solicitante no tiene espacio libre en su caché para contener el bloque recibido. Para hacer espacio, toma al azar un bloque oculto y lo envía a su origen, con lo que libera un espacio del caché. Los bloques cuyo bit *Origen* están activados nunca se eligen, pues se encuentran en su origen.

El trabajo de escritura es diferente al de lectura. Hay que distinguir tres casos. Si el bloque que contiene la palabra por escribir está presente y es la única copia en el sistema (es decir, el bit *Exclusivo* está activado), la palabra sólo se escribe de manera local.

Si el bloque necesario está presente pero no es la única copia, se envía primero un paquete de invalidación por el anillo para que las otras máquinas desechen sus copias del bloque por escribir. Cuando el paquete de invalidación se envía de regreso al solicitante, el bit *Exclusivo* se activa para ese bloque y se procede a la escritura local.

Si el bloque no está presente, se envía un paquete que combina una solicitud de lectura y una de invalidación. La primera máquina que tenga el bloque lo copia en el paquete y desecha su copia. Todas las máquinas posteriores sólo desechan el bloque de sus cachés. Cuando el paquete regresa al emisor, éste lo guarda y escribe en él.

Memnet es similar a un multiprocesador basado en bus de muchas maneras. En ambos casos, las operaciones de lectura siempre regresan el valor escrito de manera más reciente.

Además, en ambos diseños, un bloque puede estar ausente de un caché, presente en varios cachés para su lectura, o presente en un caché para escritura. Los protocolos también son similares; sin embargo, Memnet no tiene memoria global centralizada.

La mayor diferencia entre los multiprocesadores basados en bus y los basados en anillos como Memnet es que los primeros están fuertemente acoplados; con regularidad, los CPU están en un gabinete. En contraste, las máquinas en un multiprocesador basado en anillo pueden estar menos acopladas, e incluso potencialmente en escritorios dispersos en un edificio, como máquinas en una LAN, aunque este acoplamiento vago puede afectar el desempeño. Además, a diferencia de un multiprocesador basado en un bus, uno basado en un anillo (como Memnet) no tiene memoria global separada. Los cachés son lo único que existe. En ambos aspectos, los multiprocesadores basados en un anillo son casi una implantación en hardware de la memoria compartida distribuida.

Uno está tentado a decir que un multiprocesador basado en un anillo es como un ornitorrinco con pico de pato; en teoría, no debería existir, ya que combina las propiedades de dos categorías que dicen ser mutuamente exclusivas (multiprocesadores y máquinas con memoria compartida distribuida; mamíferos y aves, respectivamente). No obstante, existe, y muestra que las dos categorías no son tan distintas como uno pensaría.

6.2.4. Multiprocesadores con conmutador

Aunque los multiprocesadores basados en un bus y los basados en un anillo trabajan bien para sistemas pequeños (alrededor de 64 CPU), no se escalan bien a sistemas con cientos o miles de CPU. Al agregar CPU, en algún punto se satura el ancho de banda del bus o el anillo. Añadir más CPU no mejora el desempeño del sistema.

Se tienen dos métodos para atacar el problema del ancho de banda insuficiente:

1. Reducir la cantidad de comunicación.
2. Incrementar la capacidad de comunicación.

Hemos visto un ejemplo de intento por reducir la cantidad de comunicación mediante el ocultamiento. El trabajo adicional en esta área podría centrarse en mejorar el protocolo de ocultamiento, optimizar el tamaño del bloque, reorganizar los programas para incrementar la localidad de las referencias a memoria, etcétera.

No obstante, de forma eventual llegará el momento en que se haya utilizado cada truco del libro, pero los insaciables diseñadores querrán añadir más CPU y no habrá más ancho de banda en el bus. La única salida consiste en añadir más ancho de banda de bus. Un método consiste en cambiar la topología, pasando, por ejemplo, de un bus a dos o a un árbol o una cuadrícula. Al cambiar la topología de la red de interconexión, es posible añadir una capacidad de comunicación adicional.

Un método diferente consiste en construir el sistema como una jerarquía. Se continúan colocando algunos CPU en un bus, pero ahora se considera toda esto (CPU más el bus) como unidad. Se construye el sistema como varias unidades y se conectan éstas mediante

un bus entre ellas, como se muestra en la figura 6-6(a). Mientras más CPU se comuniquen principalmente dentro de su unidad, habrá relativamente menos tráfico entre las unidades. Si un bus entre las unidades es inadecuado, se añade un segundo bus de este tipo, o se ordenan las unidades en un árbol o cuadrícula. Si se necesita mayor ancho de banda, se reúne un bus, un árbol o una cuadrícula de unidades en una superunidad, y se separa al sistema en varias superunidades. Las superunidades se pueden conectar mediante un bus, árbol, o cuadrícula, y así en lo sucesivo. La figura 6-6(b) muestra un sistema con tres niveles de buses.

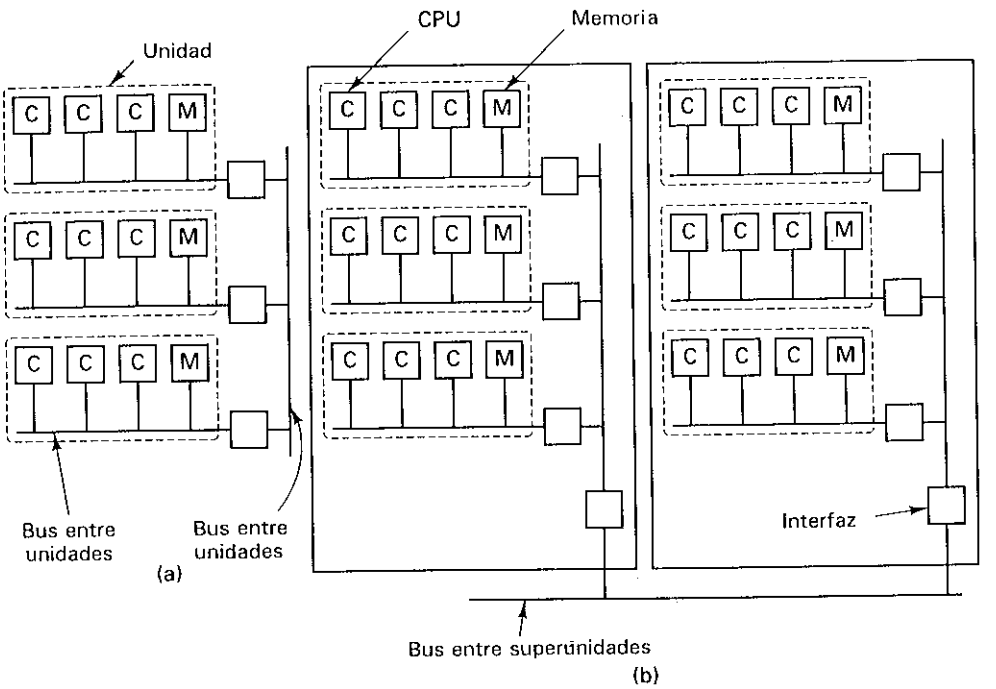


Figura 6-6. (a) Tres grupos conectados por un bus entre unidades para formar una superunidad. (b) Dos superunidades conectadas mediante un bus de superunidad.

En esta sección veremos un diseño jerárquico basado en una cuadrícula de unidades. La máquina, llamada **Dash**, fue construida como un proyecto de investigación en la Universidad de Stanford (Lenoski *et al.*, 1992). Aunque se han hecho muchas otras investigaciones con trabajos parecidos, éste es un ejemplo típico. En lo que resta de esta sección nos centraremos en el prototipo de 64 CPU que en realidad fue construido, aunque los principios de diseño fueron elegidos con cuidado para que se construya una versión más grande.

Hemos simplificado la siguiente descripción en algunos puntos para evitar los detalles innecesarios.

Un diagrama simplificado del prototipo de Dash aparece en la figura 6-7(a). Consta de 16 unidades, cada una de las cuales contiene un bus, cuatro CPU, 16M de memoria global y algo de equipo de E/S (discos, etc.). Para evitar confusión en la figura, hemos omitido el equipo de E/S y dos CPU de cada unidad. Cada CPU es capaz de husmear en su bus local, como en la figura 6-2(b), pero no en los demás buses.

El espacio total disponible de direcciones en el prototipo es 256M, dividido en 16 regiones de 16M cada una. La memoria global de la unidad 0 contiene las direcciones 0 a 16M. La memoria global de la unidad 1 contiene las direcciones 16M a 32M, etcétera. La memoria es oculta y transferida en bloques de 16 bytes, de modo que cada unidad tiene bloques de memoria de 1M dentro de su espacio de direcciones.

Directorios

Cada unidad tiene un **directorio** con un registro de las unidades que tienen copias de sus bloques. Como cada unidad tiene 1M bloques de memoria, tiene 1M entradas en su directorio, una por bloque. Cada entrada contiene un mapa de bits, con un bit por unidad, que dice si esa unidad tiene el bloque oculto. La entrada también tiene un campo de 2 bits que indica el estado del bloque. Como veremos, los directorios son esenciales en la operación de Dash. De hecho, el nombre Dash proviene de Directory Architecture for Shared memory (Arquitectura de directorios para memoria compartida).

Con 1M entradas de 18 bits cada una, el tamaño total de cada directorio es mayor de 2M bytes. Con 16 unidades, la memoria de directorio total está justamente sobre los 36M, o cerca del 14% de los 256M. Si el número de CPU por unidad se incrementa, la cantidad de memoria de directorio no cambia. De modo que el hecho de tener más CPU por unidad permite que el costo del directorio se amortice entre gran cantidad de CPU, lo que reduce el costo por CPU. También se reducen el costo del directorio y los controladores del bus por CPU. En teoría, el diseño trabaja bien con un CPU por unidad, pero el costo del directorio y el hardware del bus por CPU se incrementa.

Un mapa de bits no es la única forma de mantener un registro de la unidad que contiene a cada bloque de caché. Otro método consiste en organizar cada entrada de directorio como una lista explícita que indique las unidades que contienen al bloque de caché correspondiente. Si se comparte poco, el método de la lista requerirá menos bits, pero si se comparte mucho, se requerirán más bits. Las listas también tienen la desventaja de ser estructuras de datos con longitud variable, pero estos problemas pueden resolverse. El multiprocesador Alewif de M.I.T (Agarwal *et al.*, 1991; y Kranz *et al.*, 1993), por ejemplo, es similar a Dash en muchos aspectos, aunque utiliza listas en vez de mapas de bits es sus directorios y maneja los directorios en software.

Cada unidad en Dash se conecta a una interfaz que le permite comunicarse con otras unidades. Las interfaces se conectan mediante enlaces entre las unidades (buses primitivos) en una cuadrícula rectangular, como se muestra en la figura 6-7(a). Al agregar más unidades

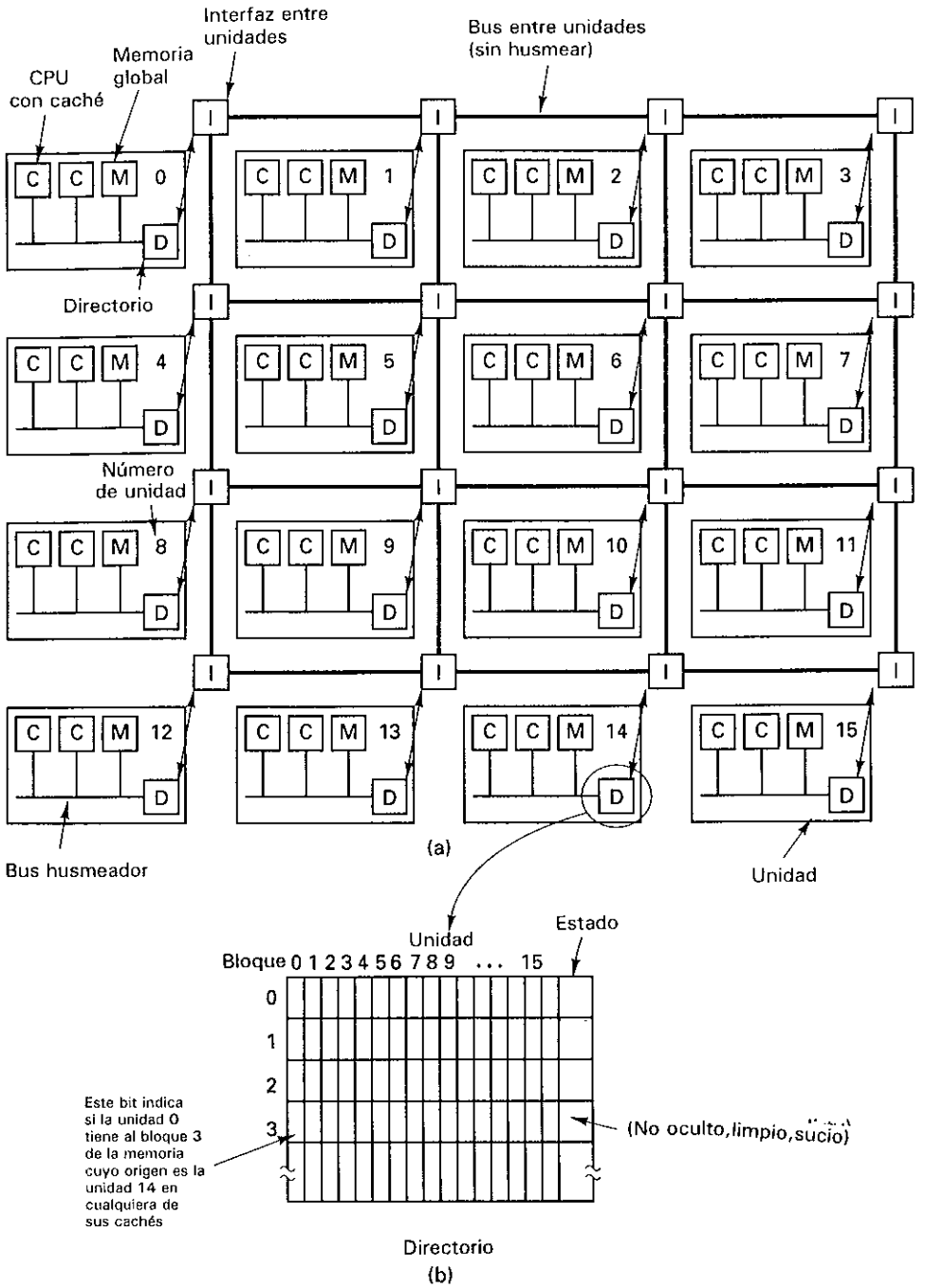


Figura 6-7. (a) Una vista simplificada de la arquitectura Dash. Cada unidad tiene en realidad cuatro CPU, pero aquí sólo mostramos dos. (b) Un directorio Dash.

al sistema, también se añaden más enlaces entre las unidades, de modo que el ancho de bandas y el sistema crecen. El sistema de enlaces entre las unidades utiliza el **ruteo de seguimiento de túneles,** lo que significa que la primera parte de un paquete puede adelantarse aún antes de recibir todo el paquete, lo que reduce la espera en cada salto. Aunque no se muestre en la figura, existen en realidad dos conjuntos de enlaces entre las unidades, uno para los paquetes de solicitud y otro para los de respuesta. Los enlaces entre las unidades no pueden ser husmeados.

Ocultamiento

El ocultamiento se hace en dos niveles: un caché de primer nivel y un caché mayor de segundo nivel. El caché de primer nivel es un subconjunto del caché de segundo nivel, por lo que sólo analizaremos este último. Cada caché (de segundo nivel) monitorea el bus local utilizando un protocolo similar al protocolo de membresía de caché de la figura 6-4.

Cada bloque de caché puede estar en alguno de los siguientes estados:

1. NO OCULTO : La única copia del bloque está en esta memoria.
2. LIMPIO: La memoria está actualizada; el bloque puede encontrarse en varios cachés.
3. SUCIO : La memoria es incorrecta; solo un caché contiene al bloque.

El estado de cada bloque de caché se guarda en el campo *Estado* de su entrada de directorio, como se muestra en la figura 6-7(b).

Protocolos

El protocolo Dash se basa en la membresía y la invalidación. En cada momento, cada bloque de caché tiene un propietario. Para un bloque NO OCULTO o LIMPIO la unidad de origen del bloque es el propietario. Para un bloque SUCIO, la unidad que contiene la única copia es el propietario. Para escribir en un bloque LIMPIO, primero hay que encontrar e invalidar todas las copias existentes. Aquí es donde entran los directorios.

Para ver cómo trabaja este mecanismo, consideremos primero como lee un CPU una palabra en memoria. Primero verifica sus propios cachés. Si ningún caché tiene la palabra, se hace una solicitud en el bus de la unidad local para ver si otro CPU de la unidad tiene el bloque con dicha palabra. Si uno la tiene, se ejecuta una transferencia del bloque entre los cachés, para colocar el bloque en el caché del CPU solicitante. Si el bloque está LIMPIO, se hace una copia; si está SUCIO, el directorio de origen es informado de que el bloque se encuentra ahora LIMPIO y compartido. De cualquier forma, un éxito de uno de los cachés satisface la instrucción pero no afecta ningún mapa de bits de directorio.

Si el bloque no está presente en los cachés de las unidades, se envía un paquete de solicitud a la unidad de origen del bloque, lo que puede determinarse al examinar los 4 bits

superiores de la dirección de memoria. La unidad de origen bien podría ser la unidad del solicitante, en cuyo caso el mensaje no se envía de forma física. El hardware para administración de directorios en la unidad de origen examina sus tablas para ver el estado que guarda el bloque. Si se encuentra NO OCULTO o LIMPIO, el hardware toma el bloque de su memoria global y lo envía de regreso a la unidad solicitante. Entonces actualiza su directorio, marchando el bloque como oculto en la unidad solicitante (si no ha sido aún marcado).

Sin embargo, si el bloque necesitado se encuentra SUCIO, el hardware del directorio observa la identidad de la unidad que contiene al bloque y transmite la solicitud hacia ese punto. La unidad que contiene el bloque SUCIO lo envía entonces a la unidad solicitante y marca como LIMPIO su copia, pues ahora está compartida. También envía una copia de regreso a la unidad de origen para que pueda actualizar la memoria y se cambie el estado del bloque a LIMPIO. Todos estos casos se resumen en la figura 6-8(a). Si un bloque está marcado como si estuviera en un nuevo estado, esto se modifica en el directorio de origen, y es este directorio el que lleva un registro del estado.

La escritura trabaja diferente. Antes de que se lleve a cabo una escritura, el CPU que realiza la escritura, debe asegurarse de ser el propietario de la única copia del bloque de caché en el sistema. Si ya tiene el bloque en su caché y éste se encuentra SUCIO, puede realizar la escritura de inmediato. Si tiene el bloque LIMPIO, se envía primero un paquete a la unidad de origen solicitando localizar e invalidar las demás copias.

Si el CPU solicitante no tiene el bloque caché, hace una solicitud en el bus local para ver si cualquiera de sus vecinos la tiene. En tal caso, se hace una transferencia entre los cachés (o de memoria a caché). Si el bloque está LIMPIO, las demás copias, si existen, la unidad de origen debe invalidarlas.

Si la transmisión local no puede encontrar una copia y el bloque está albergado en algún otro lado, se envía un paquete a la unidad de origen. Se pueden distinguir aquí tres casos. Si el bloque se encuentra NO OCULTO, se marca como SUCIO y se envía al solicitante. Si está LIMPIO, se invalidan todas las copias y se sigue entonces el procedimiento para NO OCULTO. Si está SUCIO, la solicitud es transmitida a la unidad remota que actualmente es propietaria del bloque (si es necesario). Esta unidad invalida su copia y satisface la solicitud. Los diversos casos se muestran en la figura 6-8(b).

Obviamente, el mantenimiento de la consistencia de la memoria en Dash (o en cualquier multiprocesador de gran tamaño) no es como el sencillo modelo de la figura 6-1(b). El acceso a una memoria puede requerir el envío de un gran número de paquetes. Además, para mantener consistente la memoria, lo usual es que el acceso no termine hasta que todos los paquetes hayan sido reconocidos, lo que puede tener un efecto serio en el desempeño. Para evitar estos problemas, Dash utiliza varias técnicas especiales, como dos conjuntos de enlaces entre las unidades, escrituras entubadas y semánticas de memoria diferentes de lo que uno podría esperar. Analizaremos algunos de estos temas más tarde. Por el momento, la moraleja es que esta implantación de "memoria compartida" requiere una base de datos grande (los directorios), una cantidad considerable de poder de cómputo (el hardware para administración de directorios) y un número poten-

Posición donde se encontraba el bloque

Estado del bloque	Caché de R	Caché vecino	Memoria de la unidad de origen	Caché de alguna unidad
NO OCULTO			Enviar bloque a R; marcarlo como LIMPIO y ocultarlo sólo en la unidad de R	
LIMPIO	Utilizar bloque	Copiar bloque al caché de R	Copiar el bloque de la memoria a R; marcarlo como oculto también en la unidad de R	
SUCIO	Utilizar bloque	Enviar el bloque a R y a la unidad de origen; indicar al origen que lo marque como LIMPIO y ocultarlo en la unidad de R		Enviar el bloque a R y a la unidad de origen (si está OCULTO en alguna parte); indicar al origen que lo marque como LIMPIO y también ocultarlo en la unidad de R

(a)

Posición donde se encontraba el bloque

Estado del bloque	Caché de R	Caché vecino	Memoria de la unidad de origen	Caché de alguna unidad
NO OCULTO			Enviar bloque a R; marcarlo como SUCIO y ocultarlo en la unidad R	
LIMPIO	Enviar el mensaje al origen solicitando la propiedad exclusiva en el estado SUCIO; si se otorga, utilizar bloque	Copiar e invalidar bloque; enviar mensaje al origen solicitando la propiedad exclusiva en estado SUCIO	Enviar bloque a R; invalidar todas las copias ocultas; marcarlo como SUCIO y ocultarlo solamente en la unidad de R	
SUCIO	Utilizar bloque	Transferencia entre cachés a R; invalidar la copia vecina		Enviar el bloque directamente a R; invalidar la copia oculta; el origen lo marca como SUCIO y lo oculta solamente en la unidad de R

(b)

Figura 6-8. Protocolos Dash. Las columnas muestran dónde se encontraban los bloques. Los renglones muestran su estado anterior. Los contenidos de los cuadros muestran la acción tomada. *R* se refiere al CPU solicitante. Un cuadro vacío indica una situación imposible. (a) Lecturas. (b) Escrituras.

cialmente grande de paquetes por enviar y reconocer. Veremos más adelante que la implantación de la memoria compartida distribuida tiene precisamente las mismas propiedades. La diferencia entre las dos descansa más en la técnica de implantación que en las ideas, la arquitectura o los algoritmos.

6.2.5. Multiprocesadores NUMA

Por ahora, queda muy claro que el ocultamiento del hardware en grandes multiprocesadores no es simple. El hardware y algunos intrincados protocolos deben mantener estructuras de datos complejas, como en la figura 6-8, integrados al controlador del caché o MMU.

La consecuencia inevitable es que los multiprocesadores grandes son caros y no tienen uso muy amplio.

Sin embargo, los investigadores han invertido mucho esfuerzo buscando alternativas de diseño que no requieran de elaborados esquemas de ocultamiento. Una de estas arquitecturas es el procesador **NUMA (acceso no uniforme a memoria)**. Como un multiprocesador **UMA (acceso uniforme a memoria)** tradicional, una máquina NUMA tiene un espacio de direcciones virtuales visible para todos los CPU. Cuando cualquier CPU escribe un valor en la localidad *a*, una lectura posterior por un procesador diferente regresará el valor recién escrito.

La diferencia entre las máquinas UMA y NUMA no sólo descansa en la semántica sino también en el desempeño. En una máquina NUMA, el acceso a una memoria remota es mucho más lento que el acceso a una local, y no se intenta ocultar este hecho mediante un ocultamiento en hardware. La razón entre un acceso remoto y uno local es por lo general 10:1, donde no es raro un factor de variación igual a 2 en ambos sentidos. Así, un CPU ejecuta de forma directa un programa que resida en una memoria remota, pero el programa puede ejecutarse en orden de magnitud más lento que si estuviera en la memoria local.

Ejemplos de multiprocesadores NUMA

Para aclarar el concepto de máquina NUMA, considere el ejemplo de la figura 6-9(a), Cm*, la primera máquina NUMA (Jones *et al.*, 1977). La máquina estaba formada por varias unidades, cada una con un CPU, un MMU microprogramable, un módulo de memoria y posiblemente algunos dispositivos de E/S, todos conectados mediante un bus. No había cachés, y no se husmeaba el bus. Las unidades fueron conectadas mediante buses entre las unidades, uno de los cuales se muestra en la figura.

Cuando un CPU hacía una llamada a memoria, la solicitud llegaba al MMU del CPU, que examinaba los bits superiores de la dirección para ver cuál memoria se necesitaba. Si la dirección era local, el MMU sólo hacía la solicitud en el bus local. Si correspondía a una memoria distante, el MMU construía un paquete de solicitud con la dirección (y para una escritura, la palabra dato por escribir), y lo enviaba a la unidad destino mediante un bus entre las unidades. Al recibir el paquete, el MMU destino llevaba a cabo la operación y regresaba la palabra (para una lectura) o un reconocimiento (para una escritura). Aunque era posible ejecutar un CPU por completo desde una memoria remota, el envío de un paquete por cada palabra leída y cada palabra escrita reducía la velocidad de la operación en un orden de magnitud.

La figura 6-9(b) muestra otra máquina NUMA, la BBN Butterfly. En este diseño, cada CPU se acopla de forma directa a una memoria. Cada uno de los pequeños cuadrados de la figura 6-9(b) representa un CPU más un par de memoria. Los CPU del lado derecho de la figura son los mismos que los de la izquierda. Los CPU se conectan por medio de ocho conmutadores, cada uno con cuatro puertos de entrada y cuatro de salida. Las solicitudes de memoria local se manejan de forma directa; las solicitudes remotas se cambian por

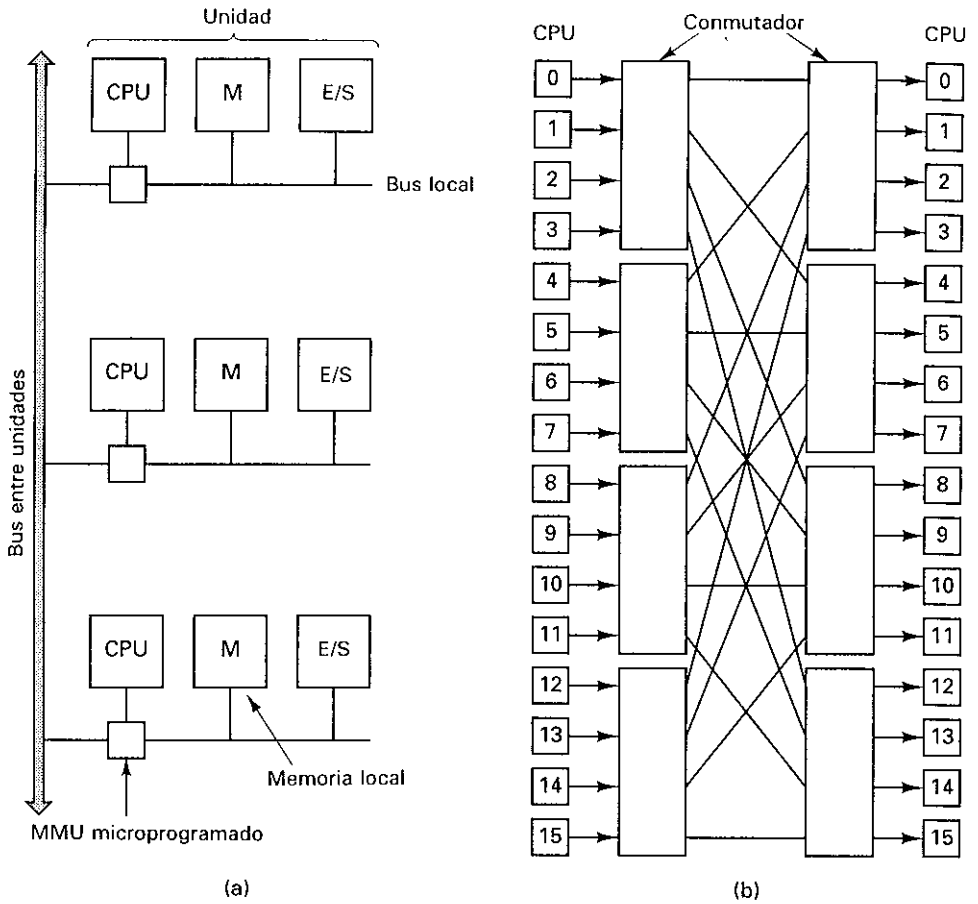


Figura 6-9. (a) Una vista simplificada del sistema Cm*. (b) BBN Butterfly. Los CPU de la derecha son los mismos que los de la izquierda (es decir, la arquitectura es en realidad un cilindro).

paquetes de solicitudes y se envían a la memoria apropiada por medio de la red de conmutación. Aquí también los programas se pueden ejecutar de manera remota, pero con una falla tremenda en el desempeño.

Aunque ninguno de estos ejemplos tiene memoria global, las máquinas NUMA pueden equiparse con una memoria no conectada con otros CPU. Por ejemplo, Bolosky *et al.*, (1989) describieron una máquina NUMA basada en un bus con una memoria global que no pertenece a otros CPU pero permite el acceso de todos ellos (además de las memorias locales).

Propiedades de los multiprocesadores NUMA

Las máquinas NUMA tienen tres propiedades claves que nos interesan:

1. Es posible el acceso a la memoria remota.
2. El acceso a la memoria remota es más lento que el de la memoria local.
3. El tiempo del acceso remoto no se oculta mediante el uso de los cachés.

Los primeros dos puntos se explican por sí mismos. El tercero exige una aclaración. En Dash y otros multiprocesadores modernos UMA, el acceso remoto suele ser más lento que el local. Lo que hace tolerable esta propiedad es la presencia del ocultamiento. Cuando se toca una palabra remota, se trae un bloque de memoria alrededor de ella hacia el caché del procesador solicitante, de modo que las llamadas posteriores tengan gran velocidad. Aunque haya un ligero retraso para manejar la falta de caché, terminar con la memoria remota puede ser un poco más caro que terminar con la memoria local. La consecuencia de esta observación es que no importa mucho que las páginas vivan en tal o cual memoria: el hardware mueve de forma automática el código y los datos adonde se necesiten (aunque una mala elección de la unidad de origen por cada página en Dash añade cierto costo).

Las máquinas NUMA no tienen esta propiedad, de modo que sí importa cuál página esté localizada en cuál memoria (es decir, en cuál máquina). El problema clave en el software de NUMA es la decisión de dónde colocar cada página para maximizar el desempeño. Más adelante resumiremos de forma breve algunas ideas de LaRowe y Ellis (1991). Otro trabajo es descrito en (Cox y Fowler, 1989; LaRowe *et al.*, 1991; y Ramanathan y Ni, 1991).

Cuando se inicia un programa en una máquina NUMA, las páginas pueden estar o no colocadas de antemano manualmente en ciertas máquinas procesadoras (sus procesadores de origen). En todo caso, cuando un CPU tiene acceso a una página que no se encuentra asociada en ese momento en su espacio de direcciones, ocurre un fallo de página. El sistema operativo nota este fallo y toma una decisión. Si la página es exclusiva para lectura, la opción es duplicar la página (es decir, crear una copia local sin perturbar la original) o asociar la página virtual en la memoria remota, lo que obliga entonces a un acceso remoto para todas las direcciones de esa página. Si la página puede utilizarse para lectura y escritura, la opción consiste en mover la página al procesador que falló (invalidando la página original) o asociar la página virtual en la memoria remota.

Las consecuencias de esto son sencillas. Si se creó una copia local (réplica o migración) y la página no se vuelve a utilizar demasiado, habrá un gasto considerable de tiempo perdido por nada. Por otro lado, si no se realiza ninguna copia, la página se asocia de manera remota, y vendrán muchos accesos, que serán muy lentos. En esencia, el sistema operativo tiene que ver si la página será utilizada con frecuencia en el futuro. Si falla, será una falla en el desempeño.

Sin importar la decisión que tome, la página será asociada, de manera local o remota, y la instrucción fallida se reestablece. Las llamadas posteriores a esa página se hacen en

hardware, sin la intervención del software. Si no se toma otra decisión, una vez tomada una mala decisión no podrá ser invertida.

Algoritmos NUMA

Para permitir la corrección de errores y la adaptación del sistema a los cambios en los patrones de referencia, los sistemas NUMA tienen por lo general un proceso demonio, el **digitalizador de páginas**, que se ejecuta en modo secundario. De manera periódica (por ejemplo, cada 4 segundos), el digitalizador de páginas reúne las estadísticas de uso acerca de las referencias locales y remotas, que se mantienen con ayuda del hardware. Cada n ejecuciones, el digitalizador de páginas evalúa las decisiones anteriores para copiar páginas o asociarlas a memorias remotas. Si las estadísticas de uso indican que una página está en el lugar incorrecto, el digitalizador elimina la asociación de la página para que la siguiente referencia cause un fallo de página, lo cual permite que se tome una nueva decisión de colocación. Si una página se mueve con frecuencia en un corto intervalo, el digitalizador puede marcar cada página como **congelada**, lo que inhibe los movimientos posteriores hasta que suceda algún evento dado (por ejemplo, algún número de segundos que hayan transcurrido).

Se han propuesto muchas estrategias para las máquinas NUMA, que difieren en el algoritmo utilizado por el digitalizador para invalidar las páginas y el algoritmo utilizado para tomar decisiones de colocación después de un fallo de página. Un algoritmo de digitalización posible consiste en invalidar cualquier página para la que hayan más referencias remotas que locales. Una prueba más fuerte consiste en invalidar una página sólo si el contador de referencias remotas ha sido mayor que el local durante las últimas k ejecuciones del digitalizador. Otras posibilidades consisten en descongelar las páginas congeladas después de transcurrir t segundos o si las referencias remotas exceden a las locales en alguna cantidad o durante cierto intervalo de tiempo.

Cuando ocurre un fallo de página, son posibles varios algoritmos, algunos de los cuales siempre incluyen réplica/migración y otros nunca incluyen réplica/migración. Un algoritmo muy sofisticado consiste en replicar o migrar a menos que la página esté congelada. Los patrones de uso reciente también puede tomarse en cuenta, al igual que el hecho de que la página esté o no esté en su máquina "de origen".

LaRowe y Ellis (1991) han comparado gran número de algoritmos y concluido que no existe una política que sea la mejor. La arquitectura de la máquina, el tamaño de la falla para un acceso remoto y el patrón de referencia del programa en cuestión juegan un papel importante para determinar el mejor algoritmo.

6.2.6. Comparación de los sistemas con memoria compartida

Los sistemas de memoria compartida cubren un espectro amplio, desde los sistemas que mantienen la consistencia por completo en hardware hasta aquellos que lo hacen en software. Hemos estudiado con detalle el extremo del hardware del espectro y hemos dado

un resumen breve del extremo del software (memoria compartida distribuida basada en páginas y memoria compartida distribuida basada en objetos). En la figura 6-10 el espectro aparece de manera explícita.

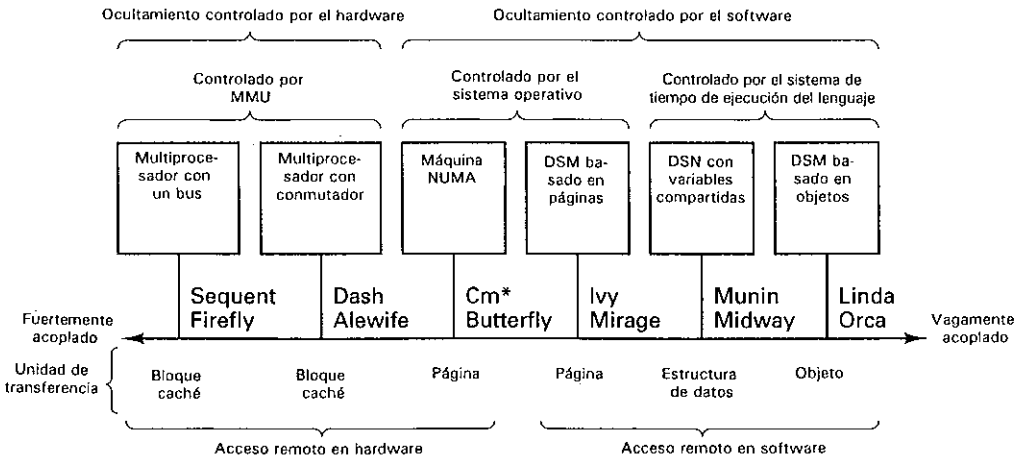


Figura 6-10. El espectro de máquinas de memoria compartida.

Del lado izquierdo de la figura 6-10 tenemos los multiprocesadores con un bus, con cachés en hardware y que mantienen la consistencia al husmear en el bus. Éstas son las máquinas de memoria compartida más sencillas y operan por completo en hardware. Muchas máquinas creadas por Sequent y otros vendedores y la estación de trabajo experimental DEC Firefly (cinco VAX en un bus común) caen dentro de esta categoría. Este diseño trabaja bien para un número pequeño o mediano de CPU, pero se degrada con rapidez al saturarse el bus.

Después vienen los multiprocesadores conmutadores, como es la máquina Stanford Dash y la máquina Alewife de M.I.T. Éstas también tienen ocultamiento de hardware pero utilizan directorios y otras estructuras de datos para llevar un registro de los CPU o unidades que tienen ciertos bloques de caché. Se utilizan algoritmos complejos para mantener la consistencia, pero como estos se guardan principalmente en el microcódigo MMU (donde las excepciones potenciales son manejadas en software), cuentan a lo más como implantaciones de "hardware".

Después vienen las máquinas NUMA. Estas son híbridas entre el control del hardware o del software. Como en los multiprocesadores, cada CPU NUMA tiene acceso a cada palabra del espacio de direcciones virtuales común sólo al leer o escribir en él. Sin embargo, a diferencia de un multiprocesador, el ocultamiento (es decir, la colocación de la página y la migración) queda controlado por el software (el sistema operativo) y no por el hardware (los MMU). Cm* (Jones *et al.*, 1977) y BBN Butterfly son ejemplos de las máquinas NUMA.

Continuando por el espectro, llegamos a las máquinas que ejecutan un sistema de memoria compartida distribuida basado en páginas como IVY (Li, 1986) y Mirage (Fleisch y Popek, 1989). Cada uno de los CPU en este sistema tiene su memoria privada y, a diferencia de las máquinas NUMA y los multiprocesadores UMA, no pueden hacer llamadas a memoria remota de forma directa. Cuando un CPU hace referencia a una palabra en el espacio de direcciones que tiene un respaldo de página localizado en la actualidad en una máquina diferente, ocurre un señalamiento al sistema operativo y la página solicitada debe ser buscada por el software. El sistema operativo adquiere la página necesaria al enviar un mensaje a la máquina donde reside en la actualidad y solicitarla. Así, tanto la colocación como el acceso se hacen en software.

Después llegamos a las máquinas que comparten sólo una porción selecta de sus espacios de direcciones, como las variables compartidas y otras estructuras de datos. Los sistemas Munin (Bennett *et al.*, 1990) y Midway (Bershad *et al.*, 1990) trabajan de esta forma. Se requiere información adicional del usuario para determinar cuáles variables son compartidas y cuáles no. En estos sistemas, el enfoque cambia de pretender que existe una memoria común, a cómo mantener consistente un conjunto de estructuras de datos distribuidas duplicadas frente a las actualizaciones, potencialmente desde todas las máquinas que utilizan los datos compartidos. En algunos casos, el hardware de paginación detecta las escrituras, lo que puede ayudar a mantener de manera eficiente la consistencia. En otros casos, el hardware de paginación no se utiliza para el manejo de consistencia.

Por último, tenemos los sistemas que trabajan con memoria compartida distribuida basada en objetos. A diferencia de los demás, aquí los programas no pueden sólo tener acceso a los datos compartidos. Tienen que recorrer métodos protegidos, lo que significa que el sistema de ejecución puede tener siempre el control de cada acceso para ayudar a mantener la consistencia. Todo se hace aquí en software, sin ningún tipo de soporte de hardware. Orca (Bal, 1991) es un ejemplo de este diseño, y Linda (Carriero y Gelernter, 1989) es similar a ésta en algunos aspectos importantes.

Las diferencias entre estos seis tipos de sistemas se resumen en la figura 6-11, que muestra desde el hardware fuertemente acoplado a la izquierda hasta el software vagamente acoplado a la derecha. Los primeros cuatro tipos ofrecen un modelo de memoria que consta de un espacio de direcciones virtuales lineal, estándar, paginado. Los primeros dos son multiprocesadores regulares y los siguientes dos hacen lo mejor para simularlos. Puesto que los cuatro primeros tipos actúan como multiprocesadores, las únicas operaciones posibles son la lectura y la escritura de palabras de memoria. En la quinta columna, las variables compartidas son especiales, pero aún el acceso es sólo mediante lecturas y escrituras normales. Los sistemas basados en objetos, con sus datos y métodos encapsulados, pueden ofrecer más operaciones generales y representar un nivel de abstracción más alto que la memoria en bruto.

La diferencia real entre los multiprocesadores y los sistemas DSM es si se puede tener acceso a los datos remotos sólo mediante la referencia a sus direcciones. En todos los multiprocesadores, la respuesta es sí. En los sistemas DSM es no: se necesita siempre la intervención del software. De manera análoga, la existencia de memoria global sin enlaces,

	Multiprocesadores			DSM		
Punto	Un bus	Con conmutador	NUMA	Basado en páginas	Con variables compartidas	Basado en objetos
¿Tiene un espacio de direcciones virtuales, compartido y lineal?	Sí	Sí	Sí	Sí	No	No
Operaciones posibles	L/E	L/E	L/E	L/E	L/E	General
¿Encapsulado y métodos?	No	No	No	No	No	Sí
¿Es posible el acceso remoto en hardware?	Sí	Sí	Sí	No	No	No
¿Es posible una memoria no conectada?	Sí	Sí	Sí	No	No	No
¿Quién convierte los accesos a memoria remota en mensajes?	MMU	MMU	MMU	Sistema operativo	Sistema de tiempo de ejec.	Sistema de tiempo de ejec.
Medio de transferencia	Bus	Bus	Bus	Red	Red	Red
La migración de datos es realizada mediante	Hardware	Hardware	Software	Software	Software	Software
Unidad de transferencia	Bloque	Bloque	Página	Página	Variable compartida	Objeto

Figura 6-11. Comparación de seis tipos de sistemas de memoria compartida.

es decir, una memoria no asociada con un CPU en particular, es posible en los multiprocesadores pero no en los sistemas DSM (ya que estos últimos son conjuntos de computadoras separadas conectadas mediante una red).

En los multiprocesadores, cuando se detecta un acceso remoto, se envía un mensaje a la memoria remota mediante un controlador de caché o MMU. En los sistemas DSM, se envía por medio del sistema operativo o el sistema de tiempo de ejecución. El medio utilizado también es diferente, siendo un bus de alta velocidad (o conjunto de buses) para los multiprocesadores y una LAN convencional (por lo general) para los sistemas DSM (aunque a veces la diferencia entre un "bus" y una "red" es discutible, principalmente en lo relativo a la cantidad de cables).

El punto siguiente se relaciona con la parte que realiza la migración de datos en caso necesario. Aquí, las máquinas NUMA son como los sistemas DSM: en ambos casos es el software, y no el hardware, el responsable de mover los datos entre las máquinas. Por último, la unidad para la transferencia de datos difiere en los seis sistemas: un bloque caché para los multiprocesadores UMA, una página para las máquinas NUMA y sistemas DSM basados en página, y una variable u objeto para los dos últimos.

6.3. MODELOS DE CONSISTENCIA

Aunque los multiprocesadores modernos tienen mucho en común con los sistemas de memoria compartida distribuida, es tiempo de dejar el tema de los multiprocesadores y continuar. En nuestra breve introducción a los sistemas DSM (Distributed Shared Memory, memoria compartida distribuida) al comienzo de éste capítulo, dijimos que tienen una o más copias de cada una de las páginas exclusivas para lectura y una copia de cada página para escritura. En la implantación más sencilla, cuando una máquina remota llama a una página para escritura, ocurre un señalamiento y la página es ocupada. Sin embargo, si algunas de las páginas que se pueden escribir está demasiado compartida, el hecho de tener solo una copia de cada una puede representar un serio cuello de botella en cuanto al desempeño.

Si se permite la existencia de varias copias, se facilita el problema del desempeño, pues basta con actualizar cualquier copia, pero esto introduce un nuevo problema: cómo mantener consistentes todas las copias. El mantenimiento de una consistencia perfecta es en especial doloroso cuando las diversas copias se encuentran en máquinas diferentes que sólo pueden comunicarse al enviar mensajes por una red lenta (comparada con las velocidades de la memoria). En algunos sistemas DSM (y multiprocesadores), la solución consiste en aceptar una menos perfecta como precio de mejor desempeño. El significado preciso de la consistencia y su relajación sin hacer insoportable la programación es un tema fundamental entre los investigadores de DSM.

Un **modelo de consistencia** es en esencia un contrato entre el software y la memoria (Adve y Hill, 1990). Dice que si el software acuerda obedecer ciertas reglas, la memoria promete trabajar de forma correcta. Si el software viola estas reglas, todo acaba y ya no se garantiza que la operación de memoria sea la correcta. Existe un espectro amplio de contratos, desde contratos que imponen sólo restricciones menores en el software hasta aquellos que hacen la programación normal casi imposible. Como probablemente usted ya ha adivinado, los contratos con restricciones menores no trabajan tan bien como los que tienen mayor restricción. Así es la vida. En esta sección estudiaremos varios modelos de consistencia utilizados en los sistemas DSM. Para mayor información, véase el artículo de Mosberger (1993).

6.3.1. Consistencia estricta

El modelo de consistencia más estricto es el de **consistencia estricta**. Se define mediante la siguiente condición:

> *Cualquier lectura a una localidad de memoria* x *regresa el valor guardado por la operación de escritura más reciente en* x.

Esta definición es natural y obvia, aunque supone de manera implícita la existencia de un tiempo global absoluto (como en la física Newtoniana) de modo que la determinación del "más reciente" no sea ambigua. Los uniprocesadores han observado tradicionalmente una

consistencia estricta y los programadores de los uniprocesos esperan que este comportamiento sea el normal. Un sistema donde el programa

```
a=1; a=2; print(a);
```

imprime 1 o cualquier valor distinto de 2 producirá con rapidez muchos programadores agitados [en este capítulo, *print* es un procedimiento que imprime su(s) parámetro(s)].

En un sistema DSM, el asunto es más complicado. Supongamos que x es una variable que está guardada sólo en la máquina B. Imagine que un proceso en la máquina A lee x en el momento T_1, lo que significa que se envía un mensaje a B para obtener x. Poco después, en T_2, un proceso en B realiza una escritura en x. Si hay una consistencia estricta, la lectura siempre debe regresar el valor anterior sin importar la posición de las máquinas y la cercanía de T_2 y T_1. Sin embargo, si T_2- T_1 es, digamos, 1 nanosegundo, y las máquinas se encuentran a una distancia de 3 metros entre sí, para que la solicitud de lectura de A a B se obtenga antes de la escritura, la señal debe viajar a 10 veces la velocidad de la luz, algo prohibido por la teoría de la relatividad de Einstein. ¿Será razonable que los programadores demanden la consistencia estricta del sistema, aunque esto requiera violar las leyes de la física?

Esto nos lleva al asunto del contrato entre el software y la memoria. Si el contrato promete la consistencia estricta de manera implícita o explícita, entonces lo mejor será que la memoria cumpla. Por otro lado, un programador que en realidad espera una consistencia estricta, de modo que sus programas fallen si no ocurre, vive en peligro. Aun en un multiprocesador pequeño, si un procesador comienza a escribir en la localidad de memoria a, y un nanosegundo más tarde otro procesador comienza a leer a, es probable que el lector obtenga el valor anterior de su caché local. Cualquiera que escriba programas que fallen bajo estas circunstancias debe quedarse después de clases a escribir un programa que imprima 100 veces: "Debo evitar las condiciones de competencia".

En un ejemplo más realista, uno puede imaginar un sistema que proporcione a los fanáticos del deporte los marcadores de los eventos deportivos en el mundo, actualizados al minuto (pero tal vez no al nanosegundo). En este caso, se podría aceptar una respuesta como si la solicitud se hubiera hecho 2 nanosegundos antes o después, en especial si esto proporciona mejor desempeño al guardar varias copias de los datos. En este caso, la consistencia estricta no se promete, ni se cumple ni se necesita.

Para estudiar la consistencia en detalle daremos varios ejemplos. Para precisar estos ejemplos, necesitamos una notación especial. En esta notación, se pueden mostrar varios procesos (P_1, P_2, etc.) a distintas alturas de la figura. Las operaciones realizadas por cada proceso se muestran en forma horizontal, de modo que el tiempo aumenta hacia la derecha. Las líneas rectas separan los procesos. Los símbolos

$W(x)a$ y $R(y)b$

significan que se han realizado una escritura a x con el valor a y una lectura desde y regresando b, respectivamente. Supondremos que el valor inicial de todas las variables en este tipo de diagramas a lo largo de este capítulo es 0. Como ejemplo, en la figura 6-12(a),

P_1 realiza una escritura a la localidad x, con el valor 1. Después, P_2 lee x y ve el 1. Este comportamiento es correcto para una memoria con consistencia estricta.

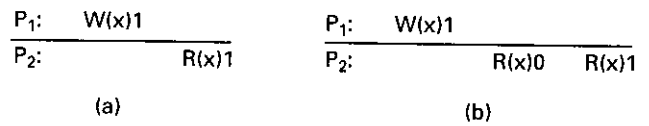


Figura 6-12. El comportamiento de dos procesos. El eje horizontal es el tiempo. (a) Memoria con consistencia estricta. (b) Memoria sin consistencia estricta.

En contraste, en la figura 6-12(b), P_2 realiza una lectura después de la escritura (posiblemente un nanosegundo después de ésta, pero aun así ocurre después de ésta), y obtiene 0. Una lectura posterior da 1. Este comportamiento es incorrecto para una memoria con consistencia estricta.

En resumen, cuando la memoria tiene consistencia estricta, todas las escrituras son visibles al instante a todos los procesos y se mantiene un orden de tiempo global absoluto. Si se cambia una localidad de memoria, todas las lecturas posteriores desde esa localidad ven el nuevo valor, sin importar qué tan pronto se haga la lectura después del cambio y sin importar los procesos que estén haciendo la lectura ni la posición de éstos. De manera análoga, si se realiza una lectura, se obtiene el valor actual, sin importar lo rápido que se realice la siguiente escritura.

6.3.2. Consistencia secuencial

Aunque la consistencia estricta es el modelo de programación ideal, es casi imposible implantarla en un sistema distribuido. Además, la experiencia muestra que los programadores suelen controlar bien los modelos más débiles. Por ejemplo, en todos los libros de texto sobre sistemas operativos se analizan las secciones críticas y el problema de exclusión mutua. Este análisis siempre incluye la advertencia de que los programas escritos en paralelo (como el problema de los productores y los consumidores) no debe establecer hipótesis acerca de las velocidades relativas de los procesos ni del intercalado de sus instrucciones en el tiempo. Si se cuenta con el hecho de que dos eventos dentro de un proceso suceden tan rápido que el otro proceso será incapaz de hacer algo, se están buscando problemas. En vez de esto, al lector se le enseña a programar de forma tal que no importe el orden exacto de ejecución de las proposiciones (de hecho, de las referencias a memoria). Cuando sea esencial el orden de los eventos, deben utilizarse semáforos u otras operaciones de sincronización. De hecho, aceptar este argumento significa aprender a vivir con modelos de memoria más débiles. Con algo de práctica, muchos programadores en paralelo podrán adaptarse.

La **consistencia secuencial** es un modelo de memoria un poco más débil que la consistencia estricta. Fue definida por primera vez por Lamport (1979), quien dijo que una memoria con consistencia secuencial es la que satisface la siguiente condición:

El resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesos fueran ejecutadas en algún orden secuencial, y las operaciones de cada proceso individual aparecen en esta secuencia en el orden especificado por su programa.

Lo que significa esta definición es que cuando los procesos se ejecutan en paralelo en diferentes máquinas (o aun en seudoparalelo en un sistema de tiempo compartido), cualquier intercalado válido es un comportamiento aceptable, pero *todos los procesos deben ver la misma serie de llamadas a memoria.* Una memoria donde un proceso (o procesos) ven un intercalado y otro proceso ve otro distinto no es una memoria con consistencia secuencial. Observe que no se habla del tiempo; es decir, no hay referencia alguna al almacenamiento "más reciente". Observe que en este contexto, un proceso "ve" las escrituras de todos los procesos pero sólo sus propias lecturas.

En la figura 6-13 se puede ver el hecho de que el tiempo no juega ningún papel. El comportamiento de una memoria que se muestra en la figura 6-13(a) cumple la consistencia secuencial, aunque la primera lectura realizada por P_2 regrese el valor inicial 0 en vez del valor nuevo 1.

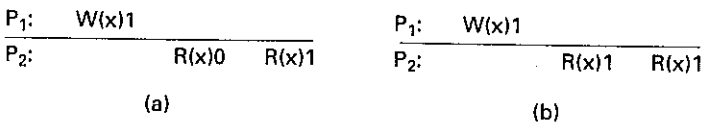


Figura 6-13. Dos resultados posibles al ejecutar el mismo programa.

La memoria con consistencia secuencial no garantiza que una lectura regrese el valor escrito por otro proceso un nanosegundo antes, un microsegundo antes, o incluso un minuto antes. Sólo garantiza que todos los procesos vean todas las referencias a memoria en el mismo orden. Si el programa que genera la figura 6-13(a) se ejecuta de nuevo, podría dar el resultado de la figura 6-13(b). Los resultados no son deterministas. Una nueva ejecución de un programa podría no proporcionar el mismo resultado, a menos que se utilicen operaciones explícitas de sincronización.

```
a = 1;          b = 1;          c = 1;
print (b, c);   print (a, c);   print (a, b);
```

(a) (b) (c)

Figura 6-14. Tres procesos paralelos.

Para aclarar este punto, consideremos el ejemplo de la figura 6-14 (Dubois *et al.*, 1988). Aquí vemos el código de tres procesos que se ejecutan en paralelo en tres procesadores distintos y que utilizan la misma memoria compartida distribuida con consistencia secuencial; todos tienen acceso a las variables *a, b* y *c*. Desde el punto de vista de referencia a la

memoria, una asignación se debe ver como una escritura, y un enunciado de impresión como una lectura simultánea de sus dos parámetros. Se supone que todos los enunciados son atómicos.

Son posibles varias secuencias intercaladas de ejecución. Con seis enunciados independientes, en principio existen 720 (6!) secuencias de ejecución posibles, aunque algunas de éstas violen el orden del programa. Consideremos las 120 (5!) secuencias que comiencen con $a = 1$. La mitad de éstas tienen *print*(a,c) antes de $b = 1$ y por lo tanto violan el orden de programación. La mitad también tiene *print*(a,b) antes que $c = 1$ y también violan el orden del programa. Solo 1/4 de las 120 secuencias (30) son válidas. También son posibles otras 30 secuencias válidas que comienzan con $b = 1$ y otras 30 que comienzan con $c = 1$, para un total de 90 secuencias de ejecución válidas. Cuatro de éstas se muestran en la figura 6-15.

(a)	(b)	(c)	(d)
a = 1;	a = 1;	b = 1;	b = 1;
print (b, c);	b = 1;	c = 1;	a = 1;
b = 1;	print (a, c);	print (a, b);	c = 1;
print (a, c);	print (b, c);	print (a, c);	print (a, c);
c = 1;	c = 1;	a = 1;	print (b, c);
print (a, c);1	print (a, b);	print (b, c);	print (a, b);
Imprime: 001011	Imprime: 101011	Imprime: 010111	Imprime: 111111
Firma: 00101	Firma: 101011	Firma: 110101	Firma: 111111

Figura 6-15. Cuatro secuencias de ejecución válidas para el programa de la figura 6-14. El eje vertical x es el tiempo, que aumenta hacia abajo.

En la figura 6-15(a), los tres procesos se ejecutan en orden, primero P_1, después P_2, y después P_3. Los otros tres ejemplos muestran otros intercalados de los enunciados en el tiempo, (diferentes, pero válidos). Cada uno de los tres procesos imprime dos variables. Como los únicos valores que puede asumir cada variable son el valor inicial (0) o el valor asignado (1), cada proceso produce una cadena de 2 bits. Los números después de *Imprime* son las salidas reales que aparecen en el dispositivo de salida.

Si concatenamos la salida de P_1, P_2 y P_3 en ese orden, obtenemos una cadena de 6 bits que caracteriza un intercalado particular de los enunciados (y por lo tanto de las referencias a memoria). Ésta es la cadena que aparece como *Firma* en la figura 6-15. Más adelante caracterizaremos cada orden por su señal en vez de por su salida.

No todos los 64 patrones de firmas están permitidos. Como ejemplo trivial, 000000 no está permitido, pues implicaría que los enunciados de impresión se ejecutaron antes que los de asignación, violando el requisito de Lamport en el sentido de que los enunciados se ejecuten según el orden del programa. Un ejemplo más sutil es 001001. Los primeros dos bits, 00, significan que b y c son ambos 0 cuando P_1 realizó su impresión. Esta situación sólo puede ocurrir cuando P_1 ejecuta ambos enunciados antes que comience P_2 o P_3. Los

siguientes dos bits, 10, indican que P_2 debe ejecutarse después de que P_1 ha comenzado pero antes de que P_3 comience. Los últimos dos bits, 01, indican que P_3 debe terminar antes de que comience P_1; pero ya hemos visto que P_1 debe ir primero. Por lo tanto, 001001 no está permitido.

En resumen, los 90 órdenes diferentes válidos para los enunciados producen varios resultados diferentes del programa (aunque menos de 64) que se permiten bajo la hipótesis de consistencia secuencial. En este caso, el contrato entre el software y la memoria consiste en que el software acepta todos ellos como válidos. En otras palabras, el software acepta los cuatro resultados que se muestran en la figura 6-15 y los demás resultados válidos como respuestas adecuadas, y trabajar de forma correcta si cualquiera de ellas ocurre. Un programa que funciona para algunos de estos resultados y no para otros viola el contrato con la memoria y es incorrecto.

Una memoria con consistencia secuencial se puede implantar en un sistema DSM o multiprocesador que duplique las páginas que se pueden escribir, garantizando que ninguna operación de memoria comienza hasta que las anteriores hayan concluido. Por ejemplo, en un sistema con un mecanismo de transmisión eficiente, confiable y por completo ordenado, todas las variables compartidas se podrían agrupar en una o más páginas y se podrían transmitir las operaciones a las páginas compartidas. No importa el orden exacto en que se intercalen las operaciones mientras todos los procesos estén de acuerdo en el orden de todas las operaciones en la memoria compartida.

Se han propuesto varios sistemas formales para expresar la consistencia secuencial (y otros modelos). Consideremos de forma breve el sistema de Ahamad *et al.* (1993). En su método, la serie de operaciones de lectura y escritura del proceso i se denotan por H_i (la historia de P_i). La figura 6-12(b) muestra dos de estas series, H_1 y H_2 para P_1 y P_2, respectivamente, como sigue:

$H_1 = W(x)1$

$H_2 = R(x)0\ R(x)1$

A el conjunto de todas estas series se le llama H.

Para obtener el orden relativo en que aparecen las operaciones por ejecutar, debemos fusionar las cadenas de operación en H en una cadena, S, donde cada operación que aparezca en H será una vez en S. De manera intuitiva, S da la orden de que las operaciones se hubieran realizado si se tuviera una memoria centralizada. Todos los valores válidos de S deben cumplir dos restricciones:

1. Debe mantenerse el orden de los programas.
2. Debe ser respetada la coherencia en la memoria.

La primera restricción significa que si un acceso para lectura o escritura, A, aparece antes que otro acceso, B, en una de las cadenas de H, A también debe aparecer antes que B en S. Si es cierta esta restricción para todos los pares de operaciones, la S resultante no mostrará operaciones en un orden que viole cualquiera de los programas.

La segunda restricción, llamada **coherencia de memoria**, significa que una lectura a alguna localidad, x, regresa siempre el valor más reciente escrito en x; es decir, el valor v escrito por el $W(x)v$ más reciente antes de $R(x)$. La coherencia de memoria examina cada localidad y la secuencia de las operaciones en ésta de manera aislada, sin contemplar las demás localidades. La consistencia, en contraste, trata de las escrituras en localidades *diferentes* y su ordenamiento.

Para la figura 6-12(b), sólo existe un valor válido de S:

$S = R(x)0\ W(x)1\ R(x)1$

Para otros ejemplos más complicados deben existir varios valores válidos de S. Se dice que el comportamiento de un programa es correcto si su secuencia de operaciones corresponde a algún valor válido de S.

Aunque la consistencia secuencial es un modelo amigable con el programador, tiene un problema serio de desempeño. Lipton y Sandberg (1988) demostraron que si el tiempo de lectura es r, el de escritura es w y el tiempo mínimo de transferencia de paquete entre los nodos es t, entonces siempre ocurre que $r+w \geq t$. En otras palabras, para cualquier memoria con consistencia secuencial, la modificación del protocolo para mejorar el desempeño de la lectura empeora el desempeño de la lectura, y viceversa. Por esta razón, los investigadores han estudiado otros modelos (más débiles). En las siguientes secciones, analizaremos algunos de ellos.

6.3.3. Consistencia causal

El modelo de **consistencia causal** (Hutto y Ahamad, 1990) representa un debilitamiento de la consistencia secuencial que hace una distinción entre los eventos potencialmente relacionados por causalidad y aquellos que no lo están.

Para ver a que se refiere la causalidad, consideremos un ejemplo de la vida diaria (para un científico de la computación). Durante una discusión de los méritos relativos de diferentes lenguajes de programación en uno de los grupos de noticias USENET, un fanático envió el mensaje: "Cualquier persona sorprendida programando en FORTRAN debe ser ejecutada". Tiempo después, un individuo sensato escribe: "Estoy en contra de la pena capital, aunque haya enorme ofensa contra el buen gusto". Debido a diversos retardos a lo largo de las rutas de propagación de mensajes, un tercer suscrito obtiene primero la respuesta y queda confundido al verla. El problema aquí es que la causalidad ha sido violada. Si el evento B es causado o influido por un evento anterior, A, la causalidad requiere que todos vean primero a A, y después vean a B.

Consideremos ahora un ejemplo de memoria. Suponga que el proceso P_1 escribe una variable x. Después P_2 lee x y escribe y. Aquí la lectura de x y la escritura de y están en potencial relacionadas de forma causal, pues el cálculo de y podría depender del valor de x leído por P_2 (es decir, el valor escrito por P_1). Por otro lado, si dos procesos escriben de forma espontánea y simultánea en dos variables, no están relacionados de forma causal. Cuando ocurre una lectura seguida por una escritura, los dos eventos están en potencia relacionados de forma causal. De manera análoga, una lectura está relacionada de forma

causal con la escritura que proporciona el dato obtenido por la lectura. Las operaciones que no están relacionadas de forma causal son **concurrentes**.

Para que una memoria sea consistente de forma causal, obedece las siguientes condiciones:

> *Las escrituras potencialmente relacionadas de forma causal son vistas por todos los procesos en el mismo orden. Las escrituras concurrentes pueden ser vistas en un orden diferente en máquinas diferentes.*

Como ejemplo de consistencia causal, considere la figura 6-16. Aquí tenemos una sucesión de eventos permitida con memoria consistente de forma causal, pero prohibida para una memoria con consistencia secuencial o con consistencia estricta. Lo que hay que observar es que las escrituras $W(x)2$ y $W(x)3$ son concurrentes, por lo que no se requiere que todos los procesos los vean en el mismo orden. Si el software falla cuando procesos diferentes ven eventos concurrentes en orden distinto, se ha violado el contrato de memoria ofrecido por la memoria causal.

P_1:	W(x)1			W(x)3		
P_2:		R(x)1	W(x)2			
P_3:		R(x)1			R(x)3	R(x)2
P_4:		R(x)1			R(x)2	R(x)3

Figura 6-16. Esta sucesión está permitida para una memoria con consistencia causal, pero no para una memoria con consistencia secuencial o una memoria con consistencia estricta.

Ahora considere un segundo ejemplo. En la figura 6-17(a) tenemos que $W(x)2$ potencialmente depende de $W(x)1$, ya que 2 puede ser un resultado de un cálculo que implique al valor leído por $R(x)1$. Las dos escrituras están relacionadas de forma causal, de modo que todos los procesos deben verlas en el mismo orden. Por lo tanto, la figura 6-17(a) es incorrecta. Por otro lado, en la figura 6-17(b), la lectura ha sido eliminada, de modo que $W(x)1$ y $W(x)2$ son ahora escrituras concurrentes. La memoria causal no necesita ordenar de forma global las escrituras concurrentes, por lo que la figura 6-17(b) es correcta.

La implantación de la consistencia causal mantiene un registro de cuáles procesos han visto y cuáles escrituras. Esto significa de hecho que debe construirse y mantenerse una gráfica de dependencia con las operaciones que dependen de otras. Hacer esto implica cierto costo.

6.3.4. Consistencia PRAM y consistencia del procesador

En la consistencia causal se permite que las escrituras concurrentes sean vistas en diferente orden en varias máquinas, aunque las relacionadas de forma causal deben verse en el mismo orden por todas las máquinas. El siguiente paso en el relajamiento de memoria

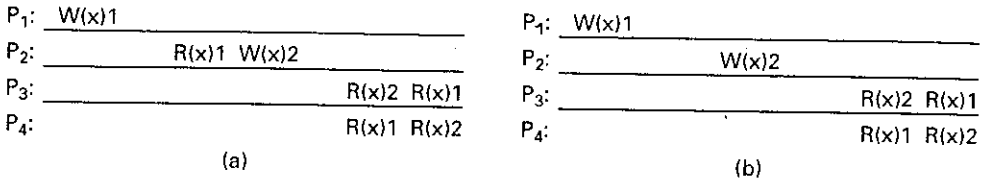


Figura 6-17. (a) Una violación de la memoria causal. (b) Una sucesión correcta de eventos en la memoria causal.

es eliminar este último requisito. Al hacer esto se tiene la **consistencia PRAM** (Pipelined RAM), sujeta a la condición:

Las escrituras realizadas por un proceso son recibidas por los otros procesos en el orden en que son realizadas, pero las escrituras de procesos diferentes pueden verse en un orden diferente por procesos diferentes.

La consistencia PRAM se debe a Lipton y Sandberg (1988). PRAM son las siglas del entubamiento del RAM ya que las escrituras realizadas por un proceso pueden entubarse; es decir, el proceso no tiene que quedarse esperando que termine cada una antes de comenzar la siguiente. Comparamos la consistencia PRAM con la consistencia causal en la figura 6-18. La sucesión de eventos que se muestran aquí está permitida para la memoria con consistencia PRAM pero no con los demás modelos más fuertes ya estudiados.

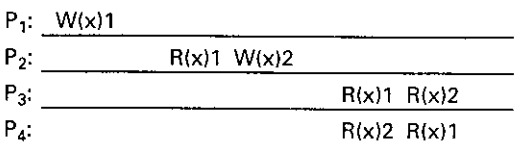


Figura 6-18. Una sucesión de eventos válida para la consistencia PRAM.

La consistencia PRAM es interesante pues es fácil de implantar. De hecho, dice que no existen garantías acerca del orden en que los diferentes procesos ven las escrituras, excepto que dos o más escrituras de una fuente llegan en orden, como si estuvieran en un entubamiento. Dicho en otras palabras, en este modelo, todas las escrituras generadas por procesos diferentes son concurrentes.

Ahora reconsideraremos los tres procesos de la figura 6-14, pero ahora con consistencia PRAM en vez de consistencia secuencial. Bajo la consistencia PRAM, los diferentes procesos podrían ver los enunciados ejecutados en un orden diferente. Por ejemplo, la figura 6-19(a) muestra como vería P_1 los eventos, mientras que la figura 6-19(b) muestra como lo vería P_2 y la figura 6-19(c) muestra la perspectiva de P_3. Para una memoria con consistencia secuencial, no se permitiría tres puntos de vista diferentes.

Si concatenamos la salida de los tres procesos, obtenemos como resultado 001001, lo cual, como ya hemos visto, es imposible con la consistencia secuencial. La diferencia fundamental entre la consistencia secuencial y la consistencia PRAM es que con la primera, aunque el orden de ejecución de los enunciados (y las referencias de memoria) no es de-

terminista, al menos todos los procesos coinciden en él. Con el segundo método, no coinciden. Los diversos procesos pueden ver las operaciones en un orden diferente.

```
  a = 1;
* print (b, c);
  b = 1;
  print (a, c);
  c = 1;
  print (a, b);

Imprime: 00
```

(a)

```
  a = 1;
  b = 1;
* print (a, c);
  print (b, c);
  c = 1;
  print (a, b);

Imprime: 10
```

(b)

```
  b = 1;
  print (a, c);
  c = 1;
* print (a, b);
  a = 1;
  print (b, c);

Imprime: 01
```

(b)

Figura 6-19. Ejecución de enunciados vista desde los tres procesos. Los enunciados señalados con asteriscos son los que realmente generan una salida.

A veces, la consistencia PRAM conduce a resultados poco intuitivos. El siguiente ejemplo, debido a Goodman (1989), fue diseñado para un modelo de memoria poco diferente (que se analiza más adelante), pero también es válido para la consistencia PRAM. En la figura 6-20, uno esperaría alguno de tres posibles resultados: se elimina a P_1, se elimina a P_2 o no se elimina ninguno (si las dos asignaciones se realizan en primer lugar). Sin embargo, con la consistencia PRAM, ambos procesos se pueden eliminar. Este resultado puede ocurrir si P_1 lee b antes de ver la asignación de b en P_2 y si P_2 lee a antes de la asignación de a en P_1. Con una memoria con consistencia secuencial, existen seis posibles intercalados de los enunciados y ninguno de ellos hace que ambos procesos se eliminen.

```
a = 1;
if (b == 0) kill (P2);
```

(a)

```
b = 1;
if (a == 0) kill (P1);
```

(b)

Figura 6-20. Dos procesos paralelos. (a) P_1. (b) P_2.

El modelo de Goodman (1989), llamado **consistencia del procesador**, es tan cercano a la consistencia PRAM que algunos autores las consideran iguales de hecho (por ejemplo, Attiya y Friedman, 1992; y Bitar, 1990). Sin embargo, Goodman dio un ejemplo que sugiere su intención de establecer una condición adicional sobre una memoria con consistencia de procesador, a saber, la coherencia de la memoria, como se describió antes: en otras palabras, para cada posición de memoria x existe un acuerdo local acerca del orden de las escrituras en x. Las escrituras en diferentes posiciones no tienen que ser vistas en el mismo orden por los diferentes procesos. Gharachorloo *et al.*, (1990) describen el uso de la consistencia del procesador en el multiprocesador Dash, pero utilizan una definición un poco diferente a la de Goodman. Las diferencias entre los modelos de consistencia PRAM y los dos modelos de consistencia del procesador son sutiles y analizadas por Ahamad *et al.*, (1993).

6.3.5. Consistencia débil

Aunque la consistencia PRAM y la del procesador proporcionan mejor desempeño que los modelos más fuertes, siguen siendo innecesariamente restrictivos para muchas aplicaciones, pues requieren que las escrituras generadas en un proceso sean vistas en todas partes en orden. No todas las aplicaciones requieren todas las escrituras y mucho menos en orden. Consideremos el caso de un proceso dentro de una sección crítica, donde se leen y se escribe en algunas variables en un ciclo. Aunque se supone que los demás procesos no tocan las variables hasta que el primer proceso salga de su sección crítica, la memoria no tiene forma de saber cuándo un proceso está en una sección crítica y cuándo no, de modo que debe propagar todas las escrituras a todas las memorias de la manera usual.

La mejor solución consiste en dejar que el proceso termine su sección crítica y garantizar entonces que los resultados finales se envíen a todas partes, sin preocuparse demasiado porque todos los resultados intermedios han sido propagados a todas las memorias en orden, o incluso si no fueron propagados. Esto se lleva a cabo mediante un nuevo tipo de variable, una **variable de sincronización**, que se utiliza con fines de sincronización. Las operaciones en ella se utilizan para sincronizar la memoria. Cuando termina una sincronización, todas las escrituras realizadas en esa máquina se propagan hacia afuera y todas las escrituras realizadas en otras máquinas son traídas hacia la máquina en cuestión. En otras palabras, toda la memoria (compartida) está sincronizada.

Dubois *et al.*, (1986) define este modelo, llamado **consistencia débil**, diciendo que tiene tres propiedades:

1. *Los accesos a las variables de sincronización son secuencialmente consistentes.*
2. *No se permite realizar un acceso a una variable de sincronización hasta que las escrituras anteriores hayan terminado en todas partes.*
3. *No se permite realizar un acceso a los datos (lectura o escritura) hasta realizar todos los accesos anteriores a las variables de sincronización.*

El punto 1 dice que todos los procesos ven los accesos a las variables de sincronización en el mismo orden. En efecto, cuando ocurre un acceso a una variable de sincronización, este hecho se transmite al mundo, y no se tiene acceso a las demás variables de sincronización hasta que éste se termine en todas partes.

El punto 2 dice que el acceso a una variable de sincronización "dirige el flujo". Obliga a terminar en todas partes las escrituras que están en progreso, terminadas o de forma parcial en algunas memorias pero no en otras. Cuando termina la variable de sincronización, también se garantiza que han terminado todas las escrituras anteriores. Al realizar una sincronización después de actualizar los datos compartidos, un proceso envía los nuevos valores a las demás memorias.

El punto 3 dice que cuando se tiene acceso a las variables ordinarias (es decir, que no son de sincronización), ya sea para lectura o escritura, se han realizado todas las sincroni-

zaciones anteriores. Al realizar una sincronización antes de leer los datos compartidos, un proceso puede estar seguro de obtener los valores más recientes.

Vale la pena mencionar que existe un poco de complejidad detrás de la palabra "realizados" aquí y en todas partes, en el contexto de DSM. Se dice que una lectura ha sido realizada cuando ninguna escritura posterior afecta el valor regresado. Se dice que una escritura ha sido realizada en el instante en que todas las lecturas posteriores regresan el valor escrito por la escritura. Se dice que una sincronización ha sido realizada cuando todas las variables compartidas han sido actualizadas. También se puede distinguir entre las operaciones que se realizan de manera local o global. Dubois *et al.*, (1988) tratan este punto con detalle.

Desde el punto de vista de la implantación, cuando el contrato entre el software y la memoria dice que sólo debe actualizarse cuando se realice un acceso a una variable de sincronización, una escritura nueva se inicia antes de terminar las anteriores y, en algunos casos, se evitan por completo las escrituras. Por supuesto, este contrato complica las cosas para el programador, pero en potencia se tendrá mejor desempeño. A diferencia de los modelos anteriores de memoria, éste apoya la consistencia en un grupo de operaciones, no en lecturas y escrituras individuales. Este modelo es más útil cuando no son comunes los accesos aislados a las variables compartidas, la mayoría de los cuales aparece por grupos (muchos accesos en un período breve y después ninguno durante un período largo).

```
int a, b,c, d, e, x, y;          /* variables */
int *p, *q;                      /* apuntadores */
int f(int *p, int *q);           /* prototipo de función */

a=x*x;                           /* a se guarda en un registro */
b=y*y                            /* b también */
c=a*a*a+b*b+a*b                  /* se utiliza posteriormente */
d=a*a*c;                         /* se utiliza posteriormente */
p=&a;                            /* p obtiene la dirección de a */
p=&b;                            /* q obtiene la dirección de b */
e=f(p, q);                       /* llamada a función */
```

Figura 6-21. Un fragmento de programa en donde algunas variables se pueden conservar en registros.

La idea de una memoria incorrecta no es nueva. Muchos compiladores también mienten. Por ejemplo, consideremos el fragmento de programa de la figura 6-21, donde todas las variables se inician con los valores adecuados. Un compilador con optimización puede decidir calcular *a* y *b* en registros y conservar los valores ahí durante cierto tiempo, sin actualizar sus posiciones en memoria. Sólo hasta que la función *f* se llama, el compilador debe colocar los valores actuales de *a* y *b* de regreso en la memoria, pues *f* podría intentar tener acceso a ellos.

El hecho de una memoria incorrecta es aceptable aquí, pues el compilador sabe lo que hace (es decir, pues el software no insiste que la memoria esté actualizada). Es claro que si existe un segundo proceso que lea la memoria sin restricciones, este esquema no funcio-

naría. Por ejemplo, si durante la asignación a *d*, el segundo proceso lee *a*, *b* y *c*, obtendría valores inconsistentes (los valores anteriores de *a* y *b*, pero el nuevo valor de *c*). Uno podría imaginar una forma especial para evitar el caos, de modo que el compilador escribiera primero en un bit particular para señalar que la memoria no está actualizada. Si otro proceso desea tener acceso a *a*, realizaría una espera ocupada de ese bit. De este modo, se podría vivir con una consistencia no tan perfecta, siempre que la sincronización fuera realizada en software y que todas las partes acataran las reglas.

Consideremos ahora una situación más compleja. En la figura 6-22(a) vemos que el proceso P_1 realiza dos escrituras en una variable ordinaria y después realiza una sincronización (indicada por la letra *S*). Si P_2 y P_3 no han sido sincronizados todavía, nada garantiza lo que verán, por lo que esta serie de eventos es válida.

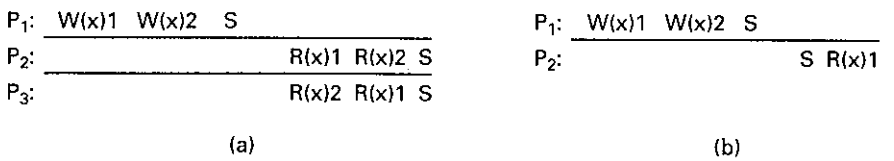


Figura 6-22. (a) Una secuencia válida de eventos para la consistencia de bit. (b) Una secuencia no válida para la consistencia débil.

La figura 6-22(b) es diferente. En este caso, se ha sincronizado P_2, lo que significa que su memoria se ha actualizado. Al leer *x*, debe obtener el valor 2. Con la consistencia débil, no puede obtener 1, como se muestra en la figura.

6.3.6. Consistencia de liberación

La consistencia débil tiene el problema de que, cuando se tiene acceso a una variable de sincronización, la memoria no sabe si esto se realiza debido a que el proceso ha terminado de escribir en las variables compartidas o está a punto de iniciar su lectura. En consecuencia, debe realizar las acciones necesarias en ambos casos, a saber, garantizar que todas las escrituras iniciadas localmente han sido terminadas (es decir, propagadas a las demás máquinas), así como recoger todas las escrituras de las demás máquinas. Si la memoria establece la diferencia entre la entrada a una región crítica y salir de ella, sería posible una implantación más eficiente. Para proporcionar esta información, se necesitan dos tipos de variables u operaciones de sincronización.

La **consistencia de liberación** (Gharachorloo *et al.*, 1990) proporciona estos dos tipos. Los accesos de **adquisición** indican a la memoria del sistema que está a punto de entrar a una región crítica. Los accesos de **liberación** dice que acaba de salir de una región crítica. Estos accesos se implantan como operaciones ordinarias sobre variables o como operaciones especiales. En cualquier caso, el programador es responsable por colocar un código explícito en el programa para indicar el momento de realizarlos; por ejemplo, llamando a

procedimientos de biblioteca como *acquire* y *release* o procedimientos como *enter_critical_region* o *leave_critical_region*.

También es posible utilizar barreras en vez de las regiones críticas con la consistencia de liberación. Una **barrera** es un mecanismo de sincronización que evita que cualquier proceso inicie la fase $n + 1$ de un programa hasta que todos los procesos terminen la fase n. Cuando un proceso llega a una barrera, debe esperar que todos los demás procesos lleguen ahí también. Cuando llega el último, todas las variables compartidas se sincronizan y continúan entonces todos los procesos. La salida de la barrera es la adquisición y la llegada es la liberación.

Además de estos accesos de sincronización, también se lee y escribe en las variables compartidas. La adquisición y liberación no tienen que aplicarse a toda la memoria, sino que protegen sólo algunas variables compartidas específicas, en cuyo caso sólo éstas se mantienen consistentes. Las variables compartidas que se mantienen consistentes son **protegidas**.

El contrato entre la memoria y el software dice que cuando el software realiza una adquisición, la memoria se asegurará de que todas las copias locales de las variables protegidas sean actualizadas de manera consistente con las remotas, en caso necesario. Al realizar una liberación, las variables protegidas que hayan sido modificadas se propagan hacia las demás máquinas. La realización de una adquisición no garantiza que los cambios realizados de manera local sean enviados a las demás máquinas de inmediato. De manera análoga, la realización de una liberación no necesariamente importa las modificaciones de las demás máquinas.

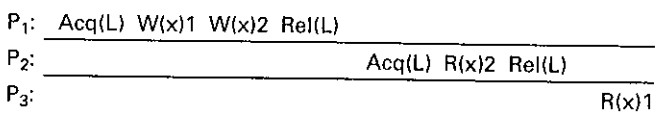


Figura 6-23. Una secuencia de eventos válida para la consistencia de liberación.

La figura 6-23 muestra una secuencia de eventos válida para la consistencia de liberación. El proceso P_1 realiza una adquisición, modifica una variable compartida dos veces, y después realiza una liberación. El proceso P_2 realiza una adquisición y lee x. Se garantiza que obtiene el valor de x al momento de la liberación, es decir, 2 (a menos que la adquisición de P_2 se realice antes de la adquisición de P_1). Si la adquisición fue realizada antes de la liberación de P_1, la adquisición tendría que retrasarse hasta que ocurra la liberación. Puesto que P_3 no puede realizar una adquisición antes de leer una variable compartida, la memoria no tiene la obligación de darle el valor actual de x, de modo que se permite que regrese 1.

Para aclarar el concepto de consistencia de liberación, describiremos de forma breve una implantación simple, en el contexto de la memoria distribuida compartida (la consistencia de liberación fue ideada en realidad para el multiprocesador Dash, pero la idea es la

misma, aunque la implantación no lo es). Para realizar una adquisición, un proceso envía un mensaje a un controlador de sincronización solicitando una adquisición sobre una cerradura particular. Si no hay competencia, se aprueba la solicitud y termina la adquisición. Después, se pueden realizar de manera local una serie arbitraria de lecturas y escrituras a los datos compartidos. Ninguno de estos se propaga a las demás máquinas. Al realizar la liberación, los datos modificados se envían a las demás máquinas que los utilizan. Después de que cada máquina ha reconocido la recepción de los datos, el controlador de sincronización es informado de la liberación. De esta manera, se pueden realizar una cantidad arbitraria de lecturas y escrituras sobre las variables compartidas con un costo fijo. Las adquisiciones y liberaciones de las diversas cerraduras ocurren de manera independiente entre sí.

Aunque el algoritmo centralizado descrito arriba puede realizar el trabajo, de ninguna manera es el único método. En general, una memoria distribuida compartida tiene consistencia de liberación si cumple las siguientes reglas:

1. *Antes de realizar un acceso ordinario a una variable compartida, deben terminar con éxito todas las adquisiciones anteriores del proceso en cuestión.*

2. *Antes de permitir la realización de una liberación, deben terminar las lecturas y escrituras anteriores del proceso.*

3. *Los accesos de adquisición y liberación deben ser consistentes con el procesador (no se pide la consistencia secuencial).*

Si se cumplen todas estas condiciones y los procesos utilizan la adquisición y la liberación de manera adecuada (es decir, en pares adquisición-liberación), los resultados de cualquier ejecución no serán diferentes de lo que ocurriría en una memoria con consistencia secuencial. De hecho, los bloques de acceso a las variables compartidas son atómicos debido a las primitivas de adquisición y liberación, con el fin de evitar el intercalado.

Una implantación diferente de la consistencia de liberación es la **consistencia de liberación con laxitud** (Keleher *et al.*, 1992). En la consistencia de liberación normal, llamada **consistencia de liberación fuerte**, para distinguir la otra variante, al realizar una liberación, el procesador que realiza ésta expulsa todos los datos modificados hacia los demás procesadores que tienen una copia en caché y que podrían necesitarlos. No existe forma de determinar si en realidad los utilizarán, de modo que para estar seguros, todos obtienen lo modificado.

Aunque el envío de todos los datos de esta manera es directo, por lo general no es eficiente. En la consistencia de liberación con laxitud, en el momento de liberación, nada se envía, sino que cuando se realiza una adquisición, el procesador que intenta realizar ésta debe obtener los valores más recientes de las variables de la máquina o máquinas que los contienen. Se puede utilizar un protocolo con marcas de tiempo para determinar cuál de las variables debe transmitirse.

En muchos programas, una región crítica se localiza dentro de un ciclo. Con la consistencia de liberación fuerte, se realiza una liberación por cada paso por el ciclo, y todos los datos modificados deben ser enviados a los demás procesadores que tienen copias de ellos. Este algoritmo desperdicia el ancho de banda e introduce un retraso innecesario. Con la consistencia de liberación con laxitud, no se hace nada al momento de la liberación. En la siguiente adquisición, el procesador determina que ya tiene todos los datos que necesita, por lo que tampoco se generan mensajes en ese momento. El resultado neto es que, con la consistencia de liberación con laxitud, no se genera tráfico alguno en la red hasta que otro procesador realiza una adquisición. Las parejas adquisición-liberación repetidas por un mismo procesador en ausencia de competencia del exterior son gratuitas.

6.3.7. Consistencia de entrada

Otro modelo de consistencia diseñado para su uso con las secciones críticas es la **consistencia de entrada** (Bershad *et al.*, 1993). Como las dos variantes de la consistencia de liberación, requiere que el programador (o compilador) utilice la adquisición y la liberación al principio y al final de cada sección crítica, respectivamente. Sin embargo, a diferencia de la consistencia de liberación, la consistencia de entrada requiere que cada variable compartida ordinaria se asocie con alguna variable de sincronización, como una cerradura o una barrera. Si se desea el acceso individual en paralelo a los elementos de un arreglo, entonces los diferentes elementos del arreglo deben asociarse con cerraduras diferentes. Cuando se realiza una adquisición sobre una variable de sincronización, sólo se pide la consistencia de las variables compartidas ordinarias protegidas por esa variable de sincronización. La consistencia de entrada difiere de la consistencia de liberación con laxitud en el hecho de que la segunda no asocia las variables compartidas con cerraduras o barreras y en el momento de la adquisición debe determinar de manera empírica las variables que necesita.

La asociación de una lista de variables compartidas a cada variable de sincronización reduce el costo asociado con la adquisición y liberación de una variable de sincronización, puesto que sólo hay que sincronizar unas cuantas variables compartidas. También permite que varias secciones críticas con variables compartidas ajenas se ejecuten de manera simultánea, lo que incrementa la cantidad de paralelismo. El precio que se paga es el costo adicional y la complejidad de la asociación de cada variable compartida con alguna variable de sincronización. La programación de esta manera también es más compleja y propensa a errores.

Las variables de sincronización se utilizan como sigue. Cada variable de sincronización tiene un propietario activo, a saber, el proceso que la adquirió por última vez. El propietario puede entrar y salir de las regiones críticas varias veces, sin tener que enviar mensajes a través de la red. Un proceso que no posee por el momento una variable de sincronización pero que desea adquirirla debe enviar un mensaje al propietario actual solicitando la posesión y los valores actuales de las variables asociadas. También es posible que varios procesos

posean de manera simultánea una variable de sincronización en un modo no exclusivo, lo que significa que pueden leer, pero no escribir en las variables dato asociadas.

Desde el punto de vista formal, una memoria exhibe la consistencia de entrada si satisface las siguientes condiciones (Bershad y Zekauskas, 1991):

1. *No se permite realizar un acceso de adquisición a una variable de sincronización con respecto de un proceso hasta que se realicen todas las actualizaciones de los datos compartidos protegidos con respecto de ese proceso.*

2. *Antes de permitir la realización de un acceso en modo exclusivo a una variable de sincronización por un proceso, ningún otro proceso debe poseer la variable de sincronización, ni siquiera en modo no exclusivo.*

3. *Después de realizar un acceso en modo exclusivo a una variable de sincronización, no se puede realizar el siguiente acceso en modo no exclusivo de otro proceso a esa variable de sincronización hasta haber sido realizado con respecto del propietario de esa variable.*

La primera condición dice que cuando un proceso realiza una adquisición, ésta podría no concluir (es decir, regresar el control al siguiente enunciado) hasta actualizar todas las variables compartidas protegidas. En otras palabras, en una adquisición, deben ser visibles todas las modificaciones remotas a los datos protegidos.

La segunda condición dice que antes de actualizar una variable compartida, un proceso debe entrar a una región crítica en modo exclusivo para garantizar que ningún otro proceso intenta actualizarla al mismo tiempo.

La tercera condición dice que si un proceso desea entrar a una región crítica en modo no exclusivo, primero debe verificar con el propietario de la variable de sincronización que protege la región crítica para buscar las copias más recientes de las variables compartidas protegidas.

6.3.8. Resumen de modelos de consistencia

Aunque se han propuesto otros modelos de consistencia, hemos analizado los principales. Difieren en sus restricciones, lo complejo de sus implantaciones, la facilidad para programarlos, y su desempeño. La consistencia estricta es la más restrictiva, pero debido a que su implantación en un sistema DSM es en esencia imposible, nunca se utiliza.

La consistencia secuencial es factible, popular entre los programadores y de uso amplio. Sin embargo, tiene el problema del desempeño pobre. La forma de darle la vuelta a este resultado es relajar el modelo de consistencia. Algunas de las posibilidades aparecen en la figura 6-24(a), con un orden aproximado de restricciones decrecientes.

Consistencia	Descripción
Estricta	Ordenamiento absoluto con respecto de tiempo de todo lo relacionado con el acceso a la memoria
Secuencial	Todos los procesos ven todos los accesos compartidos en el mismo orden
Causal	Todos los procesos ven los accesos compartidos relacionados causalmente en el mismo orden
De procesador	Consistencia PRAM + coherencia de memoria
PRAM	Todos los procesos ven las escrituras de cada procesador en el orden en que fueron ejecutadas. Las escrituras de procesadores diferentes podrían no ser vistas en el mismo orden

(a)

Débil	Los datos compartidos sólo pueden considerarse como consistentes después de realizar una sincronización
De liberación	Los datos compartidos son consistentes al salir de una región crítica
De entrada	Los datos compartidos pertenecientes a una región crítica son consistentes después de salir de una región crítica

(b)

Figura 6-24. (a) Modelos de consistencia que no utilizan las operaciones de sincronización. (b) Modelos con operaciones de sincronización.

La consistencia causal, la consistencia de procesador y la consistencia PRAM representan condiciones menos estrictas donde no existe un acuerdo global del orden de las operaciones. Los diversos procesos pueden ver diferentes secuencias de operaciones, que pueden diferir en términos de las secuencias permitidas y las prohibidas, pero en todos los casos, el programador debe evitar realizar ciertas cosas que funcionan sólo si la memoria tiene una consistencia secuencial.

Un método diferente consiste en introducir variables explícitas de sincronización, como la consistencia débil, la consistencia de liberación y la consistencia de entrada. Estos tres tipos se resumen en la figura 6-24(b). Cuando un proceso realiza una operación sobre una variable dato compartida ordinaria, no existen garantías del momento en que los resultados serán vistos por otros procesos. Las modificaciones sólo se propagan cuando se tiene acceso a una variable de sincronización. Los tres modelos difieren en el funcionamiento de la sincronización, pero en todos los casos, un proceso puede realizar varias lecturas y escrituras en una sección crítica sin llamar al transporte de datos. Al concluir la sección crítica, el resultado final se propaga a los demás procesos o está listo para la propagación en caso de que alguien más exprese su interés por ello.

En resumen, la consistencia débil, la de liberación y la de entrada requieren construcciones de programación adicional que, cuando se utilizan de la manera indicada, permiten a los programadores suponer que la memoria tiene una consistencia secuencial, cuando de

hecho no la tiene. En principio, estos tres modelos que utilizan la sincronización explícita deben poder ofrecer el mejor desempeño, pero es probable que diferentes aplicaciones produzcan resultados un tanto distintos. Se necesita mayor investigación antes de poder establecer conclusiones firmes en estos aspectos.

6.4. MEMORIA COMPARTIDA DISTRIBUIDA CON BASE EN PÁGINAS

Después de estudiar los principios de los sistemas con memoria distribuida compartida, revisaremos ahora los propios sistemas. En esta sección estudiaremos la memoria distribuida compartida "clásica", la primera de las cuales fue IVY (Li, 1986; y Li y Hudak, 1989). Estos sistemas se construyeron sobre multicomputadoras, es decir, procesadores conectados mediante una red especializada para la transferencia de mensajes, las estaciones de trabajo en una LAN o para diseños similares. El elemento esencial en este caso es que ningún procesador puede tener acceso directo a la memoria de otro procesador. Tales sistemas reciben a veces el nombre **NORMA (sin acceso a memoria remota)** en contraste con los sistemas NUMA.

La gran diferencia entre NUMA y NORMA es que el primero, cada procesador puede hacer referencia de manera directa a cada palabra en el espacio global de direcciones, sólo leyendo o escribiendo en él. Las páginas están distribuidas de manera aleatoria entre las memorias, sin afectar los resultados dados por los programas. Cuando un procesador hace referencia a una página remota, el sistema tiene la opción de traerla o utilizarla de manera remota. La decisión afecta el desempeño, pero sigue siendo correcto. Las máquinas NUMA sin verdaderos multiprocesadores; el hardware permite que cada procesador haga referencia a cada palabra del espacio de direcciones sin intervención del software.

Las estaciones de trabajo en una LAN son muy distintas de un multiprocesador. Los procesadores sólo pueden hacer referencia a su memoria local. No existe el concepto de memoria compartida global, como en un NUMA o multiprocesador UMA. Sin embargo, el objetivo del trabajo de DSM, es agregar software al sistema para permitir que una multicomputadora ejecute programados en un multiprocesador. En consecuencia, cuando un procesador hace referencia a una página remota, esa página *debe* ser traída. No existe opción, como en el caso NUMA.

Gran parte de las primeras investigaciones acerca de los sistemas DSM se dedicó a la cuestión de la forma de ejecutar los programas multiprocesadores existentes en las multicomputadoras. A veces, esto se conoce como el problema del "escritorio polvoso". La idea es inyectar nueva vida a los antiguos programas, sólo ejecutándolos en los nuevos sistemas (DSM). El concepto es en particular atractivo para las aplicaciones que necesitan todos los ciclos de CPU que puedan obtener y cuyos autores están entonces interesados en utilizar multicomputadoras de gran escala en vez de multiprocesadores de pequeña escala.

Puesto que los programas escritos para los multiprocesadores suponen por lo general que la memoria tiene consistencia secuencial, el trabajo inicial acerca de DSM fue realizado

con cuidado con el fin de proporcionar una memoria con consistencia secuencial, de modo que los antiguos programas para multiprocesadores pudieran funcionar sin modificaciones. Las experiencias posteriores han mostrado que se puede obtener mejor desempeño si se relaja el modelo de memoria, con el costo de reprogramar las aplicaciones existentes y escribir las nuevas con un estilo diferente. Regresaremos a este punto más adelante, pero primero analizaremos los principales aspectos del diseño en los sistemas DSM clásicos, del tipo IVY.

6.4.1. Diseño básico

La idea detrás de DSM es sencilla: intentar emular el caché de un multiprocesador mediante MMU y el software del sistema operativo. En un sistema DSM, el espacio de direcciones se separa en pedazos, los cuales están dispersos en todos los procesadores del sistema. Cuando un procesador hace referencia a una dirección que no es local, ocurre un señalamiento, y el software DSM trae el pedazo que contiene la dirección y reinicia la instrucción suspendida, que puede entonces concluir con éxito. Este concepto se ilustra en la figura 6-25(a), para un espacio de direcciones con 16 pedazos y cuatro procesadores, cada uno de los cuales puede contener cuatro pedazos.

En este ejemplo, si el procesador 1 hace referencia a las instrucciones o datos en los pedazos 0, 2, 5 o 9, las referencias se realizan de manera local. Las referencias a los demás pedazos provocan señalamientos. Por ejemplo, una referencia a una dirección en el pedazo 10 provocará un señalamiento al software DSM, el cual mueve entonces el pedazo 10 de la máquina 2 a la máquina 1, como se muestra en la figura 6-25(b).

6.4.2. Réplica

Una mejora al sistema básico, que ayuda al desempeño en gran medida, consiste en duplicar los pedazos exclusivos para lectura; por ejemplo, texto de programa, constantes exclusivas para lectura, u otras estructuras de datos exclusivas para lectura. Por ejemplo, si el pedazo 10 de la figura 6-25 es una sección de texto de un programa, su uso por parte del procesador 1 provoca el envío de una copia al procesador 1, sin perturbar al original en la memoria del procesador 2, como se muestra en la figura 6-25(c). De esta manera, los procesadores 1 y 2 pueden hacer referencia al pedazo 10 con la frecuencia necesaria sin causar señalamientos para traer la memoria faltante.

Otra posibilidad consiste en duplicar todos los pedazos no sólo exclusivos para lectura. Mientras se realicen lecturas, en realidad no habrá diferencia entre la duplicación de un pedazo exclusivo para lectura y uno para lectura-escritura. Sin embargo, si un pedazo duplicado súbitamente se modifica, hay que realizar una acción especial para evitar la existencia de varias copias inconsistentes. La forma de evitar la inconsistencia será analizada en las siguientes secciones.

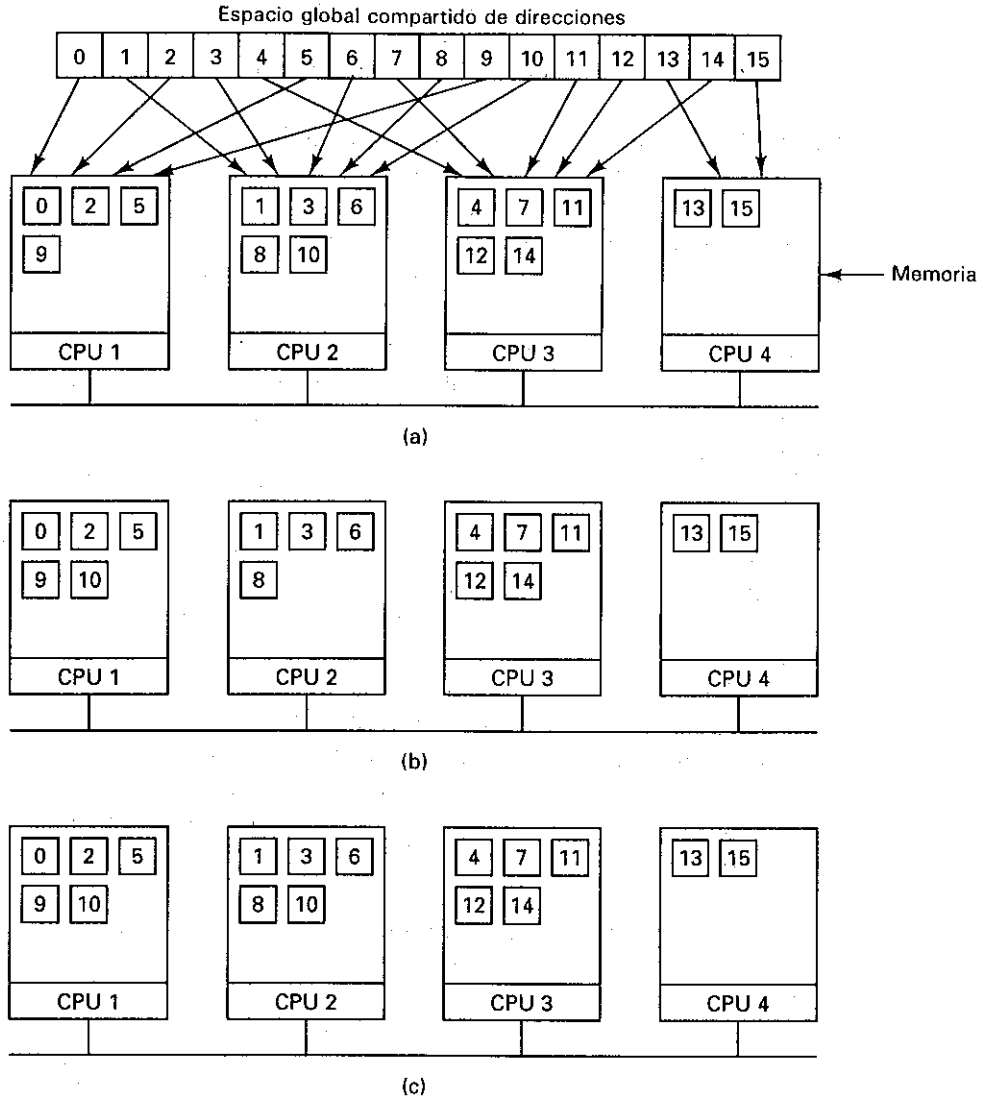


Figura 6-25. (a) Pedazos del espacio de direcciones distribuidos entre cuatro máquinas. (b) Situación después de que el CPU 1 hace referencia al pedazo 10. (c) Situación si el pedazo 10 es exclusivo para lectura y se utiliza la réplica.

6.4.3. Granularidad

Los sistemas DSM son similares a los multiprocesadores en varios aspectos fundamentales. En ambos sistemas, cuando se hace referencia a una palabra de memoria no local, se

trae un pedazo de memoria con la palabra, desde su posición actual, y se coloca en la máquina que hace la referencia (en la memoria principal o el caché, respectivamente). Un aspecto importante del diseño es el tamaño de dicho pedazo. Las posibilidades son una palabra, un bloque (unas cuantas palabras), una página o un segmento (varias páginas).

Con un multiprocesador, el transporte de una palabra o unas docenas de bytes es factible, pues el MMU conoce con exactitud la dirección de referencia y el tiempo para establecer una transferencia en el bus se mide en nanosegundos. Memnet, aunque no es estrictamente un multiprocesador, también utiliza un tamaño pequeño para el pedazo (32 bytes). Con los sistemas DSM, una granularidad tan fina es difícil o imposible, debido a la forma en que funciona MMU.

Cuando un proceso hace referencia a una palabra ausente, provoca un fallo de página. Una elección obvia consiste en traer toda la página necesaria. Además, la integración de DSM con la memoria virtual hace más sencillo el diseño total, pues se utiliza la misma unidad, la página, para ambos. Al ocurrir un fallo de página, la página faltante simplemente se trae de la otra página o del disco, de modo que el código para el manejo de los fallos de página son iguales al caso tradicional.

Sin embargo, otra opción consiste en traer una unidad más grande, digamos, una región de 2, 4 u 8 páginas, incluyendo la página necesaria. De hecho, esto simula un tamaño de página más grande. Existen ventajas y desventajas del uso de un tamaño mayor de pedazo para DSM. La principal ventaja es que debido a que el tiempo de arranque de una transferencia en la red es esencial, no tarda mucho más tiempo la transferencia de 1024 bytes que la de 512 bytes. Al transferir datos en grandes unidades, cuando hay que desplazar una gran parte del espacio de direcciones, el número de transferencias se reduce con frecuencia. Esta propiedad es en particular importante, pues muchos programas exhiben la localidad de las referencias, lo que significa que si un programa ha realizado una referencia a una palabra en una página, es probable que haga referencia a otras palabras de la misma página en un futuro inmediato.

Por otro lado, la red se bloqueará más tiempo con una transferencia mayor, bloqueando otros fallos provocados por otros procesos. Además, un tamaño de página excesivo puede introducir un nuevo problema, el hecho de **compartir de manera falsa**, que se ilustra en la figura 6-26. En este caso tenemos una página que contiene dos variables compartidas no relacionadas entre sí, *A* y *B*. El procesador 1 hace un uso extensivo de *A*, leyendo y escribiendo en ella. De manera similar, el proceso 2 utiliza *B*. En estas circunstancias, la página que contiene las variables viajará constantemente entre las dos máquinas.

El problema aquí es que aunque las variables no están relacionadas entre sí, como aparecen por accidente en la misma página, cuando un proceso utiliza una de ellas, también obtiene la otra. Mientras mayor sea el tamaño efectivo de página, ocurrirá más este fenómeno, y recíprocamente, si el tamaño efectivo de página es menor, este fenómeno ocurrirá con menor frecuencia. No ocurre nada análogo a este fenómeno en los sistemas de memoria virtual ordinarios.

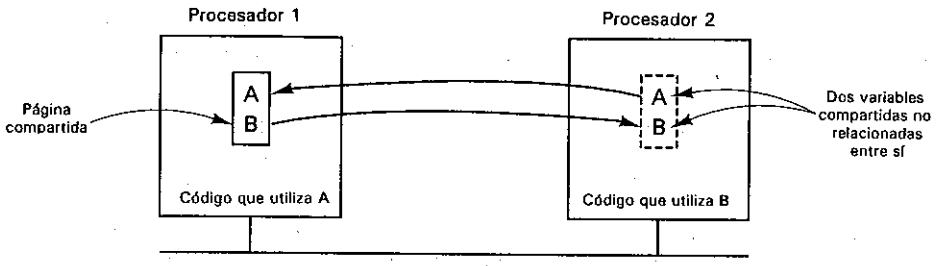


Figura 6-26. Página falsamente compartida con dos variables no relacionadas entre sí.

Los compiladores inteligentes que comprenden el problema y colocan variables en el espacio de direcciones de manera adecuada pueden ayudar a disminuir este fenómeno y mejorar el desempeño. Sin embargo, decir esto es más fácil que hacerlo. Además, si el fenómeno consiste en que el procesador 1 utilice un elemento de un arreglo y el procesador 2 utilice un elemento diferente del mismo arreglo, un compilador inteligente puede hacer poco por eliminar el problema.

6.4.4. Obtención de la consistencia secuencial

Si las páginas no se duplican, no se pretende lograr la consistencia. Existe con exactitud una copia de cada página, y ésta se desplaza de un lugar a otro de manera dinámica según se necesite. Si sólo se tiene una copia de cada página, no existe el peligro de que las diferentes copias tengan valores diversos.

Si se duplican las páginas exclusivas para lectura, tampoco existe problema alguno. Las páginas exclusivas para lectura nunca se modifican, de modo que todas las copias siempre son idénticas. Sólo se conserva una copia de cada página para lectura-escritura, por lo que también son imposibles las inconsistencias en este caso.

El caso interesante es el de las páginas de lectura-escritura duplicadas. En muchos sistemas DSM, cuando un proceso intenta leer una página remota, se crea una copia local, pues el sistema no sabe lo que hay en la página o si se puede escribir en ella. La copia local (de hecho, todas las copias) y la página original están configuradas en su MMU respectivo como exclusivas para lectura. Mientras todas las referencias sean lecturas, todo está bien.

Sin embargo, si algún proceso intenta escribir en una página duplicada, puede surgir un problema debido a que la modificación de una copia sin modificar las demás es inaceptable. Esta situación es similar a lo que ocurre en un multiprocesador, cuando un procesador intenta modificar una palabra que está presente en varios cachés, así que daremos un repaso de lo que hacen los multiprocesadores en tales circunstancias.

En general, los multiprocesadores adoptan uno de dos métodos: actualización o invalidación. Con la actualización, se permite la escritura de manera local, pero la dirección de la palabra modificada y su nuevo valor se transmiten por el bus de manera simultánea a todos los demás cachés. Cada uno de los cachés que contiene la palabra por actualizar ve

que una dirección de su caché está siendo modificada, por lo que copia el nuevo valor del bus a su caché, escribiendo sobre el valor anterior. El resultado final es que todos los cachés que contenían la palabra antes de la actualización también la contienen después de ésta, y adquieren el nuevo valor.

El otro método que adoptan los multiprocesadores es la invalidación. Cuando se utiliza esta estrategia, la dirección de la palabra por actualizar se transmite por el bus, pero el nuevo valor no. Cuando un caché ve que una de sus palabras está siendo actualizada, invalida el bloque de caché que contiene la palabra, lo que de hecho la elimina del caché. El resultado final con la invalidación es que sólo un caché contiene ahora la palabra modificada, por lo que se evitan los problemas de consistencia. Si uno de los procesadores que ahora contienen una copia inválida del bloque caché intenta utilizarla, obtendrá un fracaso del caché y traerá el bloque de un procesador que contenga una copia válida.

Aunque la implantación de estas dos estrategias es casi igual de sencilla en un multiprocesador, difieren radicalmente en un sistema DSM. A diferencia de un multiprocesador, donde el MMU sabe cuál palabra se escribirá y cuál es su nuevo valor, en un sistema DSM el software no sabe esto. Para descubrirlo, crea una copia secreta de la página que va a modificarse (se conoce el número de página), hacer que se escriba en la página, activar el bit de señalamiento al hardware, para producir un señalamiento después de cada instrucción y reiniciar el proceso suspendido. Después de una instrucción, observa el señalamiento y compara la página activa con la copia secreta que creó, para ver cuál palabra ha sido modificada. Entonces, transmite por la red un breve paquete con la dirección y el nuevo valor. Los procesadores que reciban este paquete verifican si tienen la página en cuestión y, en tal caso, la actualizan.

La cantidad de trabajo en este caso es enorme; pero lo que es peor, el esquema no es infalible. Si se realizan de manera simultánea varias actualizaciones, con origen en diferentes procesadores, éstos podrían ver las actualizaciones en diferentes órdenes, por lo que la memoria no tendría una consistencia secuencial. En un multiprocesador, no ocurre este problema, puesto que las transmisiones por el bus son por completo confiables (no se pierden mensajes) y el orden no es ambiguo.

Otro aspecto de esto es que un proceso puede tener miles de escrituras consecutivas en la misma página debido a que muchos programas exhiben la localidad de la referencia. Tomar nota de todas estas actualizaciones y transferirlas a las demás máquinas es caro en ausencia de un monitoreo del tipo del multiprocesador.

Por estas razones, los sistemas DSM basados en páginas utilizan por lo general un protocolo de invalidación en vez de uno de actualización. Son posibles varios protocolos. Adelante describiremos un ejemplo típico, en donde potencialmente se puede escribir en todas las páginas (es decir, el software DSM no sabe lo que hay en cada página).

En este protocolo, en cualquier momento, cada página está en un estado *R* (para lectura) o *W* (para lectura y escritura). El estado en que está una página se modifica conforme avanza la ejecución. Cada página tiene un propietario, a saber, el proceso que escribió de manera más reciente en dicha página. Cuando una página está en el estado *W*, existe una copia, asociada al espacio de direcciones del propietario en modo lectura-escritura. Cuando una

página está en el estado *R*, el propietario tiene una copia (con una asociación exclusiva para lectura), pero los demás procesos también podrían tener copias.

Se distinguen seis casos, como se muestra en la figura 6-27. En todos los ejemplos de la figura, el proceso *P* en el procesador 1 desea leer o escribir en una página. Los casos difieren en términos de si *P* es el propietario, si *P* tiene una copia, si otros procesos tienen copias, y el estado de la página, como se muestra.

Ahora consideremos las acciones por realizar en cada uno de los casos. En los primeros cuatro casos de la figura 6-27(a), *P* sólo realiza la lectura. En los otros cuatro, la página se asocia con su espacio de direcciones, de modo que la lectura se realiza en hardware. No ocurren señalamientos. En los casos 5 y 6, la página no se asocia, de modo que ocurre un fallo de página y el software DSM obtiene el control. Envía un mensaje al propietario solicitando una copia. Cuando la copia regresa, la página es asociada y la instrucción suspendida es reiniciada. Si el propietario tiene la página en el estado *W*, debe degradarla al estado *R*, pero puede conservar la página. En este protocolo, el otro proceso conserva la propiedad, pero en un protocolo poco diferente, ésta también podría ser transferida.

Las escrituras se manejan de manera diferente, como se muestra en la figura 6-27(b). En el primer caso, la escritura acaba de ocurrir, sin señalamientos, puesto que la página se asocia en modo lectura-escritura. En el segundo caso (sin copias), la página se cambia al estado *W* y se escribe en ella. En el tercer caso, existen más copias, por lo que primero deben ser invalidadas antes de realizar la escritura.

En los siguientes tres casos, algún otro proceso es el propietario al momento en que *P* realiza la escritura. En los tres casos, *P* debe solicitar al propietario actual que invalide las copias existentes, transferir la propiedad a *P* y enviar una copia de la página a menos que *P* ya tenga una. Sólo entonces se puede realizar la escritura. En los tres casos, *P* termina con la única copia de la página, en estado *W*.

En los seis casos, antes de realizar una escritura, el protocolo garantiza que sólo existe una copia de la página, a saber, en el espacio de direcciones del proceso que está a punto de hacer la escritura. De esta manera, se mantiene la consistencia.

6.4.5. Búsqueda del propietario

Hemos omitido algunos puntos en la descripción anterior. Uno de ellos es la forma de encontrar al propietario de la página. La solución más sencilla consiste en realizar una transmisión, y solicitar la respuesta del propietario de la página específica. Una vez que el propietario ha sido localizado de esta manera, el protocolo puede continuar de la manera anterior.

La optimización obvia consiste en no sólo preguntar quién es el propietario, sino también indicar si el emisor desea leer o escribir y si necesita una copia de la página. El propietario puede enviar entonces un mensaje, transfiriendo la propiedad y la página, según sea necesario.

La transmisión tiene la desventaja de interrumpir a cada procesador, obligándolo a inspeccionar el paquete de solicitud. Para todos los procesadores, excepto el propietario,

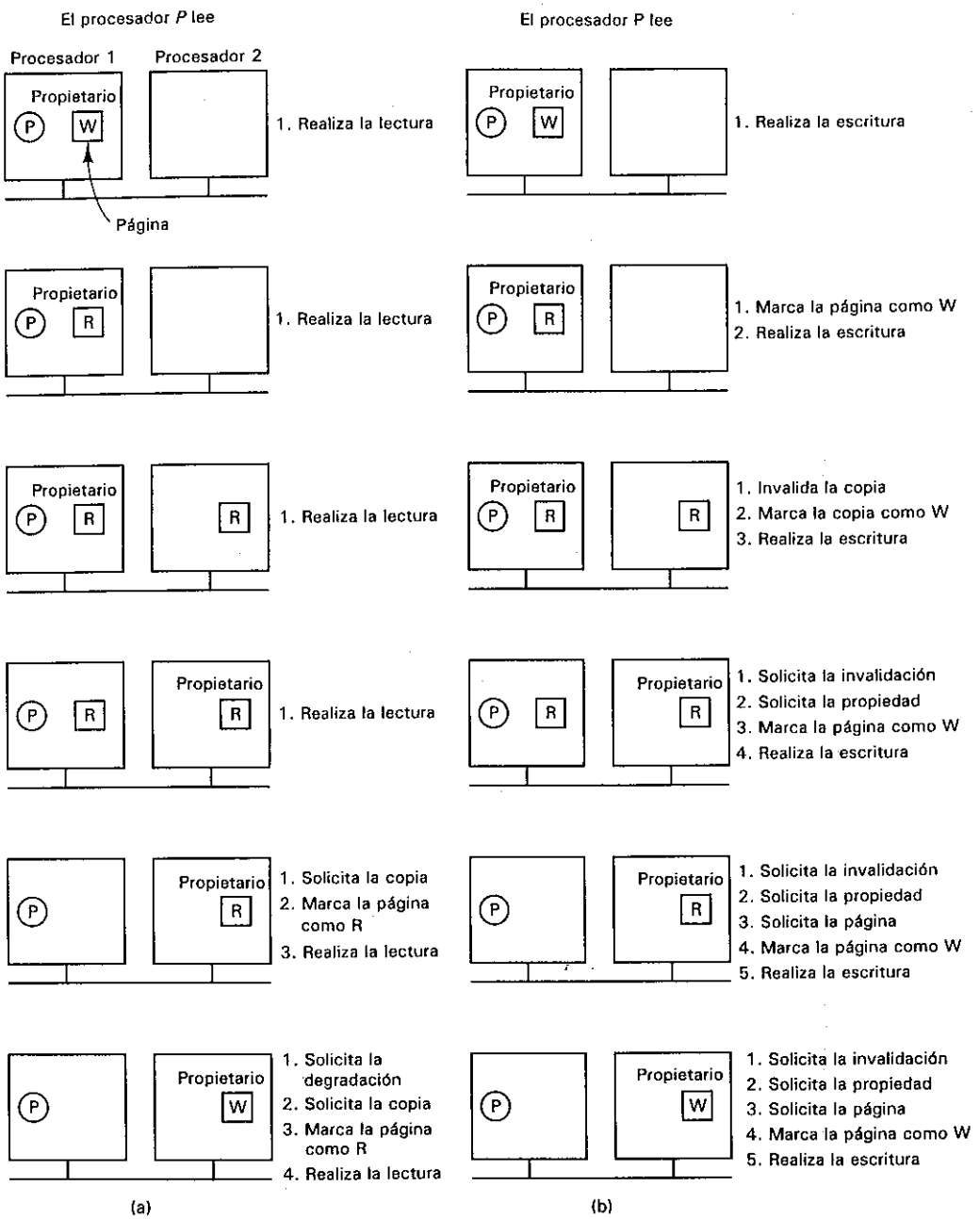


Figura 6-27. (a) El proceso *P* desea leer la página. (b) El proceso *P* desea escribir en la página

el manejo de la interrupción es en esencia un tiempo perdido. La transmisión puede utilizar un ancho de banda considerable, según el hardware.

Li y Hudak (1989) describen otras posibilidades. En la primera de ellas, un proceso es designado como controlador de páginas. Su tarea consiste en llevar un registro de quién es propietario de cada página. Cuando un proceso *P* desea leer una página, tiene que escribir en una página que no tiene o desea escribir una página que no posee, envía un mensaje al controlador de páginas indicando la operación que desea realizar y la página correspondiente. El controlador envía entonces un mensaje indicando el propietario. *P* hace contacto entonces con el propietario para obtener la página o la propiedad, según sea necesario. Se necesitan cuatro mensajes para este protocolo, como se muestra en la figura 6-28(a).

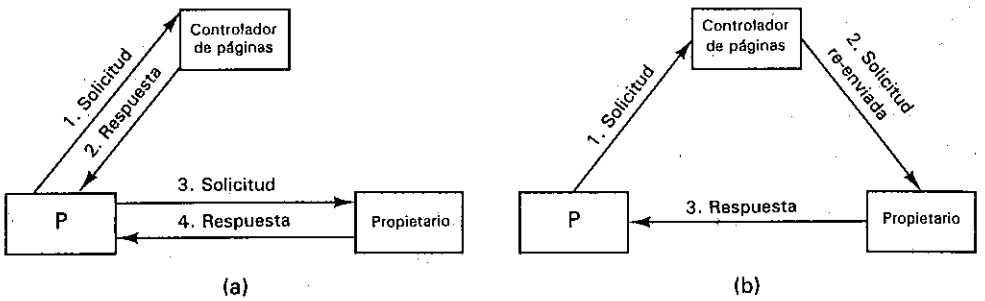


Figura 6-28. Localización de la propiedad mediante un controlador central. (b) Protocolo de cuatro mensajes. (c) Protocolo de tres mensajes.

La figura 6-28(b) muestra una optimización de este protocolo de localización del propietario. En este caso, el controlador de páginas envía la solicitud directamente al propietario, el cual contesta entonces a *P*, lo que ahorra un mensaje.

Un problema con este protocolo es la carga potencial excesiva sobre el controlador de páginas, el cual maneja todas las solicitudes recibidas. Este problema se resuelve con varios controladores de páginas en vez de uno. Sin embargo, la división del trabajo en varios controladores introduce un nuevo problema: encontrar el controlador correcto. Una solución sencilla consiste en utilizar los bits de menor orden del número de página como un índice en una tabla de controladores. Así, si se tienen ocho controladores de página, todas las páginas que terminen con 000 serán manejadas por el controlador 0, todas las páginas que terminen con 001 serán manejadas por el controlador 1, etc. También se puede utilizar una asociación diferente, por ejemplo, mediante una tabla de dispersión. El controlador de páginas utiliza las solicitudes recibidas no sólo para proporcionar respuestas sino para llevar un registro de los cambios de propiedad. Cuando un proceso dice que desea escribir en una página, el controlador registra ese proceso como el nuevo propietario.

Otro algoritmo posible consiste en hacer que cada proceso (o más probablemente, cada procesador) lleve un registro del probable propietario de cada página. Las solicitudes de propiedad se envían al probable propietario, quien las vuelve a transmitir si la propiedad ha cambiado. Si la propiedad cambia varias veces, el mensaje de solicitud tendrá que en-

viarse también varias veces. Al inicio de la ejecución y después de cada *n* cambios de propiedad, se podría transmitir la posición del nuevo propietario, para permitir que todos los procesadores actualicen sus tablas de probables propietarios.

El problema de localización del controlador también está presente en los multiprocesadores, como en Dash y también en Memnet. En ambos sistemas, esto se resuelve separando el espacio de direcciones en regiones y asignando cada región a un controlador fijo, que en esencia es la misma técnica que la solución con varios controladores ya analizada, pero utilizando los bits de orden superior del número de página como el número de controlador.

6.4.6. Búsqueda de las copias

Otro detalle importante es la forma de localizar todas las copias cuando éstas se invaliden. De nuevo, se presentan dos posibilidades. La primera consiste en transmitir un mensaje con el número de página y solicitar a todos los procesadores que contengan la página que la invaliden. Este método sólo funciona si los mensajes de transmisión son por completo confiables y nunca se pueden perder.

La segunda posibilidad consiste en que el propietario o el controlador de páginas mantengan una lista del **conjunto de copias**, indicando los procesadores que poseen tal o cual página, como se muestra en la figura 6-29. En este caso, por ejemplo, la página 4 es poseída por un proceso en el CPU 1, como se muestra en el cuadro doble en torno de 4. El conjunto de copias consta de 2 y 4, puesto que las copias de la página 4 se pueden encontrar en dichas máquinas.

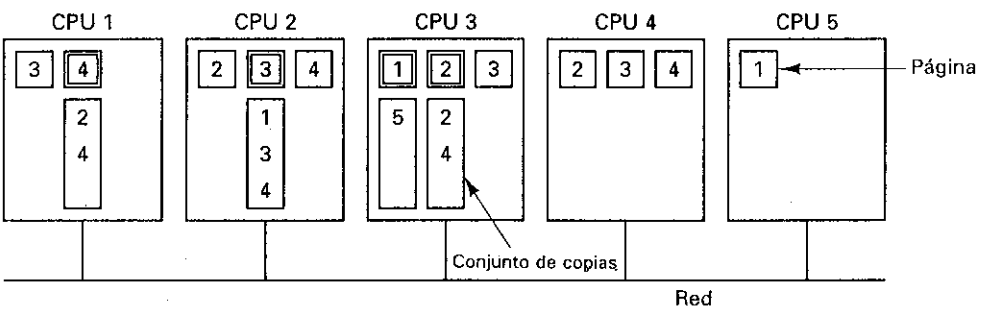


Figura 6-29. El propietario de cada página mantiene un conjunto de copias indicando los demás CPU que comparten esa página. La propiedad de la página se indica mediante los cuadros dobles.

Cuando hay que invalidar una página, el antiguo propietario, el nuevo, o el controlador de páginas, envía un mensaje a cada procesador que contenga la página y espera un reconocimiento. Cuando se ha reconocido cada mensaje, la invalidación ha terminado.

Dash y Memnet también invalidan las páginas cuando un escritor nuevo aparece súbitamente, pero lo hacen de manera diferente. Dash utiliza directorios. El proceso escritor

envía un paquete al directorio (el controlador de páginas en nuestra terminología), el cual reúne todas las copias a partir de su mapa de bits, envía cada una en un paquete de invalidación y recoge todos los reconocimientos. Memnet trae la página necesaria e invalida todas las copias transmitiendo un paquete de invalidación a través del anillo. El primer procesador que tenga una copia la coloca en el paquete y envía un bit de encabezado diciendo que está ahí. Los demás procesadores sólo invalidan sus copias. Cuando el paquete regresa por el anillo al emisor, los datos necesarios están presentes y las demás copias han desaparecido. De hecho, Memnet implanta DSM en hardware.

6.4.7. Reemplazo de página

En un sistema DSM, como en cualquier sistema que utilice una memoria virtual, puede ocurrir que una página se necesite pero que no haya un marco de página libre en la memoria para contenerla. Cuando ocurre esta situación, hay que sacar una página de la memoria para tener espacio disponible para la página necesaria. Surgen de inmediato dos subproblemas: la página que debe tomarse y el lugar dónde colocarla.

En gran medida, la elección de la página por sacar se puede realizar mediante los algoritmos tradicionales de la memoria virtual, como cierta aproximación del método del uso menos reciente (LRU). Una complicación que ocurre con DSM es que las páginas se pueden invalidar de manera espontánea (debido a la actividad de otros procesos), lo que afecta las posibles elecciones. Sin embargo, al mantener el orden LRU estimado únicamente de aquellas páginas que actualmente son válidas, se puede utilizar cualquiera de los algoritmos tradicionales.

Como con los algoritmos convencionales, es importante llevar un registro de las páginas "limpias" y las "sucias". En el contexto de DSM, una página duplicada poseída por otro proceso es siempre un candidato idóneo para sacar de la memoria, pues se sabe que existe otra copia. En consecuencia, la página no tiene que guardarse en otro lugar. Sin embargo, si se utiliza un esquema de directorios para llevar un registro de las copias, el propietario o controlador de páginas es informado de esta decisión. Si las páginas se localizan mediante una transmisión, la página sólo se puede descartar.

La segunda mejor elección es una página duplicada que posee el proceso saliente. Basta transferir la propiedad a una de las otras copias, informando a ese proceso, al controlador de páginas o a ambos, según la implantación. La propia página no tiene que transferirse, lo que produce un mensaje menor.

Si ninguna de las páginas duplicadas es un candidato adecuado, hay que elegir una página no duplicada; por ejemplo, la página válida de uso menos reciente. Existen dos posibilidades para deshacerse de ella. La primera es escribirla en un disco, si éste existe. La otra es mandarla a otro procesador.

La elección de un procesador para enviarle la página se realiza de varias maneras. Por ejemplo, a cada página se le asigna una máquina de origen, la cual la acepta, aunque esto podría implicar la reservación de una gran cantidad de espacio por lo regular no utilizado, para conservar las páginas que podrían ser enviadas ahí algún día. Otra alternativa consiste

en que el número de marcos de página libres sea empaquetado con cada mensaje enviado, de modo que cada procesador proporcione una idea la cantidad de memoria libre distribuida en la red. Un mensaje de transmisión ocasional con la cantidad exacta de marcos de página libres podría mantener estos números actualizados.

Como aspecto colateral, observe que puede existir un conflicto entre la elección de una página duplicada (que simplemente se puede descartar) y la elección de una página a la que no se ha hecho referencia durante mucho tiempo (que podría ser la única copia). Sin embargo, este mismo problema aparece en los sistemas tradicionales con memoria virtual, de modo que se aplican los mismos compromisos y heurística.

Un problema exclusivo de los sistemas DSM es el tráfico generado en la red cuando los procesos de máquinas diferentes comparten de manera activa una página para escritura, ya sea de manera falsa o verdadera. La forma adecuada para reducir el tráfico es mediante la regla de que, una vez que una página llegue a algún procesador, ésta permanezca ahí durante cierto tiempo ΔT. Si llegan solicitudes de esa página por parte de otras máquinas, éstas simplemente se forman hasta que expira el cronómetro, lo que permite que el proceso local realice muchas referencias a memoria sin interferencia.

Como es usual, es instructivo ver la forma en que se maneja el reemplazo de páginas en los multiprocesadores. En Dash, cuando se llena un caché, siempre existe la opción de escribir el bloque de nuevo en la memoria principal. En los sistemas DSM, esa posibilidad no existen aunque con frecuencia es factible utilizar un disco como el último recipiente de las páginas que nadie desea. En Memnet, cada bloque caché tiene una máquina de origen, a la que se pide que reserve cierto espacio de almacenamiento para él. Este diseño también es posible en un sistema DSM, aunque desperdicia mucho espacio en Memnet y DSM.

6.4.8. Sincronización

En un sistema DSM, como en un multiprocesador, los procesos necesitan con frecuencia sincronizar sus acciones. Un ejemplo común es la exclusión mutua, en la que sólo un proceso puede ejecutar a la vez cierta parte del código. En un multiprocesador, la instrucción TEST-AND-SET-LOCK (TSL) se utiliza con frecuencia para implantar la exclusión mutua. En su uso normal, cierta variable es 0 cuando ningún proceso está en la sección crítica y 1 cuando un proceso lo está. La instrucción TSL lee la variable y la activa en una sola operación atómica. Si el valor leído es 1, el proceso simplemente repite la instrucción TSL hasta que el proceso sale de la región crítica y hace la variable igual a 0.

En un sistema DSM, este código sigue siendo correcto, pero podría provocar un desastre en el desempeño. Si un proceso *A* está dentro de la región crítica y otro proceso *B* (en una máquina distinta) desea entrar a ella, *B* estará en un ciclo, verificando el valor de la variable, esperando que sea igual a cero. La página con la variable permanecerá en la máquina de *B*. Cuando *A* sale de la región crítica e intenta escribir 0 en la variable, obtendrá un fallo de página y solicitará la página que contiene la variable. Inmediatamente después, *B* ob-

tendrá también un fallo de página, obligando a la página a regrésar. Este desempeño es aceptable.

El problema aparece cuando otros procesos intentan entrar a la región crítica. Recuerde que la instrucción TSL modifica la memoria (escribiendo 1 en la variable de sincronización) cada vez que se ejecuta. Así, cada vez que un proceso ejecuta una instrucción TSL, debe traer toda la página que contiene la variable de sincronización del lugar donde se encuentre ésta. Si varios procesos ejecutan una instrucción TSL cada pocos cientos de nanosegundos, el tráfico en la red podría ser intolerable.

Por esta razón, se necesita con frecuencia un mecanismo adicional para la sincronización. Una posibilidad consiste en que un controlador (o controladores) de sincronización acepte mensajes solicitando entrar y salir de las regiones críticas, cerrar o eliminar la cerradura de variables, etc.,y enviar las respuestas al terminar el trabajo. Cuando no se puede entrar a una región o no se cierra una variable, no se envía una respuesta de regreso en forma inmediata, lo que provoca un bloqueo del emisor. Cuando la región está disponible o se puede cerrar la variable, se envía un mensaje de regreso. De esta manera, la sincronización se puede lograr con un mínimo de tráfico en la red, pero con el costo de centralización del control por cada cerradura.

6.5. MEMORIA COMPARTIDA DISTRIBUIDA CON VARIABLES COMPARTIDAS

La DSM basada en páginas toma un espacio normal lineal de direcciones y permite que las páginas emigren de manera dinámica sobre la red según la demanda. Los procesos tienen acceso a toda la memoria mediante las instrucciones normales de lectura y escritura y no están conscientes de la ocurrencia de fallos de página o de las transferencias en la red. Los accesos a los datos remotos son detectados y protegidos por el MMU.

Un método más estructurado consiste en compartir sólo ciertas variables y estructuras de datos necesarias para más de un proceso. De esta manera, el problema pasa de la forma de realizar la paginación sobre la red a la forma de mantener una base de datos distribuida, en potencia duplicada, consistente en las variables compartidas. Se pueden aplicar diversas técnicas en este caso, y éstas conducen con frecuencia a mejoras esenciales en el desempeño.

La primera cuestión es si las variables compartidas deben o no duplicarse; y en ese caso, si deben duplicarse de manera parcial o total. Si deben duplicarse, existe más potencial en el uso de un algoritmo de actualización que en el de un sistema DSM basado en páginas, siempre que las escrituras en las variables compartidas individuales se puedan aislar.

El uso de variables compartidas controladas de manera individual también proporciona una oportunidad importante para no compartir falsamente. Si es posible actualizar una variable sin afectar a las demás, entonces la organización física de las variables en las páginas es de menor importancia. Dos de los ejemplos más interesantes de tales sistemas son Munin y Midway, que se describen a continuación.

6.5.1. Munin

Munin es un sistema DSM que se basa fundamentalmente en objetos del software, pero que puede colocar cada objeto en una página aparte, de modo que el hardware MMU pueda utilizarse para detectar el acceso a los objetos compartidos (Bennett *et al.*, 1990; y Carter *et al.*, 1991, 1993). El modelo básico utilizado por Munin es el de varios procesadores, cada uno de ellos con espacio de dirección lineal por páginas, en el que uno o más hilos ejecutan un programa multiprocesador con ligeras modificaciones. El objetivo del proyecto Munin es el de tomar los programas multiprocesadores existentes, realizarles cambios menores y hacerlos que se ejecuten de manera eficiente en los sistemas con multicomputadoras que utilicen una forma de DSM. Se logra un buen desempeño mediante varias técnicas que se describen a continuación, incluyendo el uso de la consistencia de liberación en vez de la consistencia secuencial.

Las modificaciones consisten en anotar las declaraciones de las variables compartidas con la palabra reservada *shared*, de modo que el compilador las reconozca. También se puede proporcionar información acerca del patrón de uso esperado, para permitir el reconocimiento y optimización de ciertos casos especiales importantes. Por omisión, el compilador coloca cada variable compartida en una página separada, aunque las variables compartidas de gran tamaño, como los arreglos, ocupen varias páginas. También es posible que el programador especifique la colocación de variables compartidas del mismo tipo en Munin en la misma página. La mezcla de tipos no funciona, pues el protocolo de consistencia utilizado para una página depende del tipo de variables que estén en ella.

Para ejecutar el programa compilador, se inicia un proceso raíz en uno de los procesadores. Este proceso puede generar nuevos procesos en otros procesadores, los que se ejecutan en paralelo con el principal y se comunican entre sí mediante las variables compartidas, como lo hacen los programas multiprocesadores normales. Una vez iniciado en un procesador particular, un proceso no se puede mover.

El acceso a las variables compartidas se logra mediante las instrucciones normales de lectura y escritura del CPU. Ni se utilizan métodos de protección especiales. Si ocurre un intento por utilizar una variable compartida que no esté presente, ocurre un fallo de página y el sistema Munin obtiene el control.

La sincronización para la exclusión mutua se maneja de manera especial y está íntimamente relacionada con el modelo de consistencia de la memoria. Las variables con cerradura se pueden declarar, y se proporcionan procedimientos de biblioteca para cerrarlos y abrirlos. También se soportan las barreras, las variables de condición y otras variables de sincronización.

Consistencia de liberación

Munin se basa en una implantación software de la consistencia de liberación (fuerte). Para todo el bagaje teórico, ver Gharachorloo *et al.*, (1990). Lo que hace Munin es proporcionar las herramientas para que los usuarios estructuren sus programas en torno de las

regiones críticas, definidas de manera dinámica mediante las llamadas de adquisición (entrada) y liberación (salida). Las escrituras a las variables compartidas ocurren dentro de las regiones críticas; y las lecturas ocurren dentro o afuera. Mientras un proceso está activo dentro de una región crítica, el sistema no garantiza la consistencia de las variables compartidas, pero cuando sale de una región crítica, las variables compartidas modificadas desde la última liberación son actualizadas en todas las máquinas. Para los programas que obedecen este modelo de programación, la memoria distribuida compartida actúa como si tuviera consistencia secuencial.

Munin distingue tres clases de variables:

1. Variables ordinarias.
2. Variables de datos compartidos.
3. Variables de sincronización.

Las variables ordinarias no se comparten y sólo pueden ser leídas o escritas por el proceso que las creó. Las variables de datos compartidos son visibles para varios procesadores y parecen secuencialmente consistentes, siempre que todos los procesos las utilicen sólo en las regiones críticas. Deben ser declaradas como tales, pero su acceso es mediante las instrucciones normales para lectura y escritura. Las variables de sincronización, como las cerraduras o las barreras, son especiales, y sólo se puede tener acceso a ellas por medio de procedimientos de acceso proporcionados por el sistema, como *lock* y *unlock* para las cerraduras e *increment* y *wait* para las barreras. Estos procedimientos son los que permiten el funcionamiento de la memoria distribuida compartida.

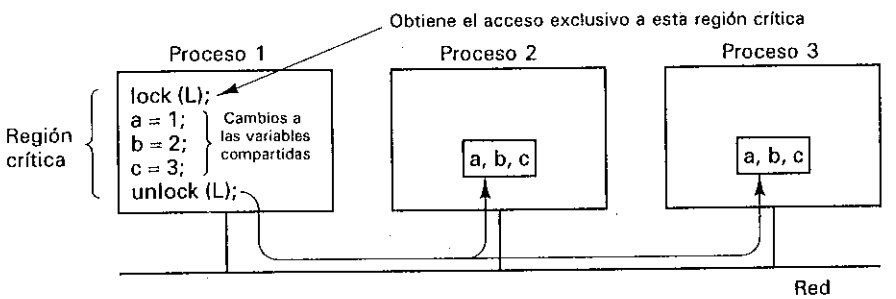


Figura 6-30. Consistencia de liberación en Munin.

La operación básica de la consistencia de liberación de Munin se muestra en la figura 6-30 para tres procesos cooperativos, cada uno de los cuales se ejecuta en una máquina distinta. En cierto momento, el proceso 1 desea entrar a una región crítica de código protegida por la cerradura *L* (todas las regiones críticas deben estar protegidas por alguna variable de sincronización). El enunciado *lock* garantiza que ningún otro proceso bien comportado está ejecutando por el momento esa región crítica. Entonces, se tiene acceso

a las tres variables compartidas *a*, *b* y *c* mediante las instrucciones normales de la máquina. Por último, se llama a *unlock* y los resultados se propagan a las otras máquinas que tienen copias de *a*, *b* o *c*. Estas modificaciones se empaquetan en un número mínimo de mensajes. Los accesos a estas variables en otras máquinas mientras el proceso 1 sigue dentro de su región crítica producen resultados indefinidos.

Protocolos múltiples

Además de utilizar la consistencia de liberación, Munin también utiliza otras técnicas para mejorar el desempeño. La principal de éstas consiste en permitir al programador que realice anotaciones en las declaraciones de las variables compartidas, clasificándolas dentro de alguna de las cuatro categorías siguientes:

1. Exclusiva para lectura.
2. Migratoria.
3. De escritura compartida.
4. Convencional.

En un principio, Munin soportaba otras categorías, pero la experiencia mostró que sólo tenían un valor marginal, de modo que se eliminaron. Cada máquina tiene un directorio con una lista de las variables, donde aparece, entre otras cosas, la categoría a la que pertenece. Para cada categoría, se utiliza un protocolo diferente.

Las variables exclusivas para lectura son las más sencillas. Cuando una referencia a una variable exclusiva para lectura provoca un fallo de página, Munin busca la variable en el directorio de variables, encuentra a su propietario, y solicita a éste una copia de la página requerida. Puesto que las páginas que contienen variables exclusivas para lectura no se modifican (después de iniciarlas), no surgen problemas de consistencia. Las variables exclusivas para lectura son protegidas por el hardware MMU. Un intento por escribir en alguna provoca un error fatal.

Las variables compartidas migratorias utilizan el protocolo de adquisición/liberación ilustrado con cerraduras en la figura 6-30. Se utilizan dentro de las regiones críticas y están protegidas por variables de sincronización. La idea es que estas variables emigren de una máquina a otra conforme se entre o salga de las regiones críticas. No se duplican.

Para utilizar una variable compartida migratoria, primero hay que adquirir su cerradura. Al leer la variable, se crea una copia de su página en la máquina que hace la referencia y se elimina la copia original. Como optimización, se puede asociar una variable compartida migratoria con una cerradura, de modo que cuando se envíe la cerradura, los datos se envíen junto con ella, eliminando los mensajes adicionales.

Una variable de escritura compartida se utiliza cuando el programador indica que es seguro el hecho de que dos o más procesos escriban en ella al mismo tiempo; por ejemplo, un arreglo en el que los diferentes procesos tienen acceso concurrente a diversos subarreglos. En un principio, las páginas que contienen variables de escritura compartida se marcan

como exclusivas para lectura, en potencia en varias máquinas a la vez. Cuando ocurre una escritura, el controlador de fallos de página crea una copia de página, su **gemelo**, marca la página como sucia, y activa al MMU para permitir las escrituras posteriores. Estos pasos se ilustran en la figura 6-31 para una palabra que al inicio tiene el valor 6 y que después se modifica a 8.

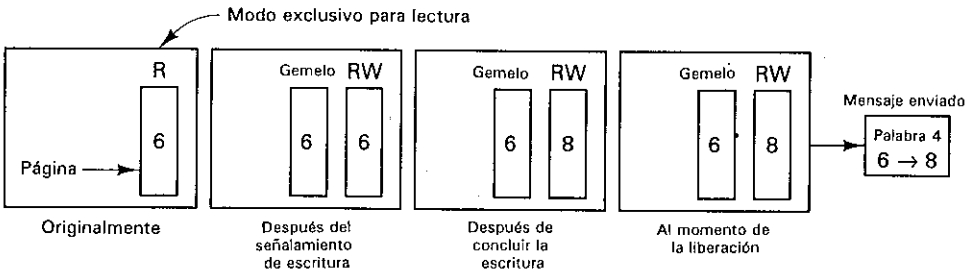


Figura 6-31. Uso de páginas generales en Munin.

Al hacer la liberación, Munin ejecuta una comparación palabra por palabra de cada página de escritura compartida sucia con su gemelo, y envía las diferencias (junto con todas las páginas migratorias) a todos los procesos que las necesiten. Entonces, restablece la protección de página para hacerla exclusiva para lectura.

Cuando una lista de diferencias llega a un proceso, el receptor verifica cada página para ver si también tiene la página modificada. Si una página no ha sido modificada, se aceptan las modificaciones recibidas. Sin embargo, si una página ha sido modificada de manera local, la copia local, su gemelo, y la página recibida correspondiente se comparan palabra por palabra. Si la palabra local ha sido modificada pero la palabra recibida no, la palabra recibida se escribe sobre la local. Si las palabras local y recibida han sido modificadas, se señala un error de tiempo de ejecución. Si no existen tales conflictos, la página fusionada reemplaza la local y la ejecución continúa.

Las variables compartidas que no tienen la nota de pertenecer a alguna de las categorías anteriores se consideran como en los sistemas DSM convencionales basados en páginas: sólo se permite una copia de cada página que se pueda escribir, y ésta se desplaza de un proceso a otro según la demanda. Las páginas exclusivas para lectura se duplican según lo necesario.

Analicemos ahora un ejemplo del uso del protocolo con varios escritores. Consideremos los programas de las figuras 6-32(a) y (b). En este caso, dos procesos están incrementando los elementos del mismo arreglo. El proceso 1 incrementa los elementos pares mediante la función f y el proceso 2 incrementa los elementos impares mediante la función g. Antes de iniciar esta fase, cada proceso se bloquea en una barrera hasta que el otro llega también a ella. Después de terminar esta fase, se bloquean en otra barrera hasta que ambos terminan.

Proceso 1

```
/* Espera al proceso 2 */
   wait_at_barrier(b);
   for (i = 0; i < n; i +=2)
         a[i] = a[i] + f(i);
/* Espera a que termine el proceso 2 */
   wait_at_barrier(b);
```

(a)

Proceso 2

```
/* Espera al proceso 1 */
 wait_at_barrier(b);
 for (i = 1; i < n; i +=2)
       a[i] = a[i] + g(i);
/* Espera a que termine el proceso 1 */
 wait_at_barrier(b);
```

(b)

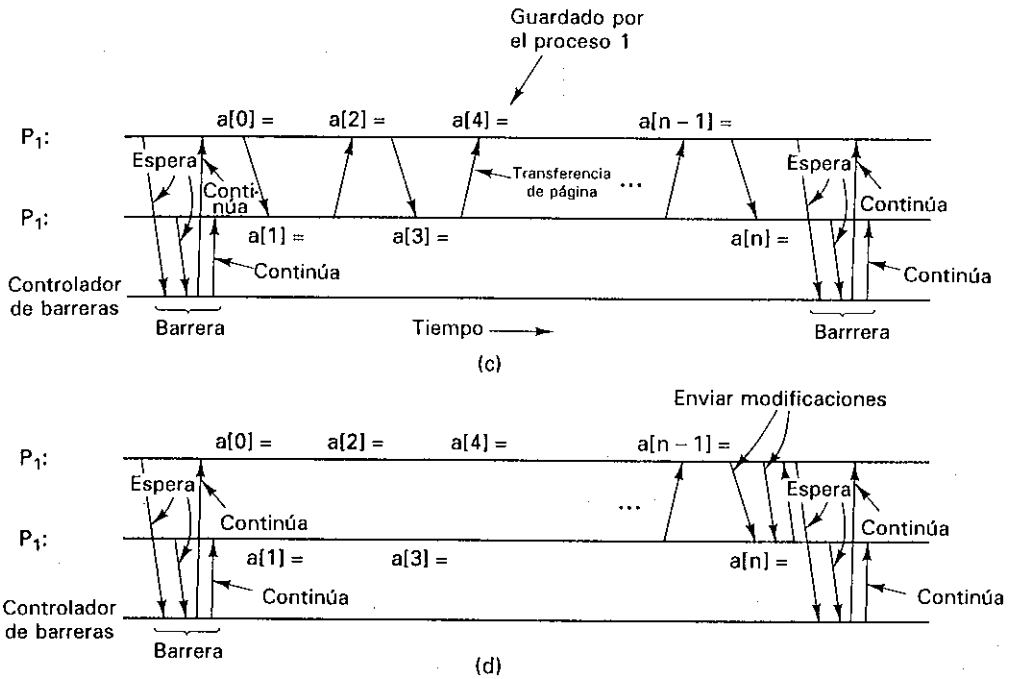


Figura 6-32. (a) Un programa que utiliza *a*. (b) Otro programa que utiliza *a*. (c) Los mensajes enviados para la memoria con consistencia secuencial. (d) Mensajes enviados para una memoria con consistencia de liberación.

Así continúan ambos durante el resto del programa. Los programas paralelos para el ordenamiento rápido y la transformada rápida de Fourier exhiben este tipo de comportamiento.

Con una memoria con consistencia secuencial pura, ambos procesos harían una pausa en la barrera, como se muestra en la figura 6-32(c). La barrera se puede implantar al hacer que cada proceso envíe un mensaje a un controlador de barreras y que se bloquee hasta que llegue la respuesta. El controlador de barreras no envía las respuestas hasta que todos los procesos hayan llegado a la barrera.

Después de pasar la barrera, el proceso 1 inicia, guardando en *a*[0]. Después, el proceso 2 intenta guardar algo en *a*[1], provocando un fallo de página para traer la página que

contiene al arreglo. Después de esto, el proceso 1 intenta guardar en $a[2]$, provocando otro fallo, y así en lo sucesivo. Con un poco de mala suerte, cada uno de estos almacenamientos requeriría la transferencia de toda una página, lo que generaría demasiado tráfico.

Con la consistencia de liberación, la situación es como se muestra en la figura 6-32(d). De nuevo, ambos procesos pasan por vez primera la barrera. El primer almacenamiento en $a[0]$ obliga a crear una página gemela para el proceso 1. De manera similar, el primer almacenamiento en $a[1]$ provoca la creación de una página gemela para el proceso 2. En este punto no se requieren transferencias de páginas entre las máquinas. A partir de ahí, cada proceso puede realizar un almacenamiento en su propia copia privada de a a voluntad, sin provocar fallos de páginas.

Cuando cada proceso llega al segundo enunciado de barrera, se calculan las diferencias entre sus valores actuales de a y los valores originales (guardados en las páginas gemelas). Esto se envía a los demás procesos que se sabe están interesados en las páginas afectadas. Estos procesos, a su vez, pueden transferirlos a otros procesos interesados, pero desconocidos para la fuente de las modificaciones. Cada uno de los procesos receptores funde las modificaciones con las de su propia versión. Pueden ocurrir conflictos en un error de tiempo de ejecución.

Después de que un proceso ha informado de las modificaciones de esta manera, envía un mensaje al controlador de barreras y espera una respuesta. Cuando todos los procesos envían sus actualizaciones y llegado a la barrera, el controlador de barreras envía las respuestas, y todos pueden continuar. De esta manera, el tráfico de páginas sólo es necesario al llegar a una barrera.

Directorios

Munin utiliza directorios para localizar las páginas que contienen variables compartidas. Cuando ocurre un fallo por una referencia a una variable compartida, Munin dispersa las direcciones virtuales que provocaron el fallo, con el fin de determinar la entrada de la variable en el directorio de variables compartidas. A partir de esa entrada, ve la categoría de la variable, si existe una copia local, y quién es el probable propietario. Las páginas de escritura compartida no necesariamente tienen un propietario. Para una variable compartida convencional, el propietario es el último proceso que adquirió el acceso para escritura. Para una variable compartida migratoria, el propietario es el proceso que la posee por el momento.

Se lleva un registro del posible propietario mediante el siguiente algoritmo. Cuando se inicia un proceso en Munin, el proceso raíz posee todas las variables compartidas. Cuando el proceso P_1 hace una referencia posterior a una variable compartida, ocurre un fallo, lo que genera un mensaje a la raíz para solicitarla. La raíz proporciona la página deseada y anota que ahora P_1 es el propietario. Si P_2 solicita la página, la raíz le indica que probablemente P_1 sea el propietario. Cuando P_2 solicita la variable a P_1, la obtiene. Si P_2 desea escribir o la página es migratoria, P_2 se convierte en el nuevo propietario y P_1 registra este hecho. El estado de los probables propietarios en este momento se muestra en la figura 6-33(a) para una variable migratoria o en la que se puede escribir.

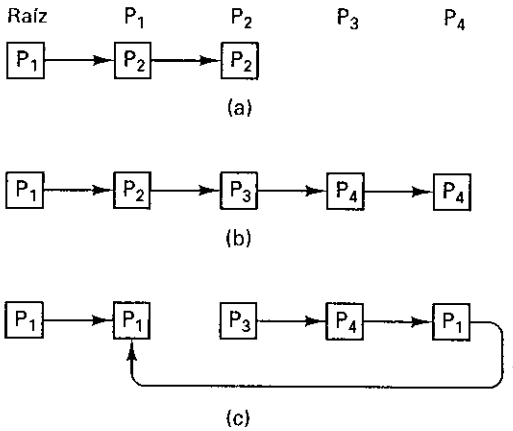


Figura 6-33. En cada instante, un proceso puede pensar que otro proceso es el probable propietario de alguna página.

Supongamos ahora que P_3 y P_4 solicitan de manera sucesiva la página. Ellos también siguen la cadena, lo que produce la figura 6-33(b). Ahora, P_1 necesita de nuevo la variable. Puesto que piensa que P_2 la tiene, envía un mensaje a P_2, sólo para ver que P_2 piensa que P_3 es el propietario. Después de seguir la cadena, P_1 obtiene la página y la cadena se ve como en la figura 6-33(c). De esta manera, cada proceso tendrá de forma eventual una idea del probable propietario, y sigue toda la cadena para determinar al propietario real.

Las direcciones también se utilizan para llevar un registro del conjunto de copias. Sin embargo, estos conjuntos no tienen que ser perfectamente consistentes. Por ejemplo, supongamos que P_1 y P_2 contienen cada uno alguna variable de escritura compartida y que cada uno de ellos conoce la existencia del otro. Entonces P_3 solicita al propietario, P_1, una copia, y la obtiene. P_3 registra a P_1 como poseedor de una copia, pero no se lo dice a P_2. Posteriormente, P_4, quien piensa que P_2 es el propietario, adquiere una copia, lo cual actualiza el conjunto de copias de P_2 para incluir a P_4. En este momento, ninguno de los procesos tiene una lista completa de quién tiene la página.

Sin embargo, es posible mantener la consistencia. Imaginemos que P_4 libera ahora una cerradura, de modo que envía las actualizaciones a P_2. El mensaje de reconocimiento de P_2 a P_4 contiene una nota diciendo que P_1 también tiene una copia. Cuando P_4 entra en contacto con P_1, conoce la existencia de P_3. De esta manera, de forma eventual descubre todo el conjunto de copias, de modo que todas las copias se pueden actualizar y pueden actualizar a su propio conjunto de copias.

Para reducir el costo al enviar actualizaciones a procesos que ya no están interesados en ciertas páginas de escritura compartida particular, se utiliza un algoritmo basado en un cronómetro. Si un proceso tiene una página, no hace referencia a ella en cierto lapso y recibe una actualización, elimina esa página. La siguiente vez que recibe una actualización de la página eliminada, el proceso indica al proceso que la actualiza que ya no posee una copia, por lo que el actualizador puede reducir el tamaño de su conjunto de copias. La cadena del

probable propietario se utiliza para denotar la copia del último recurso, que no se elimina sin determinar un nuevo propietario o escribirla en el disco. Este mecanismo garantiza que una página no sea eliminada por todos los procesos y que se pierda por ello.

Sincronización

Munin mantiene un segundo directorio con las variables de sincronización. Éstas se localizan de manera similar a la forma de localizar las variables compartidas ordinarias. Desde un punto de vista conceptual, las cerraduras actúan como si fueran centralizadas, pero de hecho se utiliza una implantación distribuida para evitar el envío de tráfico excesivo a cualquier máquina.

Cuando un proceso adquiere una cerradura, primero verifica si él mismo posee la cerradura. Si lo hace y la cerradura está libre, se otorga la solicitud. Si la cerradura no es local, se localiza mediante el directorio de sincronización, el cual mantiene un registro del probable propietario. Si la cerradura está libre, se otorga. Si no está libre, el solicitante se agrega al final de la cola. De esta manera, cada proceso conoce la identidad del proceso siguiente en la cola. Al liberar una cerradura, el propietario la transfiere al siguiente proceso de la lista.

Las barreras se implantan mediante un servidor central. Cuando se crea una barrera, recibe la cantidad de procesos que están esperándola antes de que puedan ser liberados. Cuando un proceso concluye cierta fase de su cálculo puede enviar un mensaje al servidor de barreras solicitando que lo espere. Cuando el número solicitado de procesos esté esperando, se envía un mensaje a todos ellos para liberarlos.

6.5.2. Midway

Midway es un sistema con memoria distribuida compartida cuya base consiste en compartir las estructuras de datos individuales. Es similar a Munin en varios aspectos, pero tiene ciertas características nuevas, interesantes y propias. Su objetivo es permitir que los programas multiprocesador existentes y los nuevos se ejecuten de manera eficiente en las multicomputadoras, con unos cuantos ligeros cambios en el código. Para mayor información acerca de Midway, véase (Bershad y Zekauskas, 1991; y Bershad *et al.*, 1993).

Los programas en Midway son básicamente programas convencionales escritos en C, C++, o ML, con cierta información adicional proporcionada por el programador. Los programas de Midway utilizan el paquete de hilos C de Mach para expresar el paralelismo. Un hilo puede bifurcarse en uno o más hilos. Los hijos se ejecutan en paralelo con el hilo padre y entre sí, y en potencia se ejecuta un hilo en una máquina diferente (es decir, cada hilo como proceso por separado). Todos los hilos comparten el mismo espacio lineal de direcciones, el cual contiene los datos compartidos y los datos privados. El trabajo de Midway es mantener las variables compartidas consistentes de manera eficiente.

Consistencia de entrada

La consistencia se mantiene pidiendo que todos los accesos a las variables compartidas y las estructuras de datos se realicen dentro de cierto tipo específico de sección crítica

conocido como el sistema de tiempo de ejecución de Midway. Cada una de estas secciones críticas es protegida por una variable de sincronización especial, por lo general, una cerradura, pero también posiblemente una barrera. Cada variable compartida a la que se tiene acceso en una sección crítica debe estar asociada de manera explícita con la cerradura (o barrera) de esa sección crítica mediante una llamada a procedimiento. De esta forma, cuando se entra o sale de una sección crítica, Midway conoce con precisión las variables compartidas que podrían tener un acceso en un momento dado.

Midway soporta la consistencia de entrada, que funciona de la manera siguiente. Para tener acceso a los datos compartidos, un proceso entra por lo general a una región crítica llamando a un procedimiento de biblioteca, *lock*, con una variable de cerradura como parámetro. La llamada también especifica si se requiere una cerradura exclusiva o no exclusiva. Se necesita una cerradura exclusiva cuando hay que actualizar una o más variables compartidas. Si las variables compartidas sólo serán leídas, pero no modificadas, basta una cerradura no exclusiva, la cual permite que varios procesos entren a la misma región crítica al mismo tiempo. No hay daño alguno, puesto que ninguna de las variables compartidas se puede modificar.

Al llamar a *lock*, el sistema de tiempo de ejecución de Midway adquiere la cerradura, y al mismo tiempo, actualiza todas las variables compartidas asociadas con esa cerradura. Este hecho puede requerir el envío de mensajes a otros procesos para obtener los valores más recientes. Al recibir todas las respuestas, se otorga la cerradura (suponiendo que no haya conflictos) y el proceso inicia su ejecución de la región crítica. Cuando el proceso termina la sección crítica, libera la cerradura. A diferencia de la consistencia de liberación, no se realiza comunicación alguna al momento de la liberación; es decir, las variables compartidas modificadas *no* se envían a las demás máquinas que utilizan las variables compartidas. Los datos se transfieren sólo cuando uno de sus procesos adquiere posteriormente una cerradura y solicita los valores actuales.

Para que funcione la consistencia de entrada, Midway requiere que los programas tengan tres características que no tienen los programas de multiprocesador:

1. Las variables compartidas deben declararse mediante la nueva palabra reservada *shared*.
2. Cada variable compartida debe estar asociada con una cerradura o barrera.
3. Sólo se puede tener acceso a las variables compartidas dentro de las secciones críticas.

Para hacer esto se requiere un esfuerzo adicional del programador. Si no se cumplen por completo estas reglas, no se genera un mensaje de error y el programa produce resultados incorrectos. Puesto que la programación de esta forma es propensa a errores, en especial al ejecutar antiguos programas multiprocesadores que nadie entiende en realidad, Midway también soporta la consistencia secuencial y la de liberación. Estos modelos requieren menor información detallada para la operación correcta.

La información adicional requerida por Midway debe pensarse como parte del contrato entre el software y la memoria que ya estudiamos bajo la consistencia. En efecto, si el programa está de acuerdo con ciertas reglas conocidas de antemano, la memoria promete funcionar. En caso contrario, todas las apuestas serán en contra.

Implantación

Cuando se entra en una sección crítica, el sistema de tiempo de ejecución de Midway adquiere primero la cerradura correspondiente. Para obtener una cerradura exclusiva, es necesario localizar al propietario de la cerradura, que es el último proceso que la adquirió en forma exclusiva. Cada proceso lleva el registro del probable propietario, de la misma forma en que lo hacen IVY y Munin y sigue la cadena distribuida de propietarios sucesivos hasta que encuentra el actual. Si este proceso no está utilizando en la actualidad la cerradura, se transfiere la propiedad. Si la cerradura está en uso, el proceso solicitante espera hasta que la cerradura está libre. Para adquirir una cerradura en modo no exclusivo, basta establecer contacto con cualquier proceso que lo posea en un instante dado. Las barreras se controlan mediante un controlador centralizado de barreras.

Al momento en que se adquiere la cerradura, el proceso que la adquiere actualiza su copia de todas las variables compartidas. En el protocolo más sencillo, el propietario anterior sólo las envía todas. Sin embargo, Midway utiliza una optimización para reducir la cantidad de datos por transferir. Supongamos que esta adquisición se realiza en el instante T_1 y que la adquisición anterior realizada por el mismo proceso se realizó en el instante T_0. Sólo se transfieren las variables modificadas desde T_0, puesto que el adquiriente ya tiene el resto.

Esta estrategia indica el problema de determinar la forma en que el sistema lleva el registro de lo modificado y del momento en que se modificó. Para llevar un registro de las variables compartidas que han sido modificadas, se puede utilizar un compilador especial que genere un código para mantener una tabla de tiempos de ejecución con una entrada en ella para cada variable compartida en el programa. Siempre que se actualice una variable compartida, este cambio se nota en la tabla. Si no se dispone de este compilador especial, se utiliza el hardware MMU para detectar las escrituras a los datos compartidos, como en Munin.

El momento de cada cambio se registra mediante un protocolo con marcas de tiempo basado en la relación "ocurre antes de" de Lamport (1978). Cada máquina mantiene un reloj lógico, que se incrementa cada vez que se envía un mensaje y se incluye en éste. Al llegar un mensaje, el receptor ajusta su reloj lógico con el máximo del reloj del emisor y su propio valor actual. Con estos relojes, el tiempo se divide efectivamente en intervalos definidos por las transmisiones de mensajes. Cuando se realiza una adquisición, el proceso adquiriente especifica la hora de su adquisición anterior y solicita todas las variables compartidas importantes que han cambiando desde entonces.

El uso de la consistencia de entrada implantada de esta forma tiene en potencia un excelente desempeño, puesto que la comunicación sólo ocurre cuando un proceso realiza una adquisición. Además, sólo se necesitan transferir aquellas variables compartidas no actualizadas. En particular, si un proceso entra a una región crítica, sale de ella y entra de

nuevo, no se necesita comunicación alguna. Este patrón es común en la programación paralela, de modo que la ganancia en este caso es esencial. El precio que se paga por este desempeño es una interfaz con el programador que es más compleja y propensa a errores que la utilizada por los demás modelos de consistencia.

6.6. MEMORIA COMPARTIDA DISTRIBUIDA BASADA EN OBJETOS

Los sistemas DSM basados en páginas que hemos estudiado hasta ahora utilizan el hardware MMU para señalar los accesos a las páginas faltantes. Aunque este método tiene ciertas ventajas, también tiene sus desventajas. En particular, en muchos lenguajes de programación, los datos se organizan en objetos, paquetes, módulos u otras estructuras de datos, cada una de las cuales tiene una existencia independiente de las demás. Si un proceso hace referencia a una parte de un objeto, en muchos casos se necesitará todo el objeto, por lo que tiene sentido transportar los datos a través de la red mediante unidades de objetos, no unidades de páginas.

El método de variables compartidas, utilizado en Munin y Midway, es un paso en la dirección de organizar la memoria compartida de manera más estructurada, pero sólo es un primer paso. En ambos sistemas, el programador proporciona información acerca de las variables que se comparten y las que no; además, proporciona la información del protocolo en Munin y de asociación en Midway. Los errores en estas anotaciones pueden tener consecuencias graves.

Si se avanza en la dirección de un modelo de programación de alto nivel, la programación de los sistemas DSM puede ser más sencilla y menos propensa a errores. El acceso a las variables compartidas y la sincronización mediante éstas también se integra de manera más clara. En algunos casos, también se introducen ciertas optimizaciones que son más difíciles de realizar en un modelo de programación menos abstracto.

6.6.1. Objetos

Un **objeto** es una estructura de datos encapsulada definida por el programador, como se muestra en la figura 6-34. Consta de datos internos, el **estado del objeto** y procedimientos, llamados **métodos** u **operaciones**, que operan sobre el estado del objeto. Para tener acceso u operar sobre el estado interno, el programa llama a alguno de los métodos. El método puede modificar el estado interno, regresar (parte de) el estado, o algo más. No se permite el acceso directo al estado interno. Esta propiedad es llamada **ocultamiento de la información** (Parnas, 1972). El hecho de obligar a que todas las referencias a un dato del objeto pasen por los métodos ayuda a estructurar el programa de manera modular.

En una memoria distribuida compartida basada en objetos, los procesos de varias máquinas comparten un espacio abstracto ocupado por objetos compartidos, como se muestra en la figura 6-35. La localización y administración de los objetos es controlada de manera automática por el sistema de tiempo de ejecución. Este modelo contrasta con el de los

sistemas DSM basados en páginas, como IVY, que sólo proporcionan una memoria lineal en bruto, con bytes desde 0 hasta algún máximo.

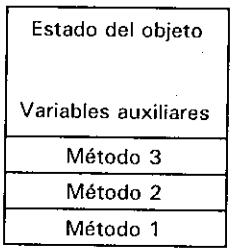


Figura 6-34. Un objeto.

Cualquier proceso llama a los métodos de cualquier objeto, sin importar la posición del proceso o del objeto. El sistema operativo y el sistema de tiempo de ejecución se encargan de que funcione la llamada a un método sin importar la posición del proceso o del objeto. Puesto que los procesos no tienen un acceso directo al estado interno de cualquiera de los objetos compartidos, aquí son posibles varias optimizaciones que no eran posibles (o al menos eran más difíciles) en la DSM basada en páginas. Por ejemplo, puesto que los accesos al estado interno están controlados de manera estricta, puede ser posible relajar el protocolo de consistencia de la memoria sin que el programador se entere de ello.

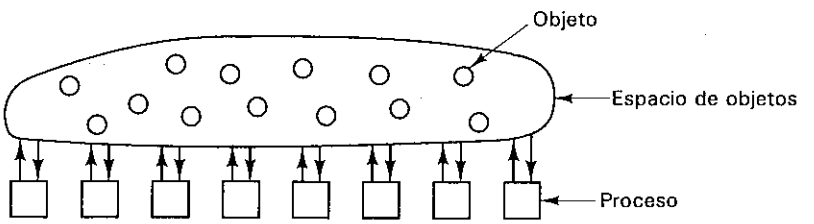


Figura 6-35. En una memoria distribuida compartida basada en objetos, los procesos se comunican al llamar a métodos sobre los objetos compartidos.

Una vez que se ha tomado la decisión de estructurar una memoria compartida como colección de objetos separados en vez de un espacio lineal de direcciones, hay que elegir otras opciones. Es probable que el aspecto más importante sea si los objetos deben duplicarse o no. Si no se utiliza la réplica, todos los accesos a un objeto pasarán por una copia, lo que es sencillo, pero puede conducir a un desempeño pobre. Al permitir que los objetos emigren de una máquina a otra, según sea necesario, podría reducirse la pérdida de desempeño al mover los objetos adonde se necesiten.

Por otro lado, si los objetos se duplican, ¿qué debería hacerse cuando se actualice una copia? Un método consiste en invalidar las demás copias, de modo que sólo permanezca

la copia actualizada. Se podrían crear copias posteriores, según la demanda. Una opción alternativa consiste en no invalidar las copias, sino actualizarlas. La DSM con variables compartidas también tiene esta opción, pero para la DSM basada en páginas, la invalidación es la única opción factible. De manera similar, la DSM basada en objetos, como la DSM con variables compartidas, permiten evitar compartir falsamente.

En resumen, la memoria distribuida compartida basada en objetos ofrece tres ventajas sobre los otros métodos:

1. Es más modular que las demás técnicas.
2. La implantación es más flexible debido al control de los accesos.
3. La sincronización y el acceso se pueden integrar de manera limpia.

La DSM basada en objetos también tiene desventajas. En primer lugar, no se puede utilizar para ejecutar antiguos programas multiprocesadores de "escritorio polvoso", los cuales suponen la existencia de un espacio lineal compartido de direcciones en el que cada proceso puede leer o escribir al azar. Sin embargo, como los multiprocesadores son relativamente nuevos, los programas multiprocesadores por los que habría que preocuparse son muy pocos.

Una segunda desventaja potencial es que, puesto que todos los accesos a los objetos compartidos deben realizarse mediante llamadas a los métodos de los objetos, se incurre en un costo adicional que no estaba presente con las páginas compartidas de acceso directo. Por otro lado, muchos expertos en ingeniería de software recomiendan los objetos como una herramienta de estructuración, incluso en máquinas individuales, y aceptan el costo como algo que vale su precio.

Adelante estudiaremos dos ejemplos un tanto diferentes de DSM basada en objetos: Linda y Orca. También existen otros sistemas distribuidos basados en objetos, como Amber (Chase *et al.*, 1989), Emerald (Jul *et al.*, 1988) y COOL (Lea *et al.*, 1993).

6.6.2. Linda

Linda proporciona los procesos en varias máquinas con una memoria distribuida compartida muy estructurada. El acceso a esta memoria es mediante un pequeño conjunto de operaciones primitivas que se agregan a los lenguajes existentes, como C y FORTRAN para formar lenguajes paralelos; en este caso, C-Linda y FORTRAN-Linda. En la siguiente descripción, nos centraremos en C-Linda, pero las diferencias conceptuales entre las variantes son pequeñas. Se puede encontrar más información acerca de Linda en (Carriero y Gelernter, 1986, 1989; y Gelernter, 1985).

Este método tiene varias ventajas sobre un nuevo lenguaje. Una de las principales ventajas es que los usuarios no tienen que aprender un nuevo lenguaje. Esta ventaja no debe ser subestimada. Una segunda ventaja es la sencillez: el cambio de un lenguaje X a X-Linda se puede lograr al agregar unas cuantas primitivas a la biblioteca y adaptando el preprocesador de Linda que alimenta con programas Linda al compilador. Por último, el sistema

Linda es muy portable a través de varios sistemas operativos y arquitecturas de máquina y ha sido implantado en muchos sistemas distribuidos y paralelos.

Espacio de n-adas

El concepto unificador detrás de Linda es el de un **espacio de n-adas** abstracto. El espacio de n-adas es global en todo el sistema, y los procesos de cualquier máquina pueden insertar o eliminar n-adas en el espacio de n-adas sin importar la forma o el lugar donde estén guardados. Para el usuario, el espacio de n-adas se ve como una enorme memoria global compartida, como vimos en la figura 6-35. La implantación real puede implicar a muchos servidores en varias máquinas, lo que describiremos posteriormente.

Una **n-ada** es como una estructura en C o un registro en Pascal. Consta de uno o más campos, cada uno de los cuales es un valor de cierto tipo soportado por el lenguaje base. Para C-Linda, los tipos de los campos incluyen a los enteros, *longint*, los números de punto flotante, así como los tipos compuestos, como los arreglos (incluyendo las cadenas) y las estructuras (pero no a otras n-adas). La figura 6-36 muestra tres n-adas como ejemplo.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Carolyn", "Elinor")
```

Figura 6-36. Tres n-adas de Linda.

Operaciones sobre las n-adas

Linda no es un sistema basado en objetos por completo generales, puesto que sólo proporciona una cantidad fija de operaciones integradas y no hay forma de definir otras nuevas. Se proporcionan cuatro operaciones sobre las n-adas. La primera, *out*, coloca una n-ada en el espacio de n-adas. Por ejemplo,

out("abc",2,5);

coloca la n-ada ("abc",2,5) en el espacio de n-adas. Los campos de *out* son por lo general constantes, variables o expresiones, como en

out ("matrix-1", i, j, 3.14);

que coloca una n-ada con cuatro campos, donde el segundo y el tercero quedan determinados por los valores actuales de las variables *i* y *j*.

Las n-adas se recuperan del espacio de n-adas mediante la primitiva *in*. Se hace referencia a ellas por su contenido más que por su nombre o su dirección. Los campos de *in* pueden ser expresiones o parámetros formales. Por ejemplo, consideremos

in("abc",2,? i);

Esta operación "busca" en el espacio de n-adas una n-ada formada por la cadena "abc", el entero 2 y un tercer campo que contenga cualquier entero (suponiendo que *i* es un entero). Si lo encuentra, elimina la n-ada del espacio de n-adas y la variable *i* recibe el valor del

tercer campo. La concordancia y eliminación son atómicas, de modo que si dos procesos ejecutan la misma operación *in* de manera simultánea, sólo una de ellas tendrá éxito, a menos que estén presentes dos o más n-adas idénticas. El espacio de n-adas podría incluso contener varias copias de la misma n-ada.

El algoritmo de concordancia utilizado por *in* es directo. Los campos de la primitiva *in*, llamados **plantilla**, se comparan (conceptualmente) con los campos correspondientes de cada n-ada en el espacio de n-adas. Ocurre una concordancia cuando se cumplen las siguientes tres condiciones:

1. La plantilla y la n-ada tienen el mismo número de campos.
2. Los tipos de los campos correspondientes son iguales.
3. Cada constante o variable en la plantilla concuerda con su campo en la n-ada.

Los parámetros formales, indicados mediante un signo de interrogación seguido por un nombre de variable o tipo, no participan en la concordancia (excepto por la verificación del tipo), aunque los que contengan un nombre de variable son asignados después de una concordancia con éxito.

Si no existe una n-ada concordante, el proceso que realiza la llamada se suspende hasta que otro proceso inserta la n-ada necesaria, en cuyo momento revive el proceso que hizo la llamada y obtiene la nueva n-ada. El hecho de que los procesos se bloqueen y eliminen su bloqueo de manera automática significa que si un proceso está por colocar una n-ada y otro está a punto de eliminarla, no importa lo que ocurra primero. La única diferencia es que si el *in* se ejecuta antes que el *out*, sólo habrá un ligero retraso hasta que la n-ada esté disponible para su eliminación.

El hecho de que los procesos se bloqueen cuando no esté presente la n-ada necesaria se puede utilizar de varias formas. Por ejemplo, se puede utilizar para implantar los semáforos. Para crear o realizar un *UP*(*V*) sobre un semáforo *S*, un proceso puede ejecutar

out("semaphore S");

Para realizar un *DOWN*(*P*), ejecuta

in("semaphore S");

El estado del semáforo *S* queda determinado por el número de n-adas ("semaphore S") en el espacio de n-adas. Si no existe ninguna, cualquier intento por obtener una se bloqueará hasta que otro proceso la proporcione.

Además de *out* e *in*, Linda tiene también la primitiva *read*, que es igual a *in*, excepto que no elimina la n-ada del espacio de n-adas. También existe una primitiva *eval*, que hace que sus parámetros sean evaluados en paralelo y que la n-ada resultante se deposite en el espacio de n-adas. Este mecanismo se puede utilizar para realizar un cálculo arbitrario. Ésta es la forma en que se crean los proceso paralelos en Linda.

Un paradigma común en Linda es el **modelo del trabajador duplicado**. Este modelo se basa en la idea de una **bolsa de tareas** con muchas tareas por realizar. El proceso principal inicia al ejecutar un ciclo que contiene

out("task-bag", job);

en donde la descripción de una tarea diferente es eliminada del espacio de n-adas en cada iteración. Cada trabajador comienza obteniendo una n-ada con la descripción de una tarea utilizando

in ("task-bag", ?job);

que se realiza entonces. Al terminar, obtiene otra. El trabajo nuevo también se puede colocar en la bolsa de tareas durante la ejecución. De esta manera sencilla, el trabajo se divide de manera dinámica entre los trabajadores, y cada trabajador se mantiene ocupado todo el tiempo, con poco costo adicional.

En ciertos aspectos, Linda es similar a Prolog, en el cual se basa, aunque de manera vaga. Ambos soportan un espacio abstracto que funciona como una especie de base de datos. En Prolog, el espacio contiene hechos y reglas; en Linda contiene n-adas. En ambos casos, los procesos proporcionan plantillas que se comparan con el contenido de la base de datos.

A pesar de estas analogías, los dos sistemas también difieren de maneras significativas. Prolog tenía la intención de programar aplicaciones de inteligencia artificial en un procesador, mientras que Linda pretendía una programación general en las multicomputadoras. Prolog tiene un complejo esquema para comparación de patrones que implica la unificación y el *backtracking*; el algoritmo de concordancia de Linda es mucho más sencillo. En Linda, una concordancia con éxito elimina la n-ada concordante del espacio de n-adas; en Prolog no. Por último, un proceso en Linda que no puede localizar una n-ada se bloquea, lo que forma la base de la sincronización entre los procesos. En Prolog, no existen procesos y los programas nunca se bloquean.

Implantación de Linda

Son posibles las implantaciones eficientes de Linda en varios tipos de hardware. Adelante analizaremos algunas de las más interesantes. Para todas las implantaciones, un preprocesador analiza el programa en Linda, extrae la información útil y la convierte al lenguaje básico en que debe estar (por ejemplo, la cadena "? int" no es permitida como un parámetro en C o en FORTRAN). El trabajo real de inserción y eliminación de las n-adas del espacio de n-adas se realiza durante la ejecución del sistema de tiempo de ejecución de Linda.

Una implantación eficiente de Linda debe resolver dos problemas:

1. La forma de simular el direccionamiento asociativo sin una búsqueda masiva.

2. La forma de distribuir las n-adas entre las máquinas y la forma de localizarlas posteriormente.

La clave para ambos problemas es observar que cada n-ada tiene una **firma de tipo**, que consta de la lista (ordenada) de los tipos de sus campos. Además, por convención, el primer campo de cada n-ada es por lo general una cadena que divide de manera eficaz al espacio

de n-adas en subespacios ajenos nombrados por la cadena. La separación del espacio de n-adas en subespacios, cada una de cuyas n-adas tiene la misma firma de tipo y el mismo primer campo, simplifica la programación y permite la existencia de ciertas optimizaciones.

Por ejemplo, si el primer parámetro de una llamada *in* o *out* es una cadena de literales, es posible determinar al momento de la compilación el subespacio sobre el que opera la llamada. Si el primer parámetro es una variable, la determinación se realiza en el momento de la ejecución. En ambos casos, esta separación significa que sólo hay que buscar en una fracción del espacio de n-adas. La figura 6-37 muestra cuatro n-adas y cuatro plantillas. Juntas forman cuatro subespacios. Para cada *out* o *in*, es posible determinar en el momento de la compilación el subespacio y el servidor de n-adas necesarios. Si la cadena inicial era una variable, la determinación del subespacio correcto tendría que retrasarse hasta el tiempo de ejecución.

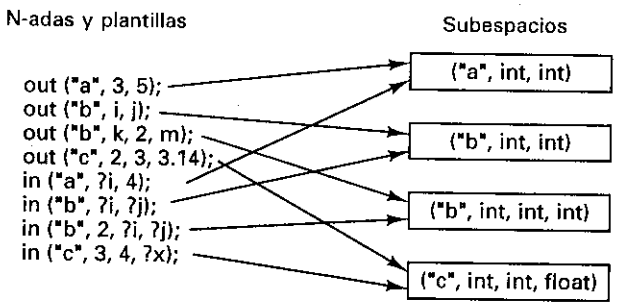


Figura 6-37. Las n-adas y las plantillas se pueden asociar con los subespacios.

Además, cada subespacio se organiza como una tabla de dispersión, utilizando su *i*-ésimo campo de la n-ada como la clave de dispersión. Si el campo *i* es una constante o una variable (pero no un parámetro formal), se puede ejecutar un *in* o un *out* al calcular la función de dispersión del *i*-ésimo campo para determinar la posición de la tabla adonde pertenece la n-ada. El conocimiento del subespacio y de la posición en la tabla eliminan la búsqueda. Si el *i*-ésimo campo de cierto *in* es un parámetro formal, la dispersión no es posible, de modo que se necesita una búsqueda completa en el subespacio, excepto por algunos casos especiales. Sin embargo, si se elige de manera adecuada el campo sobre el cual dispersar, el preprocesador puede evitar por lo general la búsqueda, la mayor parte del tiempo. También se pueden utilizar otras organizaciones en subespacios además de la dispersión para casos particulares (por ejemplo, una cola con un escritor y un lector).

También se utilizan otras optimizaciones adicionales. Por ejemplo, el esquema de dispersión descrito arriba distribuye las n-adas de un subespacio dado en charolas, para restringir la búsqueda a una sola charola. Es posible colocar las diferentes charolas en diferentes máquinas, para difundir la carga más ampliamente y para aprovechar la localidad. Si la función de dispersión es la clave módulo el número de máquinas, el número de charolas crece de manera lineal con el tamaño del sistema.

Ahora examinaremos las diversas técnicas de implantación para diferentes tipos de hardware. En un multiprocesador, los subespacios de n-adas se pueden implantar mediante tablas de dispersión en la memoria global, una por cada subespacio. Cuando se ejecuta un *in* o un *out*, se cierra el subespacio correspondiente, se inserta o elimina la n-ada y se elimina la cerradura del subespacio.

En una multicomputadora, la mejor elección depende de la arquitectura de comunicación. Si se dispone de una transmisión confiable, un candidato serio consiste en duplicar todos los subespacios por completo en todas las máquinas, como se muestra en la figura 6-38. Cuando se realiza un *out*, la nueva n-ada se transmite y entra en el subespacio adecuado en cada máquina. Para realizar un *in*, se busca en el subespacio local. Sin embargo, puesto que la conclusión con éxito de un *in* requiere la eliminación de la n-ada del espacio de n-adas, se necesita un protocolo de eliminación para eliminarla de todas las máquinas. Para evitar la competencia y los bloqueos, se puede utilizar un protocolo de compromiso de dos fases.

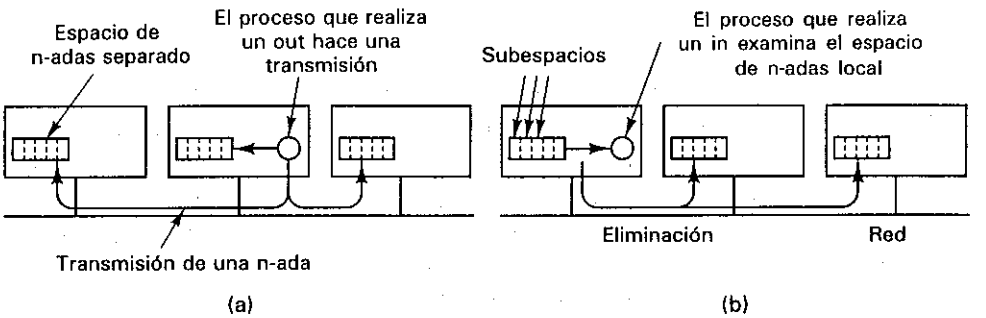


Figura 6-38. El espacio de n-adas se puede duplicar en todas las máquinas. Las líneas punteadas indican la separación del espacio de n-adas en subespacios. (a) Las n-adas se transmiten en *out*. (b) Los *in* son locales, pero las eliminaciones deben ser transmitidas.

Este diseño es directo, pero podría no escalarse bien cuando el tamaño del sistema aumente, puesto que cada n-ada debe guardarse en cada máquina. Por otro lado, el tamaño total del espacio de n-adas es con frecuencia modesto, de modo que no surgen problemas, excepto en sistemas enormes. El sistema S/Net Linda utiliza este método, pues S/Net tiene una transmisión mediante un bus rápido y confiable, paralelo basado en palabras (Carriero y Gelernter, 1986).

El diseño inverso consiste en realizar los *out* de manera local, guardando la n-ada sólo en la máquina que lo generó, como se muestra en la figura 6-39. Para realizar un *in*, un proceso debe transmitir la plantilla. Cada receptor verifica entonces si tiene una concordancia, enviando una respuesta en caso afirmativo.

Si la n-ada no está presente o si la transmisión no es recibida en la máquina que contiene a la n-ada, la máquina solicitante vuelve a transmitir la solicitud *ad infinitum*, aumentando el intervalo entre las transmisiones, hasta que se materializa una n-ada adecuada y puede

satisfacer la solicitud. Si se envían dos o más n-adas, éstas se consideran como *out* locales y las n-adas se mueven efectivamente de las máquinas que las tenían a la máquina que hizo la solicitud. De hecho, el sistema de tiempo de ejecución puede mover n-adas por su cuenta para balancear la carga. Carriero *et al.*, (1986) utilizaron este método para implantar Linda en una LAN.

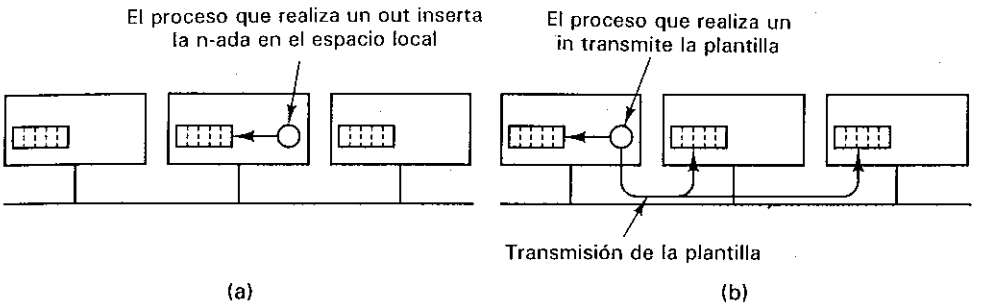


Figura 6-39. Espacio de n-adas no duplicado. (a) Se realiza un *out* de manera local. (b) Un *in* requiere la transmisión de la plantilla para encontrar una n-ada.

Estos dos métodos se combinan para producir un sistema con réplica parcial. Un ejemplo sencillo consiste en imaginar que todas las máquinas forman lógicamente un rectángulo, como se muestra en la figura 6-40. Cuando un proceso en una máquina, *A*, realiza un *out*, transmite (o envía mediante un mensaje puntual) la n-ada a todas las máquinas en su fila de la matriz. Cuando un proceso en una máquina, *B*, desea realizar un *in* transmite la plantilla a todas las máquinas en su columna. Debido a la geometría, siempre existirá exactamente una máquina que vea la n-ada y la plantilla (*C* en este ejemplo) y esa máquina realiza la concordancia y envía la n-ada al proceso que la solicita. Krishnaswamy (1991) utilizó este método para un coprocesador Linda en hardware.

Por último, consideremos la implantación de Linda en sistemas que no tienen capacidad de transmisión alguna (Bjornson, 1993). La idea básica es separar el espacio de n-adas en subespacios ajenos, creando primero una partición para cada firma de tipo, y dividiendo después cada una de estas particiones de nuevo con base en el primer campo. En potencia, cada una de las particiones resultantes puede ir a una máquina distinta, controlada por su propio servidor de n-adas, para difundir la carga. Cuando se realiza un *in* o un *out*, se determina la partición necesaria, y se envía un mensaje a esa máquina, para depositar una n-ada o para recuperar una.

La experiencia con Linda muestra que la memoria distribuida compartida se puede controlar de manera radicalmente diferente al movimiento de las páginas completas, como en los sistemas basados en páginas que estudiamos con anterioridad. También es algo distinto de las variables compartidas con consistencia de liberación o de entrada. Conforme los sistemas futuros sean más grandes y poderosos, los métodos novedosos como éste pueden conducir a nuevas perspectivas acerca de la forma de programar estos sistemas de manera más sencilla.

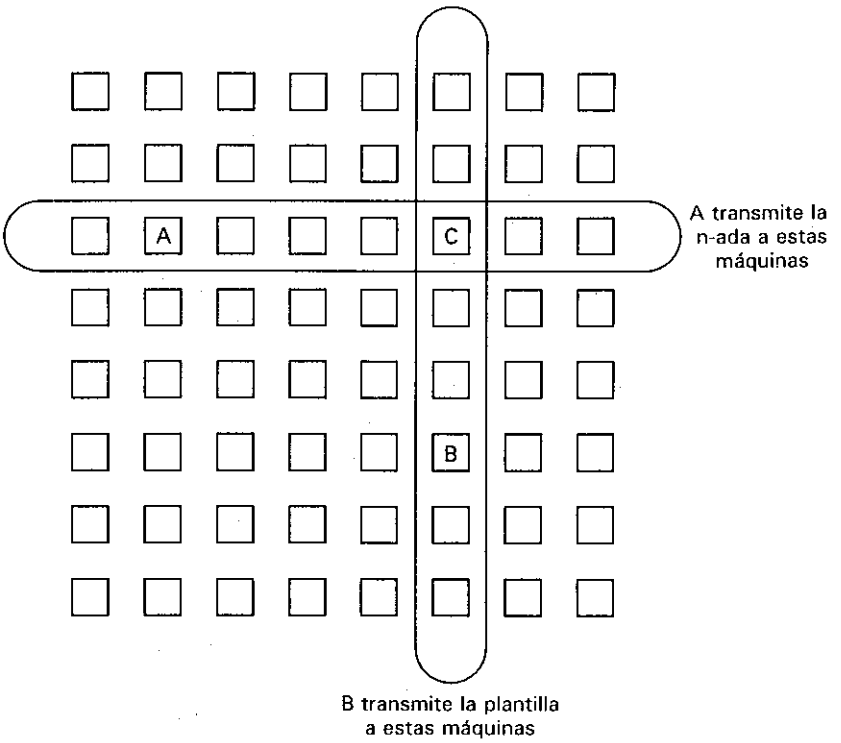


Figura 6-40. Transmisión parcial de n-adas y plantillas.

6.6.3. Orca

Orca es un sistema de programación paralela que permite a los procesos de diferentes máquinas el acceso controlado a una memoria distribuida compartida consistente en objetos protegidos (Bal, 1991; y Bal *et al.,* 1990, 1992). Estos objetos se pueden pensar como forma más poderosa (y más compleja) de las n-adas de Linda, que soportan operaciones arbitrarias en vez de solo *in* y *out*. Otra diferencia es que las n-adas de Linda se crean al vuelo, durante la ejecución, en grandes volúmenes, lo que no ocurre en Orca. Las n-adas de Linda se utilizan principalmente para la comunicación, mientras que los objetos de Orca se utilizan también para los cálculos y por lo general son más pesados.

El sistema Orca consta del lenguaje, el compilador, y el sistema de tiempo de ejecución, quien en realidad controla los objetos compartidos durante la ejecución. Aunque el lenguaje, el compilador y el sistema de tiempo de ejecución se diseñaron para trabajar juntos, el sistema de tiempo de ejecución es independiente del compilador y se podría utilizar también para otros lenguajes. Después de una introducción al lenguaje Orca, describiremos la forma

en que el sistema de tiempo de ejecución implanta una memoria distribuida compartida basada en objetos.

El lenguaje Orca

En algunos aspectos, Orca es un lenguaje tradicional cuyos enunciados secuenciales se basan de manera vaga en Modula-2. Sin embargo, es un lenguaje con tipos seguros, sin apuntadores ni sobrenombres. Las cotas de los arreglos se verifican al tiempo de ejecución (excepto cuando la verificación se realiza al momento de la compilación). Éstas y otras características similares eliminan o detectan muchos errores comunes de programación, como los almacenamientos forzados (wild stores) en la memoria. Las características del lenguaje han sido elegidas con cuidado para facilitar varias optimizaciones.

Dos características de Orca importantes para la programación distribuida son los objetos-dato (o simplemente **objetos**) compartidos y el enunciado **fork**. Un objeto es un tipo abstracto de dato, algo similar a un paquete en Ada®. Encapsula las estructuras de datos internas y los procedimientos escritos por el usuario, llamados **operaciones** (o **métodos**) para que operen sobre las estructuras de datos internas. Los objetos son pasivos; es decir, no contienen hilos a los cuales enviar mensajes. En vez de esto, los procesos tienen acceso a los datos internos de un objeto al llamar a sus operaciones. Los objetos no heredan propiedades de otros objetos, de modo que Orca se considera un lenguaje basado en objetos más que un lenguaje orientado a objetos.

Cada operación consta de una lista de parejas (protección, bloque de enunciados). Una protección es una expresión booleana que no contiene efectos colaterales, o la protección vacía, que es igual al valor *true*. Cuando se llama a una operación, se evalúan todas sus protecciones en orden indeterminado. Si todas son *false*, el proceso llamado se pospone hasta que alguna es *true*. Cuando se encuentra una protección evaluada como *true*, se ejecuta

```
Object implementation stack;                    # variable que indica la posición supe-
    top: integer;                               # rior de almacenamiento de la pila
    stack: array [integer 0..N-1] of integer;
    operation push (item: integer);             # función que no regresa nada
    begin
        stack [top] := item;                    # mete el elemento en la pila
        top := top + 1;                         # incrementa el apuntador de la pila
    end;
    operation pop(): integer;                   # función que regresa un entero
    begin
        guard top > 0 do                        # suspende si la pila está vacía
            top := top - 1;                     # decrementa el apuntador a la pila
            return stack [top];                 # regresa el elemento superior
        od;
    end;
begin
    top := 0;                                   # inicialización
end;
```

Figura 6-41. Un objeto stack simplificado, con datos internos y dos operaciones.

el bloque de enunciados que lo siguen. La figura 6-41 muestra un objeto *stack* con dos operaciones, *push* y *pop*.

Una vez definida *stack*, se pueden declarar variables de este tipo, como en

s, t: stack;

que crea dos objetos stack e inicializa la variable *top* de cada una en 0. La variable entera *k* se puede introducir en la pila *s* por medio del enunciado

s$push(k);

etcétera. La operación *pop* tiene una protección, de modo que un intento por sacar una variable de una pila vacía suspenderá al proceso que hace la llamada hasta que otro proceso introduzca algo en la pila.

Orca tiene un enunciado **fork** para crear un nuevo proceso en un procesador especificado por el usuario. El nuevo proceso ejecuta el procedimiento nombrado en el enunciado **fork**. Los parámetros, incluyendo los objetos, se pueden transferir al nuevo proceso, que es la forma en que se distribuyen los objetos entre las máquinas. Por ejemplo, el enunciado

for i **in** 1..n **do fork** foobar(s) **on** i; **od**;

genera un proceso nuevo en cada una de las máquinas 1 a *n*, ejecutando el proceso **foobar** en cada una de ellas. Como estos *n* nuevos procesos (y el padre) se ejecutan en paralelo, meten y sacan elementos de la pila compartida *s* como si estuvieran ejecutándose en un multiprocesador con memoria compartida. El sistema operativo debe encargarse de mantener la ilusión de la memoria compartida donde ésta realmente no existe.

Las operaciones sobre los objetos compartidos son atómicas y secuencialmente consistentes. El sistema garantiza que si varios procesos realizan operaciones sobre el mismo objeto compartido de manera casi simultánea, el efecto neto es que parece como si la operación se realizara de manera estrictamente secuencial, sin que una operación se efectúe hasta que termine la anterior.

Además, las operaciones aparecen en el mismo orden para todos los procesos. Por ejemplo, supongamos que deseamos aumentar el objeto *stack* con una operación nueva, *peek*, para inspeccionar el elemento superior de la pila. Entonces, si dos procesos independientes introducen 3 y 4 de manera simultánea, y todos los procesos utilizan posteriormente *peek* para examinar la parte superior de la pila, el sistema garantiza que todos los procesos verán 3 o todos los procesos verán 4. Una situación en la que algunos procesos vean 3 y otros vean 4 no puede ocurrir en un multiprocesador o una memoria distribuida compartida basada en páginas, y tampoco puede ocurrir en Orca. Si sólo se mantiene una copia de la pila, este efecto se logra de manera trivial, pero si la pila se duplica en todas las máquinas, se requiere un mayor esfuerzo, como se describe a continuación.

Aunque no lo hemos enfatizado, Orca integra los datos compartidos y la sincronización de una manera no presente en los sistemas DSM basados en páginas. Se necesitan dos tipos de sincronización en los programas paralelos. El primer tipo es la sincronización de la exclusión mutua, para evitar que dos procesos ejecuten la misma región crítica al mismo

tiempo. En Orca, cada operación sobre un objeto compartido es de hecho como una región crítica, pues el sistema garantiza que el resultado final es el mismo que se tendría al ejecutar todas las regiones críticas, una a la vez (es decir, de manera secuencial). En relación con esto, un objeto Orca es como forma distribuida de un monitor (Hoare, 1975).

El otro tipo de sincronización es la de condiciones, en la que un proceso se bloquea en espera de que se cumpla cierta condición. En Orca, la sincronización de condiciones se realiza mediante las protecciones. En el ejemplo de la figura 6-41, un proceso que intenta sacar un elemento de una pila vacía será suspendido hasta que la pila ya no esté vacía.

Administración de los objetos compartidos en Orca

La administración de objetos en Orca es controlada por el sistema de tiempo de ejecución. Funciona en las redes de transmisión (o multitransmisión) y en las redes puntuales. El sistema de tiempo de ejecución controla la réplica, migración, consistencia y llamadas a operaciones relativas a los objetos.

Cada objeto puede estar en uno de dos estados: única copia o duplicado. Un objeto en el estado de única copia sólo existe en una máquina. Un objeto duplicado está presente en todas las máquinas que contienen un proceso que lo utiliza. No se pide que todos los objetos estén en el mismo estado, de modo que algunos de los objetos utilizados por un proceso podrían estar duplicados, mientras que otros podrían ser copia única. Los objetos pueden pasar de un estado a otro durante la ejecución.

La gran ventaja de duplicar un objeto en cada máquina es que las lecturas se pueden llevar a cabo de manera local, sin tráfico o retraso en la red. Si un objeto no se duplica, todas las operaciones deben ser enviadas al mismo, y el proceso que hizo la llamada debe bloquearse en espera de la respuesta. Una segunda ventaja de la réplica es un mayor paralelismo: se pueden realizar varias operaciones de lectura al mismo tiempo. Con copia única, sólo se puede realizar una operación a la vez, lo que hace más lenta la ejecución. La principal desventaja de la réplica eso es el costo de mantener consistentes todas las copias.

Cuando un programa realiza una operación sobre un objeto, el compilador llama a un procedimiento del sistema de tiempo de ejecución, *invoke_op*, especificando el objeto, la operación, los parámetros y una bandera que indica si el objeto será modificado (llamado escritura) o no modificado (llamado lectura). La acción realizada por *invoke_op* depende de si el objeto está duplicado, si se dispone de una copia local, si será leído o se escribirá en él, y si el sistema subyacente soporta la transmisión confiable con un orden total. Hay que distinguir cuatro casos, como se muestra en la figura 6-42.

En la figura 6-42(a), un proceso desea realizar una operación sobre un objeto no duplicado que está en su propia máquina. Simplemente cierra el objeto, llama a la operación y abre al objeto. La idea de cerrarlo es para inhibir cualquier llamado remoto de manera temporal, mientras se realiza la operación local.

En la figura 6-42(b) aún tenemos un objeto con copia única, pero ahora está en alguna otra parte. El sistema de tiempo de ejecución realiza una RPC con la máquina remota solicitando que realice la operación, lo cual hace, tal vez con un ligero retraso, si el objeto

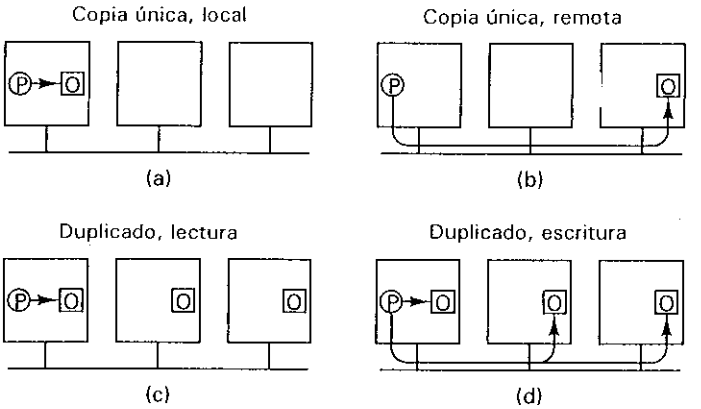


Figura 6-42. Cuatro casos para realizar una operación sobre un objeto *O*.

se cerró al llegar la solicitud. En ninguno de estos casos se distingue entre las lecturas y las escrituras (excepto que las escrituras pueden despertar a procesos bloqueados).

Si el objeto está duplicado, como en las figuras 6-42(c) y (d), siempre se dispone de una copia localmente, pero ahora importa si la operación es una lectura o una escritura. Las lecturas simplemente se realizan de manera local, sin tráfico en la red y por lo tanto sin un costo adicional.

Las escrituras sobre objetos duplicados son un poco más truculentas. Si el sistema subyacente proporciona una transmisión *confiable*, con un orden total, el sistema de tiempo de ejecución transmite el nombre del objeto, la operación, y los parámetros y bloques hasta terminar la transmisión. Todas las máquinas, incluyendo a ella misma, calculan entonces el nuevo valor.

Observe que la primitiva de transmisión debe ser confiable, lo que significa que las capas inferiores detectan y se recuperan de manera automática de los mensajes perdidos. El sistema Amoeba, con base en el cual se desarrolló Orca, tiene dicha característica. Aunque el algoritmo será descrito con detalle en el capítulo 7, lo resumiremos aquí de manera breve. Cada mensaje por transmitir se envía a un proceso especial llamado **secuenciador**, el cual le asigna un número consecutivo y después lo transmite mediante una transmisión en hardware no confiable. Siempre que un proceso note un salto en los números consecutivos, sabrá que falta un mensaje y realizará las acciones para su recuperación.

Si el sistema no tiene una transmisión confiable (o no tiene transmisión alguna), la actualización se lleva a cabo utilizando un algoritmo de copia primaria de dos fases. El proceso que hace la actualización envía primero un mensaje a la copia primaria del objeto, cerrándola y actualizándola. La copia primaria envía entonces mensajes individuales a las demás máquinas que contienen al objeto, solicitándoles que cierren sus copias. Cuando todas reconozcan haber establecido su cerradura, el proceso originador entra a la segunda

fase y envía otro mensaje indicándoles que realicen la actualización y eliminen la cerradura del objeto.

Los bloqueos son imposibles, pues aunque dos procesos intenten actualizar el mismo objeto al mismo tiempo, uno de ellos llegará primero a la copia primaria para cerrarla, y la otra solicitud se formará en una cola hasta que el objeto esté libre de nuevo. Observe además que durante el proceso de actualización, se cierran todas las copias del objeto, por lo que ningún otro proceso puede leer el valor anterior. Esta cerradura garantiza que todas las operaciones se ejecuten en un orden secuencial único y global.

Observe que este sistema de tiempo de ejecución utiliza un algoritmo de actualización, más que un algoritmo de invalidación, como la mayoría de los sistemas DSM basados en páginas. La mayor parte de los objetos son relativamente pequeños, por lo que el envío de un mensaje de actualización (el nuevo valor de los parámetros) con frecuencia no es más caro que un mensaje de invalidación. La actualización tiene la gran ventaja de permitir que ocurran las lecturas remotas posteriores sin tener que recuperar el objeto o realizar la operación de manera remota.

Ahora analizaremos brevemente el algoritmo para decidir si un objeto debe estar en una copia o duplicarse. Inicialmente, un programa Orca consta de un proceso, que tiene a todos los objetos. Cuando se bifurca, las demás máquinas son informadas de este evento y reciben las copias actuales de todos los parámetros compartidos del hijo. Cada sistema de tiempo de ejecución calcula entonces el costo esperado por duplicar, o no, cada objeto.

Para realizar este cálculo, se necesita conocer la proporción esperada de lecturas y escrituras. El compilador estima esta información mediante un examen del programa, tomando en cuenta que los accesos dentro de los ciclos cuentan más y los accesos dentro de los enunciados if cuentan menos que los demás accesos. Los costos de comunicación también se factorizan en la ecuación. Por ejemplo, un objeto con una proporción lectura/escritura de 10 en una red de transmisión será duplicado, mientras que un objeto con una proporción lectura/escritura de 1 en una red puntual será colocado en un estado de copia única, donde esta copia está en la máquina que realiza más escrituras. Para una descripción más detallada, véase (Bal y Kaashoek, 1993).

Puesto que todos los sistemas de tiempo de ejecución realizan el mismo cálculo, llegan a la misma conclusión. Si un objeto está presente en únicamente una máquina y necesita estar en todas, se disemina. Si está duplicado y ésta ya no es la mejor opción, todas las máquinas, excepto una, descartan su copia. Los objetos pueden emigrar mediante este mecanismo.

Veamos ahora cómo se logra la consistencia secuencial. Para los objetos en un estado de copia única, se establece una secuencia de todas las operaciones, de modo que la consistencia secuencial se obtiene de manera gratuita. Para los objetos duplicados, las escrituras se ordenan totalmente mediante la transmisión confiable y totalmente ordenada o mediante el algoritmo de la copia primaria. De cualquier forma, existe un acuerdo global en cuanto al orden de las escrituras. Las lecturas son locales y se pueden intercalar con las escrituras de manera arbitraria sin afectar a la consistencia secuencial.

Suponiendo que se puede realizar una transmisión confiable y totalmente ordenada en dos (más épsilon) mensajes, como en Amoeba, el esquema de Orca es muy eficiente. Los resultados sólo se envían después de terminar una operación, sin importar la cantidad de variables locales modificadas por la operación. Si uno considera a cada operación como una región crítica, la eficiencia es la misma que para la consistencia de liberación, una transmisión al final de cada región crítica.

Son posibles varias optimizaciones. Por ejemplo, en vez de sincronizar *después* de una operación, esto se podría hacer al iniciarla, como en la consistencia de entrada o en la consistencia de liberación con laxitud. La ventaja aquí es que si un proceso ejecuta una operación sobre un objeto compartido varias veces (por ejemplo, en un ciclo), no se enviarán transmisiones hasta que algún otro proceso exhiba cierto interés en el objeto.

Otra optimización consiste en suspender al proceso que realiza la llamada mientras se transmite después de una escritura que no regresa un valor (por ejemplo, *push* en nuestro ejemplo de la pila). Por supuesto, esta optimización debe hacerse de forma transparente. La información proporcionada por el compilador permite otras optimizaciones.

En resumen, el modelo Orca de la memoria distribuida compartida integra una buena práctica de ingeniería de software (objetos encapsulados), los datos compartidos, una semántica sencilla, y la sincronización de manera natural. Además, en muchos casos, es posible una implantación tan eficiente como la consistencia de liberación. Funciona mejor cuando el hardware subyacente y el sistema operativo deben proporcionar una transmisión eficiente, confiable y totalmente ordenada, y la aplicación debe tener una proporción inherente alta de las lecturas con respecto de las escrituras para los accesos a los objetos compartidos.

6.7. COMPARACIÓN

Compararemos ahora de forma breve los diversos sistemas que hemos analizado. IVY sólo intenta imitar un multiprocesador, haciendo la paginación a través de la red en vez de hacia un disco. Ofrece un modelo de memoria familiar: la consistencia secuencial, y puede ejecutar los programas multiprocesadores existentes sin modificaciones. El único problema es el desempeño.

Munin y Midway intentan mejorar el desempeño pidiendo al programador que indique las variables compartidas y mediante el uso de modelos de consistencia más débiles. Munin se basa en la consistencia de liberación, y en cada liberación transmite todas las páginas modificadas (como deltas) a otros procesos que comparten esas páginas. Midway, por otro lado, realiza la comunicación sólo cuando una cerradura cambia de propiedad.

Midway soporta sólo un tipo de variable dato compartida, mientras que Munin tiene cuatro tipos (exclusiva para lectura, migratoria, de escritura compartida y convencional). Por otro lado, Midway soporta tres protocolos de consistencia distintos (de entrada, de liberación y de procesador), mientras que Munin sólo soporta la consistencia de liberación. Un tema sujeto a discusión es si es mejor tener varios tipos de datos compartidos o varios

protocolos. Será necesaria mayor investigación antes de que podamos comprender este tema por completo.

Por último, es diferente la forma en que se detectan las escrituras a las variables compartidas. Munin utiliza el hardware MMU para señalar las escrituras, mientras que Midway ofrece una opción entre un compilador especial que registra las escrituras y hace esto a la manera de Munin, con el MMU. Una ventaja definitiva de Midway es no tener que llevar un flujo de fallos de página, especialmente dentro de las regiones críticas.

Comparemos ahora Munin y Midway con la memoria compartida basada en objetos de la variante Linda-Orca. La sincronización y el acceso a los datos en Munin y Midway son tarea del programador, mientras que ambos están fuertemente integrados en Linda y Orca. En Linda existe menos peligro de que un programador cometa un error de sincronización, puesto que *in* y *out* controlan su propia sincronización de manera interna. De manera similar, cuando se llama una operación sobre un objeto compartido en Orca, la cerradura es controlada por completo por el sistema de tiempo de ejecución, sin que el programador tenga conciencia de ello. La sincronización con condiciones (opuesta a la sincronización con exclusión mutua) no es parte del modelo de Munin o de Midway, así que el programador debe realizar todo de manera explícita. Por el contrario, es una parte integral del modelo Linda (el bloqueo cuando no está presente una n-ada) y del modelo Orca (se bloquea en una protección).

En resumen, los programadores en Munin y Midway deben realizar un mayor trabajo en el área de sincronización y consistencia, con poco soporte, y deben tener todo correcto. No existe encapsulamiento no existen métodos para proteger los datos compartidos, como en Linda y Orca. Por otro lado, Munin y Midway permiten la programación en lenguajes C o C++ ligeramente modificados, mientras que la comunicación en Linda es poco usual y Orca es un lenguaje por completo nuevo. En términos de eficiencia, Midway es mejor en términos del número y el tamaño de los mensajes transmitidos, aunque el uso de modelos de programación fundamentalmente diferentes (código C abierto, objetos y n-adas) puede conducir a algoritmos esencialmente distintos en los tres casos, lo que también puede afectar la eficiencia.

6.8. RESUMEN

Los sistemas de computo con varios CPU caen en alguna de dos categorías: los que tienen memoria compartida y los que no. Las máquinas de memoria compartida (multiprocesadores) son más fáciles de programar pero más difíciles de construir, mientras que las máquinas sin memoria compartida (multicomputadoras) son más difíciles de programar pero más fáciles de construir. La memoria distribuida compartida es una técnica para facilitar la programación de las multicomputadoras, simulando la memoria compartida en ellas.

Los pequeños multiprocesadores se basan con frecuencia en un bus, pero los de gran tamaño utilizan conmutadores. Los protocolos utilizados por los de gran tamaño requieren estructuras de datos y algoritmos complejos para mantener la consistencia de los cachés.

Los multiprocesadores NUMA evitan esta complejidad obligando al software a tomar todas las decisiones acerca de las páginas que deban colocarse en tal o cual máquina.

Es posible una implantación directa de DSM, como en IVY, pero con frecuencia tiene un desempeño mucho menor que el de un multiprocesador. Por esta razón, los investigadores han analizado varios modelos de memoria, en un intento por obtener un mejor desempeño. El estándar contra el que se miden todos los modelos es la consistencia secuencial, que significa que todos los procesos ven todas las referencias a memoria en el mismo orden. La consistencia causal, la consistencia PRAM y la consistencia del procesador debilitan el concepto de que los procesos vean a *todas* las referencias a memoria en el mismo orden.

Otro método es el de la consistencia débil, la consistencia de liberación y la consistencia de entrada, en donde la memoria no es consistente todo el tiempo, pero el programador puede obligarla a ser consistente mediante ciertas acciones, como entrar o salir de una región crítica.

Se han utilizado tres técnicas generales para proporcionar DSM. La primera estimula el modelo de memoria de multiprocesador, dando a cada proceso una memoria paginada lineal. Las páginas se mueven de una máquina a otra según sea necesario. La segunda utiliza los lenguajes de programación ordinarios con variables compartidas anotadas. La tercera se basa en los modelos de programación de alto nivel, como las n-adas y los objetos.

En este capítulo estudiamos cinco métodos distintos de DSM. IVY opera esencialmente como una memoria virtual, donde las páginas se mueven de una máquina a otra cuando ocurren los fallos de páginas. Munin utiliza varios protocolos y la consistencia de liberación para permitir que las variables individuales se compartan. Midway es similar a Munin, excepto que utiliza la consistencia de entrada en vez de la consistencia de liberación como el caso normal. Linda representa el otro extremo del espectro, con un espacio de n-adas abstracto, lejos de los detalles de paginación. Orca soporta un modelo en donde los objetos dato se duplican en varias máquinas y el acceso es mediante métodos protegidos que hacen que los objetos parezcan secuencialmente consistentes para el programador.

PROBLEMAS

1. Un sistema Dash tiene b bytes de memoria divididos entre n unidades. Cada unidad tiene p procesadores en ella. El tamaño del bloque cachés de c bytes. Dé una fórmula para la cantidad total de memoria dedicada a los directorios (excluyendo los dos bits de estado por cada entrada de directorio).
2. ¿Es realmente esencial que Dash establezca una distinción entre los estados LIMPIO y NO OCULTO? ¿Sería posible dispensar uno de los dos? Después de todo, en ambos casos, la memoria tiene una copia actualizada del bloque.
3. En el texto se afirma que son posibles varias variantes menores del protocolo de propiedad del caché de la figura 6-6. Describa una de tales variantes y dé una ventaja de su variante sobre la del texto.
4. ¿Por qué se necesita el concepto de "memoria de origen" en Memnet y no en Dash?

5. En un multiprocesador NUMA, las referencias a la memoria local tardan 100 nanosegundos y las referencias remotas 800 nanosegundos. Cierto programa realiza un total de N referencias a memoria durante su ejecución, de las cuales 1 por ciento son a una página P. Esta página es remota inicialmente, y tarda C nanosegundos en copiarse localmente. ¿Bajo qué condiciones debe copiarse localmente la página en ausencia de un uso significativo por otros procesadores?
6. Durante el análisis de los modelos de consistencia de la memoria, con frecuencia nos referimos al contrato entre el software y la memoria. ¿Por qué se necesita ese contrato?
7. Un multiprocesador tiene un bus. ¿Es posible implantar la memoria estrictamente consistente?
8. En la figura 6-13(a) se muestra un ejemplo de una memoria secuencialmente consistente. Realice un cambio mínimo a P_2 que viole la consistencia secuencial.
9. En la figura 6-14, ¿es 001110 una salida válida para una memoria secuencialmente consistente? Explique su respuesta.
10. Al final de la sección 6-2.2 analizamos un modelo formal que afirmaba que todo conjunto de operaciones sobre una memoria secuencialmente consistente se puede modelar mediante una cadena, S, de la que se pueden derivar todas las secuencias de los procesos individuales. Para los procesos P_1 y P_2 de la figura 6-16, dé todos los valores posibles de S. Ignore los procesos P_3 y P_4 y no incluya sus referencias a memoria en S.
11. En la figura 6-20, una memoria secuencialmente consistente permite seis intercalados posibles de enunciados. Enumérelos todos.
12. ¿Por qué $W(x)1$ $R(x)0$ $R(x)1$ no es válido en la figura 6-12(b)?
13. En la figura 6-14, ¿es 000000 una salida válida para una memoria con consistencia PRAM únicamente? Explique su respuesta.
14. En la mayor parte de las implantaciones de la consistencia de liberación (fuerte) en los sistemas DSM, las variables compartidas se sincronizan en una liberación, pero no en una adquisición. ¿Por qué se necesita entonces la adquisición de todos modos?
15. En la figura 6-27, suponga que el propietario de la página se localiza mediante una transmisión. ¿En cuáles casos debe enviarse la página para una lectura?
16. En la figura 6-28, un proceso puede establecer contacto con el propietario de una página por medio del controlador de páginas. Podría querer la propiedad o la propia página, que son entes independientes. ¿Existen los cuatro casos (excepto, por supuesto, el caso en que el solicitante no desea la página y no desea la propiedad)?
17. Suponga que dos variables, A y B, se localizan, por accidente, en la misma página en un sistema DSM basado en páginas. Sin embargo, ambas son variables no compartidas. ¿Es posible compartir falsamente?
18. ¿Por qué utiliza IVY un esquema de invalidación para la consistencia en vez de un esquema de actualización?
19. Algunas máquinas tienen una instrucción que, en una acción atómica, intercambia un registro con una palabra en memoria. Mediante esta instrucción, es posible implantar los semáforos para la protección de las regiones críticas. ¿Funcionarán los programas

que utilizan esta instrucción en un sistema DSM basado en páginas? En tal caso, ¿bajo qué circunstancias funcionará de manera eficiente?

20. ¿Qué ocurre si un proceso de Munin modifica una variable compartida fuera de una región crítica?

21. Dé un ejemplo de una operación *in* en Linda que no requiera una búsqueda o la dispersión para encontrar una concordancia.

22. Cuando Linda se implanta mediante la duplicación de n-adas en varias máquinas, se necesita un protocolo para eliminar n-adas. Dé un ejemplo de protocolo que no produzca competencia cuando dos procesos intentan eliminar la misma n-ada al mismo tiempo.

23. Considere un sistema Orca que se ejecuta en una red con transmisión en hardware. ¿Por qué afecta al desempeño la proporción entre las operaciones de lectura y las operaciones de escritura?

7

Estudio 1: Amoeba

En este capítulo daremos nuestro primer ejemplo de sistema operativo distribuido: Amoeba. En el siguiente analizaremos un segundo ejemplo: Mach. Amoeba es un sistema operativo distribuido: éste permite que una colección de CPU y equipo de E/S se comporten como una computadora. También proporciona elementos para la programación en paralelo si se desea. Este capítulo describe los objetivos, diseño e implantación de Amoeba. Para mayor información relativa a Amoeba, véase (Mullender *et al.*, 1990; Tanenbaum *et al.*, 1990).

7.1. INTRODUCCIÓN A AMOEBA

En esta sección daremos una introducción a Amoeba, comenzando por una breve historia y sus actuales objetivos de investigación. Después analizaremos la arquitectura de un sistema Amoeba típico. Por último comenzaremos nuestro estudio del software de Amoeba, el núcleo y los servidores.

7.1.1. Historia de Amoeba

Amoeba se originó en Vrije Universiteit, Amsterdam, Holanda, en 1981, como proyecto de investigación en cómputo distribuido y paralelo. Fue diseñado en un principio por Andrew S. Tanenbaum y tres de sus estudiantes de doctorado, Frans Kaashoek, Sape J. Mullender y Robbert van Renesse, aunque muchas otras personas contribuyeron en el diseño y la implantación. En el año de 1983, un prototipo inicial, Amoeba 1.0, tenía un nivel operacional.

A partir de 1984, el proyecto Amoeba se dividió y se estableció en un segundo grupo en el Centro de Matemáticas y Ciencias de la Computación, también en Amsterdam, bajo

la dirección de Mullender. En los años siguientes, esta cooperación se extendió a lugares de Inglaterra y Noruega con un proyecto de sistema distribuido de área amplia patrocinado por la Comunidad Europea. Este trabajo utilizó Amoeba 3.0, que, a diferencia de las versiones anteriores, se basaba en RPC. Al utilizar Amoeba 3.0, fue posible que los clientes en Tromso tuvieran un acceso transparente a los servidores en Amsterdam y viceversa.

El sistema evolucionó durante algunos años, adquiriendo características como la emulación parcial de UNIX, la comunicación en grupo y un protocolo nuevo de bajo nivel. La versión que se describe en este capítulo es Amoeba 5.2.

7.1.2. Objetivos de investigación

Muchos de los proyectos de investigación en los sistemas operativos distribuidos han partido de un sistema existente (por ejemplo, UNIX) y agregando nuevas características como el uso de redes y un sistema compartido de archivos para hacerlo más distribuido. El proyecto Amoeba siguió un método diferente. Partió de un plan limpio y desarrolló un sistema nuevo a partir de cero. La idea era tener un comienzo fresco y experimentar nuevas ideas sin tener que preocuparse por la compatibilidad retroactiva con cualquiera de los sistemas existentes. Para evitar el enorme trabajo de escribir grandes cantidades de software de aplicación, se añadió posteriormente un paquete de emulación de UNIX.

El objetivo principal del proyecto era construir un sistema operativo distribuido y transparente. Para el usuario promedio, el uso de Amoeba es igual al uso de un sistema tradicional de tiempo compartido, como UNIX. Uno entra, edita y compila programas, mueve archivos de un lugar a otro, etc. La diferencia es que cada una de estas acciones hace uso de varias máquinas, dispersas en la red. Entre estas máquinas están los servidores de procesos, los de archivos, los de directorios, los de cómputo y otros, pero el usuario no es consciente de ello. En la terminal, todo parece como un sistema ordinario de tiempo compartido.

Una distinción importante entre Amoeba y la mayoría de los demás sistemas distribuidos es que Amoeba no tiene el concepto de "máquina de origen". Cuando un usuario entra al sistema, entra a éste como un todo y no a una máquina específica. Las máquinas no tienen propietarios. El shell inicial, que se ejecuta al entrar el usuario, se ejecuta en cierta máquina arbitraria, pero al ser iniciados los comandos, en general no se ejecutan en la misma máquina donde se ejecuta el shell. En vez de esto, el sistema busca de manera automática la máquina con la menor carga para ejecutar cada nuevo comando. Durante el curso de una larga sesión en la terminal, los procesos que se ejecutan a cargo de un usuario cualquiera estarán más o menos dispersos en todas las máquinas del sistema, dependiendo de la carga, por supuesto. Es decir, Amoeba es muy transparente con respecto de la ubicación.

En otras palabras, todos los recursos pertenecen al sistema como un todo y son controlados por él. No están dedicados a usuarios específicos, excepto por periodos cortos para la ejecución de procesos individuales. Este modelo intenta proporcionar la transparencia que es el cáliz sagrado de todos los diseñadores de sistemas distribuidos.

Un ejemplo sencillo es *amake*, el reemplazo de Amoeba para el programa *make* de UNIX. Cuando el usuario escribe *amake*, se realizan todas las compilaciones necesarias,

como es de esperarse, pero el sistema (y no el usuario) determina si deben ocurrir de manera secuencial o en paralelo y la máquina o máquinas donde esto debe ocurrir. Nada de esto es visible para el usuario.

Un segundo objetivo de Amoeba es proporcionar un colchón de prueba para la realización de una programación distribuida y paralela. Aunque algunos usuarios utilizan Amoeba de la misma forma en que utilizarían cualquier otro sistema de tiempo compartido, la mayoría de los usuarios están interesados en especial en la experimentación con algoritmos, lenguajes, herramientas y aplicaciones distribuidos y paralelos. Amoeba ofrece soporte a estos usuarios al hacer que el paralelismo subyacente esté disponible para las personas que quieran aprovecharlo. En la práctica, la mayor parte de la base de usuarios actuales consta de personas que están en especial interesadas en el cómputo distribuido y paralelo en sus distintas variantes. Con este fin, se ha diseñado e implantado en Amoeba un lenguaje, Orca. Orca y sus aplicaciones se describen en (Bal, 1991; Bal *et al.*, 1990; Tanenbaum *et al.*, 1992). Sin embargo, el propio Amoeba está escrito en C.

7.1.3. La arquitectura del sistema Amoeba

Antes de describir la forma en que Amoeba está estructurado, es útil delinear en primer lugar el tipo de configuración en hardware para el que fue diseñado Amoeba, puesto que difiere un poco de lo que poseen la mayor parte de las organizaciones actuales. Amoeba se diseñó con dos hipótesis respecto al hardware:

1. Los sistemas tienen un número muy grande de CPU.
2. Cada CPU tendrá decenas de megabytes de memoria.

Estas hipótesis ya son verdaderas en ciertas instalaciones y es probable que sean ciertas en casi todas las instalaciones corporativas, académicas y gubernamentales dentro de pocos años.

La fuerza motriz detrás de la arquitectura del sistema es la necesidad de incorporar un gran número de CPU de manera directa. En otras palabras, ¿qué hacer si cada usuario puede disponer de 10 a 100 CPU? Una solución consiste en darle a cada usuario un multiprocesador personal de 10 o 100 nodos.

Aunque es posible darle a cada persona un multiprocesador, hacerlo no es una forma eficiente de gastar el presupuesto disponible. La mayor parte del tiempo, casi todos los procesadores estarán inactivos, pero algunos usuarios desearán ejecutar programas masivos paralelos y no podrán limitar con facilidad el número de ciclos inactivos del CPU, puesto que éstos se encuentran en las máquinas personales de otros usuarios.

En vez de este método del multiprocesador personal, Amoeba se basa en el modelo que se muestra en la figura 7-1. En este modelo, todo el poder de cómputo se localiza en una o más **pilas de procesadores**. Una pila de procesadores consta de varios CPU, cada uno con su propia memoria local y conexión a la red. No se necesita la memoria compartida, ni siquiera se espera que exista; pero si está presente, se le utiliza para optimizar la transferencia

de mensajes al hacer el copiado de una memoria a otra, en vez de enviar mensajes a través de la red.

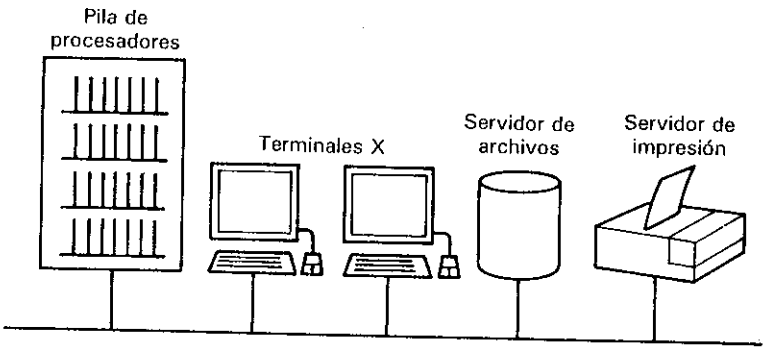


Figura 7-1. La arquitectura del sistema Amoeba.

Los CPU en una pila pueden tener distintas arquitecturas; por ejemplo, una mezcla de máquinas 680x0, 386, o SPARC. Amoeba está diseñado de forma que pueda trabajar con varias arquitecturas y sistemas heterogéneos. Incluso es posible que los hijos de un proceso se ejecuten en arquitecturas diferentes.

La pila de procesadores no es "poseída" por usuario alguno. Cuando un usuario escribe un comando, el sistema operativo asigna en forma dinámica uno o más procesadores para ejecutar ese comando. Al terminar el comando, se terminan los procesos y los recursos regresan a la pila, en espera del siguiente comando, que muy probablemente sea de un usuario diferente. Si existe una reducción en el número de procesadores en la pila, se comparte el tiempo de los procesadores individuales, de modo que los nuevos procesos sean asignados al CPU con menor carga. El punto importante en este momento es que este modelo es un poco distinto de los sistemas actuales, donde cada usuario tiene exactamente una estación de trabajo para todas sus actividades de cómputo.

La presencia esperada de memorias de gran tamaño en los sistemas futuros ha influido en el diseño de varias formas. Los compromisos espacio-tiempo se hacen con el fin de proporcionar un alto desempeño, con el costo de utilizar más memoria. Analizaremos posteriormente algunos ejemplos de esto.

El segundo elemento de la arquitectura de Amoeba es la terminal. El usuario tiene acceso al sistema a través de la terminal. Una terminal usual de Amoeba es la terminal X, con una pantalla de gran tamaño (con un mapa de bits) y un ratón. Otra alternativa consiste en que también se pueda utilizar como terminal una computadora personal o una estación de trabajo que ejecute ventanas X. Aunque Amoeba no prohibe la ejecución de los programas del usuario en la terminal, la idea detrás de este modelo es proporcionar a los usuarios terminales relativamente baratas y concentrar los ciclos de cómputo en una pila común para utilizarlos con mayor eficiencia.

Los procesadores de una pila son de forma inherente más baratos que las estaciones de trabajo, ya que sólo constan de una tarjeta y una conexión a la red. No existe un teclado, monitor o ratón y la fuente de poder puede ser compartida con muchas tarjetas. Así, en vez de comprar 100 estaciones de trabajo de alto desempeño para 100 usuarios, se pueden adquirir 50 procesadores de pila de alto desempeño y 100 terminales X por el mismo precio (dependiendo de la economía, por supuesto). Puesto que los procesadores de pila sólo se asignan cuando es necesario, un usuario inactivo sólo dispone de una barata terminal X y no de una cara estación de trabajo. Los compromisos inherentes en el modelo de la pila de procesadores *vs.* el modelo de la estación de trabajo fueron analizados en el capítulo 4.

Para evitar cualquier confusión, los procesadores de la pila no *tienen* que ser computadoras con una tarjeta. Si no se dispone de éstas, se puede designar un subconjunto de las computadoras personales existentes como los procesadores de la pila. Tampoco tienen que estar localizados en un mismo cuarto. La localización física no es importante en realidad. De hecho, los procesadores de la pila se pueden encontrar en distintos países, como analizaremos posteriormente.

Otro componente importante de la configuración de Amoeba son los servidores especializados, como los servidores de archivos, que por razones de hardware o software necesitan ejecutarse en un procesador aparte. En ciertos casos, es posible que un servidor se ejecute en un procesador de la pila, para que se inicie cuando sea necesario, pero por razones de desempeño es mejor que se ejecute todo el tiempo.

Los servidores proporcionan servicios. Un **servicio** es una definición abstracta de lo que el servidor está listo a hacer por sus clientes. Esto define lo que el cliente puede pedir y los resultados, pero no especifica el número de servidores que trabajen en forma conjunta para proporcionar el servicio. De este modo, el sistema tiene un mecanismo para proporcionar servicios tolerantes de fallas, mediante varios servidores que efectúen el servicio.

Un ejemplo es el servidor de directorios. No hay aspectos inherentes en el servidor o en el diseño del sistema que eviten que un usuario inicie un nuevo servidor de directorios en un procesador de la pila cada vez que desee buscar el nombre de un archivo. Sin embargo, esto sería ineficiente, por lo que uno o más servidores de directorios se encuentran en ejecución todo el tiempo, por lo general en máquinas dedicadas a ello para mejorar su desempeño. La decisión de tener varios servidores en ejecución permanente y otros que se inicien en forma explícita es tarea del administrador del sistema.

7.1.4. El micronúcleo de Amoeba

Después de analizar el modelo de hardware de Amoeba, pasemos al modelo de software. Amoeba consta de dos partes fundamentales: un micronúcleo que se ejecuta en cada procesador, y una colección de servidores que proporcionan la mayor parte de la funcionalidad de un sistema operativo tradicional. La estructura general se muestra en la figura 7-2.

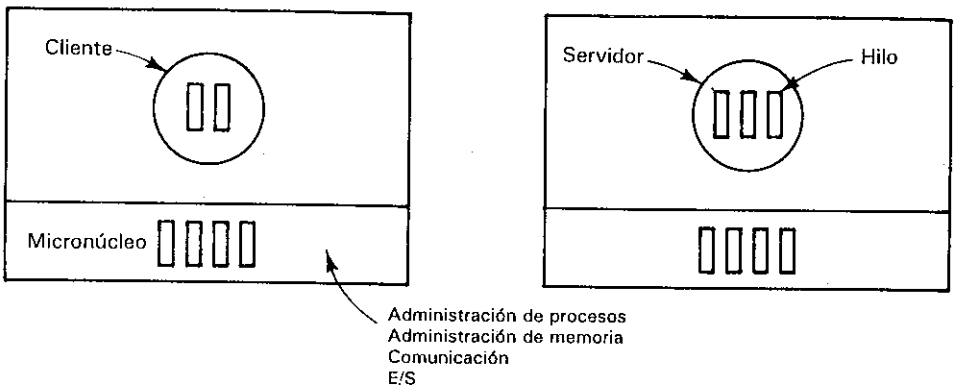


Figura 7-2. Estructura del software de Amoeba.

El micronúcleo de Amoeba se ejecuta en todas las máquinas del sistema. El mismo núcleo se utiliza en los procesadores de la pila, las terminales (suponiendo que sean computadoras en vez de terminales X) y los servidores especializados. El micronúcleo tiene cuatro funciones básicas:

1. Controlar los procesos e hilos.
2. Proporcionar el soporte de la administración de memoria de bajo nivel.
3. Soportar la comunicación.
4. Controlar la E/S de bajo nivel.

Consideraremos cada una de estas funciones a su tiempo.

Como la mayoría de los sistemas operativos, Amoeba soporta el concepto de proceso. Además, Amoeba soporta también varios hilos de control dentro de un espacio de direcciones. Un proceso con un hilo es en esencia igual a un proceso en UNIX. Tal proceso tiene un espacio de direcciones, un conjunto de registros, un contador del programa y una pila.

Por el contrario, aunque un proceso con varios hilos posee también un espacio de direcciones compartido por todos los hilos, cada uno de éstos tiene, desde el punto de vista lógicos sus propios registros, su contador del programa y su pila. De hecho, una colección de hilos de un proceso es análoga a una colección de procesos independientes en UNIX, excepto que comparten un espacio de direcciones común.

Un uso típico de varios hilos sería el de un servidor de archivos, donde cada solicitud recibida es asignada a un hilo para su procesamiento. Ese hilo podría comenzar a procesar la solicitud, después bloquearse en espera del disco y después seguir con su trabajo. Al dividir el servidor en varios hilos, cada uno de ellos puede ser secuencial, aunque deba bloquearse en espera de E/S. Sin embargo, todos los hilos pueden, por ejemplo, tener acceso a un caché compartido en software. Los hilos se pueden sincronizar mediante semáforos o mútex para evitar que dos hilos tengan acceso simultáneo al caché compartido.

La segunda tarea del núcleo es proporcionar la administración de la memoria de bajo nivel. Los hilos pueden asignar o eliminar la asignación de los bloques de memoria, llamados **segmentos**. Estos segmentos se pueden leer o escribir y ser asociados o desasociados al espacio de direcciones del proceso al cual pertenece el hilo que realiza la llamada. Un proceso debe poseer al menos un segmento, pero también puede tener varios. Los segmentos se pueden utilizar para el texto, los datos, la pila o para cualquier otro fin que desee el proceso. El sistema operativo no obliga a utilizar algún patrón particular de uso de los segmentos.

La tercera tarea del núcleo es controlar la comunicación entre los procesos. Se dispone de dos formas de comunicación: puntual y de grupo. Ambas están muy integradas entre sí, de modo que sean lo más parecidas posible.

La comunicación puntual se basa en el modelo de un cliente que envía un mensaje a un servidor y que después se bloquea hasta que el servidor envía una respuesta. Este intercambio solicitud/respuesta es la base para la construcción de todo lo demás.

La otra forma de comunicación es de grupo. Permite el envío de mensajes de una fuente a varios destinos. Los protocolos en software proporcionan una comunicación en grupo confiable y tolerante de fallas a los procesos del usuario, en presencia de mensajes perdidos u otros errores.

La cuarta función del núcleo es la administración de la E/S de bajo nivel. Para cada dispositivo de E/S conectado a una máquina, existe un controlador del dispositivo en el núcleo. Este controlador se encarga de toda la E/S del dispositivo. Los controladores están ligados con el núcleo y no se pueden cargar de manera dinámica.

Los controladores de dispositivos se comunican con el resto del sistema mediante los mensajes usuales de solicitud y respuesta. Un proceso, como por ejemplo un servidor de archivos, que necesite comunicarse con el controlador del disco, le envía mensajes de solicitud y recibe de regreso ciertas respuestas. En general, el cliente no tiene que saber que se comunica con un controlador. Desde su punto de vista, sólo se comunica con un hilo en algún lugar.

Tanto el sistema de mensajes puntuales como la comunicación en grupo hacen uso de un protocolo especializado llamado FLIP. Este protocolo es un protocolo de capas de la red y fue diseñado en especial para cubrir las necesidades del cómputo distribuido. Trabaja tanto con la unitransmisión como con la multitransmisión en redes complejas. Lo analizaremos más adelante.

7.1.5. Los servidores de Amoeba

Todo lo que no se lleva a cabo dentro del núcleo lo realizan los procesos servidores. La idea detrás de este diseño es minimizar el tamaño del núcleo y mejorar la flexibilidad. Como el sistema de archivos y otros dispositivos estándar no se integran al núcleo, éstos se pueden modificar con facilidad y se pueden ejecutar en forma simultánea varias versiones para las distintas poblaciones de usuarios.

Amoeba se basa en el modelo cliente-servidor. Los clientes son escritos en general por los usuarios, mientras que los servidores son escritos por los programadores del sistema, aunque los usuarios pueden escribir sus propios servidores si así lo desean. El concepto de objeto es central en el diseño de todo el software; un objeto es como un tipo de dato abstracto. Cada objeto consta de ciertos datos encapsulados, con ciertas operaciones definidas en ellos. Por ejemplo, los objetos archivo tienen una operación READ, entre otras.

Los objetos son controlados por los servidores. Cuando un proceso crea un objeto, el servidor que lo controla regresa al cliente una posibilidad protegida en forma cifrada para el objeto. Para el uso posterior del objeto, hay que presentar la posibilidad adecuada. Todos los objetos del sistema, tanto del hardware como del software, reciben un nombre, una protección y son controlados por las posibilidades. Algunos de los objetos soportados por esta vía son los archivos, los directorios, los segmentos de memoria, las ventanas en la pantalla, los procesadores, los discos y las unidades de cinta. Esta interfaz uniforme para todos los objetos proporciona generalidad y sencillez.

Todos los servidores estándar tienen procedimientos de resguardo en la biblioteca. Para utilizar un servidor, por lo general un cliente llama por lo regular al resguardo, el cual ordena los parámetros, envía el mensaje y se bloquea hasta recibir la respuesta. Este mecanismo oculta todos los detalles de la implantación al usuario. Se tiene un compilador de resguardo para los usuarios que deseen producir procedimientos de resguardo para sus servidores.

Tal vez el servidor más importante sea el **servidor de archivos**. Proporciona las primitivas para la administración de archivos, su creación, lectura, eliminación, etc. A diferencia de la mayoría de los servidores de archivos, los archivos creados son inmutables. Una vez creado, un archivo no se puede modificar, pero puede ser eliminado. Los archivos inmutables facilitan la réplica automática, puesto que evitan muchas de las condiciones de competencia inherentes en la réplica de archivos sujetos a modificaciones durante tal proceso.

Otro servidor importante es el **servidor de directorios**, que por ciertas oscuras razones históricas también se conoce como el **servidor soap (jabón)**. Este servidor controla los directorios y los nombres de las rutas de acceso y las asocia con las posibilidades. Para leer un archivo, un proceso le pide al servidor de directorios que busque el nombre de la ruta de acceso. En una búsqueda exitosa, el servidor de directorios regresa la posibilidad correspondiente al archivo (o algún otro objeto). Las operaciones posteriores en el archivo no utilizan al servidor de directorios, sino que van directamente al servidor de archivos. La separación del sistema de archivos en estos dos componentes aumenta la flexibilidad y hace a cada parte más sencilla, puesto que sólo tiene que controlar un tipo de objeto (directorios o archivos) y no dos.

Se tienen otros servidores estándar para el manejo de la réplica de objetos, el inicio de procesos, el monitoreo de servidores en búsqueda de fallas y la comunicación con el mundo exterior. Los servidores de los usuarios llevan a cabo gran variedad de tareas específicas a cada aplicación.

El resto del capítulo tiene la siguiente estructura. En primer lugar describiremos los objetos y las posibilidades, puesto que forman el corazón de todo el sistema. Después analizaremos el núcleo, con énfasis en la administración de los procesos, la administración de la memoria y la comunicación. Por último, examinaremos algunos de los principales servidores, como el servidor de archivos, el servidor de directorios, el servidor de réplicas y el servidor de ejecución.

7.2. OBJETOS Y POSIBILIDADES EN AMOEBA

El concepto básico y unificador subyacente en todos los servidores de Amoeba y los servicios que éstos proporcionan es el **objeto**. Un objeto es una pieza encapsulada de datos en la que se pueden llevar a cabo ciertas operaciones bien definidas. Es, en esencia, un tipo de dato abstracto. Los objetos son pasivos. No contienen procesos o métodos o alguna otra entidad activa que "haga" cosas; en lugar de esto, cada objeto es controlado por un proceso servidor.

Para llevar a cabo una operación en un objeto, un cliente hace una RPC con el servidor, donde se especifica el objeto, la operación por desarrollar y, de manera opcional, los parámetros necesarios. El servidor lleva a cabo el trabajo y regresa la respuesta. Las operaciones se llevan a cabo en forma sincronizada; es decir, después que se inicia una RPC con un servidor para obtener cierto trabajo, el hilo del cliente se bloquea hasta que el servidor responde. Sin embargo, se pueden ejecutar otros hilos en el mismo proceso.

Los clientes no son conscientes de las posiciones de los objetos que utilizan y los servidores que los controlan. Un servidor se podría ejecutar en la misma máquina que el cliente, en una distinta en la misma LAN e incluso en una máquina a cientos de kilómetros de distancia. Además, aunque la mayoría de los servidores se ejecutan como procesos del usuario, unos cuantos servidores de bajo nivel, como el servidor de segmentos (es decir, de memoria) o el servidor de procesos, se ejecutan como hilos en el núcleo. Esta distinción también es invisible para los clientes. El protocolo RPC para comunicarse con los servidores del usuario o los servidores del núcleo, ya sean locales o remotos, es idéntico en todos los casos. Así, un cliente sólo se preocupa por lo que debe realizar, no de la posición donde se guardan los objetos o donde se ejecutan los servidores. Cierto directorio contiene las posibilidades para todos los servidores de archivos accesibles junto con una especificación de la opción por omisión, de modo que un usuario pueda evitar esta opción en los casos donde esto sea importante. Por lo general, el administrador del sistema define la opción por omisión como la local.

7.2.1. Posibilidades

Los objetos reciben su nombre y protección de manera uniforme mediante boletos especiales llamados **posibilidades**. Para crear un objeto, un cliente realiza una RPC con el servidor apropiado, donde indica lo que desea. El servidor crea entonces el objeto y regresa

una posibilidad al cliente. En las operaciones siguientes, el cliente debe presentar la posibilidad para identificar al objeto. Una posibilidad no es más que un número binario de gran tamaño. El formato de Amoeba 5.2 se muestra en la figura 7-3.

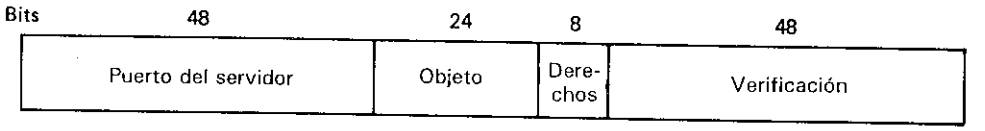


Figura 7-3. Una posibilidad en Amoeba.

Cuando un cliente desea llevar a cabo una operación en un objeto, llama a un procedimiento de resguardo, el cual construye un mensaje con la posibilidad del objeto y después hace un señalamiento al núcleo. Éste extrae el campo *Puerto del servidor* de la posibilidad y lo busca en su caché para localizar la máquina donde reside el servidor. Si el puerto no está en el caché, lo localiza mediante una transmisión que describiremos más adelante. El puerto es en sí una dirección lógica mediante la cual se puede alcanzar al servidor. Así, se asocian los puertos de servidor a cada servidor específico (o a un conjunto de servidores) y no con una máquina específica. Si un servidor se desplaza a una nueva máquina, se lleva consigo el puerto del servidor. Muchos de los puertos de servidor, como el correspondiente al servidor de archivos, se conocen en forma pública y son estables durante años. La única forma en que uno se puede dirigir a un servidor es a través de su puerto, que en un principio se elige a sí mismo.

El resto de la información en la posibilidad es ignorada por los núcleos y se transfiere al servidor para su uso. El campo *Objeto* es utilizado por el servidor para identificar el objeto específico en cuestión. Por ejemplo, un servidor de archivos puede controlar miles de archivos y utilizar el número de objeto para indicar el archivo en el cual opera en un momento dado. En cierto sentido, el campo *Objeto* de una posibilidad de un archivo es similar al número de nodo-i de UNIX.

El campo *Derechos* es un mapa de bits que indica las operaciones permitidas al propietario de una posibilidad. Por ejemplo, aunque un objeto particular soporte la lectura y la escritura, se puede construir una posibilidad particular donde se desactiven todos los bits de derechos excepto el correspondiente a READ.

El campo *Verificación* se utiliza para dar validez a la posibilidad. Éstas son controladas de forma directa por los procesos del usuario. Sin cierta forma de protección, no habría manera de evitar que los procesos del usuario moldeen las posibilidades.

7.2.2. Protección de objetos

El algoritmo básico para la protección de objetos es el siguiente. Cuando se crea un objeto, el servidor elige un campo *Verificación* al azar y lo guarda en la nueva posibilidad y dentro de sus propias tablas. Se activan todos los bits de derechos a una nueva posibilidad y esta **posibilidad del propietario** es lo que regresa al cliente. Cuando la posibilidad regresa al servidor en una solicitud para llevar a cabo una operación, se verifica el campo *Verificación*.

Para crear una posibilidad restringida, un cliente debe regresar una posibilidad al servidor, junto con una plantilla de bits para los nuevos derechos. El servidor toma el campo original *Verificación* de sus tablas, realiza un OR EXCLUSIVO con los nuevos derechos (que deben ser un subconjunto de los derechos en la posibilidad) y entonces ejecuta el resultado a través de una función de un solo sentido. Tal función, $y = f(x)$, tiene la propiedad de que dado x, es fácil encontrar y, pero dado y, para determinar x hay que hacer una búsqueda exhaustiva en todos los valores posibles de x (Evans *et al.*, 1974)

El servidor crea entonces una nueva posibilidad, con el mismo valor en el campo *Objeto*, pero con los nuevos valores en los bits de derechos en el campo *Derechos* y la salida de la función f en el campo *Verificación*. La nueva posibilidad regresa entonces al punto donde se hizo la llamada. El cliente puede enviar esta nueva posibilidad a otro proceso, si así lo desea, ya que las posibilidades se controlan por completo dentro del espacio del usuario.

El método para generar las posibilidades restringidas se muestra en la figura 7-4. En este ejemplo, el propietario ha desactivado todos los derechos excepto uno. Por ejemplo, la posibilidad restringida podría permitir que el objeto sea leído, pero nada más. El significado del campo *Derechos* es distinto para cada tipo de objeto, puesto que las propias operaciones válidas también varían entre los diversos tipos de objetos.

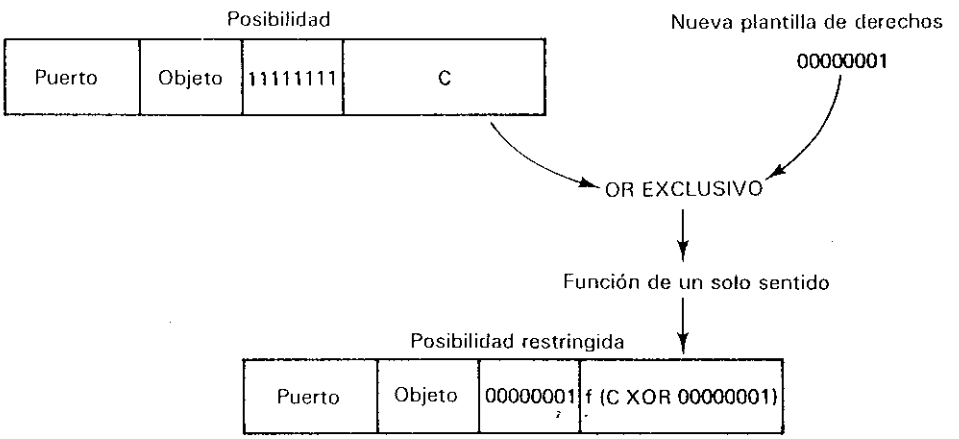


Figura 7-4. Generación de una posibilidad restringida a partir de una posibilidad del propietario.

Cuando la posibilidad restringida regresa al servidor, éste ve en el campo *Derechos* que no es una *posibilidad del propietario*, puesto que al menos un bit está desactivado. El servidor busca entonces el número aleatorio original en sus tablas, realiza un OR EXCLUSIVO de éste con el campo *Derechos* de la posibilidad y ejecuta el resultado a través de la función de un sentido. Si el resultado coincide con el campo *Verificación*, la posibilidad se acepta como válida.

Debe quedar claro del algoritmo que un usuario que intente añadir derechos que no le son permitidos simplemente invalida la posibilidad. La inversión del campo *Verificación*

en una posibilidad restringida para obtener el argumento (C XOR 00000001 en la figura 7-4) es imposible, puesto que la función f es una función de un sentido (eso es precisamente lo que significa "un sentido": no existe un algoritmo para invertirlo). Es a través de esta técnica de cifrado que las posibilidades quedan protegidas de los curiosos.

Las posibilidades se utilizan en Amoeba para dar nombres a todos los objetos y para su protección. Este mecanismo único conduce a un esquema uniforme de nombres y protección. También es por completo transparente con respecto a la posición. Para llevar a cabo una operación en un objeto, no es necesario conocer el punto donde reside. De hecho, aunque se tuviera dicho conocimiento, no hay forma de utilizarlo.

Observe que Amoeba no utiliza las listas para control de acceso para la autenticación. El esquema de protección que utiliza casi no requiere un costo administrativo. Sin embargo, en un ambiente con poca seguridad, se puede exigir un cifrado adicional (por ejemplo, cifrado de enlaces) para evitar que las posibilidades sean descubiertas en la red.

7.2.3. Operaciones estándar

Aunque muchas de las operaciones en los objetos dependen del tipo de los mismos, existen algunas operaciones que se aplican a la mayoría; éstas se muestran en la figura 7-5. Algunas de ellas necesitan que algunos bits estén activados, pero otras pueden ser llevadas a cabo por cualquier persona que pueda presentar un servidor con una posibilidad válida para uno de sus objetos.

Llamada	Descripción
Age	Lleva a cabo un ciclo de recolección de basura
Copy	Duplica el objeto y regresa una posibilidad para la copia
Destroy	Destruye el objeto y reclama su espacio de almacenamiento
Getparams	Obtiene los parámetros asociados al servidor
Info	Obtiene una cadena en ASCII que describe de manera breve al objeto
Restric	Produce una nueva posibilidad restringida para el objeto
Setparams	Configura los parámetros asociados al servidor
Status	Obtiene la información del estado actual del servidor
Touch	Pretende el objeto recién utilizado

Figura 7-5. Las operaciones estándar válidas en la mayoría de los objetos.

Es posible crear un objeto en Amoeba y entonces perder su posibilidad, por lo que se necesita un mecanismo para deshacerse de los objetos que no sean accesibles. La forma

elegida es hacer que los servidores ejecuten un recolector de basura en forma periódica, para eliminar todos los objetos no utilizados en *n* ciclos de recolección de basura. La llamada AGE inicia un nuevo ciclo de recolección de basura. La llamada TOUCH le indica al servidor que el objeto tocado sigue en uso. Cuando los objetos entran al servidor de directorios, se les toca en forma periódica, para mantener alejado al recolector de basura.

La operación COPY es una abreviatura que permite duplicar un objeto sin tener que transmitirlo. Sin esta operación, el copiado de un archivo requeriría un doble envío de éste a través de la red: del servidor al cliente y de regreso. COPY también puede buscar objetos remotos o enviar objetos a máquinas remotas.

La operación DESTROY elimina al objeto. Es evidente que necesita el derecho apropiado.

Las llamadas GETPARAMS y SETPARAMS se encargan del servidor como un todo y no de un objeto particular. Permiten que el administrador del sistema lea y escriba parámetros para el control de la operación del servidor. Por ejemplo, mediante este mecanismo se puede establecer el algoritmo para la elección de los procesadores.

Las llamadas INFO y STATUS regresan la información de estado. La primera regresa una cadena corta en ASCII, la cual describe en forma breve al objeto. La información en la cadena depende del servidor, pero en general, indica algo útil relativo al objeto (por ejemplo, en el caso de los archivos, indica el tamaño). La segunda obtiene la información relativa al servidor como un todo; por ejemplo, la cantidad de memoria libre. Esta información ayuda a un mejor monitoreo del sistema por parte del administrador del mismo.

La llamada RESTRICT genera una nueva posibilidad para el objeto, con un subconjunto de los derechos actuales, como hemos descrito.

7.3. ADMINISTRACIÓN DE PROCESOS EN AMOEBA

Un proceso en Amoeba es básicamente un espacio de direcciones y una colección de hilos que se ejecutan en él. Un proceso con un hilo es muy semejante a un proceso en UNIX o en MS-DOS,en términos de su comportamiento o su función. En esta sección explicaremos el funcionamiento de los procesos e hilos y la forma de implantarlos.

7.3.1. Procesos

Un proceso es un objeto en Amoeba. Al crear un proceso, el proceso padre obtiene una posibilidad para el proceso hijo, al igual que con cualquier otro objeto recién creado. Mediante esta posibilidad, el hijo se puede suspender, reiniciar o destruir.

La creación de un proceso en Amoeba es distinta de la de UNIX. El modelo de UNIX para la creación de un proceso hijo mediante la clonación del padre no es adecuada en un sistema distribuido, debido al costo considerable de crear primero una copia en alguna parte (FORK) para reemplazar en forma casi inmediata la copia con un programa nuevo (EXEC). En vez de esto, Amoeba permite crear un proceso nuevo en un procesador específico donde la supuesta imagen en memoria comience justo al principio. En este respecto, la creación

de un proceso en Amoeba es similar al caso de MS-DOS.Sin embargo, a diferencia de MS-DOS, un proceso puede continuar su ejecución en paralelo con su hijo, con lo cual puede crear un número arbitrario de hijos adicionales. El hijo puede, a su vez, crear sus propios hijos, lo que produce un árbol de procesos.

La administración de procesos es controlada en tres niveles distintos en Amoeba. En el nivel inferior están los servidores de procesos, hilos del núcleo que se ejecutan en cada una de las máquinas. Para crear un proceso en una máquina dada, otro proceso realiza una RPC con el servidor de procesos de esa máquina, proporcionando la información necesaria.

En el siguiente nivel tenemos un conjunto de procedimientos de biblioteca que proporcionan una interfaz más conveniente para los programas del usuario. Se tienen distintos gustos. Hacen su trabajo al llamar a los procedimientos de interfaz de bajo nivel.

Por último, la forma más sencilla de crear un proceso es utilizar el servidor de ejecución, que hace la mayor parte del trabajo para determinar el lugar donde se ejecuta el nuevo proceso. Analizaremos el servidor de ejecución en una sección posterior de este capítulo.

Algunas de las llamadas para la administración de procesos utilizan una estructura de datos llamada **descriptor del proceso** donde se proporciona la información relativa a un proceso que está por ejecutarse. Un campo del descriptor del proceso (véase la figura 7-6) indica las arquitecturas de CPU donde se puede ejecutar el proceso. En los sistemas heterogéneos, este campo es esencial para garantizar que los binarios 386 no se ejecuten en SPARC, etcétera.

Otro campo contiene la posibilidad del propietario del proceso. Cuando el proceso termina o queda incapacitado (véase más adelante), se realiza una RPC mediante esta posibilidad para informar del evento. También contiene descriptores de todos los segmentos del proceso, los cuales definen en forma colectiva su espacio de direcciones, así como los descriptores de todos sus hilos.

Por último, el descriptor del proceso contiene también un descriptor para cada hilo del proceso. El contenido de un descriptor de hilo depende de la arquitectura, pero como mínimo, contiene el contador del programa y el apuntador a la pila del hilo. También puede contener la información adicional necesaria para ejecutar el hilo, donde se incluyen otros registros, el estado del hilo y varias banderas. Los procesos más recientes contienen sólo un hilo en sus descriptores de procesos, pero los procesos incapacitados pueden haber creado más hilos antes de incapacitarse.

La interfaz de bajo nivel consta de cerca de media docena de procedimientos de biblioteca. De éstos, sólo nos interesan tres. El primero, *exec*, es el más importante. Tiene dos parámetros de entrada, la posibilidad de un servidor de procesos y un descriptor de proceso. Su función es realizar una RPC con el servidor de procesos específico para solicitar la ejecución del proceso. Si la llamada tiene éxito, una posibilidad para el nuevo proceso regresa al punto donde se hizo la llamada.

Un segundo procedimiento de biblioteca importante es *getload*. Regresa la información relativa a la velocidad del CPU, la carga actual y la cantidad de memoria libre en ese momento. El servidor de ejecución lo utiliza para determinar la mejor posición para la ejecución de un nuevo proceso.

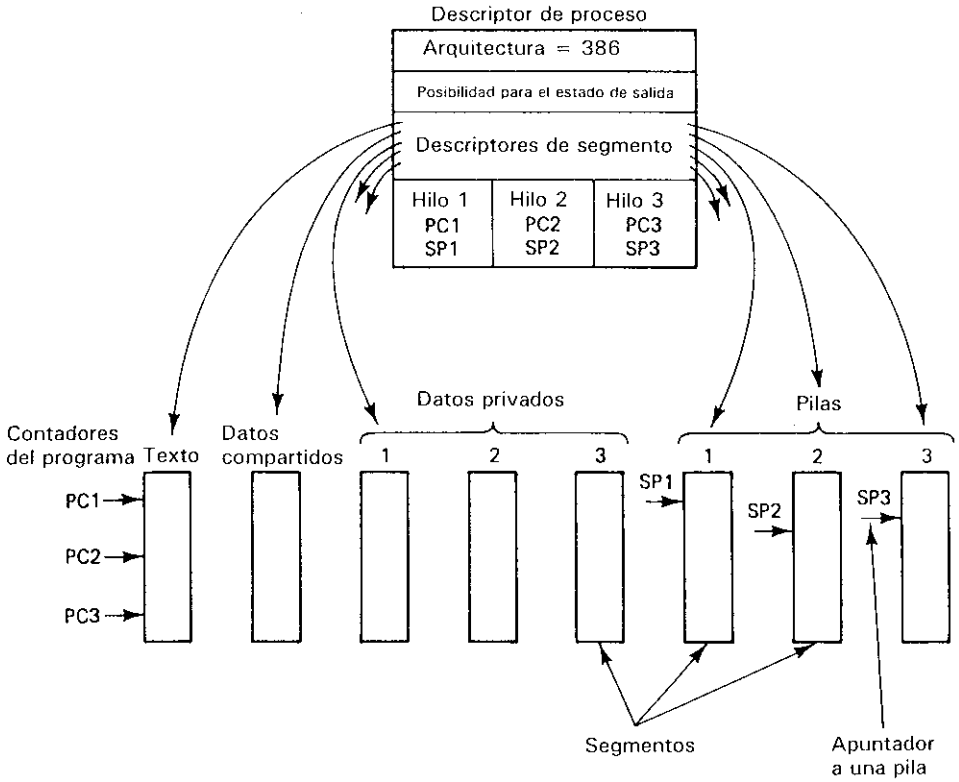


Figura 7-6. Un descriptor de proceso.

El tercer procedimiento de biblioteca importante es *stun* (incapacitar). El padre de un proceso puede suspenderse mediante una **incapacidad**. Lo más común es que el padre pueda dar la posibilidad de un proceso a un depurador, que puede incapacitarlo y volverlo a iniciar posteriormente para una depuración interactiva. Se soportan dos tipos de incapacidad: normal y de emergencia. Difieren en lo que sucede si el proceso está bloqueado en una o más RPC al momento de ser incapacitado. En una incapacidad normal, el proceso envía un mensaje al servidor que espera en ese momento, para decir, por ejemplo: "He sido incapacitado. Termina tu labor de inmediato y envíame una respuesta". Si el servidor también está bloqueado, en espera de otro servidor, el mensaje se propaga, hasta que llega al final. El servidor al final de la línea responde en forma inmediata con un mensaje de error especial. De esta forma, todas las RPC pendientes terminan de una forma casi inmediata en una forma limpia y todos los servidores terminan de manera adecuada. No se viola la estructura de anidamiento y no se necesitan "grandes saltos".

Una incapacidad de emergencia detiene el proceso al instante y no envía mensajes a los servidores que trabajan en ese momento para el proceso incapacitado. Los cálculos que

se ejecutan en los servidores se convierten en huérfanos. Cuando los servidores finalmente terminan y envían sus respuestas, éstas se descartan en última instancia.

La interfaz de procesos de alto nivel no necesita un descriptor de procesos por completo formado. Una de las llamadas, *newproc*, toma como sus primeros tres parámetros el nombre del archivo binario y los apuntadores a los arreglos del argumento y del ambiente, de manera similar al caso de UNIX. Otros parámetros proporcionan un control más detallado del estado inicial.

7.3.2. Hilos

Amoeba soporta un modelo sencillo de hilos. Al iniciar un proceso, éste tiene al menos un hilo. Durante la ejecución, el proceso puede crear más hilos y los existentes pueden terminar su labor. Así, el número de hilos es por completo dinámico. Al crearse un nuevo hilo, los parámetros de la llamada especifican el procedimiento por ejecutar y el tamaño de la pila inicial.

Aunque todos los hilos de proceso comparten el mismo texto y datos globales de un programa, cada hilo tiene su pila, su apuntador a la pila y su copia de los registros de la máquina. Además, si un hilo desea crear y utilizar variables globales a todos sus procedimientos pero invisibles para los demás hilos, se dispone de procedimientos de biblioteca para tales fines. Tales variables son **glocales**. Un procedimiento de biblioteca asigna un bloque de memoria glocal del tamaño necesario y regresa un apuntador a él. Se hace referencia a los bloques de memoria glocal mediante enteros, en vez de cadenas. Se dispone de una llamada al sistema para que un hilo adquiera su apuntador glocal.

Se dispone de tres métodos para la sincronización de hilos: señales, mútex y semáforos. Las señales son interrupciones asíncronas que se envían de un hilo a otro en el mismo proceso. Desde el punto de vista conceptual, son similares a las señales de UNIX, excepto que se envían entre hilos en vez de procesos. Las señales se pueden enviar, capturar o ignorar. Las interrupciones asíncronas entre los procesos utilizan el mecanismo de incapacidad.

La segunda forma de comunicación entre los hilos es el mútex. Un **mútex** es como un semáforo binario. Puede tener uno de dos estados, cerrado o abierto. El intento por cerrar un mútex abierto cierra éste. El hilo que realiza la llamada continúa. El intento por cerrar un mútex ya cerrado hace que el hilo que hace la llamada se bloquee hasta que otro hilo abra el mútex. Si más de un hilo espera un mútex, al momento de abrir éste, libera exactamente un hilo. Además de las llamadas para cerrar o abrir un mútex, también existe una llamada para intentar cerrar un mútex, pero si ésta no puede hacerlo durante un intervalo de tiempo dado, concluye su tiempo y regresa un código de error.

La tercera forma de comunicación entre los hilos es mediante el conteo de semáforos. Son más lentos que los mútex, pero hay ocasiones en que son necesarios. Funcionan de la manera usual, excepto que en este caso existe una llamada adicional para permitir que termine el tiempo de una operación DOWN si no tiene éxito durante cierto intervalo de tiempo.

El núcleo controla todos los hilos. La ventaja de este diseño es que cuando un hilo realiza una RPC, el núcleo puede bloquearlo y programar la ejecución de otro hilo del mismo proceso si alguno está listo. La planeación del procesamiento de los hilos se realiza mediante el uso de prioridades, donde los hilos del núcleo tienen mayor prioridad que los hilos del usuario. La planeación del procesamiento de los hilos se puede configurar con prioridades o para ejecutarse hasta terminar (es decir, que los hilos continúen su ejecución hasta bloquearse), conforme lo desee el proceso.

7.4. ADMINISTRACIÓN DE MEMORIA EN AMOEBA

Amoeba tiene un modelo de memoria en extremo sencillo. Un proceso puede tener el número de segmentos que desee y éstos se pueden localizar en cualquier parte del espacio de direcciones virtuales del proceso. Los segmentos no se intercambian ni se paginan, por lo que un proceso debe estar por completo contenido en la memoria para su ejecución. Además, aunque se utiliza el hardware MMU, cada segmento se almacena de manera adyacente a los demás en la memoria.

Aunque es posible que este diseño sea poco usual en esta época, se realizó así por tres razones: desempeño, sencillez y economía. El hecho de que un proceso esté por completo contenido en la memoria hace más rápida la RPC. Cuando hay que enviar un bloque de datos de gran tamaño, el sistema sabe que todos los datos están adyacentes, no sólo en la memoria virtual, sino también en la memoria física. Esto evita verificar si se dispone en cierto momento de todas las páginas necesarias dentro del buffer y elimina la espera de las mismas si no estuvieran dentro del buffer. De manera análoga, en la entrada, el buffer siempre está dentro de la memoria, por lo que los datos recibidos se pueden colocar ahí sin fallos de páginas. Este diseño ha permitido a Amoeba lograr tasas de transferencia muy altas para RPC de gran tamaño.

La segunda razón para este diseño es la sencillez. El hecho de no utilizar el intercambio o la paginación hace más sencillo al sistema y hace que el núcleo sea más pequeño y controlable. Sin embargo, la tercera razón es la que hace factibles a las dos primeras. La memoria será tan barata que, al cabo de pocos años, es probable que todas las máquinas Amoeba tengan cientos de megabytes de la misma. Tales memorias de gran tamaño reducirán en forma esencial la necesidad de la paginación o el intercambio, para que los programas de gran tamaño se ajusten a máquinas pequeñas.

7.4.1. Segmentos

Los procesos disponen de varias llamadas al sistema para el manejo de los segmentos. Entre las más importantes están las que permiten crear, destruir, leer y escribir segmentos. Al crear un segmento, el proceso que hizo la llamada recibe a cambio una posibilidad, la

cual es utilizada para la lectura y escritura del segmento, así como para las demás llamadas relacionadas con el segmento.

Un segmento recién creado recibe un tamaño inicial, el cual puede cambiar durante la ejecución del proceso. También puede tener un valor inicial, ya sea de otro segmento o de algún archivo.

Puesto que los segmentos se pueden leer o escribir, es posible utilizarlos para construir un servidor de archivos de la memoria principal. Para comenzar, el servidor crea un segmento del mayor tamaño posible. Puede determinar ese tamaño máximo preguntando al núcleo. Este segmento se utilizará como un disco simulado. Después, el servidor le da formato al segmento como un sistema de archivos, con todas las estructuras necesarias para llevar el registro de los archivos. Después de todo eso, se abre al público para aceptar y procesar solicitudes de los clientes.

7.4.2. Segmentos asociados

Los espacios de direcciones virtuales en Amoeba se construyen a partir de los segmentos. Al iniciar un proceso, éste debe tener al menos un segmento. Sin embargo, durante su ejecución, un proceso puede crear más segmentos y asociarlos con su espacio de direcciones en cualquier dirección virtual no utilizada. La figura 7-7 muestra un proceso con tres segmentos de memoria asociados.

Un proceso también puede desasociar segmentos. Además, un proceso puede especificar un rango de direcciones virtuales y solicitar que el rango sea desasociado, después de lo cual dichas direcciones ya no serán válidas. Al desasociar un segmento o un rango de direcciones, se regresa una posibilidad, de modo que se pueda mantener el acceso al segmento o incluso que pueda ser asociado más adelante, tal vez en una dirección virtual distinta.

Un segmento se puede asociar con el espacio de direcciones de dos o más procesos a la vez. Esto permite que los procesos operen en la memoria compartida. Sin embargo, por lo general es mejor crear un proceso con varios hilos cuando se necesite la memoria compartida. La principal razón para tener varios procesos es una mejor protección, pero si los dos procesos la comparten, lo usual es que no se desee la protección.

7.5. COMUNICACIÓN EN AMOEBA

Amoeba soporta dos formas de comunicación: RPC mediante la transferencia puntual de mensajes y la comunicación en grupo. En el nivel más bajo, una RPC consta de un mensaje de solicitud seguido de un mensaje de respuesta. La comunicación en grupo utiliza la transmisión en hardware o multitransmisión si se dispone de ésta; en caso contrario, la simula de manera transparente mediante mensajes individuales. En esta sección describiremos ambas formas de comunicación y después analizaremos el protocolo FLIP subyacente utilizado para su soporte.

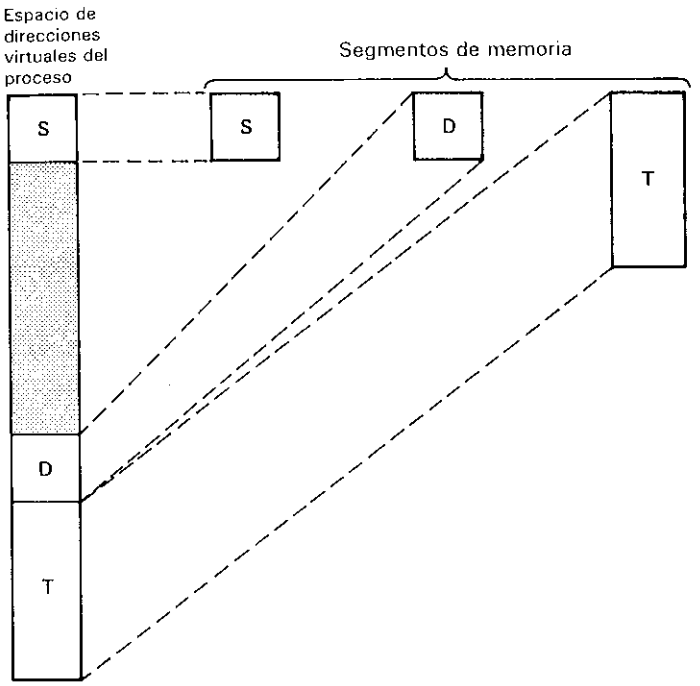


Figura 7-7. Un proceso con tres segmentos asociados en su espacio de direcciones virtuales.

7.5.1. Llamada a un procedimiento remoto (RPC)

La comunicación puntual en Amoeba consta de un cliente que envía un mensaje a un servidor, seguida de una respuesta del servidor al cliente. No es posible que un cliente envíe un mensaje y después haga otra cosa, excepto al pasar por alto la interfaz RPC, lo cual ocurre sólo bajo circunstancias muy especiales. La primitiva que envía la solicitud bloquea en forma automática al proceso que hizo la llamada hasta recibir de regreso la respuesta, lo cual obliga la existencia de cierta estructura en los programas. Las primitivas independientes *send* y *receive* se pueden pensar como la respuesta de los sistemas distribuidos al enunciado *goto*: programación spaghetti en paralelo. Deben ser evitados por los programas del usuario y utilizarse sólo mediante los sistemas de tiempo de ejecución del lenguaje que tienen requisitos de comunicación poco usuales.

Cada servidor estándar define una interfaz de procedimientos que los clientes pueden llamar. Estas rutinas de biblioteca son resguardos que empaquetan los parámetros en mensajes y llaman a las primitivas del núcleo para enviar en realidad el mensaje. Durante la transmisión de éste, el resguardo, y por tanto el hilo que hace la llamada, se bloquea. Al regresar la respuesta, el resguardo regresa al cliente el estado y los resultados. Aunque las

primitivas a nivel del núcleo están relacionadas en realidad con la transferencia de mensajes, el uso de los resguardos hace que este mecanismo aparezca como RPC a los ojos del programador, de modo que nos referiremos aquí a las primitivas básicas de comunicación como RPC, en vez de la frase más precisa "intercambio de mensajes de solicitud/respuesta".

Para que un hilo cliente realice una RPC con un hilo servidor, el cliente debe conocer la dirección del servidor. El direccionamiento se lleva a cabo al permitir que cada hilo elija un número aleatorio de 48 bits, llamado **puerto,** para que lo utilice como la dirección para los mensajes que le son enviados. Los distintos hilos de un proceso pueden utilizar diferentes puertos si así lo desean. Todos los mensajes se envían de un emisor a un puerto de destino. Un puerto no es más que cierto tipo de dirección lógica de un hilo. No existe una estructura de datos ni un espacio de almacenamiento asociados a un puerto. En ese sentido, es similar a una dirección IP o una dirección Ethernet, excepto que no está ligado a una posición física particular. El primer campo de cada posibilidad proporciona el puerto del servidor que controla el objeto (véase la figura 7-3).

Primitivas de RPC

El mecanismo RPC utiliza principalmente tres primitivas del núcleo:

1. *get_request*, indica la disposición de un servidor para escuchar a un puerto.
2. *put_reply*, llevada a cabo por un servidor cuando tiene un mensaje que desea enviar.
3. *trans,* envía un mensaje del cliente al servidor y espera la respuesta.

Los servidores utilizan las dos primeras. Los clientes utilizan la tercera para *transmitir* un mensaje y esperar una respuesta. Las tres son verdaderas llamadas al sistema; es decir, no funcionan mediante el envío de un mensaje a un hilo del servidor de comunicaciones. (Si los procesos pueden enviar mensajes, ¿por qué tendrían que contactar con un servidor con el fin de enviar un mensaje?) Sin embargo, los usuarios tienen acceso a las llamadas mediante procedimientos de biblioteca, como de costumbre.

Cuando un servidor desea irse a dormir en espera de que llegue una solicitud, llama a *get_request*. Este procedimiento tiene los tres parámetros siguientes:

```
get_request(&header, buffer, bytes)
```

El primero apunta al encabezado de un mensaje, el segundo a un buffer de datos y el tercero indica el tamaño del buffer. Esta llamada es similar a

```
read(fd, buffer, bytes)
```

en UNIX o MS-DOS, en donde el primer parámetro identifica lo que se lee, el segundo proporciona un buffer donde colocar los datos y el tercero indica el tamaño del buffer.

Cuando se transmite un mensaje a través de la red, éste contiene un encabezado y (de manera opcional) un buffer de datos. El encabezado es una estructura fija de 32 bytes y se muestra en la figura 7-8. Lo que hace el primer parámetro de la llamada *get_request* es indicar al núcleo dónde colocar el encabezado que está por recibir. Además, antes de hacer

la llamada *get_request*, el servidor debe inicializar el campo *Puerto* del encabezado para que contenga al puerto que está escuchando. Ésta es la razón por la cual el núcleo sabe cuál servidor está escuchando a cuál puerto. El encabezado recibido escribe encima del iniciado por el servidor.

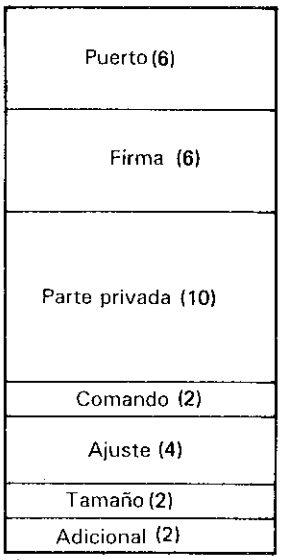


Figura 7-8. El encabezado utilizado en todos los mensajes de solicitud y respuesta en Amoeba. Los números entre paréntesis proporcionan los tamaños del campo en bytes.

Al llegar un mensaje, el servidor elimina su bloqueo. Lo normal es que primero inspeccione el encabezado para saber más de la solicitud. El campo *Firma* está reservado con fines de autenticación, pero no se utiliza por el momento.

Los demás campos no están especificados por el protocolo RPC, de modo que un cliente y un servidor se pueden poner de acuerdo para utilizarlos en la forma que deseen. Las convenciones normales son las siguientes. La mayoría de las solicitudes a los servidores contienen una posibilidad, para especificar el objeto sobre el cual se opera. Muchas respuestas también tienen una posibilidad como valor de retorno. La *parte privada* se utiliza por lo general para contener los tres campos del extremo derecho de la posibilidad.

La mayoría de los servidores soportan varias operaciones en sus objetos, como la lectura, la escritura y su destrucción. El campo *Comando* se utiliza en general para las solicitudes que indican el tipo de operación necesaria. En las respuestas, indica si la operación tuvo éxito o no; en el segundo caso, da la razón para la falla.

Los últimos tres campos contienen parámetros, si es que existen. Por ejemplo, al leer un segmento o archivo, éstos se pueden utilizar para indicar el ajuste dentro del objeto para comenzar a leer desde dicho punto y el número de bytes por leer.

Observe que para la mayoría de las operaciones, no se necesita o utiliza buffer alguno. De nuevo, en el caso de la lectura, la posibilidad del objeto, el ajuste y el tamaño caben

perfectamente dentro del encabezado. Durante la escritura, el buffer contiene los datos por escribir. Por otro lado, la respuesta a un READ contiene un buffer, mientras que la respuesta a un WRITE no.

Después que el servidor termina su trabajo, hace una llamada

```
put_reply(&header, buffer, bytes)
```

para enviar de regreso la respuesta. El primer parámetro proporciona el encabezado y el segundo el buffer. El tercero indica el tamaño del buffer. Si un servidor hace un *put_reply* sin que haya hecho antes un *get_request* no concordante con él, *put_reply* falla con un error. De manera similar, dos llamadas consecutivas *get_request* fallan. Las dos llamadas deben estar apareadas de manera correcta.

Pasemos ahora del servidor al cliente. Para realizar una RPC, el cliente llama a un resguardo, el cual hace la siguiente llamada:

```
trans(&header1, buffer1, bytes1, &header2, buffer2, bytes2)
```

Los primeros tres parámetros proporcionan la información relativa al encabezado y buffer de la solicitud saliente. Los últimos tres proporcionan la misma información de la respuesta entrante. La llamada *trans* envía la solicitud y bloquea al cliente hasta que llega la respuesta. Este diseño obliga a los procesos a apegarse de forma estricta al paradigma de comunicación RPC cliente-servidor, en forma análoga al caso de las técnicas de programación estructurada, las cuales evitan que los programadores hagan cosas que conduzcan de manera eventual a programas con una estructura pobre (como el uso sin restricciones de enunciados GOTO).

Si Amoeba trabajara en realidad de esta forma, sería posible que un intruso personificase a un servidor, al hacer un *get_request* en el puerto del servidor. Después de todo, estos puertos son públicos, ya que los clientes deben conocerlos para poder tener contacto con los servidores. Amoeba resuelve este problema con cipramiento. Cada puerto es en realidad una pareja de puertos: el **puerto de obtención,** que es privado, sólo conocido por el servidor; y el **puerto de envío** conocido por todos. Los dos están relacionados mediante una función de un sentido, F, de acuerdo con la relación

$$\text{puerto de envío} = F\,(\text{puerto de recepción})$$

La función de un sentido es de hecho la misma que se utiliza para la protección de las posibilidades, puesto que los dos conceptos no están relacionados entre sí.

Cuando un servidor ejecuta un *get_request*, el núcleo calcula el correspondiente puerto de envío y lo almacena en una tabla de puertos escuchados. Todas las solicitudes *trans* utilizan los puertos de envío, de forma que al arribar un paquete a una máquina, el núcleo compara el puerto de envío del encabezado con los puertos de envío de su tabla para ver si coinciden. El esquema es seguro, puesto que los puertos de recepción nunca aparecen en la red y no pueden obtenerse a partir de los puertos de envío de conocimiento público. Esto se muestra en la figura 7-9 y se describe con mayor detalle en (Tanenbaum *et al.*, 1986).

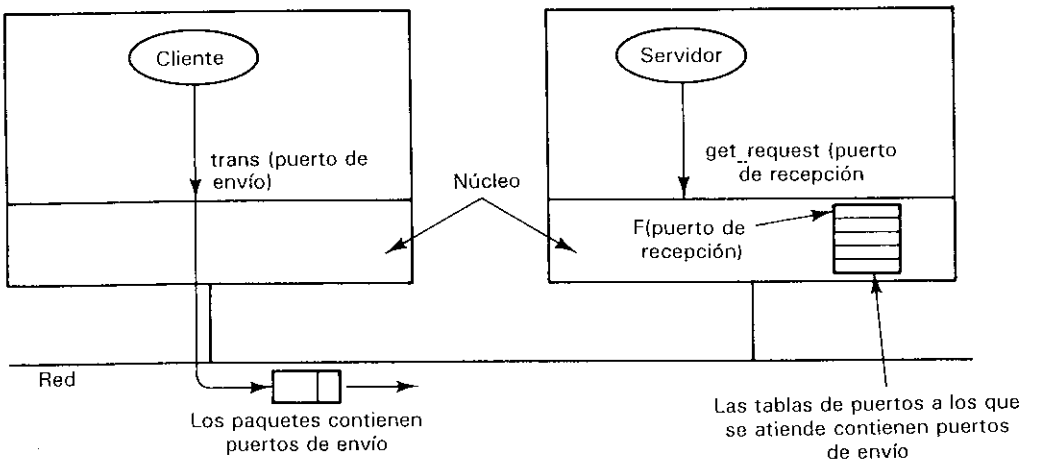


Figura 7-9. Relación entre el puerto de obtención y el puerto de envío.

La RPC de Amoeba soporta la semántica "al menos una vez". En otras palabras, cuando se lleva a cabo una RPC, el sistema garantiza que una RPC no se llevará a cabo más de una vez, incluso en el caso de fallas del servidor y un nuevo arranque rápido.

7.5.2. Comunicación en grupo en Amoeba

RPC no es la única forma de comunicación soportada por Amoeba. También soporta la comunicación en grupo. Un grupo en Amoeba consta de uno o más procesos que cooperan para llevar a cabo cierta tarea o proporcionar algún servicio. Los procesos pueden ser miembros de varios grupos al mismo tiempo. Los grupos son cerrados, lo que significa que sólo los miembros pueden realizar transmisiones al grupo. La forma usual para que un cliente tenga acceso a un servicio proporcionado por un grupo es que realice una RPC con alguno de sus miembros. Ese miembro utiliza entonces la comunicación en grupo dentro del mismo, en caso necesario, para determinar lo que debe realizar cada miembro.

Este diseño se eligió para obtener mayor grado de transparencia que en el caso de una estructura de grupos abiertos. La idea detrás de él es que los clientes utilizan por lo general la RPC para conversar con servidores individuales, por lo que también utilizan RPC para conversar con los grupos. La alternativa (grupos abiertos y el uso de RPC para comunicarse con un servidor pero utilizar comunicación en grupo para hablar con los servidores del grupo) es mucho menos transparente. (El uso de la comunicación en grupo para todo eliminaría las múltiples ventajas de RPC ya analizadas.) Una vez determinado que los clientes fuera de un grupo utilizarán RPC para hablar con él (en realidad, para hablar con un proceso del grupo), desaparece la necesidad de los grupos abiertos, por lo cual son adecuados los grupos cerrados, que además son más fáciles de implantar.

Primitivas para la comunicación en grupo

Las operaciones disponibles para la comunicación en grupo en Amoeba se muestran en la figura 7-10. *CreateGroup* crea un nuevo grupo y regresa un identificador de grupo que se utiliza en las llamadas posteriores para indicar el grupo del cual se trata. Los parámetros especifican varios tamaños y la tolerancia de fallas necesaria (el número de miembros muertos para que el grupo permanezca activo y continúe su funcionamiento de manera correcta).

Llamada	Descripción
CreateGroup	Crea un nuevo grupo y configura sus parámetros
JoinGroup	Une al proceso que hizo la llamada como miembro de un grupo
LeaveGroup	Elimina al proceso que hizo la llamada de un grupo
SendToGroup	Envía de manera confiable un mensaje a todos los miembros de un grupo
ReceiveFromGroup	Se bloquea hasta que llega un mensaje de un grupo
ResetGroup	Inicia la recuperación después de que falla un proceso

Figura 7-10. Primitivas de la comunicación en grupo en Amoeba.

JoinGroup y *LeaveGroup* permiten la entrada y salida de procesos de los grupos existentes. Uno de los parámetros de *JoinGroup* es un pequeño mensaje que se envía a todos los miembros del grupo para anunciar la llegada de un nuevo miembro. De manera análoga, uno de los parámetros de *LeaveGroup* es otro pequeño mensaje que se envía a todos los miembros para despedirse y desearle buena suerte en sus próximas actividades. La idea de los pequeños mensajes es permitir que todos los miembros del grupo sepan quiénes son sus compañeros, en caso de que estén interesados en ello; por ejemplo, para reconstruir el grupo si fallan algunos miembros. El grupo se destruye cuando el último miembro del grupo llama a *LeaveGroup*.

SendToGroup transmite de manera atómica un mensaje a todos los miembros de un grupo determinado, a pesar de que existan mensajes perdidos, buffers finitos o fallas del procesador. Amoeba soporta un ordenamiento global del tiempo, por lo que si dos procesos llaman a *SendToGroup* de manera casi simultánea, el sistema garantiza que todos los miembros del grupo recibirán los mensajes en el mismo orden. Esto queda garantizado; los programadores pueden contar con ello. Si las dos llamadas fueran con exactitud simultáneas, se declara como primera la que envía su paquete con éxito en primer lugar a través de la LAN. En términos de la semántica analizada en el capítulo 6, este modelo corresponde a la consistencia secuencial, no a la consistencia estricta.

ReceiveFromGroup intenta obtener un mensaje de un grupo determinado. Si no se dispone de un mensaje (es decir, uno guardado en un buffer por el núcleo), el proceso que hace la llamada se bloquea en espera de él. Si ya existe un mensaje disponible, el proceso que hizo la llamada lo recibe sin mayor retraso. El protocolo garantiza que sin fallas del procesador, no se perderán de manera irremediable los mensajes. También se puede hacer que el protocolo tolere las fallas, con un costo adicional, como analizaremos posteriormente.

La última llamada, *ResetGroup*, se utiliza para la recuperación en caso de fallas. Especifica el número mínimo de miembros de un grupo. Si el núcleo puede establecer contacto con el número solicitado de procesos y puede reconstruir el grupo, regresa el tamaño del nuevo grupo. En caso contrario, falla y la recuperación queda a cargo del programa usuario.

El protocolo de transmisión confiable de Amoeba

Analicemos ahora la forma en que Amoeba implanta la comunicación en grupo. Amoeba trabaja mejor en las LAN que soportan la multitransmisión o la transmisión simple (o ambas, como es el caso de Ethernet). Para hacer más sencilla la exposición, nos centraremos en la transmisión, aunque de hecho la implantación utiliza la multitransmisión cuando debe evitar molestar a las máquinas no interesadas en el mensaje enviado. Se supone que la transmisión mediante el hardware es buena, pero no perfecta. En la práctica, es rara la pérdida de paquetes, pero en ciertas ocasiones ocurre la sobre-ejecución del receptor. Puesto que estos errores pueden aparecer, no pueden ser ignorados simplemente, por lo que el protocolo está diseñado para enfrentarlos.

La idea fundamental que forma la base de la implantación de la comunicación en grupo es la **transmisión confiable**. Esto quiere decir que si un proceso usuario transmite un mensaje (por ejemplo, mediante *SendToGroup*), el mensaje proporcionado por el usuario se envía de manera correcta a todos los miembros del grupo, incluso aunque el hardware pueda perder paquetes. Por el momento, supondremos que los procesadores no tienen fallas. Más adelante consideraremos el caso de los procesadores no confiables. La siguiente descripción es apenas un bosquejo. Para mayores detalles, véase (Kaashoek y Tanenbaum, 1991; y Kaashoek *et al.,* 1989). Otros protocolos de transmisión confiable se analizan en (Birman y Joseph, 1987a; Chang y Maxemchuk, 1984; García-Molina y Spauster, 1991; Luan y Gligor, 1990; Melliar-Smith *et al.*, 1990; y Tseung, 1989).

La configuración hardware/software necesaria para la transmisión confiable en Amoeba se muestra en la figura 7-11. El hardware de todas las máquinas es idéntico por lo general y todas ejecutan de igual forma el mismo núcleo. Sin embargo, cuando la aplicación inicia, una de las máquinas se elige como secuenciador (como un comité que elige un presidente). Si el secuenciador falla en cierto momento, los demás miembros eligen uno nuevo. Se conocen muchos algoritmos de elección; como por ejemplo, elegir el proceso con la máxima dirección en la red. En una sección posterior de este capítulo analizaremos la tolerancia de fallas.

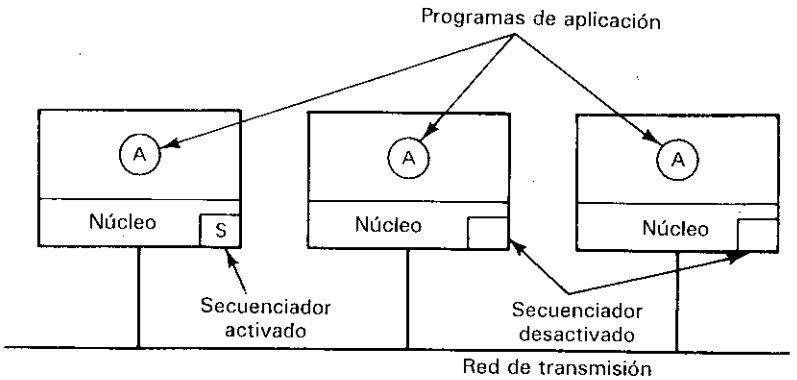


Figura 7-11. Estructura del sistema para la comunicación en grupo en Amoeba.

Se puede utilizar la siguiente secuencia de eventos para lograr una transmisión confiable:

1. El proceso usuario hace un señalamiento al núcleo y le transfiere el mensaje.
2. El núcleo acepta el mensaje y bloquea al proceso usuario.
3. El núcleo envía un mensaje puntual al secuenciador.
4. Cuando el secuenciador recibe el mensaje, le asigna el siguiente número secuencial disponible, coloca este número en un campo reservado para él en el encabezado y transmite el mensaje (y el número secuencial).
5. Cuando el núcleo emisor ve que el mensaje ha sido transmitido, elimina el bloqueo del proceso para que continúe su ejecución.

Consideremos ahora cada uno de estos pasos con más detalle. Cuando el proceso de cierta aplicación ejecuta una primitiva de transmisión, como *SendToGroup*, ocurre un señalamiento al núcleo. El núcleo bloquea entonces al proceso que hace la llamada y construye un mensaje que contiene el encabezado proporcionado por el usuario y los datos dados por la aplicación. El encabezado contiene el tipo de mensaje (*Solicitud de transmisión* en este caso), un identificador único del mensaje (que se utiliza para detectar los duplicados), el número de la última transmisión recibida por el núcleo (que por lo general recibe el nombre de **reconocimiento a cuestas**) y alguna información adicional.

El núcleo envía el mensaje al secuenciador mediante un mensaje puntual normal, a la vez que inicia un cronómetro. Si la transmisión regresa antes de que se termine el tiempo de éste (el caso normal), el núcleo emisor detiene el cronómetro y regresa el control al proceso que hizo la llamada. En la práctica, ese caso ocurre cerca del 99% de las veces, puesto que las LAN son muy confiables.

Por otro lado, si la transmisión no regresa antes de que se termine el tiempo, el núcleo supone que el mensaje o la transmisión se han perdido. De cualquier forma, transmite de nuevo el mensaje. Si se perdió el mensaje original, no se hará ningún daño y el segundo (o posterior) intento realizará una transmisión normal. Si el mensaje llegó al secuenciador y fue transmitido, pero el emisor perdió la transmisión, el secuenciador detectará la nueva transmisión como un duplicado (mediante el identificador del mensaje) y simplemente indicará al emisor que todo está bien. El mensaje no se transmite una segunda vez.

Una tercera posibilidad es que una transmisión regrese antes de que expire el tiempo, pero que sea la transmisión equivocada. Esta situación aparece cuando dos procesos intentan transmitir de manera simultánea. Uno de ellos, *A*, llega primero al secuenciador y se envía su mensaje. *A* ve la transmisión y elimina el bloqueo de su programa de aplicación. Sin embargo, su competidor, *B*, ve la transmisión de *A* y se da cuenta de que ha fallado en lograr llegar primero. Sin embargo, *B* sabe que es probable que su mensaje llegue al secuenciador (puesto que los mensajes perdidos son raros), donde se formará en una cola y será transmitido posteriormente. Así, *B* acepta la transmisión de *A* y sigue en espera de que regrese su propia transmisión o que termine el tiempo de su cronómetro.

Consideremos ahora lo que ocurre en el secuenciador si éste recibe una *solicitud de transmisión*. Primero se verifica si el mensaje es una retransmisión, en cuyo caso se informa al emisor que la transmisión ya se ha llevado a cabo, como hemos mencionado. Si el mensaje es nuevo (caso normal), se le asigna el siguiente número secuencial y el contador secuencial se incrementa en uno para la siguiente vez. Entonces, el mensaje y su identificador se guardan en un **buffer de historia** y se transmite el mensaje. Éste se transfiere también a la aplicación que se ejecuta en la máquina del secuenciador (puesto que la transmisión no provoca una interrupción en la máquina que realizó la transmisión).

Por último, consideremos lo que ocurre si un núcleo recibe una transmisión. En primer lugar, el número secuencial se compara con el número secuencial de la transmisión más reciente. Si el primer número es una unidad mayor que el segundo (caso normal), esto indica que no se han perdido transmisiones, por lo que el mensaje se transfiere al programa de aplicación, si éste está esperando. Si no está esperando, el mensaje se almacena en un buffer hasta que el programa llame a *ReceiveFromGroup*.

Supongamos que la transmisión recibida tiene el número secuencial 25, mientras que la anterior tiene el número 23. El núcleo es alertado de inmediato ante el hecho de que el número 24 se ha perdido, por lo que envía un mensaje puntual al secuenciador solicitando una retransmisión particular del mensaje perdido. El secuenciador busca el mensaje en el buffer de historia y lo envía. Al arribar, el núcleo receptor procesa 24 y 25 y los transfiere al programa de aplicación en orden consecutivo. Así, el único efecto de un mensaje perdido es un pequeño retraso (normalmente). Todos los programas de aplicación ven las transmisiones en el mismo orden, incluso aunque se pierdan los mensajes.

El protocolo confiable de transmisión se muestra en la figura 7-12. Aquí, el programa de aplicación que se ejecuta en la máquina *A* envía un mensaje *M* a su núcleo para su transmisión. El núcleo envía el mensaje al secuenciador, donde se le asigna el número secuencial 25. El mensaje (que contiene el número secuencial 25) se envía entonces a todas

las máquinas y también se transfiere al programa de aplicación del propio secuenciador. Este mensaje transmitido se denota por *M25* en la figura.

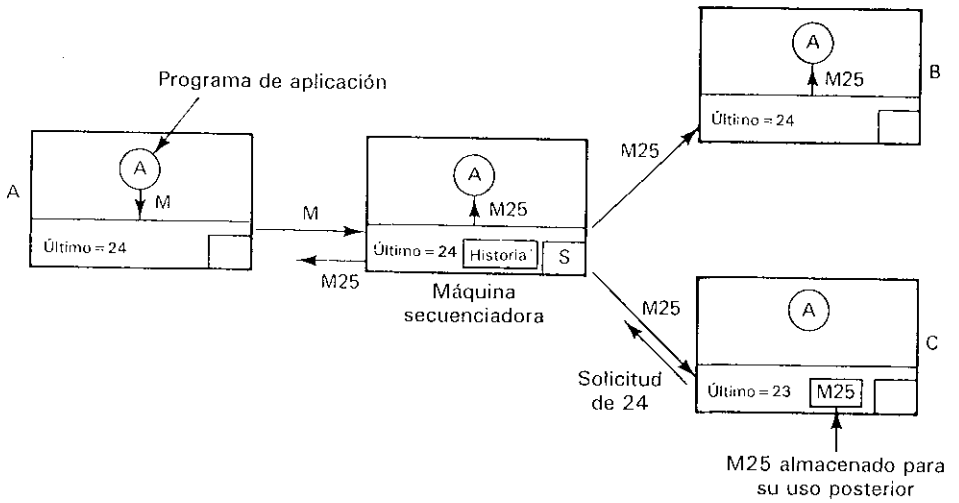


Figura 7-12. La aplicación de la máquina *A* envía un mensaje al secuenciador, el cual le añade un número secuencial (25) y lo transmite. En *B* se acepta, pero en *C* se almacena hasta que 24, que se ha perdido, se pueda recuperar por medio del secuenciador.

El mensaje *M25* llega a las máquinas *B* y *C*. En la máquina *B*, el núcleo ve que ya ha procesado todas las transmisiones hasta la número 24, de modo que transfiere de manera inmediata *M25* a su programa de aplicación. Sin embargo, en *C*, el último mensaje recibido fue el 23 (se debe haber perdido el 24), por lo que *M25* se almacena en un buffer del núcleo y se envía al secuenciador un mensaje puntual solicitando 24. Sólo después de recibir la respuesta y dársela al programa de aplicación se podrá transmitir el mensaje *M25*.

Ahora analicemos el manejo del buffer de historia. A menos que se haga algo por evitarlo, el buffer de historia se llenará muy pronto. Sin embargo, si el secuenciador supiera que todas las máquinas han recibido de manera correcta todas las transmisiones de la 0 a la 23, éstas se pueden eliminar de su buffer de historia.

Se dispone de varios mecanismos para permitir al secuenciador que descubra esta información. El mecanismo básico es que cada mensaje *Solicitud de transmisión* enviado al secuenciador cargue un reconocimiento a cuestas *k* para indicar que todas las transmisiones hasta la *k* se han recibido en forma correcta. De esta forma, el secuenciador puede mantener una tabla a cuestas cuyo índice sea el número de máquina, donde se indique la última transmisión recibida por cada una de ellas. Cuando el buffer de historia se empiece a colmar, el secuenciador puede revisar dicha tabla para encontrar el valor más pequeño. Puede entonces descartar con seguridad todos los mensajes hasta ese valor.

Si una máquina guarda silencio durante un intervalo de tiempo poco usual, el secuenciador no conocerá su estado. Para informar al secuenciador, se le pide que envíe un mensaje

corto de reconocimiento cuando no transmita mensajes durante cierto período. Además, el secuenciador puede enviar un mensaje *Solicitud de estado* donde indique a las demás máquinas que deben enviar un mensaje con el número de la última transmisión recibida. De esta forma, el secuenciador puede actualizar su tabla a cuestas y truncar su buffer de historia.

Aunque los mensajes *Solicitud de estado* son raros en la práctica, ocurren, con lo que el número promedio de mensajes necesarios para una transmisión confiable es mayor de 2, incluso aunque no se pierdan mensajes. El efecto crece con el número de máquinas.

Existe un punto sutil del diseño referente a este protocolo que hay que aclarar. Existen dos formas de realizar la transmisión. En el método 1 (ya descrito), el usuario envía un mensaje puntual al secuenciador, el cual lo transmite a su vez. En el método 2, el usuario transmite el mensaje, junto con un identificador. Cuando el secuenciador ve esto, transmite un mensaje especial *Aceptado* con el identificador único y su número secuencial recién asignado. Una transmisión sólo es "oficial" cuando se envía el mensaje *Aceptado*. Los dos métodos se comparan en la figura 7-13.

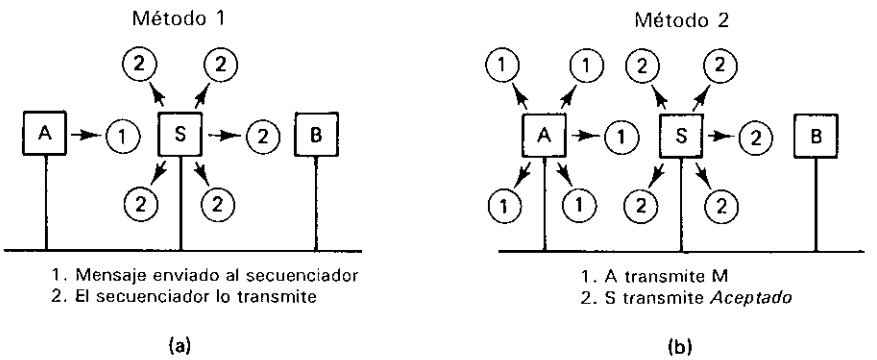


Figura 7-13. Dos métodos para realizar una transmisión confiable.

Estos protocolos son por lógica equivalentes, pero tienen distintas características de desempeño. En el método 1, cada mensaje aparece completo en la red dos veces, una vez hacia el secuenciador y otra desde el secuenciador. Así, un mensaje con una longitud de m bytes consume $2m$ bytes de ancho de banda de la red. Sin embargo, sólo se trasmite la segunda vez, por lo que cada máquina usuario es interrumpida una vez (para el segundo mensaje).

En el método 2, el mensaje completo sólo aparece una vez en la red, junto con un pequeño mensaje *Aceptado* de parte del secuenciador, por lo que tan sólo se consume la mitad del ancho de banda. Por otro lado, cada máquina se interrumpe dos veces, una para el mensaje y otra para *Aceptado*. Así, el método 1 desperdicia ancho de banda para reducir

las interrupciones en comparación con el método 2. Uno de los métodos se puede preferir sobre el otro, según el tamaño promedio de los mensajes.

En resumen, este protocolo permite realizar una transmisión confiable en una red no confiable, mediante dos mensajes por cada transmisión confiable. Cada transmisión es indivisible y todas las aplicaciones reciben todos los mensajes en el mismo orden, sin importar el número de ellos que se pierdan. Lo peor que puede pasar es que se presente un pequeño retraso cuando se pierda un mensaje, lo cual ocurre rara vez. Si dos procesos intentan transmitir al mismo tiempo, uno de ellos llega en primer lugar al secuenciador y gana. El otro verá que una transmisión de su competidor regresa del secuenciador y se dará cuenta de que su solicitud está formada en una cola y que aparecerá en breve, por lo que simplemente se dedica a esperarla.

Comunicación en grupo tolerante de fallas

Hasta este momento hemos supuesto que ninguno de los procesadores falla. De hecho, este protocolo está diseñado para soportar la pérdida de una colección arbitraria de k procesadores (incluido el secuenciador), donde k (el grado de tolerancia) se selecciona al crear el grupo. Mientras más grande sea k, se necesitará más redundancia y la operación será cada vez más lenta que el caso normal, por lo que el usuario debe elegir con cuidado k. Adelante daremos el esquema del algoritmo de recuperación. Para mayores detalles, véase (Kaashoek y Tanenbaum, 1991).

Cuando un procesador falla, al principio nadie detecta este evento. Sin embargo, tarde o temprano, alguno de los núcleos notará que los mensajes enviados a la máquina fallida no son reconocidos, por lo que el núcleo señala al procesador fallido como muerto y al grupo como inutilizable. Todas las primitivas de comunicación en grupo en esa máquina fallarán (regresan un estado de error) hasta reconstruir el grupo.

Poco después de observar el problema, uno de los procesos usuario que ha obtenido un retorno de error llama a *ResetGroup* para iniciar la recuperación. Ésta se lleva a cabo en dos etapas (García-Molina, 1982). En la primera, se elige un proceso como coordinador. En la segunda, el coordinador reconstruye al grupo y actualiza todos los demás procesos. En ese momento continúa la operación normal.

En la figura 7-14(a) vemos un grupo de seis máquinas, de las cuales la máquina 5, el secuenciador, acaba de fallar. Los números de los cuadros indican el último mensaje recibido de forma correcta por cada máquina. Dos máquinas, 0 y 1, detectan al mismo tiempo la falla del secuenciador y ambas llaman a *ResetGroup* para iniciar la recuperación. Esta llamada hace que el núcleo envíe un mensaje a todos los demás miembros para que participen en la recuperación, solicitándoles el número secuencial máximo de los mensajes advertidos por ellos. En este momento se descubre que dos procesos se han declarado a sí mismos coordinadores. Aquél que ha visto el mensaje con el número secuencial mayor gana. En caso de un empate, gana el que tenga la mayor dirección en la red. Esto produce un solo coordinador, como se muestra en la figura 7-14(b).

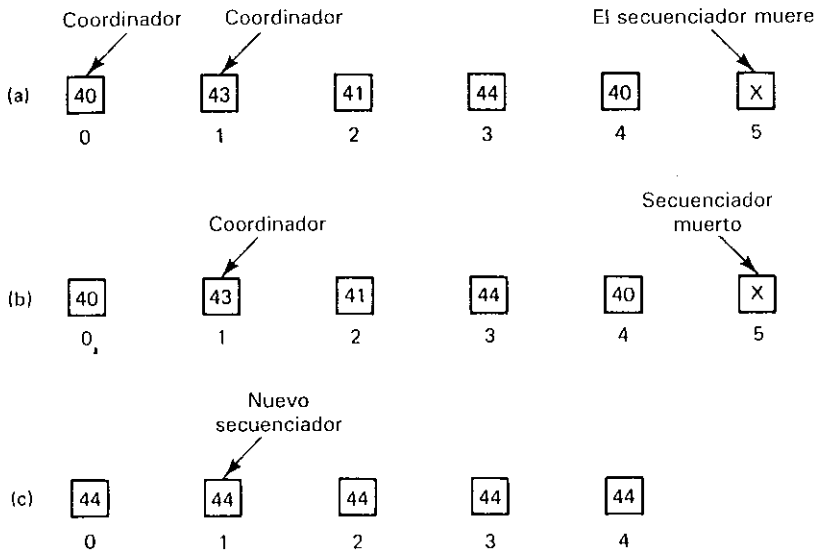


Figura 7-14. (a) El secuenciador falla. (b) Se selecciona un coordinador. (c) Recuperación.

Una vez que el coordinador entra en funciones, éste recoge de los demás miembros los mensajes que podrían haberse perdido. Ahora está actualizado y puede convertirse en el nuevo secuenciador. Construye un mensaje *Resultados* donde se anuncia como secuenciador e indica a los demás el máximo número secuencial. Cada miembro puede entonces solicitar los mensajes perdidos. Cuando un miembro esté actualizado, envía un mensaje de reconocimiento al nuevo secuenciador. Cuando el nuevo secuenciador tenga un reconocimiento de todos los miembros sobrevivientes, sabrá que todos los mensajes han sido entregados en orden a los programas de aplicación, por lo que descarta su buffer de historia y puede continuar la operación normal.

Queda pendiente otro problema: ¿Cómo obtiene el coordinador los mensajes perdidos si el secuenciador falla? La solución consiste en el valor de k, el grado de tolerancia elegido durante la creación del grupo. Si $k = 0$ (caso no tolerante de fallas), sólo el secuenciador mantiene un buffer de historia. Sin embargo, si k es mayor que 0, $k + 1$ máquinas tienen un buffer de historia actualizado. Así, si falla una colección arbitraria de k máquinas, se garantiza que al menos sobrevive un buffer de historia y éste le puede proporcionar al coordinador los mensajes que necesite. Las demás máquinas pueden mantener sus buffers de historia mediante el examen de la red.

Existe un problema más por resolver. Por lo general, una primitiva *SendToGroup* termina con éxito si el secuenciador ha recibido y transmitido o aprobado el mensaje. Si $k > 0$, este protocolo es insuficiente para sobrevivir a k fallas arbitrarias. En vez de esto, se utiliza una versión ligeramente modificada del método 2. Cuando el secuenciador ve un mensaje

M que acaba de transmitirse, no transmite de inmediato un mensaje *Aceptado*, como lo haría en el caso $k = 0$, sino que espera hasta que los k núcleos de número más pequeño reconozcan que lo han visto y almacenado. Sólo hasta ese momento el secuenciador transmite el mensaje *Aceptado*. Puesto que ahora se sabe que $k + 1$ máquinas (incluido el secuenciador) han almacenado *M* en su buffer de historia, aunque fallen k máquinas, *M* no se perderá.

Como en el caso usual, ninguno de los núcleos transfiere *M* a su programa de aplicación hasta no ver el mensaje *Aceptado*. Como este mensaje no se genera hasta que $k + 1$ máquinas hayan almacenado *M*, se garantiza que si una máquina obtiene *M*, todas las obtendrán en cierto momento. De esta forma, siempre es posible recuperarse de la pérdida de cualesquiera k máquinas. De manera colateral, para mejorar la operación en el caso $k > 0$, siempre que se guarde algún dato en un buffer de historia, se transmite un pequeño paquete de control para anunciar este hecho al mundo.

En resumen, el esquema de comunicación en grupo de Amoeba garantiza la transmisión atómica con un ordenamiento global con respecto del tiempo, aunque fallen k máquinas, donde k es un valor elegido por el usuario durante la creación del grupo. Este mecanismo proporciona una base fácil de comprender para la realización de programación distribuida. Se utiliza en Amoeba con el fin de soportar la memoria compartida distribuida basada en objetos, para el lenguaje de programación Orca y otras facilidades. También se puede implantar de manera eficiente. Las mediciones hechas con CPU 68030 en una Ethernet de 10Mb/segundo muestran que es posible controlar de manera continua 800 transmisiones confiables por segundo (Tanenbaum *et al.*, 1992).

7.5.3. El protocolo Internet Fast Local (FLIP)

Amoeba utiliza un protocolo adaptado llamado **FLIP** (siglas en inglés del **Protocolo local rápido Internet**) para la transmisión real de los mensajes. Este protocolo controla tanto la RPC como la comunicación en grupo y está por debajo de ellos en la jerarquía de protocolos. En términos de OSI, FLIP es un protocolo con capas de red, mientras que la RPC es más un protocolo de sesión o transporte sin conexión (es discutible la posición exacta, puesto que OSI fue diseñado para la redes orientadas a conexiones). Desde un punto de vista conceptual, FLIP se puede reemplazar por cualquier otro protocolo con capas de red, como IP, pero esto provocaría la pérdida de transparencia por parte de Amoeba. Aunque FLIP fue diseñado en el contexto de Amoeba, se pretende que sea útil en otros sistemas. En esta sección describiremos su diseño e implantación.

Requisitos de protocolo para los sistemas distribuidos

Antes de entrar en los detalles de FLIP, es útil comprender algunas de las razones para su diseño. Después de todo, existen ya muchos protocolos, de modo que hay que justificar la invención de uno nuevo. En la figura 7-15 enumeramos los principales requisitos que

debe cumplir un protocolo para un sistema distribuido. En primer lugar, el protocolo debe soportar tanto la RPC como la comunicación en grupo de manera eficiente. Si la red subyacente tiene transmisión o multitransmisión en hardware, como es el caso de Ethernet, el protocolo debe utilizar esto para la comunicación en grupo. Por otro lado, si la red no tiene estas características, la comunicación en grupo debe seguir funcionando de la misma forma, aunque la implantación deba ser distinta.

Elemento	Descripción
RPC	El protocolo debe soportar RPC
Group communication	El protocolo debe soportar la comunicación en grupo
Process migration	Los procesos deben poder llevar con ellos sus direcciones
Security	Los procesos no deben suplantar a otros procesos
Network management	Se necesita un soporte para la reconfiguración automática
Wide-area networks	El protocolo también debe funcionar en redes de área amplia

Figura 7-15. Características deseables para un protocolo de sistema distribuido.

Una característica cada vez más importante es el soporte de la migración de un proceso. Un proceso debe poder desplazarse de una máquina a otra, incluso a una máquina en una red distinta, sin que nadie lo note. Los protocolos tales como OSI, X.25 y TCP/IP, los cuales utilizan las direcciones de las máquinas para identificar los procesos, dificultan la migración de éstos, puesto que un proceso no puede cargar su dirección al desplazarse.

La seguridad también es un aspecto importante. Aunque los puertos de envío y recepción proporcionan la seguridad en Amoeba, también debe existir un mecanismo de seguridad en el protocolo de paquetes de forma que se pueda utilizar en el caso de los sistemas operativos que no tengan direcciones seguras del tipo cifrado que posee Amoeba.

Otro punto donde la mayoría de los protocolos existentes tienen malas calificaciones es la administración de la red. No deberían ser necesarias unas tablas complejas de configuración para indicar las conexiones entre las distintas redes. Además, si la configuración cambia, debido a la desaparición o restablecimiento de compuertas, el protocolo se debe adaptar a la nueva configuración de manera automática.

Por último, el protocolo debe funcionar tanto en las redes de área local como en las redes de área amplia. En particular, se debe utilizar el mismo protocolo en ambas.

La interfaz FLIP

El protocolo FLIP y su arquitectura asociada fueron diseñados para cubrir todos estos requisitos. En la figura 7-16 se muestra una configuración típica FLIP. Aquí vemos cinco

máquinas, dos en una Ethernet y tres en un anillo de fichas. Cada máquina tiene un proceso usuario, desde *A* hasta *E*. Una de las máquinas se conecta a ambas redes y funciona de manera automática como una compuerta. Las compuertas también pueden ejecutar clientes y servidores, como los demás nodos.

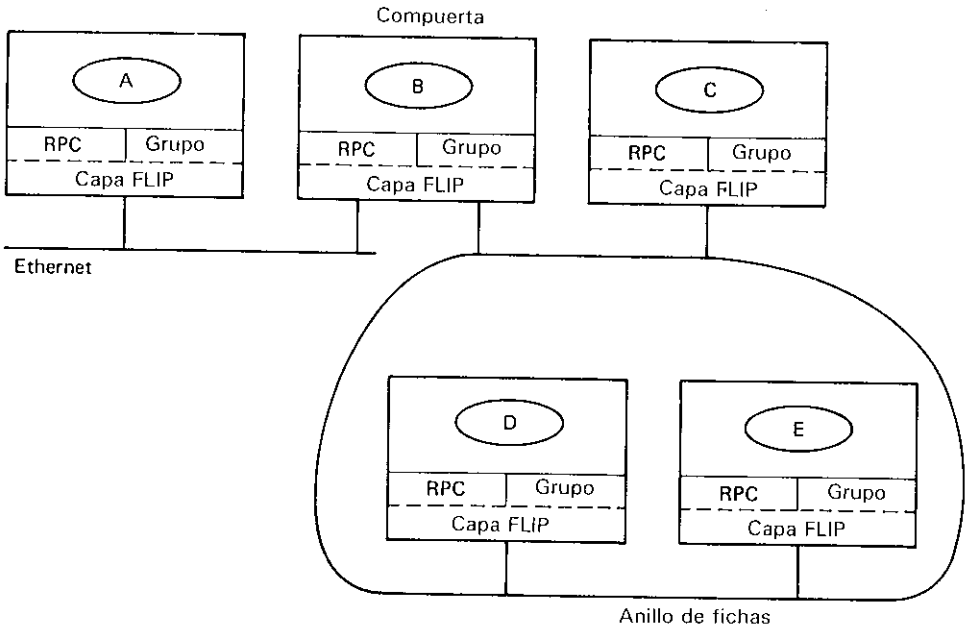


Figura 7-16. Un sistema FLIP con cinco máquinas y dos redes.

El software está estructurado como se muestra en la figura 7-16. El núcleo contiene dos capas. La capa superior controla las llamadas de los procesos usuario para los servicios de RPC o la comunicación en grupo. La capa inferior controla el protocolo FLIP. Por ejemplo, cuando un cliente llama a *trans,* hace un señalamiento al núcleo. La capa RPC examina el encabezado y el buffer, construye un mensaje mediante ambos y transfiere éste hacia la capa FLIP para su transmisión.

Toda la comunicación de bajo nivel en Amoeba se basa en las **direcciones FLIP**. Cada proceso tiene exactamente una dirección FLIP: un número aleatorio de 64 bits elegido por el sistema durante la creación del proceso. Si el proceso llega a emigrar, lleva su dirección FLIP con él. Si la red se configura de nuevo en cierto momento, de modo que todas las máquinas tengan nuevos números o direcciones en la red (en hardware), las direcciones FLIP permanecen sin modificación. Es el hecho de que una dirección FLIP identifica de manera única a un proceso (y no a una máquina) lo que hace la comunicación en Amoeba insensible a los cambios en la topología de la red y el direccionamiento en la misma.

Una dirección FLIP es en realidad dos direcciones, una pública y otra privada, relacionadas por

Dirección-pública = DES(dirección-privada)

donde DES son las siglas en inglés del estándar para el cipramiento de datos (Data Encryption Standard). Para calcular la dirección pública por medio de la dirección privada, ésta última se utiliza como clave DES para cifrar un bloque de ceros con una longitud de 64 bits. Dada una dirección pública, no es factible calcular la correspondiente dirección privada. Los servidores atienden a las direcciones privadas, pero los clientes envían a direcciones públicas, de manera similar al funcionamiento de los puertos de envío y recepción, pero en un nivel inferior.

FLIP está diseñado no sólo para funcionar con Amoeba, sino con otros sistemas operativos. También existe una versión de UNIX, y no existe una razón para que no funcione con MS-DOS. La seguridad proporcionada por el esquema de las direcciones pública y privada también funciona para la comunicación entre sistemas UNIX mediante FLIP, de manera independiente a Amoeba.

Además, FLIP está diseñado para su integración dentro del hardware; por ejemplo, como parte del circuito de interfaz de la red. Por esta razón, se ha especificado una interfaz precisa con la capa inmediata superior. La interfaz entre la capa FLIP y la capa por encima de ella (que llamaremos la capa RPC) tiene nueve primitivas, siete para el tráfico de salida y dos para el tráfico de entrada. Cada una de ellas tiene un procedimiento de biblioteca que la llama. Las siete llamadas se muestran en la figura 7-17.

	Descripción	Dirección
Init	Asignar una entrada en la tabla	↓
End	Regresar una entrada de la tabla	↓
Register	Atender a una dirección FLIP	↓
Unregister	Detener la atención a una dirección FLIP	↓
Unicast	Enviar un mensaje puntual	↓
Multicost	Enviar un mensaje multicost	↓
Broadcast	Enviar un mensaje de transmisión	↓
Receive	Paquete recibido	↑
Notdeliver	Recepción de un paquete no entregado	↑

Figura 7-17. Llamadas soportadas por la capa FLIP.

La primera, *init*, permite a la capa RPC asignar una entrada en una tabla e iniciarla con apuntadores a dos procedimientos (o bien, en una implantación en hardware, dos vectores de interrupción). Estos procedimientos son los que se llaman cuando arriban paquetes normales o paquetes no entregables, respectivamente. *End* libera la entrada cuando la máquina se desactiva.

Se llama a *register* para anunciar una dirección FLIP del proceso a la capa FLIP. Se le llama cuando se inicia un proceso (o al menos, en el primer intento por obtener o enviar un mensaje). La capa FLIP ejecuta de manera inmediata la dirección pública ofrecida mediante la función DES y guarda la dirección pública en sus tablas. Si un paquete entrante se dirige a la dirección FLIP pública, será transmitido a la capa RPC para su entrega. La llamada *unregister* elimina una entrada de las tablas de la capa FLIP.

Las siguientes tres llamadas son para el envío de mensajes puntuales, los mensajes de multitransmisión o de transmisión simple, respectivamente. Ninguno de ellos garantiza la entrega. Para hacer que la RPC sea confiable, se utilizan los reconocimientos. Para que la comunicación en grupo sea confiable, aun con pérdida de paquetes, se utiliza el protocolo del secuenciador ya analizado.

Las últimas dos llamadas son para el tráfico de entrada. La primera es para los mensajes que se originan en cualquier parte, dirigidos a esta máquina. La segunda es para los mensajes enviados por esta máquina pero que regresan como no entregados.

Aunque la interfaz FLIP pretende servir principalmente para uso de las capas RPC y de transmisión dentro del núcleo, también es visible para los procesos usuario, en caso de que tengan necesidad de una comunicación en bruto.

Operación de la capa FLIP

Los paquetes transferidos por la capa RPC o la capa para la comunicación en grupo (véase la figura 7-16) a la capa FLIP se direccionan mediante las direcciones FLIP, por lo que la capa FLIP convierte estas direcciones en la red para su transmisión real. Para llevar a cabo esta función, la capa FLIP mantiene una tabla de ruteo como la que se muestra en la figura 7-18. Por el momento, esta tabla se tiene en software, pero los diseñadores de futuros circuitos podrían implantarla en hardware.

Cada vez que un paquete llegue a una máquina, primero se controla en la capa FLIP, la cual extrae de él la dirección FLIP y la dirección en la red del emisor. También se registra el número de saltos hechos por el paquete. Puesto que el contador de saltos sólo se incrementa si un paquete es antecedido por una compuerta, el contador de saltos indica el número de compuertas por las que ha pasado el paquete. Así, este contador es una medida burda de la lejanía de la fuente. (En realidad, las cosas son poco mejores que esto, ya que las redes lentas cuentan como varios saltos.) Si la dirección FLIP no está presente en la tabla de ruteo, entra en ella. Esta entrada se puede utilizar después para el envío de paquetes *hacia* esa dirección FLIP, puesto que ahora se conocen su número y dirección en la red.

Dirección FLIP	Dirección en la red	Contador de saltos	Bit de seguridad	Edad

Figura 7-18. La tabla de ruteo de FLIP.

Un bit adicional en cada paquete indica si la trayectoria del paquete está por completo contenida en redes confiables. Esto es controlado por las compuertas. Si el paquete ha pasado por una o más redes no confiables, hay que cifrar los paquetes hacia la dirección fuente si se desea total seguridad. En las redes confiables no es necesario el cifrado.

El último campo de cada entrada de la tabla de ruteo proporciona la edad de dicha entrada. Su valor es 0 si el paquete se recibe de la dirección FLIP correspondiente. Todas las edades se incrementan de manera periódica. Este campo permite a la capa FLIP determinar una entrada adecuada en la tabla para eliminarla si la tabla se colma (un número grande indica que no ha habido tráfico durante mucho tiempo).

Localización de los puertos de envío

Para ver el funcionamiento de FLIP en el contexto de Amoeba, consideremos un sencillo ejemplo con la configuración de la figura 7-16. *A* es un cliente y *B* es un servidor. Con FLIP, toda máquina que tenga conexiones con dos o más redes se considera de manera automática una compuerta, por lo que no importa el hecho de que *B* se ejecute en una máquina compuerta.

Al crear *B*, el núcleo elige una nueva dirección aleatoria FLIP para él y la registra con la capa FLIP. Después de iniciar, *B* se inicia a sí mismo y hace un *get_request* en su puerto de envío, lo que produce un señalamiento al núcleo. La capa RPC busca el puerto de envío en su caché del puerto de envío al puerto de recepción (o lo calcula si no encuentra el dato) y hace una nota donde indica que un proceso atiende a ese puerto. Entonces se bloquea hasta recibir una solicitud.

Más tarde, *A* hace un *trans* en su puerto de envío. Su capa RPC busca en sus tablas para ver si conoce la dirección FLIP del proceso servidor que atiende al puerto de envío. Puesto que esto no es así, la capa RPC envía un paquete especial de transmisión para encontrarlo. Este paquete tiene un contador máximo de saltos para garantizar que la transmisión se restringe a su propia red. (Cuando una compuerta observa un paquete cuyo contador de saltos sea igual al contador máximo, el paquete se descarta y no puede seguir adelante.) Si la transmisión falla, termina el tiempo de la capa RPC emisora e intenta de nuevo con un contador máximo una unidad mayor, etc., hasta que localiza al servidor.

Cuando el paquete de transmisión llega a la máquina de *B*, su capa RPC envía de regreso una respuesta donde anuncia su dirección FLIP. Este paquete, al igual que todos los paquetes recibidos, hace que la capa FLIP de *A* cree una entrada para esa dirección FLIP antes de transferir el paquete de respuesta a la capa RPC. La capa RPC crea entonces una entrada en sus propias tablas, donde asocia el puerto de envío con la dirección FLIP. Entonces envía la solicitud al servidor. Puesto que la capa FLIP tiene ahora una entrada para la dirección FLIP del servidor, puede construir un paquete con la dirección apropiada en la red y enviarla sin más acciones. Las solicitudes posteriores al puerto de envío del servidor utilizan el caché de la capa RPC para encontrar la dirección FLIP y la tabla de ruteo de la capa FLIP para encontrar la dirección en la red. Así, la transmisión sólo se utiliza la primera vez que se tiene contacto con el servidor. Después de eso, las tablas del núcleo proporcionan toda la información necesaria.

En resumen, la localización de un puerto de envío necesita dos asociaciones:

1. Del puerto de envío a la dirección FLIP (lo cual se realiza en la capa RPC).
2. De la dirección FLIP a la dirección en la red (lo cual se realiza en la capa FLIP).

La razón de este proceso de dos etapas es doble. En primer lugar, FLIP se diseñó como protocolo general para su uso en los sistemas distribuidos, donde se incluyen sistemas no Amoeba. Puesto que estos sistemas no utilizan por lo general puertos del tipo de Amoeba, la asociación de los puertos de envío con las direcciones FLIP no están integradas en la capa FLIP. Otros usuarios de FLIP pueden utilizar de manera directa las direcciones FLIP.

En segundo lugar, un puerto de envío en realidad identifica un *servicio* y no un *servidor*. Un servicio puede ser proporcionado por varios servidores, con el fin de mejorar el desempeño y la confiabilidad. Aunque todos los servidores atiendan al mismo puerto de envío, cada uno tiene su propia dirección FLIP privada. Cuando una capa RPC de un cliente realiza una transmisión para determinar la dirección FLIP correspondiente a un puerto de envío, cualquiera o todos los servidores pueden responder. Puesto que cada servidor tiene una dirección FLIP diferente, cada respuesta crea una entrada distinta de una tabla de ruteo. Todas las respuestas se transfieren a la capa RPC, la cual elige una de ellas para utilizarla.

La ventaja de este esquema sobre el hecho de tener un caché (puerto, dirección en la red) es que permite a los servidores emigrar hacia nuevas máquinas o hacer que sus máquinas entren a nuevas redes sin tener que solicitar una nueva configuración manual, como sería el caso, por ejemplo, de TCP/IP. En este caso, existe una fuerte analogía con el caso de una persona que se cambia de casa y se le asigna en su nueva residencia el mismo número telefónico que tenía anteriormente. (Como observación, Amoeba no soporta la migración de procesos, pero se prevé que esta característica esté disponible en versiones futuras.)

La ventaja sobre el caso donde los clientes y los servidores utilizan de manera directa las direcciones FLIP es la protección que ofrece la función de un sentido que se emplea para derivar los puertos de envío de los puertos de recepción. Además, si un servidor falla, elegirá una nueva dirección FLIP cuando arranque de nuevo. Los intentos por utilizar la antigua dirección FLIP expirarán, lo cual permitirá a la capa RPC indicar la falla al cliente.

Este mecanismo es el que permite garantizar una semántica "a lo más uno". Sin embargo, el cliente puede intentar de nuevo con el mismo puerto de envío si así lo desea, puesto que esto no es necesario que se invalide al fallar el servidor.

FLIP en las redes de área amplia

FLIP también funciona de manera transparente en las redes de área amplia. En la figura 7-19 tenemos tres redes de área local conectadas mediante una red de área amplia. Supongamos que el cliente *A* desea realizar una RPC con el servidor *E*. En primer lugar, la capa RPC de *A* intenta localizar el puerto de envío mediante un contador máximo de saltos igual a uno. Cuando esto falla, intenta de nuevo con un contador máximo igual a dos. Esta vez, *C* permite el paso del paquete de transmisión a través de todas las compuertas conectadas a la red de área amplia, es decir, *D* y *G*. De hecho, *C* simula la transmisión a través de la red de área amplia mediante el envío de mensajes individuales a las demás compuertas. Cuando esta transmisión no puede hacer contacto con el servidor, se envía una tercera transmisión, esta vez con un contador máximo igual a tres. Esta vez se tiene éxito. La respuesta contiene la dirección en la red y la dirección FLIP de *E*, que entran a la tabla de ruteo de *A*. A partir de este momento, la comunicación entre *A* y *E* ocurre mediante una comunicación puntual normal. No se necesitan más transmisiones.

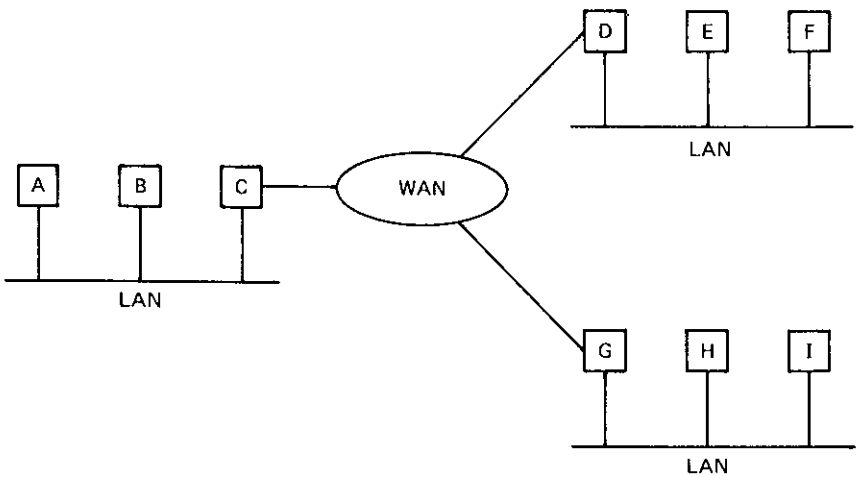


Figura 7-19. Tres LAN conectadas mediante una WAN.

La comunicación a través de la red de área amplia se encapsula en todo protocolo necesario para la red. Por ejemplo, en una red TCP/IP, *C* podría tener conexiones con *D* y *G* abiertas todo el tiempo. Otra alternativa es que la implantación decida cerrar cualquier conexión no utilizada durante cierto periodo.

Aunque este método no se escala bien a miles de LAN, funciona más o menos bien para números modestos. En la práctica, se desplazan pocos servidores, por lo que una vez localizado un servidor mediante una transmisión, las solicitudes posteriores utilizan las entradas del caché. Por medio de este método, un número importante de máquinas en todo el mundo pueden trabajar juntas de forma por completo transparente. Una RPC con un hilo en el espacio de direcciones del proceso que hizo la llamada y una RPC a un hilo al otro lado del planeta se realizan de la misma manera.

La comunicación en grupo también utiliza FLIP. Cuando se envía un mensaje a varios destinos, FLIP utiliza la multitransmisión o la transmisión simple en hardware en aquellas redes donde estén disponibles. En aquellas donde no estén disponibles, se simula la transmisión mediante el envío de mensajes individuales, tal como hemos visto en la red de área amplia. La capa FLIP elige el mecanismo, con la misma semántica del usuario en todos los casos.

7.6. LOS SERVIDORES DE AMOEBA

La mayoría de los servicios de los sistemas operativos tradicionales (como el servidor de archivos) se implantan en Amoeba como procesos servidores. Aunque sería posible disponer de una colección arbitraria de servidores, cada uno con su propio modelo del mundo, pronto se decidió proporcionar un modelo de lo que hace un servidor para lograr uniformidad y sencillez. Aunque esto es voluntario, la mayoría de los servidores siguen este modelo. En esta sección describiremos el modelo y algunos ejemplos de los servidores fundamentales de Amoeba.

Todos los servidores estándar de Amoeba se definen mediante un conjunto de procedimientos de resguardo. Los resguardos más recientes se definen en **AIL** (siglas en inglés del **Lenguaje de Interfaz de Amoeba**), aunque los más antiguos están escritos a mano en C. Los procedimientos de resguardo son generados por el compilador AIL a partir de las definiciones de resguardo y se colocan entonces en la biblioteca, de manera que los usuarios los puedan utilizar. De hecho, los resguardos definen de manera precisa los servicios que proporciona un servidor, así como sus parámetros. En nuestro siguiente análisis, nos referiremos con frecuencia a los resguardos.

7.6.1. El servidor de archivos

Como todos los sistemas operativos, Amoeba tiene un sistema de archivos. Sin embargo, a diferencia de la mayoría de los demás, la elección del sistema de archivos no está dictada por el sistema operativo. El sistema de archivos se ejecuta como colección de procesos servidores. Los usuarios que no quieran utilizar los estándares, pueden escribir los suyos propios. El núcleo no sabe, ni le preocupa, cuál de ellos es el sistema de archivos "real". De hecho, los diversos usuarios pueden tener sistemas de archivos diferentes e incompatibles entre sí al mismo tiempo, si así lo desean.

El sistema de archivos estándar consta de tres servidores, el **servidor de archivos**, que controla el espacio de almacenamiento de archivos; el **servidor de directorios**, que se encarga de los nombres de los archivos y del manejo de los directorios; y el **servidor de réplicas**, el cual controla la réplica de archivos. El sistema de archivos se ha separado en estos componentes independientes para lograr mayor flexibilidad y hacer que cada uno de los servidores tenga una implantación directa. En esta sección analizaremos el servidor de archivos y en las siguientes los otros dos.

De manera breve, un proceso cliente puede crear un archivo mediante la llamada *create*. El servidor de archivos responde con el envío de una posibilidad, la cual se puede utilizar en las llamadas posteriores a *read* para recuperar todo o parte del archivo. En la mayoría de los casos, el usuario le dará al archivo un nombre en ASCII y la pareja (nombre en ASCII, posibilidad) se dará al servidor de directorios para su almacenamiento en un directorio, pero esta operación no tiene que ver con el servidor de archivos.

El servidor de archivos se diseñó para ser muy rápido (de ahí el nombre en inglés, **bullet server**, servidor bala). También se diseñó para su ejecución en máquinas con memorias primarias de gran tamaño y enormes discos, en vez de máquinas para usuario final, donde la memoria siempre es escasa. La organización es un poco distinta de la mayoría de los servidores de archivos convencionales. En particular, los archivos son **inmutables**. Una vez creado un archivo, no puede ser modificado en lo sucesivo. Puede ser eliminado y un nuevo archivo se puede crear en su lugar, pero el nuevo archivo tiene una posibilidad diferente de la correspondiente al anterior. Este hecho hace más sencilla la réplica automática, como veremos en seguida. También es adecuado para el uso en discos ópticos de gran capacidad con una sola escritura.

Puesto que los archivos no se pueden modificar después de su creación, el tamaño de un archivo siempre se conoce al momento de ésta. Esta propiedad permite que los archivos se almacenen en bloques adyacentes en el disco y también en la memoria caché principal. Mediante este tipo de almacenamiento adyacente, los archivos se pueden leer hacia la memoria en una sola operación del disco y pueden ser enviados a los usuarios en un solo mensaje de respuesta RPC. Estas simplificaciones llevan a un alto desempeño.

Así, el modelo conceptual detrás del sistema de archivos consiste en que un cliente crea todo un archivo en su propia memoria y entonces lo retransmite en una RPC al servidor de archivos, el cual lo almacena y regresa una posibilidad con la que se puede tener acceso a él posteriormente. Para modificar este archivo (por ejemplo, editar un programa o documento), el cliente envía de regreso la posibilidad y solicita el archivo, el cual se envía (de manera ideal) en una RPC a la memoria del cliente. El cliente puede entonces modificar el archivo localmente de la forma que lo desee. Cuando termina, envía el archivo al servidor (de manera ideal) en una RPC, lo que provoca la creación de un nuevo archivo y el regreso de una nueva posibilidad. En este momento, el cliente puede solicitar al servidor que destruya el archivo original o que conserve el archivo anterior como respaldo.

Como una concesión a la realidad, el servidor de archivos también soporta clientes que tienen muy poca memoria como para recibir o enviar archivos completos en una RPC. Durante la lectura, es posible pedir una sección de un archivo, determinada mediante un

ajuste y un contador de bytes. Esta característica permite que los clientes lean archivos mediante la unidad de tamaño que consideren más conveniente.

La escritura de un archivo en varias operaciones se complica por el hecho de que se garantiza que los archivos del servidor de archivos son inmutables. Este problema se resuelve mediante la introducción de dos tipos de archivos, los **archivos no comprometidos**, que están en proceso de creación, y los **archivos comprometidos**, que son permanentes. Los archivos no comprometidos se pueden modificar, pero los comprometidos no. Una RPC que realiza un *create* debe especificar si el archivo se compromete en forma inmediata o no.

En ambos casos, se hace una copia del archivo en el servidor y se regresa una posibilidad para el archivo. Si el archivo no está comprometido, puede ser modificado mediante posteriores RPC; en particular, se le pueden añadir secciones. Cuando las añadiduras y demás cambios se hayan concluido, el archivo se puede comprometer, en cuyo momento se convierte en inmutable. Para enfatizar la naturaleza transitoria de los archivos no comprometidos, éstos no se pueden leer. Sólo se pueden leer los archivos comprometidos.

La interfaz del servidor de archivos

El servidor de archivos soporta las seis operaciones que se muestran en la figura 7-20, junto con otras tres reservadas para el administrador del sistema. Además, también son válidas las operaciones estándar que se muestran en la figura 7-5. Se tiene acceso a todas estas operaciones mediante llamadas a procedimientos de resguardo de la biblioteca.

Llamada	Descripción
Create	Crea un nuevo archivo; también lo puede comprometer
Read	Lee todo o parte de un archivo específico
Size	Regresa el tamaño de un archivo específico
Modify	Escribe sobre *n* bytes de un archivo no comprometido
Insert	Inserta o añade *n* bytes a un archivo no comprometido
Delete	Elimina *n* bytes de un archivo no comprometido

Figura 7-20. Llamadas al servidor de archivos.

El procedimiento *create* proporciona algunos datos, los cuales se colocan en un nuevo archivo cuya posibilidad regresa dentro de la respuesta. Si el archivo está comprometido (determinado por un parámetro), se puede leer pero no modificar. Si no está comprometido, no se puede leer hasta que se comprometa, pero puede ser modificado o añadírsele algún elemento.

La llamada *read* puede leer todo o parte de cualquier archivo comprometido. Especifica el archivo por leer, al proporcionar una posibilidad para dicho archivo. La presentación de

la posibilidad es una prueba de que la operación es permitida. El servidor de archivos no realiza verificaciones con base en la identidad del cliente. De hecho, ni siquiera conoce la identidad del cliente. La llamada *size* toma una posibilidad como parámetro e indica el tamaño del archivo correspondiente.

Las últimas tres llamadas funcionan en los archivos no comprometidos. Permiten la modificación del archivo, mediante la escritura sobre él, la inserción o eliminación de bytes. Se pueden hacer varias llamadas en serie. La última llamada puede indicar por medio de un parámetro que desea comprometer al archivo.

El servidor de archivos también soporta tres llamadas especiales para el administrador del sistema, que debe presentar una superposibilidad especial. Estas llamadas envían el caché de la memoria principal al disco, permiten la compactación del mismo y reparan los sistemas de archivos dañados.

Las posibilidades generadas y utilizadas por el servidor de archivos utilizan el campo *Derechos* para proteger las operaciones. Por ejemplo, de esta forma se puede crear una posibilidad que permita que un archivo sea leído pero no destruido.

Implantación del servidor de archivos

El servidor de archivos mantiene una tabla de archivos con una entrada por cada uno de éstos, similar a la tabla de nodos-i de UNIX, como se muestra en la figura 7-21. Toda la tabla se lee en la memoria cuando arranca el servidor de archivos y se mantiene ahí mientras el servidor de archivos esté ejecutándose.

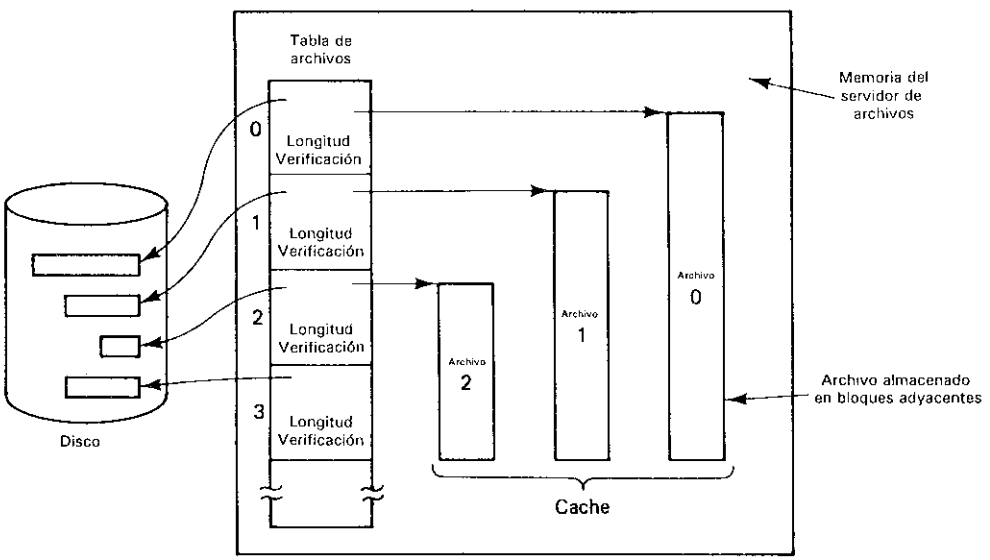


Figura 7-21. Implantación del servidor de archivos.

En términos generales, cada entrada de la tabla contiene dos apuntadores y una longitud, más cierta información adicional. Un apuntador tiene la dirección en disco del archivo y el otro tiene la dirección en la memoria principal, si el archivo en cuestión está en el caché de la memoria principal en ese momento. Todos los archivos se almacenan en bloques adyacentes, tanto en el disco como en el caché, de modo que basta con un apuntador y una longitud. A diferencia de UNIX, no se necesitan bloques directos o indirectos.

Aunque esta estrategia desperdicia espacio debido a la fragmentación externa, tanto en la memoria como en el disco, tiene la ventaja de su mayor sencillez y alto desempeño. Un archivo en disco se puede leer a la memoria en una operación, al máximo de velocidad del disco y se puede transmitir en la red al máximo de velocidad de ésta. Como las memorias y los discos se están haciendo cada vez más grandes y más baratos, es probable que el costo de la memoria desperdiciada sea aceptable en términos de la velocidad proporcionada.

Cuando un proceso cliente desea leer un archivo, envía la posibilidad correspondiente al archivo hacia el servidor de archivos. El servidor extrae el número de objeto de la posibilidad y lo utiliza como índice en la tabla de archivos para localizar la entrada del archivo. La entrada contiene el número aleatorio utilizado en el campo *Verificación* de la posibilidad, el cual se utiliza a su vez para verificar la validez de la posibilidad. Si no es válida, la operación termina con un código de error. Si es válida, se busca todo el archivo en el disco y se pasa al caché, a menos de que ya esté ahí. El espacio del caché se controla mediante LRU, pero la hipótesis implícita es que el caché es lo bastante grande como para contener todo el conjunto de archivos en uso en ese momento.

Si se crea un archivo y se pierde su posibilidad, nunca se podrá tener acceso al archivo y permanecerá en su lugar para siempre. Para evitar esta situación, se utilizan los tiempos de espera. Si no se tiene acceso a un archivo no comprometido durante 10 minutos, este archivo se elimina y se libera su entrada en la tabla. Si la entrada se utiliza después para otro archivo, pero la posibilidad se presenta 15 minutos más tarde, el campo *Verificación* detectará el hecho de que el archivo ha cambiado y se rechazará la operación en el archivo anterior. Este método es aceptable, puesto que lo normal es que los archivos existan en un estado no comprometido durante unos cuantos segundos.

Para el caso de los archivos comprometidos, se utiliza un método menos draconiano. A cada archivo (en una entrada de la tabla de archivos) se le asocia un contador, el cual se inicializa con el valor *TIEMPO_MÁXIMO_DE_VIDA*. De manera periódica, un demonio hace una RPC con el servidor de archivos, para solicitarle que lleve a cabo la operación estándar *age* (véase la figura 7-5). Esta operación hace que el servidor de archivos recorra toda la tabla de archivos, para decrementar cada contador en uno. Se destruye cualquier archivo cuyo contador tenga el valor cero y se recupera su espacio en disco, en tablas y en caché.

Para evitar que este mecanismo elimine archivos en uso, se dispone de otra operación, *touch*. A diferencia de *age*, la cual se aplica a todos los archivos, *touch* se aplica a un archivo específico. Su función es regresar el valor del contador a *TIEMPO_MÁXIMO_DE_VIDA*. *Touch* se llama de manera periódica para el caso de todos los archivos enumerados en cualquier directorio, para evitar que expire su tiempo. Por lo general, se ejecuta la instrucción

touch una vez cada hora y se elimina un archivo no tocado durante 24 horas. Este mecanismo elimina los archivos perdidos (es decir, los archivos que no se encuentran en algún directorio).

El servidor de archivos se puede ejecutar en el espacio del usuario, como un proceso ordinario. Sin embargo, si se ejecuta en una máquina de uso exclusivo, sin otros procesos en esa máquina, se puede lograr una pequeña mejoría en el desempeño al colocarlo dentro del núcleo. La semántica no se ve afectada por este movimiento. El cliente ni siquiera puede decir dónde está localizado.

7.6.2. El servidor de directorios

El servidor de archivos, como hemos visto, sólo controla el almacenamiento de archivos. La colocación de nombres y otros objetos se controlan mediante el **servidor de directorios**. Su principal función consiste en proporcionar una asociación entre los nombres legibles para los humanos (ASCII) y las posibilidades. Los procesos pueden crear uno o más directorios, cada uno de los cuales puede contener varios renglones. Cada renglón describe un objeto y contiene tanto al objeto como su posibilidad. Se dispone de operaciones para la creación y eliminación de directorios, adición y eliminación de renglones y búsqueda de nombres en directorios. A diferencia de los archivos, los directorios *no* son inmutables. Se pueden añadir o eliminar entradas de los directorios existentes.

Los propios directorios son objetos y están protegidos por posibilidades, al igual que los demás objetos. Las operaciones en directorios, como la búsqueda de nombres y la adición de nuevas entradas, son protegidas por bits en el campo *Derechos*, de la manera usual. Las posibilidades de los directorios se pueden almacenar en otros directorios, lo cual permite la existencia de árboles jerárquicos de directorios y estructuras más generales.

Aunque el servidor de directorios se puede utilizar simplemente para almacenar parejas (nombre del archivo, posibilidad), también puede soportar un modelo más general. En primer lugar, una entrada de un directorio puede nombrar a cualquier tipo de objeto descrito mediante una posibilidad, no sólo un archivo o un directorio. El servidor de directorios no sabe ni se preocupa por el tipo de objetos que controlan las posibilidades. Las entradas de un directorio pueden corresponder varios tipos de objetos y estos pueden estar dispersos en todo el mundo. No existe el requisito de que los objetos de un directorio sean del mismo tipo o que deban ser controlados por el mismo servidor. Cuando se busca y utiliza una posibilidad, su servidor se localiza mediante una transmisión, como describimos en la sección anterior relativa a FLIP.

Cadena en ASCII	Conjunto de posibilidades	Propietario	Grupo	Otros
Correo	▭ ▭ ▭	1111	0000	0000
Juegos	▭ ▭ ▭	1111	1110	1110
Exámenes	▭ ▭ ▭	1111	0000	0000
Artículos	▭ ▭ ▭	1111	1100	1000
Comités	▭ ▭ ▭	1111	1010	0010

Figura 7-22. Un directorio típico controlado por el servidor de directorios.

En segundo lugar, un renglón podría contener no una posibilidad, sino todo un conjunto de posibilidades, como se muestra en la figura 7-22. En general, estas posibilidades son para copias idénticas del objeto y son controlados por distintos servidores. Cuando un proceso busca un nombre, se le da todo el conjunto de posibilidades. Para ver la utilidad de tal característica, consideremos el procedimiento de biblioteca *open* para la apertura de un archivo. Busca un archivo y obtiene a cambio un conjunto de posibilidades. Entonces intenta con cada una de las posibilidades, hasta que encuentra una cuyo servidor esté vivo. De esta manera, si un objeto no está disponible, se puede utilizar otro en su lugar, sin que el programa principal se dé cuenta de ello. Debe quedar claro que este mecanismo funciona mejor cuando los archivos son inmutables, ya que en este caso no existe peligro de que alguno de ellos sea modificado después de su creación.

En tercer lugar, un renglón puede contener varias columnas, cada una de las cuales forma un dominio de protección distinto con derechos distintos. Por ejemplo, un directorio podría tener una columna para el propietario, una para el grupo del propietario y otra para las demás personas, con el fin de simular el esquema de protección de UNIX. Una posibilidad de un directorio es en realidad una posibilidad para una columna específica de un directorio, lo que permite al propietario, grupo o demás personas tener distintos permisos. Puesto que el conjunto subyacente de posibilidades es el mismo para todas las columnas de un renglón, sólo se necesita almacenar los bits de derechos para cada columna. Las posibilidades reales se pueden calcular cuando sea necesario.

La organización de un ejemplo de directorio, con cinco entradas, se muestra en la figura 7-22. Este directorio tiene un renglón para cada uno de los cinco nombres de archivo almacenados en él. El directorio también tiene tres columnas, cada una de las cuales representa un dominio distinto de protección; en este caso, para el propietario, el grupo del propietario y todas las demás personas. Cuando el propietario de un directorio libera una posibilidad para, digamos, la última columna, el receptor no tiene acceso a las posibilidades más poderosas de las primeras dos columnas.

Como hemos mencionado, los directorios pueden contener las posibilidades de otros directorios. Esto permite construir no sólo árboles, sino también gráficas de directorios generales. Un uso evidente de esta poderosa característica es el de colocar la posibilidad de un archivo en dos o más directorios, con lo que se crean varios enlaces a éste. Estas posibilidades también pueden tener derechos distintos, lo que permite a las personas compartir un archivo con distintos permisos de acceso, algo imposible de hacer en UNIX.

En cualquier sistema distribuido, en particular uno que se desee utilizar en las redes de área amplia, es difícil tener el concepto de un directorio raíz global. En Amoeba, cada usuario tiene su propio directorio raíz, como se muestra en la figura 7-23. Este directorio contiene las posibilidades no sólo de los subdirectorios privados del usuario, sino también de varios directorios públicos, los cuales contienen programas del sistema y otros archivos compartidos.

Algunos de los directorios en la raíz de cada usuario son similares a los directorios de UNIX, como *bin, dev*, etc. Sin embargo, otros son por completo diferentes. Uno de ellos es *home*, que es el directorio de origen del usuario.

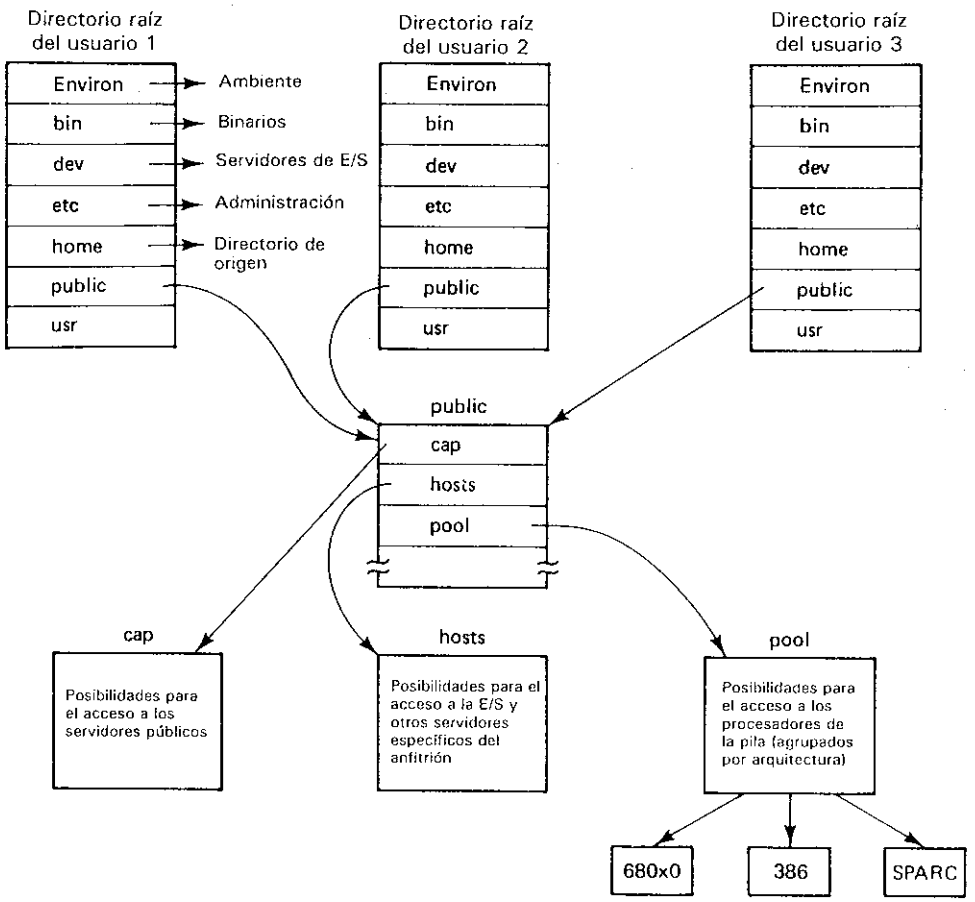


Figura 7-23. Versión simplificada de la jerarquía de directorios de Amoeba.

Otro directorio es *public*, el cual contiene el inicio del árbol público compartido. Entre este tipo de directorios están *cap, hosts* y *pool*. Cuando un proceso desea tener contacto con el servidor de archivos, con el servidor de directorios o con cualquier otro servidor, por ejemplo, para crear un nuevo objeto, debe contar con una posibilidad genérica para hablar con ese servidor. Estas posibilidades se mantienen en */public/cap*.

Otro de los directorios en *public* es *hosts*, el cual contiene un directorio para cada máquina del sistema. Este directorio contiene las posibilidades de varios servidores que se pueden encontrar en un anfitrión, como un servidor de discos, un servidor de terminales, un servidor de procesos, un servidor de números aleatorios, etc. Por último, *pool* contiene las posibilidades para los procesadores de una pila, agrupados por la arquitectura del CPU. Se dispone de un mecanismo para que cada usuario sólo utilice un conjunto de procesadores de una pila.

La interfaz del servidor de directorios

Las principales llamadas al servidor de directorios se muestran en la figura 7-24. Las dos primeras, *create* y *delete*, se utilizan para crear y eliminar directorios, respectivamente. Cuando se crea un directorio, se regresa su posibilidad, al igual que el caso de la creación de un archivo. Esta posibilidad se puede insertar después en cualquier otro directorio para construir una jerarquía. Esta interfaz de bajo nivel permite un control máximo sobre la forma de la gráfica de nombres. Puesto que muchos programas se conforman con trabajar con árboles de directorios convencionales, se dispone de un paquete de biblioteca para facilitar esta tarea.

Llamada	Descripción
Create	Crea un nuevo directorio
Delete	Elimina un directorio o una entrada de un directorio
Append	Añade una nueva entrada a un directorio específico
Replace	Reemplaza una entrada de directorio
Lookup	Regresa el conjunto de posibilidades correspondientes a un nombre específico
Getmasks	Regresa las plantillas correctas para la entrada dada
Chmod	Modifica los bits de derechos en una entrada de directorio existente

Figura 7-24. Las principales llamadas del servidor de directorios.

Es importante observar que la eliminación de una entrada de un directorio no es igual a la destrucción del propio objeto. Si se elimina una posibilidad de un directorio, el propio objeto continúa su existencia. Por ejemplo, la posibilidad se coloca en otro directorio. Para deshacerse del objeto, hay que destruirlo de manera explícita.

Para añadir una nueva entrada a un directorio, sea un archivo, un directorio o algún otro objeto, se utiliza la llamada *append*. Como la mayoría de las llamadas al servidor de directorios, especifica la posibilidad del directorio por utilizar (al cual se añade una sección), así como la posibilidad por colocar en el directorio y los bits de derechos para todas las columnas. Se puede escribir sobre una entrada existente mediante *replace*; por ejemplo, cuando se edita un archivo y se va a utilizar la nueva versión en vez de la anterior.

La operación más común en los directorios es *lookup*, la cual toma como parámetros una posibilidad para un directorio (columna) y una cadena en ASCII; regresa el correspondiente conjunto de posibilidades. La apertura de un archivo para su lectura requiere que primero se busquen sus posibilidades.

Las últimas dos operaciones de la lista son para la lectura y escritura de las plantillas correctas para todas las columnas de un renglón determinado por la cadena.

También existen unas cuantas operaciones sobre los directorios. La mayoría están relacionadas con la búsqueda o reemplazo de varios archivos a la vez. Estas operaciones pueden ser útiles para la implantación de las transacciones atómicas relacionadas con varios archivos.

Implantación del servidor de directorios

El servidor de directorios es un componente crítico en el sistema Amoeba, por lo que su implantación es tolerante de fallas. La estructura de datos básica es un arreglo de parejas de posibilidades en una partición simple del disco. Este arreglo no utiliza el servidor de archivos, puesto que se debe actualizar con frecuencia y se pensó que el costo sería excesivo.

Cuando se crea un directorio, el número de objeto colocado en su posibilidad es un índice para dicho arreglo. Cuando se presenta una posibilidad del directorio, el servidor inspecciona el número de objeto contenido en él y lo utiliza para buscar la pareja correspondiente de posibilidades en el arreglo. Estas posibilidades son para archivos idénticos, almacenados en servidores de archivos diferentes, cada uno de los cuales contiene el directorio y el campo *Verificación*, el cual se utiliza para verificar la autenticidad de la posibilidad del directorio.

Cuando se modifica un directorio, se crea un archivo nuevo para él y se escribe sobre los arreglos en la partición simple del disco. La segunda copia se crea más tarde mediante un hilo en ejecución secundaria. Se destruyen entonces todos los directorios anteriores. Aunque este método tiene un costo adicional, proporciona un grado mayor de confiabilidad que los sistemas de archivos tradicionales. Además, lo común es que los servidores de directorios vengan en parejas, donde cada uno cuenta con su propio arreglo de parejas de posibilidades (en distintos discos), para evitar un desastre si una de las particiones del disco está dañada. Los dos servidores se comunican para mantenerse sincronizados. También es posible la ejecución con uno de ellos. El modelo de dos servidores se muestra en la figura 7-25.

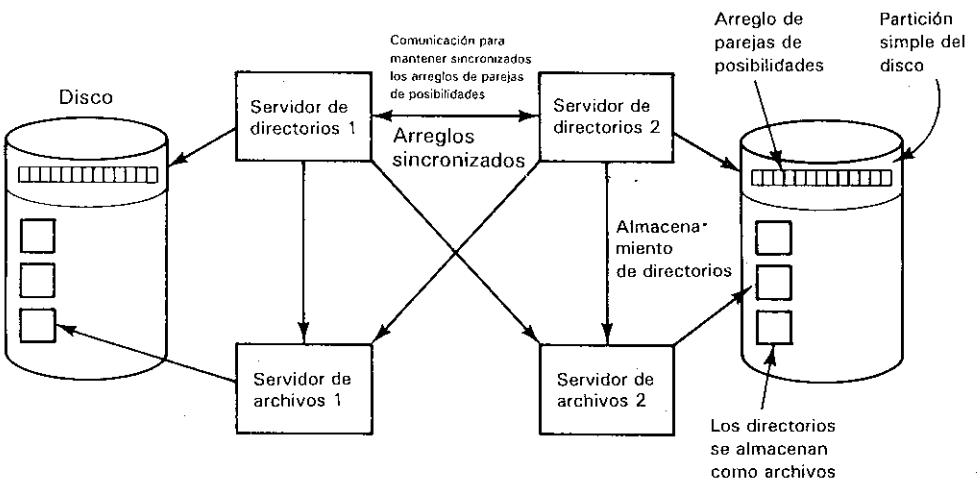


Figura 7-25. Una pareja de servidores de directorios. Todos los datos se almacenan dos veces en diferentes servidores de archivos.

En la figura 7-22, el conjunto de posibilidades se muestra siendo almacenado una vez por cada renglón, aunque existan varias columnas. Esta organización se utiliza en la realidad. En la mayoría de los casos, la columna *propietario* contiene bits de derechos todos iguales a uno, de modo que las posibilidades del conjunto son posibilidades reales del propietario (es decir, el campo *Verificación* no se ha ejecutado mediante la función de un sentido). Cuando se busca un nombre en otra columna, el propio servidor de directorios calcula la posibilidad restringida mediante un XOR del campo *derechos* tomado de la entrada del directorio y el campo *Verificación* tomado de la posibilidad del propietario. El resultado se pasa por la función de un solo sentido y regresa al proceso que hizo la llamada.

Este método elimina la necesidad de almacenar un gran número de posibilidades. Además, el servidor de directorios captura en un caché las posibilidades de uso intenso para evitar uno innecesario de la función de un sentido. Si el conjunto de posibilidades no contiene posibilidades del propietario, entonces hay que llamar al servidor para calcular las posibilidades restringidas, puesto que el servidor de directorios no tiene acceso al campo original *Verificación*.

7.6.3. El servidor de réplicas

Los objetos controlados por el servidor de directorios se pueden duplicar en forma automática mediante el servidor de réplicas. Éste practica lo que se llama **réplica retardada**. Lo que esto significa es que cuando se crea un archivo o algún otro objeto, al principio sólo se hace una copia. Entonces se puede llamar al servidor de réplicas, para producirlas idénticas, cuando tenga tiempo. En vez de recibir llamadas directas, el servidor de réplicas se mantiene en ejecución en un plano secundario todo el tiempo, examinando partes específicas del sistema de directorios en forma periódica. Cuando encuentra una entrada de directorio que supuestamente deba contener *n* posibilidades pero que contiene menos, se pone en contacto con los servidores correspondientes y ordena la creación de copias adicionales. Aunque el servidor de réplicas se puede utilizar para copiar cualquier tipo de objeto, funciona mejor para el caso de los objetos inmutables, como los archivos. La ventaja es que los objetos inmutables no pueden cambiar durante el proceso de réplica, por lo que puede trabajar con seguridad en un plano secundario, aunque tarde un tiempo considerable. Los objetos mutables podrían cambiar durante el proceso de réplica, agregando mayor complejidad para evitar la inconsistencia.

Además, el servidor de réplicas ejecuta los mecanismos de maduración y de recolección de basura utilizados por el servidor de archivos y otros servidores. De manera periódica, toca cada objeto que esté bajo el control del servidor de directorios, para evitar que su tiempo expire. También envía los mensajes *age* a los servidores con el fin de decrementar todos los contadores de los objetos y recolectar la basura de aquellos cuyo contador tenga el valor cero.

7.6.4. El servidor de ejecución

Cuando el usuario escribe un comando (por ejemplo, *sort*) en la terminal, hay que tomar dos decisiones:

1. ¿En qué tipo de arquitectura se debe ejecutar el proceso?
2. ¿Cuál procesador se debe elegir?

La primera pregunta se refiere a si el proceso se debe ejecutar en una 386, SPARC, 680x0, etc. La segunda se relaciona con la elección del CPU específico y depende de la carga y disponibilidad de memoria de los procesadores candidatos. El **servidor de ejecución** ayuda a tomar esas decisiones.

Cada servidor de ejecución controla una o varias pilas de procesadores. Una pila de procesadores se representa mediante un directorio llamado **pooldir**, el cual contiene subdirectorios para cada una de las arquitecturas de CPU soportadas. Los subdirectorios contienen las posibilidades para el acceso a los servidores de procesos para cada una de las máquinas en la pila. Un ejemplo de ordenamiento se muestra en la figura 7-26. También se pueden ordenar los procesadores de otras formas, entre las que se encuentran las pilas mixtas o traslapadas; además, una pila se puede subdividir en subpilas.

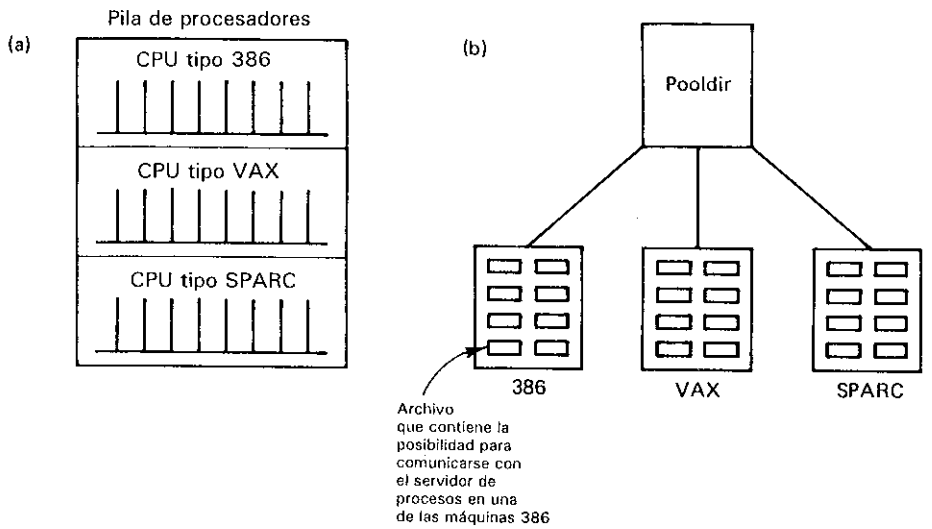


Figura 7-26. (a) Una pila de procesadores. (b) El pooldir correspondiente.

Cuando el shell desea ejecutar un programa, busca en */bin* para encontrar, digamos, *sort*. Si *sort* está disponible para varias arquitecturas, *sort* no será un archivo, sino un directorio que contiene programas ejecutables para cada arquitectura disponible. El shell hace entonces una RPC con el servidor de ejecución para enviarle todos los descriptores de proceso disponibles y solicitarle que elija una arquitectura y un CPU específicos.

El servidor de ejecución busca entonces en pooldir para ver lo que puede ofrecer. La selección se hace más o menos así. En primer lugar, se calcula la intersección de los descriptores de proceso y los procesadores de la pila. Si existen descriptores de proceso (es

decir, programas binarios) para la 386, SPARC y 680x0 y este servidor de ejecución controla procesadores 386, SPARC y VAX, las únicas posibilidades son la 386 y la SPARC, de modo que las demás máquinas quedan eliminadas como candidatos.

En segundo lugar, el servidor de ejecución verifica si alguna de las máquinas candidato tiene memoria suficiente para ejecutar el programa. Las que no lo tengan se eliminan. El servidor de ejecución mantiene un registro del uso de la memoria y el CPU para cada uno de los procesadores de la pila mediante llamadas periódicas *getload* a cada una, donde solicita esos valores, de modo que los números en las tablas del servidor de ejecución se renuevan de manera continua.

En tercer y último lugar, para cada una de las máquinas restantes se hace una estimación del poder de cómputo que pueden dedicar al nuevo programa. La heurística utiliza como entrada el poder de cómputo total conocido del CPU y el número de hilos activos que en ese momento se ejecutan en él. Por ejemplo, si una máquina de 20 MIPS tiene cuatro hilos activos, entonces la adición de un quinto proceso significa que cada uno de ellos, incluido el nuevo, tendrán como promedio 4 MIPS. Si otro procesador tiene 10 MIPS y un hilo, entonces en esa máquina el nuevo programa puede esperar 5 MIPS. El servidor de ejecución elige el procesador que pueda entregar un mayor número de MIPS y regresa la posibilidad para que se puedan comunicar su servidor de procesos y el proceso que hizo la llamada. Este último utiliza esta posibilidad para crear el proceso, como ya se describió en la sección 7.3.

7.6.5. El servidor de arranque

Como otro ejemplo de servidor de Amoeba, consideremos el **servidor de arranque**. Éste se utiliza para proporcionar cierto grado de tolerancia de fallas en Amoeba, mediante la verificación de todos los servidores que deberían estar en ejecución y aquellos que en realidad lo estén, además de que toma medidas correctivas en caso de que no estén trabajando. Un servidor interesado en la sobrevivencia de las fallas se puede incluir en el archivo de configuración del servidor de arranque. Cada entrada indica la frecuencia y la forma en que el servidor de arranque debe realizar una encuesta. Mientras el servidor responda de manera correcta, el servidor de arranque no realiza ninguna acción posterior.

Sin embargo, si el servidor no responde después de cierto número de intentos, el servidor de arranque lo declara muerto e intenta reiniciarlo. Si eso falla, entonces asigna de alguna manera un nuevo procesador de la pila donde inicia una nueva copia. De esta forma, los servicios críticos vuelven a arrancar de manera automática si llegan a fallar. El servidor de arranque se puede duplicar a sí mismo, para protegerse de sus propias fallas.

7.6.6. El servidor TCP/IP

Aunque Amoeba utiliza el protocolo FLIP de manera interna para lograr un alto desempeño, necesita hablar TCP/IP, por ejemplo, para la comunicación con las terminales X, para enviar y recibir correo de otras máquinas que no sean de tipo Amoeba, así como para

interactuar con otros sistemas Amoeba por medio de Internet. Para que Amoeba pueda hacer esto, se dispone de un servidor TCP/IP.

Para establecer una conexión, un proceso en Amoeba hace una RPC con el servidor TCP/IP dando una dirección TCP/IP. El proceso que hace la llamada se bloquea hasta que se establece o niega la conexión. En la respuesta, el servidor TCP/IP proporciona una posibilidad para el uso de la conexión. Las RPC posteriores pueden enviar y recibir paquetes de la máquina remota sin que el proceso Amoeba tenga conciencia del uso de TCP/IP. Este mecanismo es menos eficiente que FLIP, por lo que sólo se utiliza cuando no es posible utilizar FLIP.

7.6.7. Otros servidores

Amoeba soporta otros servidores, entre los que se encuentran un servidor de discos (utilizado por el servidor de directorios para el almacenamiento de sus arreglos de parejas de posibilidades), varios servidores de E/S, un servidor con la hora actual y un servidor de números aleatorios (útil para la generación de puertos, posibilidades y direcciones FLIP). El servidor *Navaja del ejército suizo* se encarga de muchas actividades por realizar en cierto momento futuro, después de la iniciación de los procesos. Los servidores de correo se encargan de la entrada y salida del correo electrónico.

7.7. RESUMEN

Amoeba es un nuevo sistema operativo diseñado para hacer que una colección de computadoras independientes aparezca ante sus usuarios como un sistema de tiempo compartido. En general, los usuarios no están conscientes del sitio donde se ejecutan sus procesos (o incluso el tipo de CPU utilizado), ni del sitio donde se almacenan sus archivos o las copias de ellos mantenidas por razones de disponibilidad y desempeño. Sin embargo, los usuarios explícitamente interesados en la programación paralela pueden explotar la existencia de varios CPU mediante la subdivisión de un solo trabajo en varias máquinas.

Amoeba se basa en un micronúcleo que controla los procesos de bajo nivel, la administración de la memoria, la comunicación y la E/S. El sistema de archivos y el resto del sistema operativo se pueden ejecutar como procesos usuario. Esta división del trabajo mantiene el núcleo pequeño y sencillo.

Amoeba tiene un mecanismo para nombrar y proteger a todos los objetos: las posibilidades. Cada posibilidad contiene los derechos que indican las operaciones permitidas mediante su uso. Las posibilidades se protegen por cifrado mediante las funciones de un solo sentido. Cada una contiene un campo para la suma de verificación, la cual garantiza la seguridad de la posibilidad.

Se soportan tres mecanismos de comunicación: RPC y FLIP simple para la comunicación puntual y la comunicación confiable en grupo para la comunicación entre varias partes. La RPC garantiza una semántica "al menos una vez". La comunicación en grupo se basa

en la transmisión confiable proporcionada por el algoritmo del secuenciador. Ambos mecanismos se soportan sobre el protocolo FLIP y están fuertemente integrados. El FLIP simple sólo se utiliza en circunstancias especiales.

El sistema de archivos de Amoeba consta de tres servidores: el servidor de archivos para el almacenamiento de éstos, el servidor de directorios para otorgar nombres y el servidor de réplicas para la réplica de archivos. El servidor de archivos mantiene archivos inmutables que se almacenan en bloques adyacentes, en el disco y en el caché. El servidor de directorios es un servidor tolerante de fallas que asocia cadenas en ASCII con las posibilidades. El servidor de réplicas controla la réplica retrasada.

PROBLEMAS

1. Los diseñadores de Amoeba supusieron que pronto se dispondrá de memoria en grandes cantidades a bajo precio. ¿Qué efecto tuvo esta hipótesis en el diseño?
2. Mencione una ventaja y una desventaja del modelo de pila de procesadores comparado con el modelo del multiprocesador personal.
3. Enumere tres funciones del micronúcleo de Amoeba.
4. Algunos servidores de Amoeba se pueden ejecutar tanto en el espacio del núcleo como en el espacio del usuario. Sus clientes no pueden decir cuál es la diferencia (excepto por el tiempo). ¿Qué tiene Amoeba para que sea imposible para los clientes decir la diferencia?
5. Un usuario malicioso intenta adivinar el puerto de recepción del servidor de archivos, mediante la elección de un número aleatorio de 48 bits y el paso de éste a través de la función de un solo sentido (que es conocida), para después ver si se obtiene el puerto de recepción. Cada intento tarda 1 milisegundo. ¿Cuánto tardaría en obtener el puerto de recepción en promedio?
6. ¿Cómo puede saber un servidor que una posibilidad es una posibilidad del propietario, en contraposición a una posibilidad restringida? ¿Cómo se verifican las posibilidades del propietario?
7. Si una posibilidad no tiene posibilidad del propietario, ¿cómo verifican su validez los servidores?
8. Explique lo que es una variable glocal.
9. ¿Por qué tiene la llamada *trans* parámetros para el envío y la recepción? ¿No sería mejor y más sencillo tener dos llamadas, *send_request y get_reply*, una para el envío y otra para la recepción?
10. Amoeba afirma que garantiza la semántica "a lo más una" en la RPC. Supongamos que tres servidores de archivos ofrecen el mismo servicio. Un cliente hace una RPC con uno de ellos, el cual lleva a cabo la solicitud y después falla. Entonces, la RCP se repite con otro servidor, lo cual implica que el trabajo se realizó dos veces. ¿Es esto posible? En tal caso, ¿qué significa la garantía? En caso contrario, ¿cómo se evita?
11. ¿Para qué necesita el secuenciador un buffer de historia?

12. En el texto se presentaron dos algoritmos para la transmisión en Amoeba. En el método 1, el emisor envía un mensaje puntual al secuenciador, el cual lo transmite. En el método 2, el emisor realiza la transmisión y el secuenciador transmite entonces un pequeño paquete de reconocimiento. Consideremos una red de 10 Mb/segundo donde el procesamiento de una interrupción llegada en forma de paquete tarda 500 microsegundos, de forma independiente al tamaño del paquete. Si todos los paquetes de datos tienen 1K bytes, y los paquetes de reconocimiento tienen 100 bytes, ¿qué ancho de banda y cuánto tiempo de CPU se utilizan en cada 1 000 transmisiones mediante los dos métodos?

13. ¿Qué propiedad del direccionamiento FLIP permite controlar la migración de procesos y la reconfiguración automática de la red de manera directa?

14. El servidor de archivos soporta los archivos inmutables para sus usuarios. ¿Son también inmutables las propias tablas del servidor de archivos?

15. ¿Para qué tiene el servidor de archivos archivos comprometidos y no comprometidos?

16. En Amoeba, se pueden crear enlaces con un archivo mediante la colocación de posibilidades con derechos distintos en los diversos directorios. Esto le otorga distintos permisos a los usuarios. Esta característica no está presente en UNIX. ¿Por qué?

8

Estudio 2: Mach

Nuestro segundo ejemplo de sistema operativo moderno se basa en un micronúcleo, Mach. Comenzaremos con un vistazo a su historia y su evolución a partir de sistemas anteriores. Después examinaremos con cierto detalle el propio micronúcleo, con énfasis en los procesos y los hilos, la administración de la memoria y la comunicación. Por último, analizaremos la emulación de UNIX. Se puede encontrar mayor información relativa a Mach en (Acetta *et al.*, 1986; Baron *et al.*, 1985; Black *et al.*, 1992; Boykin *et al.*, 1993; Draves *et al.*, 1991; Rashid, 1986a; Rashid, 1986b; y Sansom *et al.*, 1986).

8.1. INTRODUCCIÓN A MACH

En esta sección daremos una breve introducción a Mach. Comenzaremos con su historia y objetivos. Después describiremos los principales conceptos del micronúcleo de Mach y el servidor principal que se ejecuta en el núcleo.

8.1.1. Historia de Mach

Las primeras raíces de Mach van hasta un sistema llamado **RIG (Rochester Intelligent Gateway)** que se inició en la Universidad de Rochester en 1975 (Ball *et al.*, 1976). RIG fue escrito para una minicomputadora de 16 bits de Data General llamada Eclipse. Su principal objetivo de investigación era demostrar que se podían estructurar los sistemas operativos de manera modular, como una colección de procesos que se comuniquen entre sí mediante la transferencia de mensajes, incluso a través de una red. El sistema se diseñó y se construyó, lo cual mostró la factibilidad de dichos sistemas operativos.

Cuando uno de sus diseñadores, Richard Rashid, salió de Rochester y se trasladó a Carnegie Mellon University (CMU) en 1979, él deseaba continuar desarrollando sistemas operativos con transferencia de mensajes, pero en un hardware más moderno. Se consideraron varias máquinas. La elegida fue la PERQ, una de las primeras estaciones de trabajo de ingeniería, con una pantalla dada por un mapa de bits, un ratón y una conexión de red. También era microprogramable. El nuevo sistema operativo para la PERQ se llamó **Accent**. Superó a RIG al añadir protección, además de que podía operar de manera transparente a través de la red; tenía una memoria virtual de 32 bits y otras características. Se echó a andar una versión inicial en 1981.

En 1984, Accent era utilizado en la PERQ 150, pero perdió terreno debido a UNIX. Esta observación hizo que Rashid iniciara un proyecto de sistemas operativos de tercera generación llamado **Mach**. Al hacer que Mach fuese compatible con UNIX, él esperaba poder utilizar el enorme volumen disponible de software para UNIX. Además, Mach tenía varias mejoras en relación con Accent: los hilos, un mejor mecanismo de comunicación entre procesos, soporte de multiprocesador y un sistema de memoria virtual muy imaginativo.

En esa época, la DARPA (siglas en inglés de la Agencia de Proyectos de Investigación Avanzada del Departamento de Defensa de los Estados Unidos) andaba a la caza de un sistema operativo que soportara multiprocesadores, como parte de la Iniciativa de Cómputo Estratégico. Se eligió a CMU y, con fondos importantes de DARPA, Mach se desarrolló todavía más. Mach era una versión modificada de 4.1 BSD, con características adicionales insertadas para la comunicación y la administración de la memoria. Cuando se dispuso de 4.2 BSD y 4.3 BSD, el código de Mach se combinó con ellos para producir versiones actualizadas. Aunque este método produjo un núcleo de gran tamaño, garantizó la absoluta compatibilidad con el UNIX de Berkeley, un objetivo importante para DARPA.

La primera versión de Mach apareció en 1986, para la VAX 11/784, un multiprocesador de 4 CPU. Poco después, se realizaron las versiones para la IBM PC/RT y la Sun 3. Para el año de 1987, Mach también se ejecutaba en los multiprocesadores Encore y Sequent. Aunque Mach tenía la posibilidad del uso de redes, en esa época se concebía más como un sistema de una máquina o multiprocesador, que como un sistema operativo distribuido transparente para la colección de máquinas en una LAN.

Después, se instituyó la Open Software Foundation, un consorcio de vendedores de computadoras dirigidos por IBM, DEC y Hewlett-Packard, con el fin de arrebatarle el control de UNIX a su creador, AT&T, quien en ese tiempo trabaja de cerca con Sun Microsystems para desarrollar el sistema V versión 4. Los miembros de OSF temían que esta alianza podría darle a Sun una ventaja competitiva sobre ellos. Después de algunos pasos en falso, OSF eligió a Mach 2.5 como la base para su primer sistema operativo, OSF/1. Aunque Mach 2.5 y OSF/1 contenían grandes secciones del código de Berkeley y AT&T, la esperanza era que al menos se lograra el control de la dirección en que se encaminaba UNIX.

En 1988, el núcleo de Mach 2.5 era grande y monolítico, debido a la presencia de gran parte del código UNIX de Berkeley en el núcleo. En 1988, CMU eliminó todo el código de Berkeley del núcleo y lo colocó en el espacio del usuario. Lo que restaba era un micronúcleo sólo con Mach. En este capítulo nos centraremos en el micronúcleo de Mach 3 y un emulador del sistema operativo BSD UNIX a nivel usuario. Sin embargo, la dificultad es que Mach sigue en desarrollo, por lo que cualquier descripción es a lo más como una foto instantánea. Por fortuna, gran parte de las ideas básicas analizadas en este capítulo son relativamente estables, pero algunos de los detalles podrían cambiar con el tiempo.

8.1.2. Objetivos de Mach

Mach ha evolucionado de manera considerable a partir de su primera encarnación como RIG. Los objetivos del proyecto también han variado con el tiempo. Los actuales objetivos principales de Mach se pueden resumir de la manera siguiente:

1. Proporcionar una base para la construcción de otros sistemas operativos (por ejemplo, UNIX).
2. Soporte de un espacio de direcciones ralo y de gran tamaño.
3. Permitir el acceso transparente a los recursos de la red.
4. Explotar el paralelismo tanto en el sistema como en las aplicaciones.
5. Hacer que Mach se pueda transportar a una colección más grande de máquinas.

Estos objetivos abarcan tanto la investigación como el desarrollo. La idea es explorar los multiprocesadores y los sistemas distribuidos, a la vez que se puedan emular los sistemas ya existentes, como UNIX,MS-DOS y el sistema operativo de Macintosh.

La mayor parte del trabajo inicial en Mach se centraba en los sistemas con un procesador o con un multiprocesador. Cuando Mach se diseñó, pocos sistemas tenían el soporte para los multiprocesadores. Aún ahora, pocos sistemas multiprocesadores distintos de Mach son independientes de la máquina.

8.1.3. El micronúcleo de Mach

El micronúcleo de Mach se diseñó como base para emular UNIX y otros sistemas operativos. La emulación se lleva a cabo mediante una capa del software que se ejecuta fuera del núcleo, en el espacio del usuario, como se muestra en la figura 8-1. Cada emulador consta de una parte que está presente en el espacio de direcciones de los programas de aplicación, así como uno o más servidores que se ejecutan de manera independiente a los programas de aplicación. Hay que observar que se pueden ejecutar varios emuladores al mismo tiempo, por lo que es posible ejecutar programas en 4.3BSD, el Sistema V y MS-DOS, en la misma máquina al mismo tiempo.

El núcleo de Mach, al igual que otros micronúcleos, proporciona la administración de procesos, la administración de la memoria, la comunicación y los servicios de E/S. Los archivos, directorios y demás funciones tradicionales del sistema operativo se controlan en

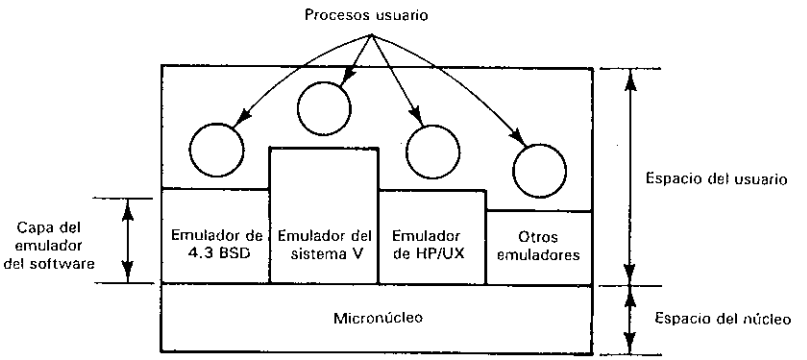


Figura 8-1. El modelo abstracto para la emulación de UNIX mediante Mach.

el espacio del usuario. La idea subyacente en el núcleo de Mach es proporcionar los mecanismos necesarios para que el sistema funcione, pero dejando la política a los procesos a nivel usuario.

El núcleo controla cinco abstracciones principales:

1. Procesos.
2. Hilos.
3. Objetos de la memoria.
4. Puertos.
5. Mensajes.

Además, el núcleo controla algunas otras abstracciones, ya sea relacionadas con éstas o menos centrales en el modelo.

Un proceso es básicamente un ambiente donde se lleva a cabo la ejecución. Tiene un espacio de direcciones que contiene el texto y datos del programa y por lo general una o más pilas. El proceso es la unidad básica para la asignación de recursos. Por ejemplo, un canal de comunicación siempre es "poseído" por un proceso.

Debemos mencionar que a lo largo de este capítulo nos apegaremos a la terminología tradicional, aunque esto signifique desviarnos de la terminología utilizada en los artículos relativos a Mach (por ejemplo, es claro que Mach tiene procesos, pero se les llama "tareas").

Un hilo en Mach es una entidad ejecutable. Tiene un contador del programa y un conjunto de registros asociados a él. Cada hilo es parte con exactitud de un proceso. Un proceso con un hilo es similar a un proceso tradicional (por ejemplo, UNIX).

Un concepto exclusivo de Mach es el de **objeto de memoria**, una estructura de datos que se puede asociar con el espacio de direcciones de un proceso. Los objetos de memoria ocupan una o más páginas y forman la base del sistema de memoria virtual de Mach. Cuando un proceso intenta hacer referencia a un objeto de memoria no presente en la memoria física principal, obtiene un fallo de página. Como en todos los sistemas operativos, el núcleo

captura este fallo. Sin embargo, a diferencia de otros sistemas, el núcleo de Mach puede enviar un mensaje a un servidor a nivel usuario para buscar la página faltante.

La comunicación entre procesos en Mach se basa en la transferencia de mensajes. Para recibir éstos, un proceso usuario pide al núcleo que cree para él un tipo de buzón protegido, llamado **puerto**. El puerto se almacena dentro del núcleo y forma parte de una cola compuesta por una lista ordenada de mensajes. Las colas no tienen un tamaño fijo, pero por razones de control de flujo, si más de *n* mensajes están formados en una cola, un proceso que intente hacer un envío a ésta se suspende, para que el puerto se pueda vaciar. El parámetro *n* se puede configurar para cada puerto.

Un puerto recibe la facultad de enviar a (o recibir de) alguno de sus puertos a otro proceso. Este permiso toma la forma de una **posibilidad** y no sólo incluye un apuntador al puerto, sino también una lista de derechos que el otro proceso tiene con respecto del puerto (por ejemplo, el derecho SEND). Una vez otorgado este permiso, el otro proceso puede enviar mensajes al puerto, el cual puede entonces ser leído por el primer proceso. Toda la comunicación en Mach utiliza este mecanismo. Mach no soporta un mecanismo completo de posibilidades; los puertos son los únicos objetos para los que existen las posibilidades.

8.1.4. El servidor BSD UNIX de Mach

Como hemos descrito, los diseñadores de Mach modificaron el UNIX de Berkeley para ejecutarlo en el espacio del usuario, como programa de aplicación. Esta estructura tiene varias ventajas significativas sobre un núcleo monolítico. En primer lugar, al separar el sistema en una parte que controla la administración de los recursos (el núcleo) y otra que controla las llamadas al sistema (el servidor de UNIX), ambas partes son más sencillas y fáciles de mantener. En cierta forma, esta separación recuerda la división del trabajo en el sistema operativo Mainframe VM/370 de IBM, donde el núcleo simula una colección de 370, cada una de las cuales ejecuta un sistema operativo de un usuario.

En segundo lugar, al colocar a UNIX en el espacio del usuario, puede hacerse muy independiente de la máquina, mejorando con ello su portabilidad a una amplia gama de computadoras. Todas las dependencias de la máquina se pueden eliminar de UNIX y ocultarse en el núcleo de Mach.

En tercer lugar, como hemos mencionado, se pueden ejecutar varios sistemas operativos en forma simultánea. Por ejemplo, en una 386, Mach puede ejecutar un programa en UNIX y otro programa en MS-DOS al mismo tiempo. De forma similar, es posible probar un nuevo sistema operativo experimental y ejecutar un sistema operativo de producción al mismo tiempo.

En cuarto lugar, se puede añadir al sistema la operación en tiempo real, puesto que todos los obstáculos tradicionales que UNIX presenta en el trabajo en tiempo real, como la desactivación de interrupciones para la actualización de las tablas críticas se eliminan en su conjunto o se desplazan al espacio del usuario. El núcleo se puede estructurar con cuidado para no tener este tipo de dificultades en las aplicaciones de tiempo real.

Por último, este orden se puede utilizar para proporcionar una mayor seguridad entre los procesos, en caso necesario. Si cada proceso tiene su propia versión de UNIX, es muy difícil que un proceso pueda husmear en los archivos de otro proceso.

8.2. ADMINISTRACIÓN DE PROCESOS EN MACH

La administración de los procesos en Mach se encarga de los procesos, los hilos y la planificación. En esta sección analizaremos cada uno a su tiempo.

8.2.1. Procesos

Un proceso en Mach consta principalmente de un espacio de direcciones y una colección de hilos que se ejecutan en ese espacio de direcciones. Los procesos son pasivos. La ejecución se asocia con los hilos. Los procesos se utilizan para recolectar en recipientes convenientes todos los recursos relacionados con un grupo de hilos que cooperan entre sí.

La figura 8-2 ilustra un proceso en Mach. Además de un espacio de direcciones y los hilos, tiene ciertos puertos y otras propiedades. Todos los puertos que se muestran en la figura tienen funciones especiales. El **puerto del proceso** se utiliza para la comunicación con el núcleo. Muchos de los servicios del núcleo que pueden ser solicitados por un proceso se realizan mediante el envío de un mensaje al puerto del proceso, en vez de hacer una llamada al sistema. Este mecanismo se utiliza en todo Mach para reducir el número de las llamadas al sistema reales a un mínimo estricto. En este capítulo analizaremos un pequeño número de ellas, para dar una idea de lo que son.

En general, el programador no es consciente de si un servicio necesita una llamada al sistema o no. Todos los servicios, incluso aquellos cuyo acceso se realiza mediante llamadas al sistema o mediante una transferencia de mensajes, tienen procedimientos de resguardo en la biblioteca. Estos son los procedimientos que se describen en los manuales y que son llamados por los programas de aplicación. Los procedimientos se generan mediante una definición de servicio mediante el compilador **MIG (Mach Interface Generator)**.

El **puerto de arranque** se utiliza al comienzo para la iniciación de todos los parámetros de un proceso. El primer proceso lee el puerto de arranque para aprender los nombres de los puertos del núcleo que proporcionan los servicios esenciales. Los procesos de UNIX también lo utilizan para la comunicación con el emulador de UNIX.

El **puerto de excepción** es utilizado por el sistema para informar de las excepciones causadas por el proceso. Las excepciones típicas son la división entre cero y la ejecución de una instrucción no válida. El puerto indica al sistema el lugar donde envía el mensaje de excepción. Los depuradores también utilizan los puertos de excepción.

Los **puertos registrados** se utilizan por lo general para proporcionar al proceso una forma de comunicación con los servidores estándar del sistema. Por ejemplo, el servidor de nombres permite presentar una cadena y obtener el puerto correspondiente a ciertos servidores básicos.

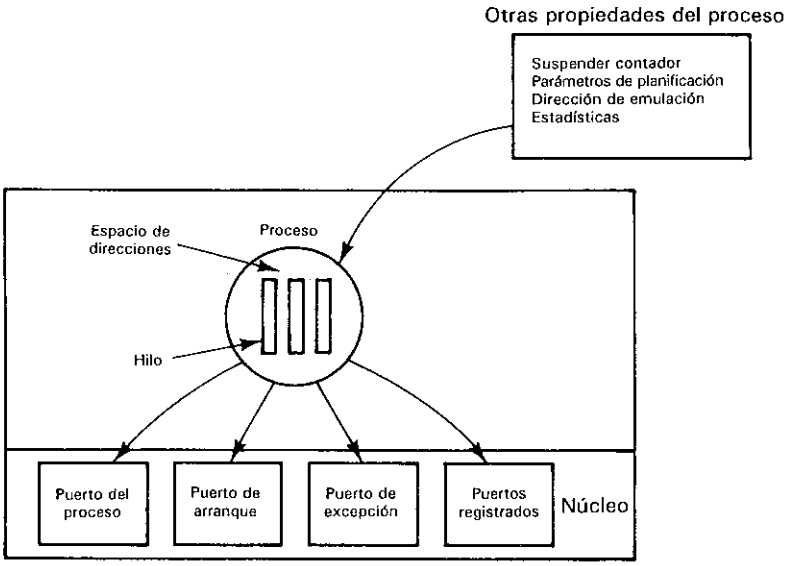


Figura 8-2. Un proceso en Mach.

Los procesos también tienen otras propiedades. Un proceso puede ser ejecutable o estar bloqueado, de manera independiente al estado de sus hilos. Si un proceso es ejecutable, entonces los hilos que también sean ejecutables se pueden planificar y ejecutar. Si un proceso está bloqueado, sus hilos no se pueden ejecutar, sin importar el estado que tengan.

Los elementos relativos a cada proceso también incluyen parámetros de planificación. Entre ellos se encuentran la capacidad de especificar los procesadores donde se pueden ejecutar los hilos del proceso. Esta característica es más útil en un sistema multiprocesador. Por ejemplo, el proceso puede utilizar esta facultad para provocar que cada hilo se ejecute en un procesador distinto, para provocar la ejecución de todos en el mismo procesador, o cualquier situación intermedia. Además, cada proceso tiene una prioridad predefinida ajustable. Cuando se crea un hilo, el nuevo hilo recibe esa prioridad. También es posible modificar la prioridad de todos los hilos existentes.

Se puede configurar una dirección de emulación para indicar al núcleo la localización de los controladores de llamadas al sistema en el espacio de direcciones del proceso. El núcleo necesita conocer esas direcciones para controlar las llamadas al sistema UNIX que deben ser emuladas. Éstas se configuran una vez iniciado el emulador de UNIX y son heredadas por todos los hijos del emulador (es decir, todos los procesos UNIX).

Por último, cada proceso tiene estadísticas asociadas a él, como la cantidad de memoria consumida, los tiempos de ejecución de sus hilos, etc. Un proceso interesado en esa información puede obtenerla al enviar un mensaje al puerto del proceso.

También es importante mencionar algo de lo que no tiene Mach. Un proceso no tiene una identificación del usuario, una gid, plantilla de señales, directorio raíz, directorio de trabajo o arreglo con los descriptores de archivos, todo lo cual sí poseen los procesos en UNIX. Toda esta información se controla en el paquete de emulación, de modo que el núcleo no sabe nada con respecto de ella.

Primitivas de la administración de los procesos

Mach proporciona un pequeño número de primitivas para la administración de los procesos. La mayor parte de éstas se realizan mediante el envío de mensajes al núcleo por medio del puerto del proceso, en vez de hacer llamadas reales al sistema. En la figura 8-3 se muestran las llamadas más importantes. Éstas, como todas las llamadas de Mach, tienen prefijos que indican el grupo al cual pertenecen, pero los hemos omitido aquí (y en las siguientes tablas) por brevedad.

Llamada	Descripción
Create	Crea un nuevo proceso, heredando ciertas propiedades
Terminate	Elimina un proceso específico
Suspend	Incrementa el contador de suspensión
Resume	Decrementa el contador de suspensión. Si es 0, libera el bloqueo del proceso
Priority	Establece la prioridad para los hilos actuales o futuros
Assign	Indica el procesador donde deben ejecutarse los hilos nuevos
Info	Regresa información acerca del tiempo de ejecución, uso de la memoria, etcétera
Threads	Regresa una lista de los hilos del proceso

Figura 8-3. Algunas llamadas para la administración de procesos en Mach.

Las primeras dos llamadas de la figura 8-3 se refieren a la creación y destrucción de procesos, respectivamente. La llamada para la creación de procesos especifica un proceso prototipo, que no necesariamente es el proceso que hizo la llamada. El hijo es una copia del prototipo, excepto que la llamada tiene un parámetro que indica si el hijo debe o no heredar el espacio de direcciones del padre. Si no hereda este espacio, los objetos (por ejemplo, el texto, los datos iniciados y una pila) se pueden asociar con dicho espacio de direcciones posteriormente. En un principio, el hijo no tiene hilos. Sin embargo, obtiene

de manera automática un puerto de proceso, uno de arranque y uno de excepción. Los demás puertos no se heredan de manera automática, ya que cada uno sólo puede tener un lector.

Los procesos se pueden suspender y reasumir bajo el control del programa. Cada proceso tiene un contador, que se incrementa mediante la llamada *suspend* y se decrementa mediante la llamada *resume*, que puede bloquearlo o eliminar un bloqueo. Si el contador es 0, el proceso se puede ejecutar. Si es positivo, se suspende. Un contador es más general que un bit y ayuda a evitar condiciones de competencia.

Las llamadas *priority* y *assign* dan al programador el control sobre la forma y el lugar de ejecución de sus hilos en los sistemas multiprocesadores. La planificación del CPU se logra mediante prioridades, de modo que el programador tenga un control fino sobre la decisión de cuáles sean los hilos más importantes y cuáles los menos. La llamada *assign* permite controlar el o los hilos que se deban ejecutar en un CPU específico o en cierto grupo de CPU.

Las últimas dos llamadas de la figura 8-3 regresan la información relativa al proceso. La primera de ellas proporciona información estadística y la segunda regresa una lista de todos los hilos.

8.2.2. Hilos

Las entidades activas en Mach son los hilos. Estos ejecutan instrucciones y controlan sus registros y espacio de direcciones. Cada hilo pertenece con exactitud a un proceso. Este no puede llevar nada a cabo si no tiene uno o más hilos.

Todos los hilos de un proceso comparten el espacio de direcciones y todos los recursos a todo lo ancho del proceso, los cuales se muestran en la figura 8-2. Sin embargo, los hilos también tienen recursos particulares de cada uno de ellos. Uno de éstos es el **puerto del hilo**, análogo al puerto del proceso. Cada hilo tiene su puerto del hilo, que utiliza para llamar ciertos servicios del núcleo específicos para hilos, como la salida al terminar un hilo. Puesto que los puertos son recursos a todo lo ancho de los procesos, cada hilo tiene acceso a los puertos de sus hermanos, por lo que cada hilo controla a los demás en caso necesario.

Los hilos de Mach son controlados por el núcleo; es decir, son lo que a veces se conoce como hilos pesados en vez de ligeros (hilos correspondientes en su totalidad al espacio del usuario). El núcleo crea y destruye los hilos mediante la actualización de ciertas estructuras de datos de dicho núcleo. Estas estructuras proporcionan los mecanismos básicos para el manejo de múltiples actividades dentro de un espacio de direcciones. Lo que haga el usuario con estos mecanismos es su responsabilidad.

En un sistema con un CPU, los hilos se controlan mediante el tiempo compartido; primero se ejecuta uno y después otro. En un multiprocesador, pueden existir varios hilos activos al mismo tiempo. Este paralelismo hace que la exclusión mutua, la sincronización y la planificación sean más importantes de lo normal, puesto que el desempeño y lo correcto de los procesos se convierten ahora en asuntos cruciales. Puesto que se pretende la ejecución de Mach en multiprocesadores, estos aspectos han recibido una atención especial.

Como proceso, un hilo puede ser ejecutable o estar bloqueado. El mecanismo también es similar: un contador por cada hilo, el cual se puede incrementar o decrementar. Cuando es 0, el hilo es ejecutable. Cuando es positivo, el hilo debe esperar hasta que otro lo reduzca a 0. Este mecanismo permite que los hilos controlen el comportamiento de los demás.

Se dispone de varias primitivas. La interfaz básica del núcleo proporciona cerca de dos docenas de primitivas para hilos, muchas de las cuales se encargan de controlar con detalle la planificación. Por arriba de estas primitivas se pueden construir varios paquetes de hilos.

El método de Mach es el paquete de **hilos C**. Este paquete pretende que todas las primitivas de hilos del núcleo estén disponibles para los usuarios en forma sencilla y conveniente. No tiene toda la potencia que ofrece la interfaz del núcleo, pero es suficiente para el común de los programadores. También está diseñado para ser portable a una amplia gama de sistemas operativos y arquitecturas.

El paquete de hilos C proporciona 16 llamadas para el manejo directo de los hilos. Las más importantes se enumeran en la figura 8-4. La primera, *fork*, crea un nuevo hilo en el mismo espacio de direcciones del hilo que hizo la llamada. Ejecuta el procedimiento especificado por un parámetro en vez del código del padre. Después de la llamada, el hilo padre continúa su ejecución en paralelo con el hijo. El hilo se inicia con una prioridad y en un procesador determinado mediante los parámetros de planificación del proceso, como hemos analizado.

Llamada	Descripción
Fork	Crea un nuevo hilo que ejecuta el mismo código que el hilo padre
Exit	Termina al hilo de la llamada
Join	Suspende al proceso que hace la llamada hasta que un hilo realice su salida
Detach	Anuncia que nunca se hará una operación join en el hilo (nunca se esperará por él)
Yield	El CPU desiste de manera voluntaria
Self	Regresa la identidad del hilo que hace la llamada

Figura 8-4. Las llamadas a los hilos C para el manejo directo de los hilos.

Cuando un hilo termina su trabajo, llama a *exit*. Si el padre está interesado en esperar a que el hilo termine, puede llamar a *join* para bloquearse también hasta que un hilo hijo determinado termine su ejecución. Si el hilo ya ha terminado, el padre continúa de manera inmediata. Estas tres llamadas son más o menos similares a las llamadas al sistema FORK, EXIT y WAITPID de UNIX.

La cuarta llamada, *detach*, no existe en UNIX. Proporciona una forma para anunciar que nunca se esperará la conclusión de un hilo determinado. Si ese hilo llega a existir, su pila y demás información de estado serán eliminados en forma inmediata. Por lo general, esta

operación de limpieza sólo ocurre después de que el padre ha realizado un *join* con éxito. En un servidor, se podría dar inicio a un nuevo hilo para dar servicio al recibir una solicitud. Al terminar, el hilo ejecuta *exit*. Puesto que no hay necesidad de esperar al hilo inicial, el hilo servidor debe ser eliminado.

La llamada *yield* es una indicación al planificador en el sentido de que el hilo no tiene algo útil que hacer por el momento y que espera la ocurrencia de cierto evento antes de poder continuar. Un planificador inteligente tomaría la indicación y ejecutaría otro hilo. En Mach, que por lo general planifica sus hilos con prioridad, *yield* sólo es un factor de optimización. En los sistemas con planificación sin prioridades, es esencial que un hilo sin trabajo libere el CPU, para que otros hilos se puedan ejecutar.

Por último, *self* regresa la identidad de quien hizo la llamada, en forma similar al caso GETPID en UNIX.

Las demás llamadas (que no aparecen en la figura) permiten nombrar a los hilos, y que el programa controle el número de hilos y los tamaños de sus pilas, proporcionando interfaces con los hilos del núcleo y el mecanismo de transferencia de mensajes.

La sincronización se lleva a cabo mediante mútex y variables de condición. Las primitivas de los mútex son *lock, trylock* y *unlock*. También se dispone de primitivas para asignar y liberar mútex. Los mútex funcionan como semáforos binarios, proporcionando la exclusión mutua, pero sin llevar información.

Las operaciones sobre las variables de condición son *signal, wait* y *broadcast*, las cuales permiten que los hilos se bloqueen en espera de cierta condición y que sean despertados posteriormente cuando otro hilo provoca la ocurrencia de dicha condición.

Implantación de los hilos C en Mach

Se dispone de varias implantaciones de los hilos C en Mach. La original lleva todo a cabo en el espacio del usuario, dentro de un proceso. Este método utilizaba el tiempo compartido para todos los hilos C en un hilo del núcleo, como se muestra en la figura 8-5(a). Este método también se puede utilizar en UNIX o en cualquier otro sistema que no proporcione un soporte del núcleo. Los hilos se ejecutaban como co-rutinas, lo que significa que se planificaban sin prioridades. Un hilo podía conservar el CPU durante el tiempo que quisiera o pudiera. Para el problema de los productores y los consumidores, el productor podría llenar el buffer y después bloquearse, dándole al consumidor la oportunidad para su ejecución. Sin embargo, para otras aplicaciones, los hilos debían llamar a *yield* de vez en cuando para darle oportunidad a los demás hilos.

El paquete original de implantación adolecía de un problema inherente a la mayoría de los paquetes de hilos en el espacio del usuario que no tienen soporte del núcleo. Si un hilo hace una llamada al sistema con bloqueo, como una lectura de la terminal, todo el proceso se bloquea. Para evitar esta situación, el programador debe evitar las llamadas al sistema con bloqueo. En el UNIX de Berkeley, existe una llamada SELECT que se puede utilizar para indicar si alguna característica está pendiente, pero toda la situación es algo confusa.

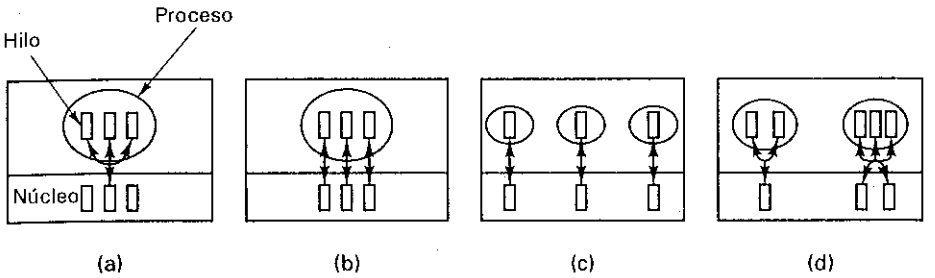


Figura 8-5. (a) Todos los hilos C utilizan un hilo del núcleo. (b) Cada hilo C tiene su propio hilo del núcleo. (c) Cada hilo C tiene su propio proceso de un solo hilo. (d) Asociación arbitraria de los hilos del usuario con los hilos del núcleo.

Una segunda implantación consiste en utilizar un hilo Mach por cada hilo C, como se muestra en la figura 8-5(b). Estos hilos se planifican sin prioridades. Además, en un multiprocesador, se podrían ejecutar en realidad en paralelo, en varios CPU. De hecho, también es posible hacer una conexión con un multiplexor de m hilos del usuario con n hilos del núcleo, aunque el caso más usual es que $m = n$.

Un tercer paquete de implantación tiene un hilo por cada proceso, como se muestra en la figura 8-5(c). Los procesos se configuran de tal forma que sus espacios de direcciones se asocien con la misma memoria física, lo que permite compartir de la misma forma que las anteriores implantaciones. Esta implantación sólo se utiliza cuando se necesita un uso especializado de la memoria virtual. El método tiene la desventaja de que los puertos, los archivos en UNIX y otros recursos exclusivos de cada proceso no se pueden compartir, lo cual limita su valor de manera apreciable.

El cuarto paquete permite que una cantidad arbitraria de hilos del usuario se asocien con una cantidad arbitraria de hilos del núcleo, como se muestra en la figura 8-5(d).

El principal valor práctico del primer método es que, debido a que no existe un verdadero paralelismo, las ejecuciones sucesivas dan resultados reproducibles, lo que permite una depuración más fácil. El segundo método tiene la ventaja de su sencillez y se utilizó durante mucho tiempo. El tercero no se utiliza por lo general. El cuarto, aunque más complejo, proporciona máxima flexibilidad y es el de uso normal en la actualidad.

8.2.3. Planificación

La planificación de Mach se ha visto muy influida por su objetivo de ejecución en multiprocesadores. Puesto que un sistema con un procesador es en realidad un caso particular de un multiprocesador (con un CPU), nuestro análisis se centrará en la planificación en sistemas de multiprocesadores. Para mayor información, véase (Black, 1990).

Los CPU de un multiprocesador pueden ser asignados a **conjuntos de procesadores** por medio del software. Cada CPU pertenece con exactitud a un conjunto de procesadores. También los hilos se pueden asignar a conjuntos de procesadores mediante el software.

Así, cada conjunto de procesadores tiene una colección de CPU a su disposición y una colección de hilos que necesitan poder de cómputo. El trabajo del algoritmo de planificación es asignar hilos a CPU de manera justa y eficaz. Para los fines de la planificación, cada conjunto de procesadores es un mundo cerrado, con sus propios recursos y sus propios clientes, de manera independiente a los demás conjuntos de procesadores.

Este mecanismo le da a los procesos un gran control sobre sus hilos. Un proceso puede asignar un hilo importante a un conjunto de procesadores con un CPU sin hilos adicionales, lo cual garantiza que el hilo se ejecuta todo el tiempo. También puede reasignar hilos de manera dinámica a conjuntos de procesadores durante el trabajo, para mantener balanceada la carga de trabajo. Aunque no es probable que el compilador promedio utilice esta facultad, la podría utilizar un sistema para el manejo de una base de datos o un sistema de tiempo real.

La planificación de los hilos en Mach se basa en las prioridades. Las prioridades son números de 0 hasta un número máximo (por lo general, 31 o 127), donde 0 representa la máxima prioridad y 31 o 127 la mínima. Esta prioridad inversa proviene de UNIX. Cada hilo tiene tres prioridades asignadas a él. La prioridad es una prioridad base, que cada hilo puede establecer por su cuenta, dentro de ciertos límites. La segunda prioridad es el mínimo valor numérico que el hilo puede establecer como prioridad base. Puesto que el uso de un valor mayor produce un peor servicio, lo normal será que un hilo establezca su valor como el mínimo permitido, a menos que de manera intencional intente retrasarse con respecto de otros hilos. La tercera prioridad es la prioridad actual, la cual se utiliza con fines de planificación. El núcleo la calcula como la suma de la prioridad base con una función basada en el uso reciente del CPU por parte del hilo.

Los hilos de Mach son visibles para el núcleo, al menos cuando se utiliza el modelo de la figura 8-5(b). Cada hilo compite con otros hilos por los ciclos de un CPU, sin importar el proceso al que pertenezcan. El núcleo no toma en cuenta cuál hilo pertenece a cuál proceso al tomar las decisiones de planificación.

A cada conjunto de procesadores se le asocia un arreglo de colas de ejecución, como se muestra en la figura 8-6. El arreglo tiene 32 colas, correspondientes a los hilos con prioridades de la 0 a la 31. Cuando un hilo de prioridad *n* es ejecutable, se le coloca al final de la cola *n*. Un hilo no ejecutable no está presente en ninguna de las colas de ejecución.

Cada cola de ejecución tiene tres variables asociadas a ella. La primera es un mútex que se utiliza para cerrar la estructura de datos. Se utiliza para garantizar que un CPU controla las colas en cada momento. La segunda es el contador del número de hilos en todas las colas juntas. Si este contador es 0, no hay trabajos por realizar. La tercera variable es un indicador de la posición del hilo con máxima prioridad. Se garantiza que ningún hilo está en la prioridad máxima, pero el más alto podría estar en una prioridad menor. Esta indicación permite la búsqueda del hilo con máxima prioridad para evitar colas vacías en la parte superior.

Además de las colas de ejecución globales de la figura 8-6, cada CPU tiene su propia cola de ejecución local. Cada una de estas colas locales tiene aquellos hilos que están limitados de manera permanente a dicho CPU; por ejemplo, por ser controladores de dis-

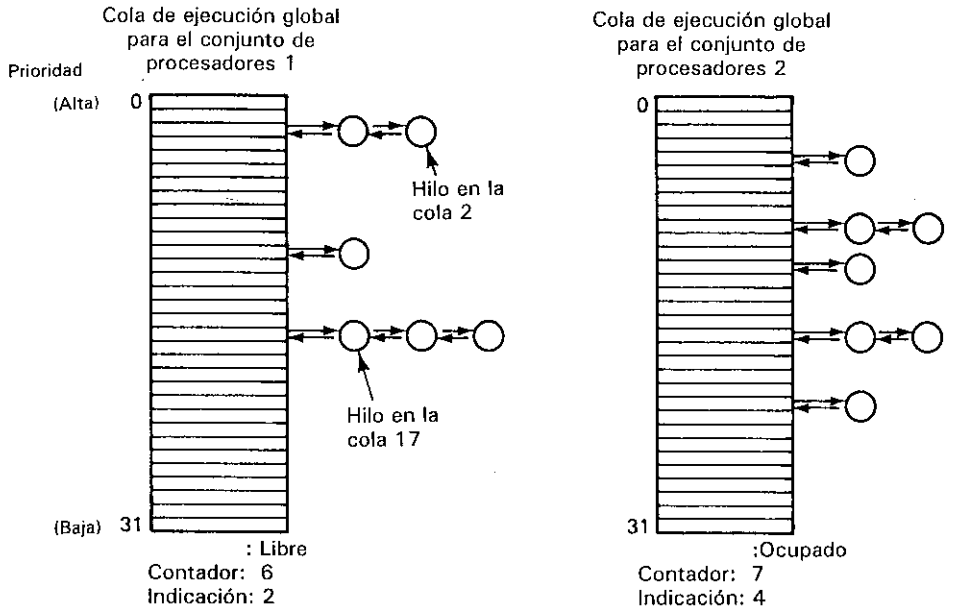


Figura 8-6. Las colas globales de ejecución para un sistema con dos conjuntos de procesadores.

positivos para los dispositivos de E/S conectados a ese CPU. Estos hilos sólo se ejecutan en un CPU, por lo que sería incorrecta su colocación en una cola global de ejecución (puesto que podrían ser elegidos por el CPU "incorrecto").

Ahora podemos describir el algoritmo básico de planificación. Cuando un hilo se bloquea, sale o agota su quantum, el CPU donde se ejecuta busca en primer término en su cola de ejecución local para ver si existen hilos activos. Esta verificación sólo necesita de la inspección de la variable de conteo asociada a la cola local. Si es diferente de cero, el CPU comienza la búsqueda en la cola del hilo con máxima prioridad, a partir de la cola especificada por la indicación. Si la cola local está vacía, se aplica el mismo algoritmo a la cola global, sólo con una diferencia: la cola global se debe cerrar antes de buscar en ella. Si no existen hilos para su ejecución en alguna cola, se ejecuta un hilo especial inactivo hasta que algún hilo se declare listo.

Si se encuentra un hilo ejecutable, se le planifica y ejecuta durante un quantum. Al final del quantum, se verifican las colas local y global para ver si son ejecutables otros hilos con su prioridad u otra mayor, en el entendido de que todos los hilos de la cola local tienen mayor prioridad que los hilos de la cola global. Si se encuentra un candidato adecuado, ocurre una conmutación de hilos. Si no, se ejecuta el hilo durante otro quantum. Los hilos también pueden tener prioridades. En los multiprocesadores, la longitud del quantum es variable, según el número de hilos ejecutables. Mientras existan más hilos ejecutables y menos CPU, menor será el quantum. Este algoritmo proporciona tiempos cortos de res-

puesta para las solicitudes cortas, incluso en los sistemas con carga pesada, pero proporciona una alta eficacia (es decir, quanta de gran tamaño) en los sistemas con poca carga.

En cada marca de reloj, el CPU incrementa en una pequeña cantidad el contador de prioridades del hilo que en la actualidad se encuentra en ejecución. Al subir el valor, la prioridad baja y el hilo se desplazará a una cola con un número más grande (es decir, con menor prioridad). Los contadores de prioridad se decrementan en la medida del tiempo transcurrido.

Para ciertas aplicaciones, un gran número de hilos podrían estar trabajando juntos para resolver un problema y podría ser importante una planificación detallada para un control detallado. Mach proporciona un "gancho" para darle a los hilos cierto control adicional sobre su planificación (además de los conjuntos de procesadores y las prioridades). El gancho es una llamada al sistema que permite a un hilo reducir su prioridad al mínimo absoluto durante cierto número de segundos. Esto permite que otros hilos tengan la oportunidad de ejecutarse. Al terminarse el tiempo, la prioridad regresa a su valor original.

Esta llamada al sistema tiene otra propiedad interesante: puede nombrar a su sucesor si así lo desea. Por ejemplo, después de enviar un mensaje a otro hilo, el hilo emisor puede entregar al CPU y solicitar que el hilo receptor pueda ejecutarse a continuación. Este mecanismo, conocido como **planificación manos-fuera**, pasa por arriba de las colas de ejecución. Si se utiliza de manera sabia, puede mejorar el desempeño. El núcleo también lo utiliza en ciertas circunstancias, como medio de optimización.

Mach se puede configurar para realizar una planificación por afinidad, pero por lo general esta opción no está activa. Si lo está, el núcleo planifica un hilo en el CPU donde acaba de ejecutar algo, con la esperanza de que parte de su espacio de direcciones se encuentre en el caché de ese CPU. La planificación por afinidad sólo se aplica a los multiprocesadores.

Por último, algunas versiones soportan otros algoritmos de planificación, incluyendo algoritmos útiles para las aplicaciones de tiempo real.

8.3. ADMINISTRACIÓN DE MEMORIA EN MACH

Mach tiene un poderoso sistema para la administración de la memoria, muy elaborado y muy flexible, basado en la paginación, con características que se encuentran poco en otros sistemas operativos. En particular, separa las partes independientes de la máquina de dicho sistema para la administración de la memoria de las partes dependientes de la máquina, de manera poco usual pero en extremo clara. Esta separación hace que la administración de la memoria sea más portable que en otros sistemas. Además, el sistema para la administración de la memoria interactúa de manera muy cercana con el de comunicación, que analizaremos en la siguiente sección.

El aspecto de la administración de memoria de Mach que establece la diferencia con los demás es que el código se divide en tres partes. La primera es el módulo *pmap*, que se ejecuta en el núcleo y se encarga del manejo del MMU. Configura los registros de MMU

y las tablas de páginas del hardware, además de capturar todos los fallos de página. Este código depende de la arquitectura MMU y debe ser escrito de nuevo para cada una de las máquinas nuevas a las que se traslade Mach. La segunda parte es el código del núcleo independiente de la máquina, encargado del procesamiento de los fallos de página, el manejo de los mapas de direcciones y el reemplazo de páginas.

La tercera parte del código de administración de la memoria se ejecuta como proceso usuario llamado **administrador de la memoria** o a veces **paginador externo**. Controla la parte lógica (en contraposición a la parte física) del sistema para la administración de memoria, principalmente el manejo de los espacios de respaldo (disco). Por ejemplo, mantiene un registro de las páginas virtuales en uso, las páginas virtuales en la memoria principal y la posición de almacenamiento de las páginas en el disco cuando no se encuentran en la memoria principal.

El núcleo y el administrador de la memoria se comunican por medio de un protocolo bien definido, lo que permite a los usuarios escribir sus propios administradores de memoria. Esta división del trabajo permite a los usuarios implantar sistemas de paginación con fines especiales y escribir sistemas con requisitos particulares. También tiene el potencial para hacer más pequeño y más sencillo el núcleo, al desplazar gran parte del código hacia el espacio del usuario. Por otro lado, también podría volverlo más complejo, puesto que el núcleo se protege a sí mismo de los administradores de memoria erróneos o maliciosos; además, con dos entes activos que realizan el manejo de la memoria, existe el peligro de condiciones de competencia.

8.3.1. Memoria virtual

El modelo conceptual de memoria que ven los procesos usuario de Mach es un espacio de direcciones virtuales grande y lineal. Para la mayoría de los circuitos de CPU de 32 bits, el espacio de direcciones va de la dirección 0 a la dirección $2^{31}-1$ puesto que el núcleo utiliza la mitad superior para sus fines particulares. El espacio de direcciones se soporta mediante la paginación. Puesto que ésta se diseñó con el fin de dar la ilusión de una memoria ordinaria, un poco más de lo que en realidad es, en principio no habría nada más que decir acerca de la forma en que Mach controla el espacio de direcciones virtuales.

En realidad, hay mucho que decir. Mach proporciona un control fino de la forma de uso de las páginas virtuales (para los procesos interesados en ello). Para comenzar, el espacio de direcciones se puede utilizar de manera rala. Por ejemplo, un proceso podría tener docenas de secciones del espacio de direcciones virtuales en uso, cada una a varios megabytes de distancia de su vecino más cercano, con grandes huecos de direcciones no utilizadas entre las secciones.

En teoría, todo espacio de direcciones virtuales se puede utilizar de esta manera, por lo que la capacidad de utilizar varias secciones dispersas no es en realidad una propiedad de la arquitectura del espacio de direcciones virtuales. En otras palabras, cualquier máquina de 32 bits debería permitir a un proceso tener una sección de 50K de datos espaciados cada 100 megabytes, desde 0 hasta el límite de 4 gigabytes. Sin embargo, en muchas implanta-

ciones, se mantiene una tabla lineal de páginas desde 0 hasta la máxima página utilizada dentro de la memoria del núcleo. En una máquina con un tamaño de página de 1K, esta configuración necesita 4 millones de entradas en la tabla de páginas, lo que la hace cara, si no es que imposible. Incluso con una tabla de páginas de varios niveles, tal uso ralo por lo menos es inconveniente. Con Mach, la intención es soportar por completo los espacios ralos de direcciones.

Para determinar las direcciones virtuales en uso, Mach proporciona la forma de asignar y liberar las secciones del espacio de direcciones virtuales, llamadas **regiones**. La llamada de asignación puede especificar una dirección base y un tamaño, en cuyo caso se asigna la región indicada; o bien, puede simplemente especificar un tamaño, en cuyo caso el sistema encuentra un rango de direcciones adecuado y regresa su dirección base. Una dirección virtual sólo es válida si cae dentro de una región asignada. Un intento por utilizar una dirección entre las regiones asignadas produce un señalamiento, el cual, sin embargo, puede ser capturado por el proceso si así se desea.

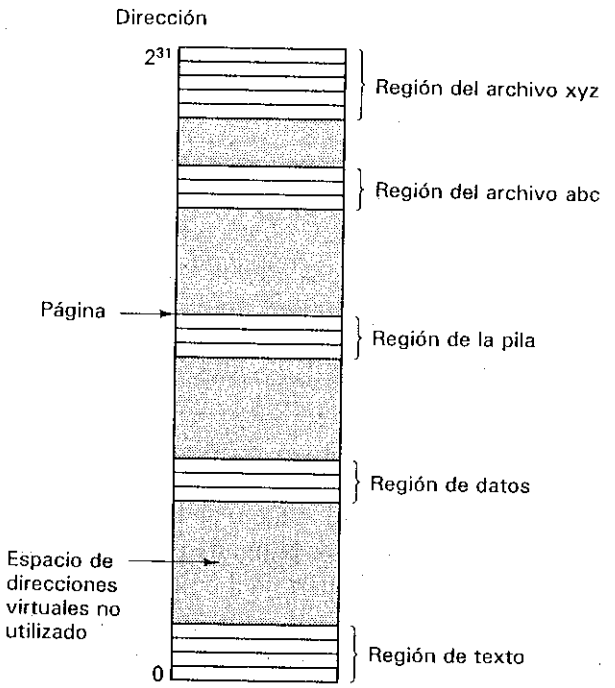


Figura 8-7. Un espacio de direcciones con regiones asignadas, objetos asociados y direcciones no utilizadas.

Un concepto fundamental relacionado con el uso del espacio de direcciones virtuales es el **objeto de memoria**. Un objeto de memoria puede ser una página o un conjunto de ellas, un archivo o alguna otra estructura de datos más particular. Un objeto de memoria

se puede asociar con una parte no utilizada del espacio de direcciones virtuales para formar una nueva región, como se muestra en la figura 8-7. Cuando un archivo es asociado con el espacio de direcciones virtuales, se puede leer y escribir mediante las instrucciones usuales de la máquina. Los archivos asociados se paginan de la manera usual. Cuando un proceso termina, sus archivos asociados aparecen automáticamente de nuevo en el sistema de archivos, con todos los cambios realizados en ellos al ser asociados. También es posible desasociar archivos u otros objetos de memoria de forma explícita, para liberar sus direcciones virtuales, de manera que éstas estén disponibles para una asignación o asociación posterior.

Por otro lado, la asociación de un archivo no es la única forma para tener acceso a él. También se pueden leer de manera convencional. Sin embargo, aún en ese caso, la biblioteca puede asociar los archivos a espaldas del usuario, en vez de leerlos mediante el sistema de E/S. Esto permite que las páginas del archivo utilicen el sistema de memoria virtual, en vez de utilizar buffers de uso exclusivo en todas partes del sistema.

Llamada	Descripción
Allocate	Hace que una región del espacio de direcciones virtuales sea utilizable
Deallocate	Invalida una región del espacio de direcciones virtuales
Map	Asocia un objeto de memoria en el espacio de direcciones virtuales
Copy	Crea una copia de una región en otra dirección virtual
Inherit	Configura los atributos de herencia de cierta región
Read	Lee datos del espacio de direcciones virtuales de otro proceso
Write	Escribe datos en el espacio de direcciones virtuales de otro proceso

Figura 8-8. Selección de algunas llamadas en Mach para el manejo de la memoria virtual.

Mach soporta varias llamadas para el manejo de los espacios de direcciones virtuales. Las principales aparecen en la figura 8-8. Ninguna de ellas es una verdadera llamada al sistema; todas escriben mensajes en el puerto del proceso que hizo la llamada.

La primera llamada, *allocate*, hace utilizable una región del espacio de direcciones virtuales. Un proceso puede heredar un espacio de direcciones virtuales ya asignado y puede asignar más, pero cualquier intento por hacer referencia a una dirección no asignada fracasará. La segunda llamada, *deallocate*, invalida una región (es decir, la elimina del mapa de memoria) lo que permite asignarla de nuevo o asociarle algo mediante la llamada *map*.

La llamada *copy* copia un objeto de memoria en una región nueva. El original permanece sin cambios. De esta forma, un objeto de memoria puede aparecer varias veces en el espacio de direcciones. Desde el punto de vista conceptual, la llamada a *copy* no es distinta del

copiado del objeto mediante un ciclo programado. Sin embargo, *copy* se implanta de manera optimizada, mediante páginas compartidas, para evitar el copiado físico.

La llamada *inherit* afecta la forma en que las regiones se heredan durante la creación de nuevos procesos. El espacio de direcciones se puede configurar de modo que se hereden algunas regiones y otras no. Analizaremos esto en la siguiente sección.

Las llamadas *read* y *write* permiten que un hilo tenga acceso a la memoria virtual perteneciente a otro proceso. Estas llamadas necesitan que el proceso que hizo la llamada posea el puerto del proceso correspondiente al proceso remoto, algo que los procesos pueden transferir a sus amigos si así lo desean.

Además de las llamadas de la figura 8-8, existen otras más. Estas llamadas se ocupan en principio de la recuperación y establecimiento de atributos, de los modos de protección y varios tipos de información estadística.

8.3.2. Memoria compartida

El hecho de compartir es muy importante en Mach. No se necesita un mecanismo especial para que un proceso comparta objetos: todos ven el mismo espacio de direcciones de manera automática. Si uno de ellos tiene acceso a una parte de los datos, todos lo harán. Es más interesante la posibilidad de que dos o más procesos compartan los mismos objetos de memoria o bien que compartan páginas de datos. A veces, el hecho de compartir es importante en los sistemas con un CPU. Por ejemplo, en el problema clásico de los productores y los consumidores, podría ser recomendable que el productor y el consumidor sean distintos procesos y que incluso compartan un buffer común, de modo que el productor coloque datos en él y el consumidor pueda retirar datos de él.

En los sistemas multiprocesadores, el hecho de compartir los objetos entre dos o más procesos es con frecuencia más importante. En muchos casos, un problema es resuelto por una colección de procesos cooperativos que se ejecutan en paralelo en varios CPU (en contraposición al tiempo compartido en un CPU). Estos procesos necesitan tener acceso a buffers, tablas u otras estructuras de datos de manera continua, para hacer su trabajo. Es esencial que el sistema operativo comparta esta información. Por ejemplo, las primeras versiones de UNIX no tenían esta capacidad, aunque se le añadió posteriormente.

Por ejemplo, consideremos un sistema que analiza imágenes digitalizadas de satélite de la tierra en tiempo real, conforme se transmiten a su base. Tal análisis consume tiempo y la misma imagen se analiza para pronosticar el tiempo, la predicción del tiempo de cosecha y el seguimiento de la contaminación. Al recibir cada imagen, se guarda como archivo.

Se dispone de un multiprocesador para hacer el análisis. Puesto que los programas meteorológico, agrícola y ambiental son todos distintos y fueron escritos por personas diferentes, no es razonable que todos sean hilos del mismo proceso. En vez de esto, cada uno es un proceso independiente y cada uno asocia la fotografía activa en su espacio de direcciones, como se muestra en la figura 8-9. Observe que el archivo que contiene la fotografía puede ser asociado con una dirección virtual diferente en cada proceso. Aunque cada página está presente sólo una vez en la memoria, puede aparecer en el mapa de páginas

de cada proceso en un lugar distinto. De esta forma, los tres procesos pueden trabajar en el mismo archivo al mismo tiempo de manera conveniente.

Otro de los usos importantes del hecho de compartir es la creación de procesos. Como en UNIX, la forma básica para crear un proceso en Mach es como una copia de un proceso ya existente. En UNIX, una copia es siempre un clon del proceso que ejecuta la llamada al sistema FORK, mientras que en Mach el hijo puede ser un clon de un proceso distinto (el prototipo). De cualquier modo, el hijo es una copia de algún otro proceso.

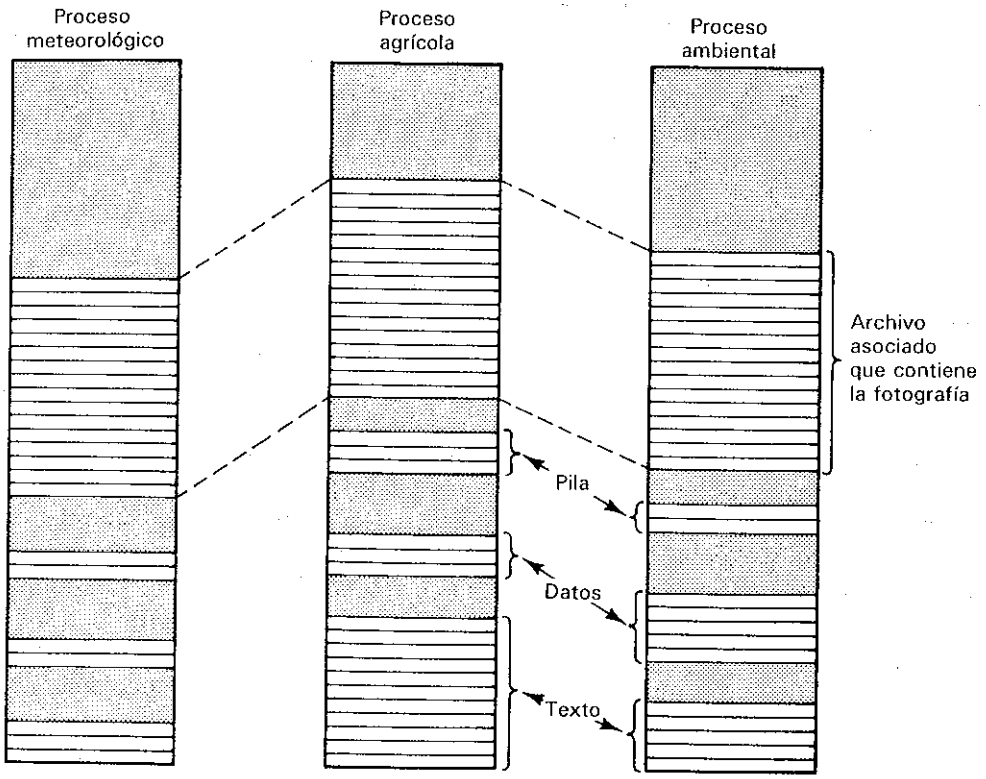


Figura 8-9. Tres procesos que comparten un archivo asociado.

Una vía para crear el hijo consiste en copiar todas las páginas necesarias y asociar las copias con el espacio de direcciones del hijo. Aunque este método es válido, es innecesariamente caro. Por lo general, el texto del programa es exclusivo para lectura, por lo que no se modifican y también algunas partes de los datos podrían ser exclusivas para lectura. No existe razón para copiar páginas exclusivas para lectura, puesto que su asociación a ambos procesos cumplen con los requisitos del trabajo. Las páginas donde se puede escribir no siempre se comparten, puesto que la semántica de la creación de procesos (al menos en UNIX) dice que en el momento de la creación el hijo y el padre son idénticos, pero los cambios posteriores no son visibles en el espacio de direcciones del otro.

Además, ciertas regiones (por ejemplo, ciertos archivos asociados) podrían no ser necesarias en el hijo. ¿Por qué tener que arreglar las cosas de modo que estén presentes en el hijo si no son necesarias?

Para lograr estos objetivos, Mach permite que los procesos asignen un **atributo de herencia** a cada región en su espacio de direcciones. Distintas regiones pueden tener distintos atributos. Se dispone de tres valores:

1. La región no es utilizada en el proceso hijo.
2. La región es compartida entre el proceso prototipo y el hijo.
3. La región en el proceso hijo es una copia del prototipo.

Si una región tiene el primer valor, la región correspondiente del hijo no se asigna. Las referencias a él se consideran como referencias a cualquier otra memoria no asignada (generan señalamientos). El hijo es libre de asignar la región para sus propios fines, o bien asociar ahí un objeto de memoria.

La segunda opción es el hecho verdadero de compartir. Las páginas de la región están presentes en el espacio de direcciones del prototipo y en el espacio de direcciones del hijo. Los cambios hechos por cualquiera de ellos son visibles al otro. Esta opción no se utiliza para la implantación de la llamada al sistema FORK de UNIX, pero se utiliza con frecuencia para otros fines.

La tercera posibilidad consiste en copiar todas las páginas en la región y asociarlas al espacio de direcciones del hijo. FORK utiliza esta opción. En los hechos, Mach no copia en realidad sino que utiliza un astuto truco llamado **copiado durante la escritura**. Coloca todas las páginas necesarias en el mapa de la memoria virtual del hijo, pero las señala como exclusivas para la lectura, como se muestra en la figura 8-10. Mientras el hijo sólo lea referencias a estas páginas, todo funciona bien.

Sin embargo, si el hijo intenta escribir en cualquiera de las páginas, ocurre un fallo de protección. El sistema operativo crea entonces una copia de la página y asocia ésta en el espacio de direcciones del hijo, para reemplazar la página exclusiva para lectura que estaba ahí. La nueva página se señala como accesible para la lectura-escritura. En la figura 8-10(b), el hijo ha intentado escribir en la página 7. Esta acción provoca que la página 7 se copie a una página 8 y que la página 8 se asocie al espacio de direcciones en lugar de la página 7. La página 8 se señala como accesible para la lectura-escritura, de modo que las posteriores escrituras no provoquen un señalamiento.

El proceso de copiado durante la escritura tiene varias ventajas sobre el método de realizar la copia al mismo tiempo en que se crea el nuevo proceso. En primer lugar, algunas de las páginas son exclusivas para lectura, de modo que no existe necesidad de copiarlas. En segundo lugar, es posible que no se haga referencia a otras páginas, de modo que aunque se pudiese escribir en ellas, no tienen que copiarse. En tercer lugar, otras páginas podrían estar listas para la escritura, pero el hijo las podría liberar en vez de utilizarlas. También en este caso es importante evitar la copia. De esta forma, sólo hay que copiar aquellas páginas en las que el hijo en realidad escriba.

El copiado durante la escritura también tiene ciertas desventajas. La administración es más compleja, puesto que el sistema lleva un registro del hecho de que ciertas páginas son genuinamente exclusivas para lectura, de modo que una escritura sería un error de programación, mientras que otras páginas se deben copiar al escribir en ellas. En segundo lugar, el copiado durante la escritura requiere de varios señalamientos al núcleo, uno por cada página que se escriba en último término. Según el hardware, un señalamiento al núcleo seguido de una copia de varias páginas podría no ser tan caro como varios señalamientos al núcleo, cada uno de ellos seguido de un copiado de una página. Por último, el copiado durante la escritura no trabaja bien en una red, puesto que siempre se requiere del transporte físico.

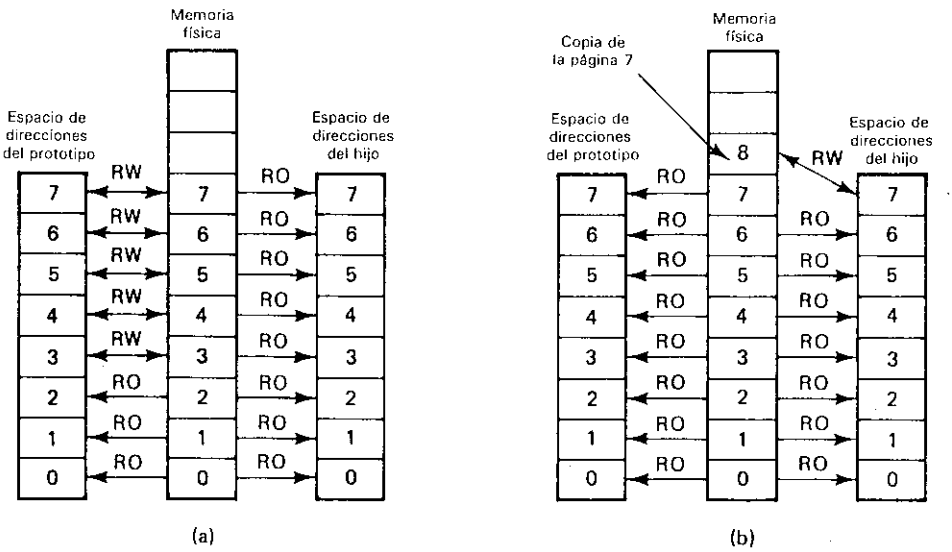


Figura 8-10. Operación del copiado durante la escritura. (a) Después del FORK, todas las páginas del hijo se marcan como exclusivas para lectura (RO; RW indica que se permite la lectura-escritura). (b) Cuando el hijo escribe la página 7, se crea una copia.

8.3.3. Administradores externos de la memoria

Al principio de nuestro análisis de la administración de la memoria en Mach, mencionamos de manera breve la existencia de administradores de memoria a nivel usuario. Ahora los analizaremos con más detalle. Cada objeto de memoria asociado a un espacio de direcciones de un proceso tiene un administrador de memoria externo que lo controle. Las diversas clases de objetos de la memoria se controlan mediante distintos administradores de memoria. Cada uno de estos puede implantar su propia semántica, determinar la posición donde almacenar las páginas mientras no se encuentren en la memoria y proporcionar sus propias reglas acerca de lo que ocurra con los objetos después de liberarlos.

Para asociar un objeto a un espacio de direcciones de un proceso, éste envía un mensaje a un administrador de memoria para pedirle que realice la asociación. Se necesitan tres

puertos para hacer el trabajo. El primero, el **puerto del objeto**, es creado por el administrador de memoria y será utilizado posteriormente por el núcleo para informarle al administrador acerca de los fallos de página y otros eventos relacionados con el objeto. El segundo, el **puerto de control**, es creado por el propio núcleo de modo que el administrador responda a estos eventos (muchos de ellos requerirán de una acción por parte del administrador de la memoria). El uso de distintos puertos obedece al hecho de que los puertos son unidireccionales. El puerto del objeto es escrito por el núcleo y leído por el administrador de memoria; el puerto de control trabaja en la otra dirección. El tercer puerto, el **puerto del nombre**, se utiliza como un tipo de nombre para identificar al objeto. Por ejemplo, un hilo puede darle al núcleo una dirección virtual y preguntar por la región a la que pertenece. La respuesta es un apuntador al puerto del nombre. Si las direcciones pertenecen a la misma región, serán identificadas mediante el mismo puerto del nombre.

Cuando el administrador de memoria asocia un objeto, proporciona la posibilidad correspondiente al puerto del objeto como uno de sus parámetros. El núcleo crea entonces los otros dos puertos y envía un mensaje inicial al puerto del objeto para informarle de los puertos de control y de nombre. El administrador de memoria envía entonces una respuesta para indicarle al núcleo los atributos del objeto y también para informarle de si debe o no mantener el objeto en su caché después de liberarlo. En principio, todas las páginas del objeto se señalan de modo que no se puedan leer ni escribir, para obligar a un señalamiento en el primer uso.

En este momento, el administrador de memoria hace una lectura en el puerto del objeto y se bloquea. El administrador permanece inactivo hasta que el núcleo le solicita una acción mediante la escritura de un mensaje en el puerto del objeto. El hilo que asoció el objeto elimina su bloqueo y puede ejecutarse.

Tarde o temprano, el hilo intentará leer o escribir una página perteneciente al objeto de memoria. Esta operación provoca un fallo de página y un señalamiento al núcleo, éste enviará entonces un mensaje al administrador de memoria por medio del puerto del objeto, para indicar la página a la que se hace referencia y solicitar que se le proporcione. Este mensaje es asíncrono, puesto que el núcleo no intenta bloquear alguno de sus hilos en espera de un proceso usuario que podría no contestar. Mientras espera una respuesta, el núcleo suspende al hilo fallido y busca otro hilo para ejecutarlo.

Cuando el administrador de memoria se entera del fallo de página, verifica si la referencia es válida. En caso contrario, envía un mensaje de error al núcleo. Si es válida, el administrador obtiene la página mediante un método adecuado para el objeto en cuestión. Si el objeto es un archivo, el administrador de memoria busca la dirección correcta y lee la página en su propio espacio de direcciones. Después envía una respuesta de regreso al núcleo mediante un apuntador a la página. El núcleo asocia la página con el espacio de direcciones del hilo fallido. Entonces se puede eliminar el bloqueo del hilo. Este proceso se repite el número de veces necesarias para cargar las páginas requeridas.

Para garantizar que existe una oferta continua de marcos de página libres, un hilo demonio de paginación en el núcleo despierta de manera periódica y verifica el estado de la memoria. Si no existe un número suficiente de marcos libres, elige una página para

liberarla mediante el algoritmo de la segunda posibilidad. Si la página está limpia, por lo general se descarta. Si la página está sucia, el demonio la envía al administrador de memoria encargado del objeto de la página. Se espera que el administrador de memoria escriba la página en el disco e indique cuando termine de hacer esto. Si la página pertenece a un archivo, el administrador de memoria busca primero el ajuste de la página en el archivo, después lo escribe en la posición correspondiente.

Las páginas se pueden señalar como **preciosas**, en cuyo caso nunca se descartan, aunque estén limpias. Siempre serán regresadas a su administrador de memoria. Las páginas preciosas se pueden utilizar, por ejemplo, para las páginas compartidas a través de la red para las que sólo exista una copia que nunca debe ser descartada. La comunicación también puede ser iniciada por el administrador de memoria; por ejemplo, cuando se realice una llamada al sistema SYNC para dirigir el caché de regreso al disco.

Es importante observar que el demonio de paginación es parte del núcleo. Aunque el algoritmo para el reemplazo de páginas es por completo independiente de la máquina, si se tiene una memoria por completo ocupada por páginas poseídas por distintos administradores de memoria, no existe una forma adecuada para que uno de ellos decida la página a eliminar. El único método posible es repartir de manera estadística los marcos para página entre los distintos administradores, y dejar que cada uno de ellos realice su reemplazo de páginas sobre su conjunto. Sin embargo, como los algoritmos globales son con frecuencia más eficientes que los locales, no se utiliza este punto de vista. Trabajos posteriores han investigado este tema (Harty y Cheriton, 1992; y Subramian, 1991).

Además de los administradores de memoria para los archivos asociados y otros objetos especializados, también existe un administrador de memoria predefinido para la memoria paginada "ordinaria". Cuando un proceso asigna una región del espacio de direcciones virtuales mediante la llamada *allocate*, de hecho asocia un objeto controlado por el administrador por omisión. Este administrador proporciona el número necesario de páginas, las cuales se rellenan con ceros. Utiliza un archivo temporal para el intercambio de espacio, en vez de un área independiente de intercambio, como en el caso de UNIX.

Para tener una idea del funcionamiento del administrador de memoria externo, se utiliza un protocolo estricto para la comunicación entre núcleo y administradores de memoria. Este protocolo consta de un pequeño número de mensajes que el núcleo puede enviar a un administrador de memoria y un pequeño número de respuestas que el administrador de memoria envía de regreso al núcleo. Toda comunicación es iniciada por el núcleo en forma de un mensaje asíncrono en un puerto de objeto para cierto objeto de la memoria. Posteriormente, el administrador de memoria envía una respuesta asíncrona en el puerto de control.

La figura 8-11 enumera los principales tipos de mensajes que el núcleo puede enviar a los administradores de memoria. Cuando un objeto se asocia mediante la llamada *map* de la figura 8-8, el núcleo envía un mensaje *init* al administrador de memoria adecuado para que éste se inicie a sí mismo. El mensaje especifica los puertos a utilizar para el análisis posterior del objeto. Las solicitudes de páginas por parte del núcleo y la entrega de una página utilizan *data_request* y *data_write*, respectivamente. Éstas controlan el tráfico de las páginas en ambas direcciones y como tales son las llamadas más importantes.

Data_unlock es una solicitud del núcleo al administrador de memoria para que abra una página cerrada de modo que el núcleo la pueda utilizar para otro proceso. *Lock_completed* señala el fin de una secuencia *lock_request* que se describe más adelante. Por último, *terminate* indica al administrador de la memoria que el objeto mencionado en el mensaje ya no se utiliza y que por tanto se puede eliminar de la memoria. Existen algunas llamadas específicas del administrador de memoria por omisión, así como unas cuantas que controlan los atributos y los errores.

Llamada	Descripción
Init	Inicia un nuevo objeto de memoria recién asociado
Data_request	Da al núcleo una página determinada para el manejo de una falla de página
Data_write	Toma una página de la memoria y la escribe fuera de ella
Data_unlock	Abre una página de modo que el núcleo la pueda utilizar
Lock_completed	El anterior Lock_request ha concluido
Terminate	Se informa que dicho objeto ya no está en uso

Figura 8-11. Selección de algunos de los mensajes del núcleo a los administradores externos de la memoria.

Los mensajes de la figura 8-11 van del núcleo a el administrador de la memoria. Las respuestas que se muestran en la figura 8-12 van en sentido contrario, del administrador de memoria al núcleo. Son respuestas que el administrador de memoria puede utilizar para responder a las anteriores solicitudes.

Llamada	Descripción
Set_attributes	Respuesta a init
Data_provided	Aquí está la página solicitada (respuesta a Data_request)
Data_unavailable	No está disponible página alguna (respuesta a Data_request)
Lock_request	Solicita al núcleo que limpie, elimine o cierre páginas
Destroy	Destruye un objeto que ya no es necesario

Figura 8-12. Selección de tipos de mensajes de los administradores externos de la memoria al núcleo.

La primera, *set_attributes*, es una respuesta a *init*. Indica al núcleo que está listo para controlar un objeto recién asociado. La respuesta también proporciona bits de modo para el objeto e indica al núcleo si debe o no almacenar el objeto en el caché, aunque no tenga

asociado proceso alguno. Las siguientes dos son respuestas a *data_request*. Esa llamada pide al administrador de memoria que proporcione una página. La respuesta depende del hecho de que pueda o no proporcionarla. La primera proporciona la página y la segunda no.

Lock_request permite al administrador de memoria pedirle al núcleo que limpie algunas páginas; es decir, que envíe las páginas de modo que se puedan escribir al disco. Esta llamada se utiliza también para modificar el modo de protección de las páginas (leer, escribir, ejecutar). Por último, *destroy* se utiliza para indicar al núcleo que cierto objeto ya no es necesario.

Es importante observar que cuando el núcleo envía un mensaje a un administrador de memoria, en realidad hace una llamada. Aunque se logra flexibilidad de esta manera, algunos diseñadores de sistemas consideran poco elegante que el núcleo llame a programas del usuario para que lleven a cabo servicios para él. Estas personas creen por lo general en un sistema jerárquico, donde las capas inferiores proporcionan servicios a las capas superiores y no viceversa.

8.3.4. Memoria compartida distribuida en Mach

El concepto de administrador de memoria externo de Mach conduce a una implantación de una memoria compartida distribuida basada en el uso de páginas. En esta sección describiremos en forma breve algo del trabajo realizado en esta área. Para más detalles, véase (Forin *et al.*, 1989). Para revisar el concepto fundamental, la idea es tener un espacio de direcciones virtuales, lineal, compartido por todos los procesos que se ejecuten en computadoras que no tienen memoria física compartida. Cuando un hilo hace referencia a una página que no tiene, provoca un fallo de página. En cierto momento se localiza dicha página y se envía a la máquina fallida, donde se instala de forma que el hilo pueda continuar su ejecución.

Puesto que Mach ya dispone de administradores de memoria para distintas clases de objetos, es natural introducir un nuevo objeto en la memoria, la página compartida. Las páginas compartidas se controlan mediante uno o más administradores de memoria especiales. Una posibilidad consiste en tener un administrador de memoria que controle a todas las páginas compartidas. Otra es tener un administrador diferente para cada página compartida o para una colección de páginas compartidas, para repartir la carga.

Otra posibilidad más es tener distintos administradores de memoria para páginas con diferente semántica. Por ejemplo, un administrador de memoria podría garantizar la coherencia total de la memoria, lo que significa que un *read* después de un *write* siempre verá los datos más recientes. Otro administrador de memoria podría ofrecer una semántica más débil, como por ejemplo, que un *read* nunca regrese datos que no hayan sido actualizados durante los últimos 30 segundos.

Consideremos el caso más básico: una página compartida, un control centralizado y la total coherencia de la memoria. Todas las demás páginas son locales en una máquina. Para implantar este modelo, necesitamos un administrador de memoria que dé servicio a todas las máquinas en el sistema. Lo llamaremos el servidor DSM (siglas en inglés de memoria

compartida distribuida). El servidor DSM controla las referencias a la página compartida. Los administradores de memoria convencionales controlan las demás páginas. Hasta ahora hemos supuesto de manera implícita que el administrador o administradores de la memoria que dan servicio a una máquina son locales con respecto a dicha máquina. De hecho, puesto que la comunicación es transparente en el caso de Mach, un administrador de memoria no tiene que residir en la máquina cuya memoria administra.

Siempre se puede leer o escribir en la página compartida. Si se puede leer, se puede duplicar en varias máquinas. Si se puede escribir en ella, sólo existe una copia. El servidor DSM siempre conoce el estado de la página compartida, al igual que la máquina o máquinas activas. Si se puede leer en la página, DSM tiene su copia válida.

Supongamos que se puede leer la página y que cierto hilo intenta leerla. El servidor DSM simplemente envía una copia a tal máquina, actualiza sus tablas para indicar un lector más y termina. La página será asociada a la nueva máquina para su lectura.

Ahora supongamos que uno de los lectores intenta escribir en la página. El servidor DSM envía un mensaje al núcleo o núcleos que tienen la página para solicitarla. No es necesario transferirla, puesto que el servidor DSM tiene su copia válida. Lo único que necesita es un reconocimiento de que la página no se utiliza. Cuando todos los núcleos hayan liberado la página, se da una copia al escritor junto con un permiso exclusivo para su uso (con fines de escritura).

Si alguien más desea entonces la página (cuando se puede escribir en ella), el servidor DSM indica al propietario actual que se detenga y la envíe de regreso. Al llegar la página, se le puede dar a uno o más lectores o a un escritor. Se tienen muchas variantes de este algoritmo centralizado, como el hecho de no pedir la página de regreso sino hasta que la máquina que la esté utilizando la tenga durante un período mínimo. También es posible una solución distribuida.

8.4. COMUNICACIÓN EN MACH

El objetivo de la comunicación en Mach es soportar una gama de estilos de comunicación de manera confiable y flexible (Draves, 1990). Controla la transferencia asíncrona de mensajes, RPC, flujos de bytes y otras formas. El mecanismo de comunicación entre procesos de Mach se basa en el de sus antecesores, RIG y Accent. Debido a esta evolución, el mecanismo utilizado ha sido optimizado para el caso local (un nodo) en vez del caso remoto (sistema distribuido).

En primer lugar explicaremos el caso de un nodo con gran detalle, para después regresar a la forma en que esto se amplió al caso de las redes. Debe observarse que en estos términos, un multiprocesador es un nodo, por lo que la comunicación entre procesos en distintos CPU dentro del mismo multiprocesador utiliza el caso local.

8.4.1. Puertos

La base de toda la comunicación en Mach es una estructura de datos en el núcleo llamada **puerto**. Un puerto es en esencia un buzón protegido. Cuando un hilo de un proceso desea

comunicarse con un hilo de otro procesos, el hilo emisor escribe el mensaje al puerto y el hilo receptor lo obtiene de él. Cada puerto está protegido para garantizar que sólo los procesos autorizados puedan enviar y recibir de él.

Los puertos soportan la comunicación unidireccional, como los entubamientos en UNIX. Un puerto que se puede utilizar para enviar una solicitud de un cliente a un servidor no se puede utilizar para enviar la respuesta del servidor al cliente. Se necesita un segundo puerto para la respuesta.

Los puertos soportan los flujos de mensajes confiables y secuenciales. Si un hilo envía un mensaje a un puerto, el sistema garantiza su entrega. Los mensajes nunca se pierden debido a errores, desbordamiento u otras causas (al menos si no existen fallas generales del sistema). También se garantiza que los mensajes enviados por un hilo llegan a su destino en el orden en que fueron enviados. Si dos hilos escriben en el mismo puerto de manera intercalada, el sistema no garantiza de modo alguno cierta secuencia de los mensajes, puesto que se puede llevar a cabo cierto almacenamiento en un buffer debido a cerraduras y otros factores.

A diferencia de los entubamientos, los puertos soportan los flujos de mensajes y no los flujos de bytes. Los mensajes no se concatenan. Si un hilo escribe cinco mensajes de 100 bytes en un puerto, el receptor siempre los verá como cinco mensajes distintos y nunca como un mensaje de 500 bytes. Por supuesto, un software de nivel más alto puede ignorar las fronteras de los mensajes si no son importantes para él.

En la figura 8-13 se muestra un puerto. Cuando se crea un puerto, se le asignan 64 bytes de espacio de almacenamiento en el núcleo, los cuales se mantienen hasta la destrucción del puerto, ya sea en forma explícita o implícita bajo ciertas condiciones; por ejemplo, cuando todos los procesos que lo utilizan han terminado su trabajo. El puerto contiene los campos que se muestran en la figura 8-13 y unos cuantos más.

Los mensajes no se almacenan en realidad dentro del propio puerto, sino en otra estructura de datos del núcleo, la **cola de mensajes**. El puerto contiene un contador del número de mensajes presentes en ese momento en la cola, así como el máximo permitido. Si el puerto pertenece a un conjunto de puertos, en él está presente un apuntador a la estructura de datos del conjunto de puertos. Como mencionamos antes de manera breve, un proceso puede otorgar a otros la posibilidad de utilizar sus puertos. Por varias razones, el núcleo debe conocer el número y tipo de estas posibilidades presentes en cada momento, de modo que el puerto almacena los contadores.

Si ocurren errores durante el uso del puerto, se informa de éstos mediante el envío de mensajes a otros puertos cuyas posibilidades se almacenen ahí. Los hilos se pueden bloquear durante la lectura de un puerto, por lo que se incluye un apuntador a la lista de hilos bloqueados. También es importante encontrar la posibilidad de lectura del puerto (sólo existe una), de modo que también está presente esa información. Si el puerto es de proceso, el siguiente campo contiene un apuntador al proceso al cual pertenece. Si es un puerto de hilo, el campo contiene un apuntador a la estructura de datos del núcleo correspondiente al hilo, etc. También se necesitan unos cuantos campos más que no describiremos aquí.

Cuando un hilo crea un puerto, obtiene un entero que identifica a éste, similar a un descriptor de archivo en UNIX. Este entero se utiliza en las llamadas posteriores que envían

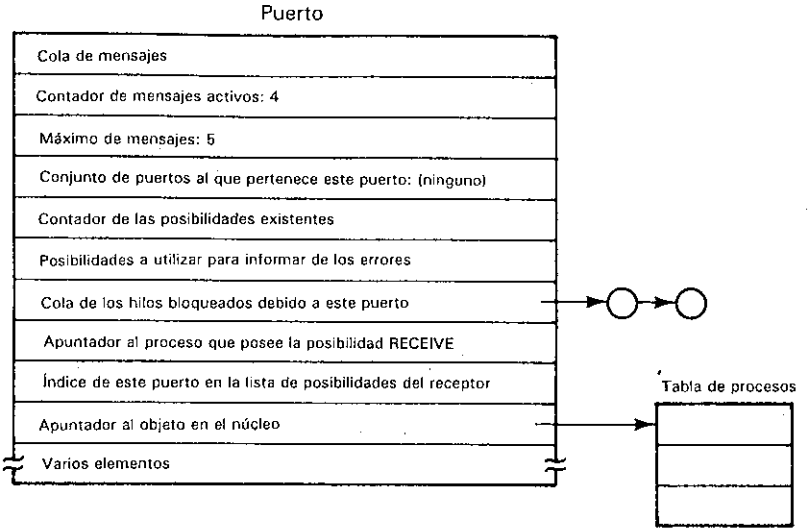


Figura 8-13. Un puerto en Mach.

mensajes al puerto o reciben mensajes de él, con el fin de identificar el puerto por utilizar. Se lleva un registro de los puertos por cada proceso, no por cada hilo, por lo que si un hilo crea un puerto y obtiene de regreso el entero 3 para su identificación, otro hilo del mismo proceso no obtendrá 3 para la identificación de un nuevo puerto. De hecho, el núcleo ni siquiera mantiene un registro del hilo que crea un puerto dado.

Un hilo puede transferir el acceso a un puerto a otro hilo de un proceso diferente. Es claro que no puede hacer esto simplemente al colocar el entero adecuado en un mensaje, del mismo modo que un proceso en UNIX no transfiere un descriptor de archivo para la salida estándar a través de un entubamiento, mediante la escritura del entero 1 en éste. El mecanismo exacto que se utiliza está protegido por el núcleo y será analizado en una sección posterior. Por el momento, basta saber que se puede llevar a cabo.

En la figura 8-14 vemos una situación en la que dos procesos, *A* y *B*, tienen acceso al mismo puerto. El proceso *A* envía un mensaje al puerto y *B* lee dicho mensaje. El encabezado y cuerpo del mensaje se copian de manera física de *A* al puerto y más adelante, del puerto a *B*.

Los puertos se pueden agrupar en **conjuntos de puertos** si así se desea. Un puerto puede pertenecer a lo más a un conjunto de puertos. Es posible leer de un conjunto de puertos (pero no escribir en uno de ellos). Por ejemplo, un servidor utiliza este mecanismo para leer gran número de puertos al mismo tiempo. El núcleo regresa un mensaje de uno de los puertos en el conjunto. No se puede comprometer a elegir alguno de los puertos. Si todos están vacíos, el servidor se bloquea. De esta forma, un servidor mantiene un puerto distinto para cada uno de los muchos objetos que soporta y obtiene mensajes de cualquiera de ellos sin tener que dedicar un hilo a cada uno. La implantación correspondiente forma a todos los mensajes del conjunto de puertos en una cadena. En la práctica, la única diferencia

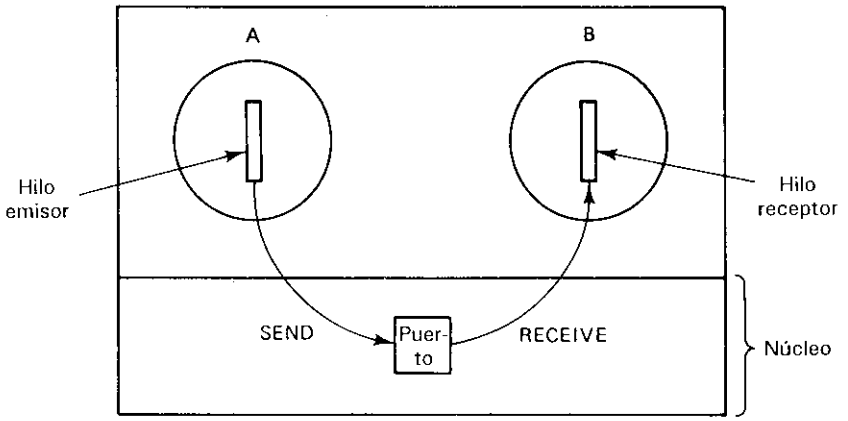


Figura 8-14. La transferencia de mensajes se realiza a través de un puerto.

entre la recepción de un puerto y la recepción de un conjunto de puertos es que en esta última, el puerto real de envío es identificado por el receptor y en el primer caso no.

Algunos puertos se utilizan de manera especial. Todo proceso tiene un **puerto de proceso** necesario para la comunicación con el núcleo. La mayoría de las "llamadas al sistema" asociadas con los procesos (véase la figura 8-3) se realizan mediante la escritura de mensajes en este puerto. De manera análoga, cada hilo tiene su puerto para llevar a cabo las "llamadas al sistema" relacionadas con los hilos. La comunicación con los controladores de E/S también utiliza el mecanismo de los puertos.

Posibilidades

En una primera aproximación, el núcleo mantiene para cada proceso una tabla de todos los puertos a los cuales tiene acceso dicho proceso. Esta tabla se mantiene segura dentro del núcleo, donde los procesos usuario no puedan alcanzarla. Los procesos se refieren a los puertos mediante su posición en esta tabla; es decir, la entrada 1, la entrada 2, etc. Estas entradas de la tabla son posibilidades clásicas. Nos referiremos a ellas como posibilidades y a la lista que las contiene la llamaremos **lista de posibilidades**.

Cada proceso tiene con exactitud una lista de posibilidades. Cuando un hilo pide al núcleo crear un puerto para él, el núcleo lo hace e introduce una posibilidad para el hilo en la lista de posibilidades del proceso al que pertenece el hilo. El hilo que hace la llamada y todos los demás hilos del mismo proceso tienen igual acceso a la posibilidad. El entero que regresa al hilo como identificación de la posibilidad es por lo general un índice en la lista de posibilidades (pero también puede ser un entero grande, como una dirección de la máquina). Llamaremos a este entero **nombre de la posibilidad** (o a veces simplemente posibilidad, donde el contexto indicará que hablamos del índice y no de la propia posibilidad). Siempre es un entero de 32 bits y nunca es una cadena.

Cada posibilidad no sólo consta de un apuntador a un puerto, sino también de un campo de derechos que indica los accesos posibles al puerto por parte del propietario de la posibilidad. (Todos los hilos de un proceso se consideran propietarios por igual de las posibilidades del proceso.) Existen tres derechos: RECEIVE, SEND y SEND-ONCE. El derecho RECEIVE permite al propietario leer mensajes del puerto. Ya hemos mencionado que la comunicación en Mach es unidireccional. Esto significa en realidad que en cualquier instante, sólo un proceso puede tener el derecho RECEIVE de un puerto. Se puede transferir una posibilidad con un derecho RECEIVE a otro proceso, pero esto hace que sea eliminado de la lista de posibilidades del emisor. Así, para cada puerto sólo existe un receptor potencial.

Una posibilidad con el derecho SEND permite al propietario enviar mensajes al puerto dado. Muchos procesos poseen la posibilidad de realizar envíos a un puerto. Esta situación es algo similar al sistema bancario de la mayoría de los países: cualquier persona que conozca un número de cuenta bancario puede depositar dinero en esa cuenta, pero sólo el propietario puede hacer retiros.

El derecho SEND-ONCE también permite el envío de un mensaje, pero solamente una vez. Después de hacer el envío, el núcleo destruye la posibilidad. Este mecanismo se utiliza para los protocolos solicitud-respuesta. Por ejemplo, un cliente desea algo de un servidor, por lo que crea un puerto para el mensaje de respuesta. Entonces envía al servidor un mensaje de solicitud, el cual contiene una posibilidad (protegida) para el puerto de respuesta, con el derecho SEND-ONCE. Después de que el servidor envía la respuesta, se libera la lista de posibilidades y el nombre queda disponible para una nueva posibilidad en el futuro.

Los nombres de las posibilidades tienen significado dentro de un proceso. Es posible que dos procesos tengan acceso al mismo puerto, pero que utilicen distintos nombres para él, de la misma forma que dos procesos UNIX pueden tener acceso al mismo archivo abierto, pero que utilicen distintos descriptores de archivo para leerlo. En la figura 8-15, ambos procesos tienen una posibilidad para enviar hacia el puerto *Y*, pero en *A* es la posibilidad 3 y en *B* es la posibilidad 4.

Una lista de posibilidades está ligada a un proceso específico. Cuando ese proceso termina o es eliminado, también se elimina su lista de posibilidades. Los puertos para los cuales posee una posibilidad con el derecho RECEIVE ya no se pueden utilizar y por lo tanto se destruyen, aunque contengan mensajes no entregados (y que ya no podrán entregarse).

Si los distintos hilos de un proceso adquieren la misma posibilidad varias veces, sólo se crea una entrada en la lista de posibilidades. Para llevar un registro del número de veces que cada uno está presente, el núcleo mantiene un contador de referencia por cada puerto. Cuando se elimina una posibilidad, se decrementa el contador de referencia. Sólo cuando este contador llega a 0 se elimina la posibilidad de la lista. Este mecanismo es importante, debido a que distintos hilos pueden adquirir o liberar posibilidades sin el conocimiento de los demás; por ejemplo, la biblioteca de emulación de UNIX y el programa en ejecución.

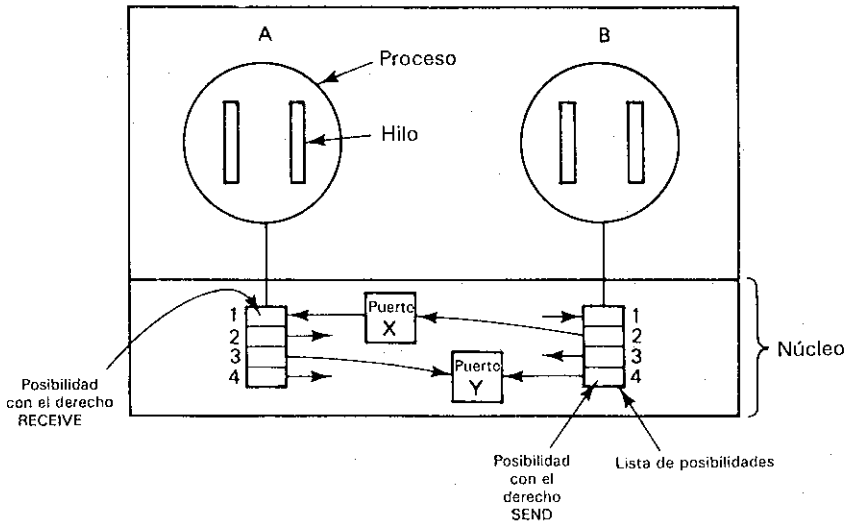


Figura 8-15. Lista de posibilidades.

Cada entrada de la lista de posibilidades es uno de los cuatro elementos siguientes:

1. Una posibilidad para un puerto.
2. Una posibilidad para un conjunto de puertos.
3. Una entrada nula.
4. Un código que indica que el puerto que se encontraba en ese lugar está muerto ahora.

La primera posibilidad ha sido explicada en detalle. La segunda permite que un hilo lea de un conjunto de puertos sin que tenga conciencia de que el nombre de la posibilidad está respaldado por un conjunto y no por un puerto. La tercera contiene una posición para indicar que la entrada correspondiente no está en uso. Si se asigna una entrada a un puerto que se destruye más adelante, la posibilidad se reemplaza mediante una entrada nula para señalarla como no utilizada.

Por último, la cuarta opción indica los puertos que ya no existen pero para los cuales existen posibilidades con derechos SEND. Por ejemplo, cuando se elimina un puerto debido a que termina el proceso propietario de la posibilidad RECEIVE correspondiente, el núcleo busca todas las posibilidades SEND y las señala como muertas. Los intentos por realizar envíos a posibilidades nulas y muertas fallan, con un código de error apropiado. Al terminar todas las posibilidades SEND de un puerto, por la razón que sea, el núcleo (de manera opcional) envía un mensaje al receptor para indicar que no existen emisores y que no recibirá ningún mensaje.

Primitivas para la administración de puertos

Mach proporciona cerca de 20 llamadas para la administración de puertos. Todas ellas se llaman mediante el envío de un mensaje a un puerto de proceso. En la figura 8-16 aparece una muestra de las más importantes.

Llamada	Descripción
Allocate	Crea un puerto e inserta su posibilidad en la lista de posibilidades
Destroy	Destruye un puerto y elimina su posibilidad de la lista
Deallocate	Elimina una posibilidad de la lista de posibilidades
Extrac_right	Extrae la n-ésima posibilidad de otro proceso
Insert_right	Inserta una posibilidad en la lista de posibilidades de otro proceso
Move_member	Desplaza una posibilidad a un conjunto de posibilidades
Set_qlimit	Establece el número de mensajes que puede contener un puerto

Figura 8-16. Selección de algunas llamadas para la administración de puertos en Mach.

La primera, *allocate*, crea un puerto nuevo e introduce su posibilidad en la lista de posibilidades del proceso que hace la llamada. La posibilidad es para la lectura del puerto. Se regresa un nombre de posibilidad para poder utilizar el puerto.

Las siguientes dos revierten el trabajo de la primera. *Destroy* elimina una posibilidad. Si es una posibilidad RECEIVE, se destruye el puerto y todas las demás posibilidades correspondientes a él de todos los procesos se señalan como muertas. *Deallocate* decrementa el contador de referencia asociado con una posibilidad. Si es 0, se elimina la posibilidad, pero el puerto permanece intacto. *Deallocate* sólo se puede utilizar para eliminar las posibilidades SEND o SEND-ONCE, o para posibilidades muertas.

Extract_right permite que un hilo seleccione una posibilidad de la lista de posibilidades de otro proceso e insertarla en su propia lista. Por supuesto, el hilo que hace la llamada necesita tener acceso al puerto de proceso que controla al otro proceso (por ejemplo, su propio hijo). *Insert_right* actúa en sentido contrario. Permite que un proceso tome una de sus propias posibilidades y la añada a (por ejemplo) la lista de posibilidades de un hijo.

La llamada *move_member* se utiliza para administrar los conjuntos de puertos. Puede añadir o eliminar un puerto a un conjunto de puertos. Por último, *set_qlimit* determina el número de mensajes que puede contener un puerto. Cuando se crea un puerto, el valor predefinido es de 5 mensajes, pero este número se puede incrementar o disminuir por medio de esta llamada. Los mensajes pueden tener cualquier tamaño, puesto que no se almacenan de forma física en el propio puerto.

8.4.2. Envío y recepción de mensajes

La finalidad de contar con los puertos es enviarles mensajes. En esta sección analizaremos la forma en que se envían, se reciben y su contenido. Mach tiene una llamada al sistema para el envío y recepción de mensajes. La llamada está contenida en un procedimiento de biblioteca llamado *mach_msg*. Tiene siete parámetros y un gran número de opciones. Para dar una idea de su complejidad, puede regresar 35 mensajes de error diferentes. Adelante daremos un bosquejo de algunas de sus posibilidades. Por fortuna, se utiliza principalmente en procedimientos generados por el compilador de resguardo, en vez de ser escritos a mano.

La llamada *mach_msg* se utiliza para el envío y la recepción. Puede enviar un mensaje a un puerto y regresar el control al proceso que hace la llamada de manera inmediata, momento en el cual el proceso que hace la llamada puede modificar el buffer de mensajes sin afectar los datos enviados. También puede intentar recibir un mensaje de un puerto, bloquearse si el puerto está vacío o desistir después de cierto tiempo. Por último, puede combinar estas dos operaciones, al enviar primero un mensaje y después bloquearse hasta recibir de regreso una respuesta. En este último modo, se puede utilizar *mach_msg* para RPC.

Una llamada típica a *mach_msg* sería:

```
mach_msg(&hdr,options,send_size,rcv_size,rcv_port,timeout, notify_port);
```

El primer parámetro, *hdr*, es un apuntador al mensaje por enviar o a la posición donde se colocará el mensaje por recibir, o ambos. El mensaje comienza con un encabezado fijo, al cual le sigue directamente el cuerpo del mensaje. Esta distribución se muestra en la figura 8-17. Más adelante explicaremos los detalles del formato del mensaje, pero por el momento sólo observaremos que el encabezado contiene un nombre de posibilidad para el puerto de destino. Esta información es necesaria para que EL NÚCLEO pueda indicar la posición donde envía el mensaje. Cuando se lleva a cabo un RECEIVE puro, el encabezado no se llena, puesto que el mensaje recibido se escribe encima de él.

El segundo parámetro de la llamada *mach_msg*, *options*, contiene un bit que determina si va a enviar un mensaje y otro para determinar si va a recibirlo. Si ambos se activan, se realiza una RPC. Otro bit activa un tiempo de espera, dado en milisegundos por medio del parámetro *timeout*. Si la operación solicitada no se puede llevar a cabo durante el intervalo del tiempo de espera, la llamada regresa con un código de error. Si la parte SEND de una RPC agota su tiempo (por ejemplo, si el puerto de destino está ocupado durante mucho tiempo), ni siquiera se intenta la parte RECEIVE.

Otros bits de *options* permiten que un SEND que no se puede terminar de manera inmediata regrese el control, enviando un informe del estado a *notify_port*. Aquí pueden ocurrir todos los tipos de errores si la posibilidad de *notify_port* no es adecuada o se modifica antes de que pueda ocurrir la notificación. Incluso es posible que la llamada arruine al

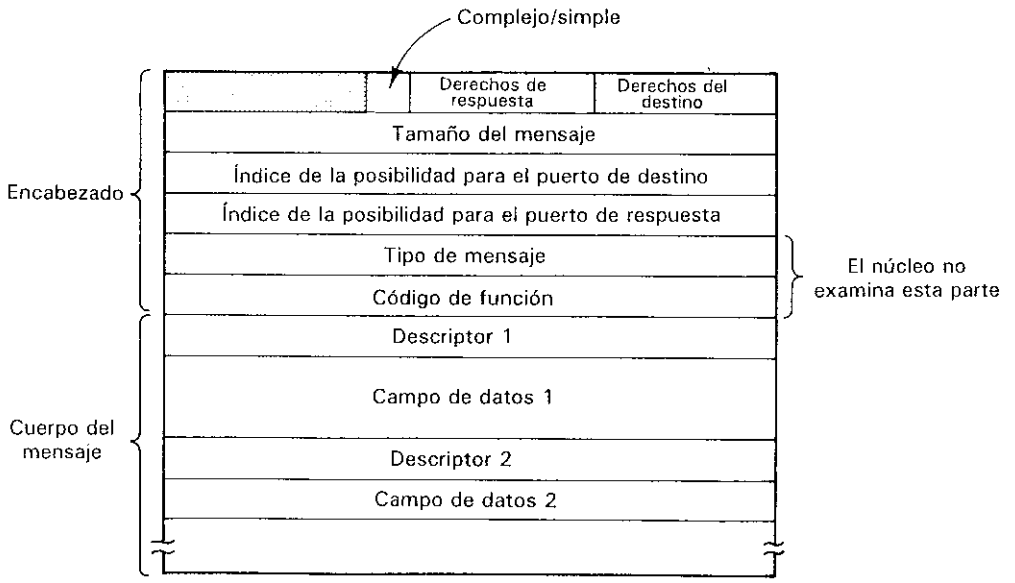


Figura 8-17. El formato de los mensajes en Mach.

propio *notify_port* (las llamadas pueden tener efectos colaterales complejos, como veremos más adelante).

La llamada *mach_msg* se aborta a la mitad del camino mediante una interrupción de software. Otro bit *options* indica si se debe desistir o intentar de nuevo.

Los parámetros *send_size* y *rcv_size* indican el tamaño del mensaje por enviar y el número de bytes disponibles para almacenar el mensaje por recibir, respectivamente. *Rcv_port* se utiliza para recibir mensajes. Es el nombre de la posibilidad del puerto o conjunto de puertos a los cuales se escucha.

Ahora pasemos al formato del mensaje de la figura 8-17. La primera palabra contiene un bit para indicar si el mensaje es simple o complejo. La diferencia es que los mensajes simples no pueden acarrear posibilidades o apuntadores protegidos, mientras que los complejos sí. Los mensajes simples requieren de un trabajo menor por parte del núcleo y por lo tanto son más eficaces. Ambos tipos de mensajes tienen una estructura definida por el sistema, la cual se describe más adelante. El campo *tamaño del mensaje* indica el tamaño combinado del encabezado y el cuerpo. Esta información es necesaria para la transmisión y para el receptor.

A continuación vienen dos nombres de posibilidades (es decir, índices a la lista de posibilidades del emisor). El primero especifica el puerto de destino y el segundo puede proporcionar un puerto de respuesta. Por ejemplo, en una RPC cliente-servidor, el campo de destino designa al servidor y el campo de respuesta indica al servidor el puerto donde debe enviar la respuesta.

Los últimos dos campos del encabezado no son utilizados por el núcleo. Un software de más alto nivel puede utilizarlos de la manera que desee. Por convención, se utilizan para especificar el tipo de mensaje y proporcionar un código de función o de operación, (por ejemplo, para el caso de un servidor, se puede especificar aquí si la solicitud es para una lectura o una escritura). Este uso está sujeto a modificaciones en el futuro.

Cuando un mensaje se envía y recibe con éxito, se copia en el espacio de direcciones del destino. Sin embargo, puede ocurrir que el puerto de destino esté por completo ocupado. Lo que ocurra en este caso depende de las distintas opciones y los derechos asociados al puerto de destino. Puede ocurrir que el emisor esté bloqueado y que simplemente espere hasta disponer de espacio en el puerto. Otra posibilidad es que el emisor agote su tiempo. En ciertos casos, puede exceder el límite del puerto y realizar el envío de cualquier manera.

Es importante mencionar algunos aspectos relativos a la recepción de mensajes. Por ejemplo, si un mensaje recibido es más grande que el buffer, ¿qué debe hacerse con él? Se tienen dos opciones: eliminarlo o hacer que falle la llamada *mach_msg* pero que regrese el tamaño, lo cual permite al proceso que hace la llamada que intente de nuevo con un tamaño más grande.

Si se bloquean varios hilos que intentan leer el mismo puerto y llega un mensaje, el sistema elige uno de ellos para obtener el mensaje. El resto permanece bloqueado. Si el puerto leído es en realidad un conjunto de puertos, es posible que cambie la composición del conjunto si uno o más hilos están bloqueados. Tal vez éste no sea el lugar para exponer todos los detalles, pero baste decir que existen reglas precisas que gobiernan ésta y otras situaciones similares.

Formatos de mensaje

Un cuerpo de mensaje puede ser simple o complejo, controlado por un bit de encabezado, como se mencionó antes. Los mensajes complejos se estructuran como se muestra en la figura 8-17. El cuerpo de un mensaje complejo consta de una serie de parejas (descriptor, campo de datos). Cada descriptor indica lo que contiene el campo de datos que lo sigue. Los descriptores vienen en dos formatos, que sólo difieren en el número de bits que contiene cada campo. El formato normal de un descriptor se muestra en la figura 8-18. Especifica el tipo de elemento que le sigue, su tamaño y el número de ellos (un campo de datos puede contener varios elementos del mismo tipo). Los tipos disponibles incluyen simples bits y bytes, enteros de varios tamaños, palabras de máquina sin estructura alguna, colecciones de booleanos, números de punto flotante, cadenas y posibilidades. Con esta información, el sistema puede intentar realizar conversiones entre las máquinas, cuando las máquinas fuente y destino tienen diferentes representaciones internas. Esta conversión no la lleva a cabo el núcleo, sino el servidor de mensajes de la red (descrito más adelante). Esto también sirve para el transporte entre nodos (realizado también por el servidor de mensajes de la red), incluso para los mensajes simples.

Uno de los elementos más interesantes que puede estar contenido en un campo de datos es una posibilidad. Mediante los mensajes complejos es posible copiar o transferir una

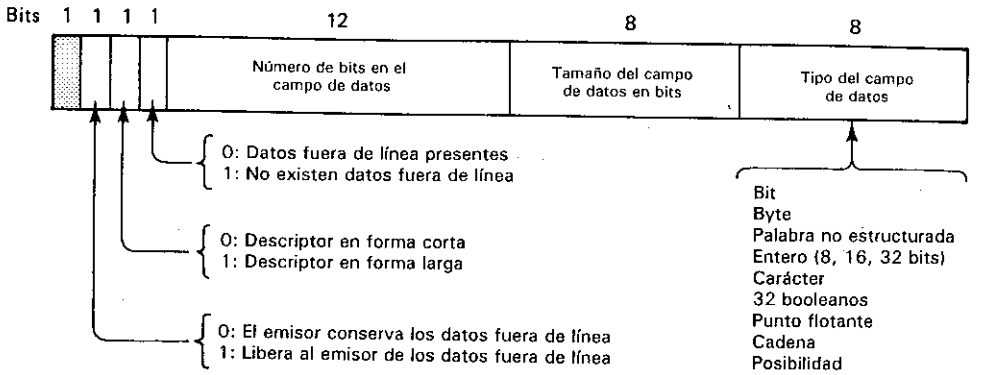


Figura 8-18. Descriptor de campo de un mensaje complejo.

posibilidad de un proceso a otro. Puesto que las posibilidades son objetos del núcleo protegidos en Mach, se necesita un mecanismo protegido para desplazarlos.

Este mecanismo es el siguiente. Un descriptor puede especificar que la palabra inmediata en el mensaje contiene el nombre de una de las posibilidades del emisor y que ésta se va a transferir al proceso receptor e insertar en la lista de posibilidades de éste. El descriptor también especifica si se va a copiar la posibilidad (la original no se modifica) o a desplazar (se elimina el original).

Además, ciertos valores del *tipo del campo de datos* solicitan al núcleo que modifique los derechos de la posibilidad mientras realiza la copia o el traslado. Por ejemplo, una posibilidad RECEIVE puede cambiar a una posibilidad SEND o SEND-ONCE, de modo que el receptor podrá enviar una respuesta a un puerto para el que el emisor sólo tenga una posibilidad RECEIVE. De hecho, la forma normal para establecer la comunicación entre dos procesos es que uno de ellos cree un puerto y entonces envíe la posibilidad RECEIVE del puerto al otro, modificándola por una posibilidad SEND en el trayecto.

Para ver cómo funciona el transporte de posibilidades, consideremos la situación de la figura 8-19(a). Aquí vemos dos procesos, *A* y *B*, con tres y una posibilidades, respectivamente. Todas son posibilidades RECEIVE. La numeración empieza en 1, ya que la entrada 0 es el puerto nulo. Uno de los hilos en *A* envía un mensaje a *B*, el cual contiene la posibilidad 3.

Cuando llega el mensaje, el núcleo inspecciona el encabezado y ve que se trata de un mensaje complejo. Entonces comienza a procesar los descriptores en el cuerpo del mensaje, uno por uno. En este ejemplo sólo existe un descriptor, para una posibilidad, con instrucciones para modificarlo a una posibilidad SEND (o tal vez SEND-ONCE). El núcleo asigna un espacio libre en la lista de posibilidades del receptor, que en este ejemplo es el espacio 2, y modifica el mensaje de modo que la palabra posterior al descriptor es ahora 2 en vez de 3. Cuando el receptor obtiene el mensaje, ve que tiene una nueva posibilidad de nombre (índice) 2. Puede utilizar esa posibilidad de manera inmediata (por ejemplo, para el envío de un mensaje de respuesta).

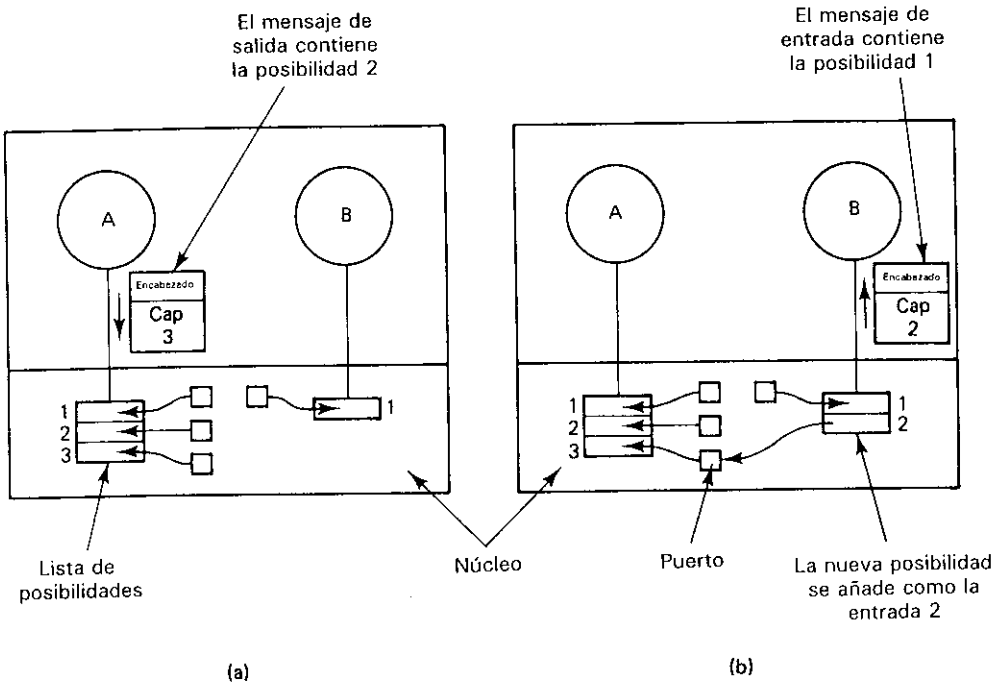


Figura 8-19. (a) Situación antes de enviar la posibilidad. (b) Situación después de que ésta llega a su destino.

Existe un último aspecto de la figura 8-18 que no hemos analizado todavía: **los datos fuera de línea**. Mach proporciona una forma para transferir datos en bruto de un emisor a un receptor, sin realizar copiado alguno (en una máquina o un multiprocesador). Si el bit de datos fuera de línea está activo en el descriptor, la palabra posterior al descriptor contiene una dirección y los campos de tamaño y número del descriptor, lo cual proporciona un contador de bytes de 20 bits. Juntos especifican una región del espacio de direcciones virtuales del emisor. Se utiliza la forma larga del descriptor para regiones de mayor tamaño.

Cuando el mensaje llega al receptor, el núcleo eligé una parte no asignada del espacio de direcciones virtuales del mismo tamaño que los datos fuera de línea y asocia las páginas del emisor con el espacio de direcciones del receptor, señalándolas como de "copiado durante la escritura". La palabra de dirección posterior al descriptor se modifica para reflejar la dirección donde se localiza la región en el espacio de direcciones del receptor. Este mecanismo proporciona una forma de mover bloques de datos con velocidades muy altas, puesto que no se necesita copiado alguno, excepto por el encabezado del mensaje y el cuerpo de dos palabras (el descriptor y la dirección). Según el valor de un bit en el descriptor, la región se elimina o mantiene dentro del espacio de direcciones del emisor.

Aunque este método es muy eficiente para copias entre procesos en una máquina (o entre los CPU en un multiprocesador), no es muy útil para la comunicación en una red ya

que las páginas deben ser copiadas si se utilizan, aunque sólo se vayan a leer. Así, se pierde la capacidad para transmitir los datos de manera lógica sin trasladarlos de manera física. El copiado durante la escritura requiere también que los mensajes estén alineados en las fronteras de las páginas y que tengan una longitud dada por un número entero de páginas para mejores resultados. Las páginas fraccionarias permiten al receptor ver datos antes o después de los datos fuera de línea, que no debería ver.

8.4.3. El servidor de mensajes de la red

Todo lo que hemos dicho hasta ahora acerca de la comunicación en Mach se limita a la comunicación dentro de un nodo, ya sea un CPU o un nodo multiprocesador. La comunicación a través de la red se controla mediante servidores a nivel usuario llamados **servidores de mensajes de la red**, que son algo similar a los administradores externos de memoria ya analizados. Cada máquina en un sistema distribuido Mach ejecuta un servidor de mensajes de la red, los cuales funcionan juntos para administrar los mensajes entre las máquinas, con la intención de simular los mensajes dentro de las máquinas lo mejor que puedan.

Un servidor de mensajes de la red es un proceso de varios hilos que lleva a cabo varias funciones. Entre éstas se encuentran la interfaz con los hilos locales, el envío de mensajes a través de la red, la traducción de los tipos de datos de la representación de una máquina a la de otra, la administración de las posibilidades de manera segura, las notificaciones remotas, proporcionar un servicio sencillo de búsqueda de nombres a todo lo ancho de la red y la administración de la autenticación de otros servidores de mensajes de la red. Los servidores de mensajes de la red pueden utilizar una variedad de protocolos, según la red a la que estén conectados.

El método básico mediante el cual se envían los mensajes a través de la red se muestra en la figura 8-20. Aquí tenemos un cliente en la máquina *A* y un servidor en la máquina *B*. Antes de que el cliente pueda hacer contacto con el servidor, hay que crear un puerto en *A* para que funcione como un representante del servidor. El servidor de mensajes de la red tiene la posibilidad RECEIVE para este puerto. Un hilo dentro de él escucha de manera constante a este puerto (y a otros puertos remotos, que juntos forman un conjunto de puertos). Este puerto aparece como el cuadro pequeño en el núcleo de *A*.

El transporte de un mensaje del cliente al servidor requiere de cinco pasos, numerados del 1 al 5 en la figura 8-20. En primer lugar, el cliente envía un mensaje al puerto representante del servidor. En segundo lugar, el servidor de mensajes de la red obtiene dicho mensaje. Puesto que este mensaje es estrictamente local, se envían datos fuera de línea y el copiado durante la escritura funciona de la manera usual. En tercer lugar, el servidor de mensajes de la red busca el puerto local, 4 en el ejemplo, en una tabla que asocia los puertos representantes con los **puertos de la red**. Una vez determinado el puerto de la red, el servidor de mensajes de la red busca su posición en otras tablas. Entonces construye un mensaje en la red con el mensaje local, más ciertos datos fuera de línea, y lo envía a través de la LAN al servidor de mensajes de la red en la máquina del servidor. En ciertos casos,

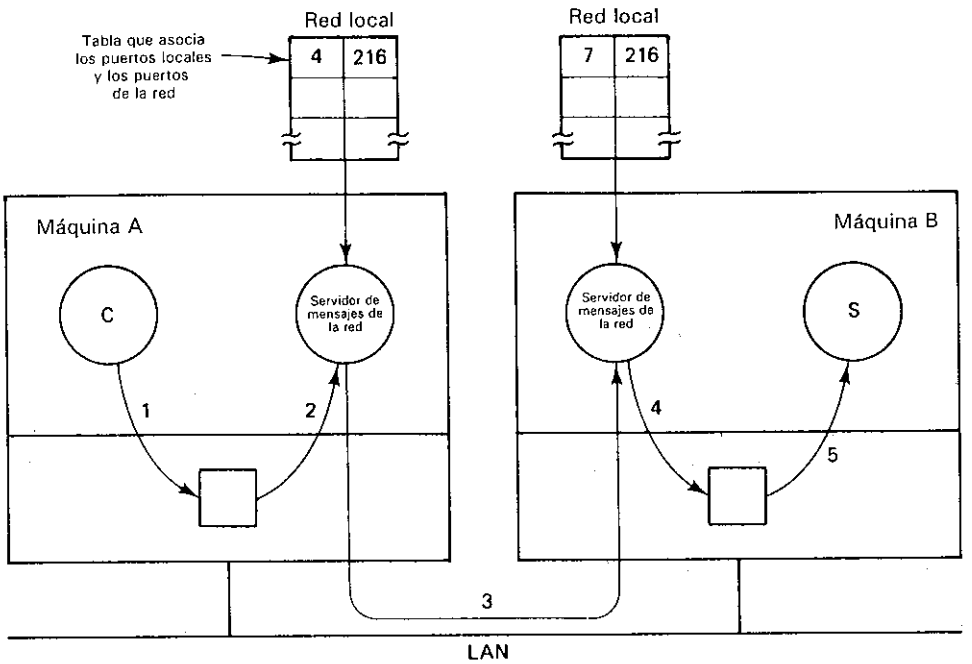


Figura 8-20. La comunicación entre máquinas en Mach se lleva a cabo en cinco pasos.

el tráfico entre los servidores de mensajes de la red debe cifrarse para mayor seguridad. El módulo de transporte se encarga de separar el mensaje en paquetes y encapsularlos en los recipientes del protocolo adecuado.

Cuando el servidor remoto de mensajes de la red obtiene el mensaje, busca el número del puerto de la red contenido en él y lo asocia a un número de puerto local. En el paso 4, escribe el mensaje en el puerto local recién buscado. Por último, el servidor lee el mensaje del puerto local y lleva a cabo la solicitud. La respuesta sigue la misma trayectoria en dirección opuesta.

Los mensajes complejos requieren un poco más de trabajo. Para los campos ordinarios de datos, es el servidor de mensajes de la red de la máquina servidor el que lleva a cabo la conversión, en caso de ser necesario; por ejemplo, tomando en cuenta un orden diferente de los bytes en las dos máquinas. También hay que procesar las posibilidades. Cuando una posibilidad se envía a través de la red, se le asigna un número de puerto de la red y tanto el servidor de mensajes fuente como el destino crean entradas para ella en sus tablas de asociación. Si estas máquinas desconfían entre sí, serán necesarios procedimientos de autenticación elaborados, para convencerse de la verdadera identidad de cada una.

Aunque la idea de transmitir mensajes de una máquina a la otra por medio de un servidor a nivel usuario ofrece cierta flexibilidad, se paga un precio sustancial en el desempeño, en

comparación con una implantación pura en el núcleo, utilizada por la mayoría de los demás sistemas distribuidos. Para resolver este problema, se está desarrollando una nueva versión del paquete de comunicación en una red (el código NORMA), que se ejecuta dentro del núcleo y logra una comunicación más rápida. De manera eventual reemplazará al servidor de mensajes en la red.

8.5. EMULACIÓN DE UNIX EN MACH

Mach tiene varios servidores que se ejecutan por arriba de él. Tal vez el más importante es un programa que contiene gran parte del UNIX de Berkeley (por ejemplo, en esencia todo el código del sistema de archivos) dentro de él. Este servidor es el principal emulador de UNIX (Golub *et al.*, 1990). Este diseño es un reconocimiento de la historia de Mach como versión modificada del UNIX de Berkeley.

La implantación de la emulación de UNIX en Mach consta de dos partes, el servidor de UNIX y una biblioteca de emulación de llamadas al sistema, como se muestra en la figura 8-21. Al arrancar el sistema, el servidor de UNIX indica al núcleo que capture todos los señalamientos de llamadas al sistema y los dirija hacia una dirección dentro de la biblioteca de emulación del proceso UNIX que realiza la llamada al sistema. A partir de ese momento, cualquier llamada al sistema hecha por un proceso UNIX hará que el control pase de manera temporal al núcleo, y de inmediato a su biblioteca de emulación. En el momento en que el control se otorga a la biblioteca de emulación, todos los registros de la máquina recuperan los valores que tenían al momento del señalamiento. Este método de "rebotar" el núcleo de regreso al espacio del usuario se denomina a veces el **mecanismo del trampolín**.

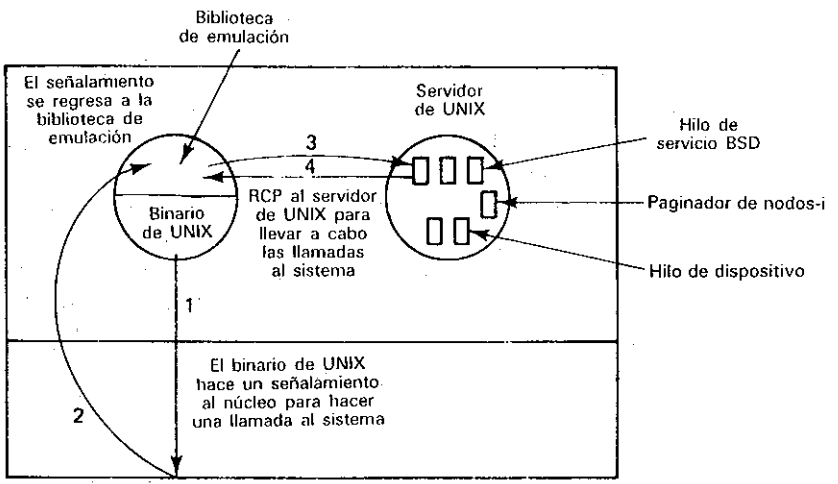


Figura 8-21. La emulación de UNIX en Mach utiliza el mecanismo de trampolín.

Una vez que la biblioteca de emulación obtiene el control, examina los registros para determinar qué llamada al sistema se invocó. Entonces hace una RPC con otro proceso, el servidor de UNIX, para realizar el trabajo. Cuando termina, el programa usuario recupera el control. Esta transferencia de control no tiene que pasar por el núcleo.

Cuando el proceso *init* produce hijos, éstos heredan de manera automática la biblioteca de emulación y el mecanismo de trampolín, por lo que también puede hacer llamadas al sistema UNIX. La llamada al sistema EXEC se implantó de modo que no reemplaza la biblioteca de emulación, sino sólo la parte del programa UNIX correspondiente al espacio de direcciones.

El servidor UNIX se implanta como colección de hilos C. Aunque algunos hilos controlan los cronómetros, el uso de la red y otros dispositivos de E/S, la mayoría de los hilos controlan las llamadas al sistema BSD, llevando a cabo solicitudes a cargo de los emuladores dentro de los procesos UNIX. La biblioteca de emulación se comunica con estos hilos mediante la comunicación entre procesos usual de Mach.

Cuando un mensaje llega al servidor UNIX, un hilo inactivo lo acepta, determina el proceso del cual proviene, extrae el número y parámetros de la llamada al sistema, la lleva a cabo y, por último, envía de regreso la respuesta. La mayoría de los mensajes corresponde con exactitud a una llamada al sistema BSD.

Un conjunto de llamadas al sistema que no funciona de esta manera son las llamadas de E/S al archivo. *Podrían* haberse implantado de esta forma, pero por razones de desempeño, se utilizó un método diferente. Al abrir un archivo, éste se asocia de manera directa con el espacio de direcciones del proceso que hace la llamada, de modo que la biblioteca de emulación pueda llegar a él de manera directa, sin tener que hacer una RPC con el servidor de UNIX. Por ejemplo, para satisfacer una llamada al sistema READ, la biblioteca de emulación localiza los bytes para leer en el archivo asociado, localiza el buffer del usuario y simplemente hace una copia del primero al último de la forma más rápida posible.

Si las páginas del archivo no están en la memoria, ocurrirán fallos de página durante el ciclo de copiado. Cada fallo hará que Mach envíe un mensaje al administrador externo de memoria para respaldar el archivo UNIX asociado. Este administrador de memoria es un hilo dentro del servidor de UNIX, llamado **paginador de nodos_i**. Obtiene la página del archivo del disco y hace que sea asociada al espacio de direcciones del programa de aplicación. También sincroniza las operaciones en los archivos abiertos por varios procesos UNIX de manera simultánea.

Aunque este método de ejecución de programas UNIX parece un tanto barroco, distintas mediciones han mostrado que se compara de manera favorable con las implantaciones adicionales mediante un núcleo monolítico (Golub *et al.*, 1990). El trabajo futuro se centrará en la separación del servidor de UNIX en varios servidores con funciones más específicas. En cierto momento, es posible que se elimine el servidor único de UNIX, aunque esto depende de la forma en que se desarrolle el trabajo con varios servidores a través del tiempo.

8.6. RESUMEN

Mach es un sistema operativo basado en un micronúcleo. Se diseñó con el fin de proporcionar una base para la construcción de nuevos sistemas operativos y la emulación de

los ya existentes. También proporciona una forma flexible de extender UNIX a los multiprocesadores y los sistemas distribuidos.

Mach se basa en los conceptos de procesos, hilos, puertos y mensajes. Un proceso en Mach es un espacio de direcciones y una colección de hilos que se ejecutan en él. Las entidades activas son los hilos. El proceso es simplemente un recipiente para ellos. Cada proceso e hilo tiene un puerto al que se puede escribir para hacer que las llamadas al núcleo se lleven a cabo, lo que elimina la necesidad de las llamadas directas al sistema.

Mach tiene un sistema de memoria virtual muy elaborado, con objetos de memoria que se pueden asociar o desasociar de los espacios de direcciones, respaldado por administradores de memoria externos a nivel usuario. De esta forma, se puede escribir o leer de los archivos en forma directa, por ejemplo. Los objetos de memoria se pueden compartir de varias maneras, entre las cuales se encuentra el copiado durante la escritura. Los atributos de herencia determinan las partes del espacio de direcciones de un proceso que deben transferirse a sus hijos.

La comunicación en Mach se basa en los puertos, objetos del núcleo que contienen mensajes. Todos los mensajes se dirigen a los puertos, a los cuales se tiene acceso mediante las posibilidades; éstas se almacenan dentro del núcleo y se hace referencia a ellas mediante enteros de 32 bits, que son por lo general índices a listas de posibilidades. Los puertos se pueden transferir de un proceso a otro al incluirlos en los mensajes complejos.

La emulación del UNIX BSD se realiza mediante una biblioteca de emulación que vive en el espacio de direcciones de cada proceso UNIX. Su trabajo es capturar las llamadas al sistema reflejadas hacia ella por el núcleo y transferirlas al servidor UNIX para que se lleven a cabo. Unas cuantas llamadas se controlan en forma local, dentro del espacio de direcciones del proceso. Se están desarrollando otros emuladores de UNIX.

Amoeba y Mach tienen muchos aspectos en común, pero también varias diferencias. Ambos tienen procesos e hilos y se basan en la transferencia de mensajes. Amoeba tiene como primitiva a la transmisión confiable, mientras que Mach no; pero Mach tiene paginación según la demanda, y Amoeba no. En general, Amoeba está más orientado a hacer que una colección de máquinas distribuidas actúe como una computadora, mientras que Mach está más orientado hacia el uso eficaz de los multiprocesadores. Ambos se están desarrollando de manera continua y sin duda se verán modificados con el curso del tiempo.

PROBLEMAS

1. Mencione una diferencia entre un proceso con dos hilos y dos procesos con un hilo cada uno pero que comparten el mismo espacio de direcciones; es decir, el mismo conjunto de páginas.
2. ¿Qué ocurre si se realiza un *join* con uno mismo?
3. Un hilo de Mach crea dos nuevos hilos como hijos, *A* y *B*. El hilo *A* hace una llamada *detach* y *B* no. Ambos hilos realizan su salida y el padre ejecuta un *join*. ¿Qué ocurre?

4. Las colas de ejecución global de la figura 8-6 se deben cerrar antes de comenzar la búsqueda. ¿También deben cerrarse las colas locales (que no se muestran en la figura) antes de iniciar la búsqueda? ¿Por qué sí o por qué no?
5. Cada una de las colas globales de ejecución tiene un mútex para su cerradura. Suponga que un multiprocesador particular tiene un reloj global que provoca interrupciones del reloj en todos los CPU de manera simultánea. ¿Qué implicaciones tiene esto para el planificador de Mach?
6. Mach soporta el concepto de un conjunto de procesadores. ¿En cuál tipo de máquinas tiene más sentido este concepto? ¿Para qué se utiliza?
7. Mach soporta tres atributos de herencia para las regiones del espacio de direcciones virtuales. ¿Cuáles de ellos se necesitan para que el FORK de UNIX funcione de manera correcta?
8. Un pequeño proceso tiene todas sus páginas en memoria. Existe la suficiente memoria disponible como para hacer diez copias más del proceso. Éste produce un hijo. ¿Es posible que el hijo obtenga un fallo de página o de protección?
9. ¿Por qué cree que existe una llamada para copiar una región de la memoria virtual (véase la figura 8-8)? Después de todo, cualquier hilo lo puede copiar al quedarse en un ciclo de copiado.
10. ¿Por qué se ejecuta el algoritmo para el reemplazo de páginas en el núcleo en vez de ejecutarse en un administrador externo de la memoria?
11. Dé un ejemplo donde sea deseable que un hilo libere un objeto en su espacio de direcciones virtuales.
12. ¿Pueden dos procesos tener una posibilidad RECEIVE para el mismo puerto al mismo tiempo? ¿Qué ocurre con las posibilidades SEND?
13. ¿Sabe un proceso si el puerto de donde lee es en realidad un conjunto de puertos? ¿Importa eso?
14. Mach soporta dos tipos de mensajes, simples y complejos. ¿Se necesitan en realidad los mensajes complejos, o son solamente una optimización?
15. Conteste ahora la pregunta anterior en el caso de las posibilidades SEND-ONCE y los mensajes fuera de línea. ¿Son esenciales para el correcto funcionamiento de Mach?
16. En la figura 8-15, el mismo puerto tiene un nombre distinto en procesos diferentes. ¿Qué problemas puede causar esto?
17. Mach tiene una llamada al sistema que permite a un proceso solicitar que los señalamientos que no sean de tipo Mach se pasen a un controlador especial, en vez de provocar la eliminación del proceso. ¿Para qué sirve esta llamada al sistema?

9

Estudio 3: Chorus

Nuestro tercer ejemplo de un sistema operativo moderno se basa en un micronúcleo, Chorus. La estructura de este capítulo es similar a la de los capítulos anteriores: primero una breve historia, luego un panorama del micronúcleo, y después una revisión detallada de la administración de procesos, la administración de la memoria y la comunicación. Después de ello, estudiaremos la forma en que Chorus enfrenta la emulación de UNIX. A continuación está una sección acerca de la programación distribuida orientada a objetos en Chorus. Concluiremos con una breve comparación de Amoeba, Mach y Chorus. El lector puede encontrar más información relativa a Chorus en (Abrossimov *et al.*, 1989, 1992; Armand y Dean, 1992; Batlivala, *et al.*, 1992; Bricker *et al.*, 1991; Gien y Grob, 1992; y Rozier *et al.*, 1988).

9.1. INTRODUCCIÓN A CHORUS

En esta sección resumiremos la forma en que Chorus ha evolucionado a lo largo de los años, analizaremos sus objetivos de forma breve y después daremos una introducción técnica a su micronúcleo y dos de sus subsistemas. En las secciones posteriores describiremos el micronúcleo y los subsistemas con mayor detalle. La documentación de Chorus utiliza una terminología poco estándar. En este capítulo utilizaremos los nombres estándar pero daremos los términos de Chorus entre paréntesis.

9.1.1. Historia de Chorus

Chorus surgió del instituto francés de investigación INRIA en 1980, como proyecto de investigación en sistemas distribuidos. Desde entonces han aparecido cuatro versiones, numeradas del 0 al 3. La idea detrás de la versión 0 era la de modelar aplicaciones distri-

buidas como colección de **actores**, en esencia procesos estructurados, cada uno de los cuales alternaban entre la realización de una transacción atómica y la ejecución de un paso de comunicación. En realidad, cada actor era un autómata de estado finito macroscópico. Cada máquina del sistema ejecutaba el mismo núcleo, el cual controlaba a los actores, la comunicación, los archivos y los dispositivos de E/S. La versión 0 fue escrita en Pascal UCSD interpretado y se ejecutó en una colección de máquinas 8086 conectadas mediante una red de anillo. Fue operacional a mediados de 1982.

La versión 1, que se utilizó de 1982 a 1984, se centró en la investigación del multiprocesador. Fue escrita para el multiprocesador francés SM90, que constaba de 8 CPU 68020 de Motorola en un bus común. Un CPU ejecutaba UNIX; las otras siete ejecutaban Chorus y utilizan el CPU de UNIX para los servicios del sistema y E/S. Varias SM90 se conectaban mediante Ethernet. El software era similar al de la versión 0, añadiendo mensajes estructurados y algo de soporte para la tolerancia de fallas. La versión 1 fue escrita en Pascal compilado en vez de interpretado, y se distribuyó a cerca de una docena de universidades y compañías para su uso experimental.

La versión 2 (1984-1986) fue una reescritura fundamental del sistema, en C. Se diseñó de modo que las llamadas al sistema fuesen compatibles con UNIX en el nivel del código fuente, lo que significa que podía recompilar los programas existentes en UNIX en Chorus y ejecutarlos en él. El núcleo de la versión 2 se rediseñó por completo, pasando la mayor funcionalidad posible de éste al código del usuario, y cambiando el núcleo por lo que ahora se conoce como micronúcleo. La emulación de UNIX se realizaba mediante varios procesos, para el manejo de la administración de procesos, la de archivos y la de dispositivos, respectivamente. Se agregó un soporte para las aplicaciones distribuidas, incluyendo la ejecución remota y los protocolos para la asignación de nombres y la localización distribuidas.

La versión 3 se inició en 1987. Esta versión marcó la transición de un sistema de investigación a un producto comercial, ya que los diseñadores de Chorus salieron de INRIA y formaron una compañía, Chorus Systèmes, para seguir desarrollando y comercializar Chorus. Se hicieron numerosos cambios técnicos en la versión 3, incluyendo un refinamiento del micronúcleo y de su relación con el resto del sistema. Desaparecieron los últimos vestigios del modelo del actor, con sus transacciones atómicas y se introdujo la llamada a procedimientos remotos (RPC) como el modelo de comunicación usual. Los procesos en modo núcleo también aparecieron.

Para que Chorus fuese un producto comercial viable, se aumentó la capacidad de emulación de UNIX. Se agregó la compatibilidad binaria, de modo que los programas en UNIX se ejecutaran sin compilarse de nuevo. Parte de la emulación de UNIX, que se realizaba en el micronúcleo, pasó al subsistema de emulación, que se hizo al mismo tiempo más modular. El manejo de excepciones se modificó para manejar las señales de UNIX de manera correcta.

Se mejoró su desempeño. Además, el sistema fue parcialmente reescrito en C++. Además, se hizo más portátil y se implantó en varias arquitecturas diferentes. La versión 3 también tomó varias ideas de otros micronúcleos de sistemas distribuidos, entre las que destacan el sistema de comunicación entre procesos, el diseño de la memoria virtual y los

paginadores externos de Mach, y el uso de capacidades ralas para la asignación global de nombres y la protección de Amoeba.

9.1.2. Objetivos de Chorus

Los objetivos del proyecto Chorus han evolucionado junto con el sistema. En un principio, se trataba de una investigación puramente académica, diseñada para explorar nuevas ideas en el cómputo distribuido con base en el modelo del actor. Al pasar el tiempo, se volvió más comercial, y se cambió el énfasis. Los objetivos actuales se pueden resumir como sigue:

1. Emulación de UNIX de alto rendimiento.
2. Uso en sistemas distribuidos.
3. Aplicaciones de tiempo real.
4. Integración de la programación orientada a objetos en Chorus.

Como sistema comercial, buena parte del trabajo se centraba en el seguimiento de los estándares en evolución de UNIX, portando el sistema a nuevos circuitos CPU, y mejorando el desempeño. La compañía desea que Chorus se vea como alternativa al UNIX de AT&T, con nueva ingeniería, un mantenimiento más sencillo y orientado a las necesidades futuras del usuario.

Un segundo tema fundamental es la necesidad de la distribución. Chorus pretende que los programas en UNIX se ejecuten en una colección de máquinas conectadas mediante una red. Para soportar las aplicaciones distribuidas, se han agregado varias extensiones al modelo de programación. Algunas de éstas, como la comunicación basada en mensajes, se ajustan muy bien al modelo existente. Otras, como la introducción de hilos, requieren repensar las características existentes, como el manejo de las señales en UNIX.

Una tercera dirección es la introducción de un soporte para las aplicaciones de tiempo real. El enfoque en este caso permite que los programas de tiempo real se ejecuten (parcialmente) en modo núcleo y que tengan acceso directo al micronúcleo, sin software de por medio. Aquí también son importantes el control de las interrupciones por parte del usuario y el algoritmo de planificación.

Por último, otro objetivo es la introducción de la programación orientada a objetos en Chorus de manera clara, sin molestar a los subsistemas y aplicaciones existentes. La forma de hacer esto se describe con detalle en una sección posterior de este capítulo.

9.1.3. Estructura del sistema

Chorus está estructurado en capas, como se muestra en la figura 9-1. En la parte inferior está el micronúcleo (llamado sólo **núcleo** en la documentación de Chorus). Proporciona la mínima administración de los nombres, procesos, hilos, memoria y comunicación. Se tiene acceso a estos servicios mediante llamadas al micronúcleo. Existen más de 100 llamadas.

Los procesos de las capas superiores proporcionan el resto del sistema operativo. Cada máquina de un sistema distribuido basado en Chorus ejecuta una copia idéntica del micronúcleo de Chorus.

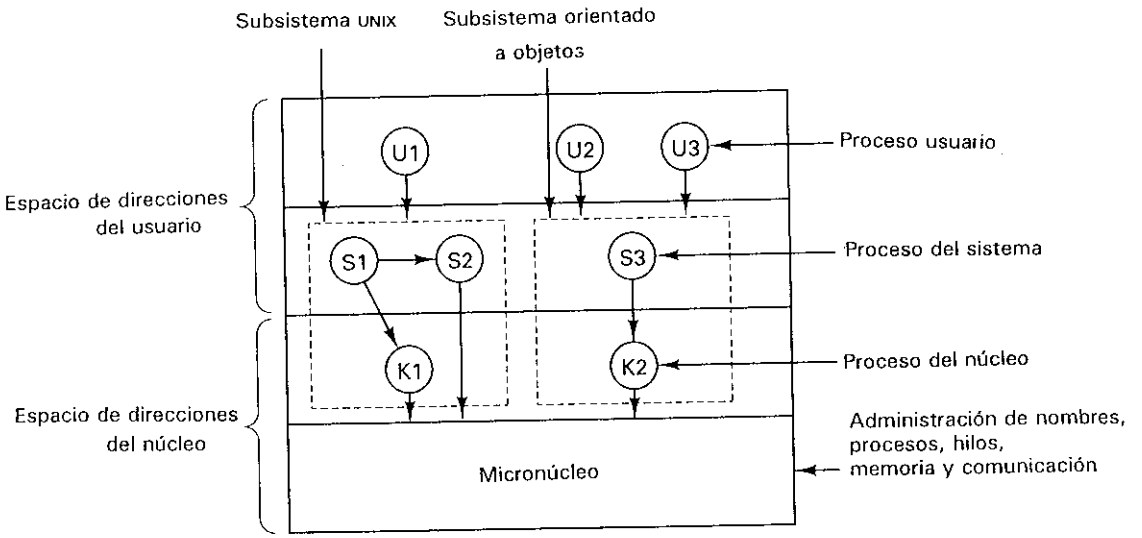


Figura 9.1. Chorus está estructurado en capas, con un micronúcleo, los subsistemas y los procesos usuario.

Por arriba del micronúcleo, pero operando también en el modo núcleo, están los **procesos del núcleo**. Estos procesos se cargan y eliminan de manera dinámica durante la ejecución del sistema y proporcionan una forma de extender la funcionalidad del micronúcleo sin que aumenten de manera permanente su tamaño y complejidad. Puesto que estos procesos comparten el espacio del núcleo con el micronúcleo y también entre ellos, deben cambiar de posición después de cargarse. Pueden llamar al micronúcleo para obtener servicios, y también se llaman entre sí.

Por ejemplo, los controladores de interrupciones se escriben como procesos del núcleo. En una máquina con una unidad de disco al momento de iniçiar el sistema, se cargará el proceso controlador de interrupciones del disco. Cuando ocurran las interrupciones de disco, éstas serán controladas por este proceso. En las estaciones de trabajo sin disco, el controlador de interrupciones del disco no se necesita y por lo tanto no se carga. La capacidad para cargar y descargar los procesos del núcleo de manera dinámica permite configurar el software del sistema de manera acorde con el hardware, sin tener que volver a compilar o a ligar el micronúcleo.

La siguiente capa contiene los **procesos del sistema**. Estos se ejecutan en modo usuario, pero envían mensajes a los procesos del núcleo (y entre sí) y realizan llamadas al micronúcleo, como se muestra mediante las flechas de la figura 9-1. Una colección de procesos del núcleo y de sistema trabajan de manera conjunta para formar un **subsistema**. En la

figura 9.1, los procesos *S1, S2* y *K1* forman un subsistema y los procesos *S3* y *K2* forman otro. Un subsistema presenta una interfaz bien definida a sus usuarios, como la interfaz de llamadas al sistema UNIX. Un proceso de cada subsistema es el administrador y controla la operación del subsistema.

Arriba de los subsistemas están los **procesos usuario**. Por ejemplo, las llamadas al sistema realizadas por un proceso usuario *U1* podrían ser capturadas por *K1* y pasar a *S1* o *S2* para su procesamiento. Estos, a su vez, podrían utilizar los servicios del micronúcleo, en caso necesario. Los subsistemas permiten construir nuevos (o viejos) sistemas operativos sobre el micronúcleo de manera modular, y permiten la existencia de varias interfaces con el sistema operativo, en una misma máquina y al mismo tiempo.

El micronúcleo sabe cuál subsistema está utilizando cada proceso usuario (si utiliza alguno) y garantiza que se restringe a la realización de las llamadas al sistema ofrecidas por ese subsistema. No se permiten las llamadas directas de los procesos usuario al micronúcleo, excepto por aquellas llamadas definidas como válidas por el subsistema. Los procesos de tiempo real se pueden ejecutar como procesos del sistema, en vez de procesos usuario, y con ello utilizar por completo el micronúcleo sin intervención ni un costo excesivo.

9.1.4. Abstracciones del núcleo

El núcleo (llamado el micronúcleo en nuestra nomenclatura) proporciona y controla seis abstracciones fundamentales que forman la base de Chorus. Estos conceptos son los procesos, los hilos, las regiones, los mensajes, los puertos, los grupos de puerto y los identificadores únicos. Estos se ilustran en la figura 9.2. Los **procesos** de Chorus (que siguen llamándose **actores** en la documentación de Chorus) son en esencia iguales a los procesos de los demás sistemas operativos. Son recipientes que encapsulan a los recursos. Un proceso posee ciertos recursos, y cuando el proceso desaparece, también desaparecen sus recursos.

Dentro de un proceso, pueden existir uno o más **hilos**. Cada hilo es similar a un proceso que tiene su propia pila, su apuntador a la pila, contador del programa y registros. Sin embargo, todos los hilos de un proceso comparten el mismo espacio de direcciones y otros recursos disponibles en todo el proceso. En principio, los hilos de un proceso son independientes entre sí. En un multiprocesador, se podrían ejecutar varios hilos al mismo tiempo, en diferentes CPU. Los tres tipos de procesos pueden tener varios hilos.

Cada proceso tiene un espacio de direcciones, que por lo general van de 0 a cierta dirección máxima, como $2^{32}-1$. Todos los hilos de un proceso tienen acceso a este espacio de direcciones. Un rango consecutivo de direcciones es una **región**. Cada región está asociada con alguna pieza de datos, como un programa o archivo. En los sistemas que soportan la memoria virtual y la paginación, las regiones se pueden paginar. Las regiones juegan un papel fundamental en la administración de la memoria en Chorus.

Los hilos de procesos diferentes (en potencia en diversas máquinas) se comunican mediante la transferencia de **mensajes**. Los mensajes tienen una parte fija y un cuerpo de

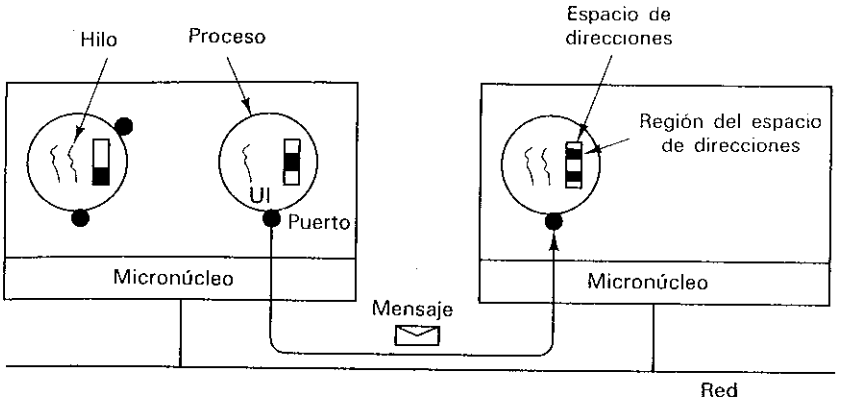


Figura 9.2. Procesos, hilos, regiones, mensajes y puertos, con identificadores únicos.

tamaño variable, ambos opcionales. El cuerpo no está tipificado y contiene cualquier información colocada ahí por el emisor. Un mensaje no se envía a un hilo, sino a una estructura intermedia llamada **puerto**. Un puerto es un buffer para la recepción de mensajes, que contiene los mensajes recibidos por un proceso pero que aún no han sido leídos. Al igual que un hilo, una región o algún otro recurso, en cualquier momento, cada puerto pertenece a un proceso. Sólo ese proceso puede leer sus mensajes. Los puertos se agrupan para formar grupos de puertos. Analizaremos estos puertos posteriormente.

La última abstracción del núcleo se refiere a los nombres. La mayor parte de los recursos del núcleo (por ejemplo, los procesos y los puertos) reciben su nombre mediante un **identificador único (UI)** de 64 bits. Una vez que un UI se ha asignado a un recurso, se garantiza que nunca será reutilizado para otro recurso, ni siquiera en una máquina diferente un año después. Esta unicidad se garantiza al codificar en cada UI el sitio (máquina o multiprocesador) donde se creó la UI más un número de época y un contador válida en la misma. El número de época se incrementa cada vez que se arranca el sistema.

Los UI son simplemente números binarios y en sí no están protegidos. Los procesos pueden enviar UI a otros procesos en mensajes o guardarlos en archivos. Cuando se transfiere un UI a través de la red y el receptor intenta tener acceso al objeto correspondiente, la información de la posición en el UI se utiliza como un indicio de la probable posición del objeto.

Puesto que los UI son grandes y su uso es caro, se utilizan los **identificadores locales** o **LI** dentro de un proceso para identificar los recursos, de manera similar al uso de enteros pequeños como descriptores de archivos para identificar archivos abiertos en UNIX.

Las abstracciones del núcleo no son las únicas utilizadas por Chorus. También son importantes otras tres abstracciones que son controladas de manera conjunta por el núcleo y los subsistemas: las posibilidades, los identificadores de protección y los segmentos. Una **posibilidad** es el nombre de un recurso controlado por un subsistema (o, en unos cuantos casos, por el núcleo). Consta de la UI de 64 bits de un puerto perteneciente a ese subsistema y una clave de 64 bits asignada por el subsistema, como se muestra en la figura 9-3.

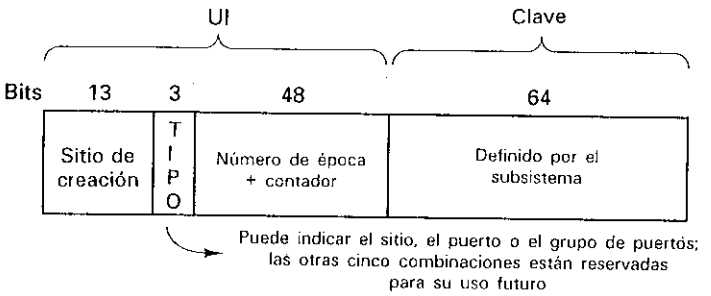


Figura 9.3. Una posibilidad en Chorus.

Cuando un subsistema crea un objeto, como archivo, regresa a quien hizo la llamada la posibilidad del objeto. A partir de esta posibilidad, ese proceso, o cualquier proceso posterior que adquiera la posibilidad, determina la UI de un puerto al que se envían los mensajes para solicitar operaciones sobre el objeto. La clave de 64 bits incluye esos mensajes para indicar al subsistema el objeto al cual se hace referencia. Dentro de los 64 bits se incluye un índice de las tablas del subsistema, para identificar al objeto. Se eligen otros bits de manera aleatoria para dificultar la estimación de las posibilidades válidas. Como las UI, las posibilidades pueden transferirse con libertad en mensajes y archivos. Este esquema de nombres fue tomado de Amoeba (Tanenbaum *et al.*, 1986).

La administración de la memoria en Chorus se basa en dos conceptos, las regiones ya descritas y los segmentos. Un **segmento** es una serie lineal de bytes identificados mediante una posibilidad. Cuando un segmento se asocia con una región, los bytes de ese segmento son accesibles a los hilos del proceso de la región, simplemente leyendo o escribiendo direcciones en la región. Los programas, archivos y otras formas de datos se guardan como segmentos en Chorus y se pueden asociar con regiones. Las llamadas al sistemas también pueden leer y escribir en un segmento, aunque éste no esté asociado con una región. En la figura 9.4 se muestra un ejemplo de espacio de direcciones.

9.1.5. Estructura del núcleo

Una vez descritas las principales abstracciones proporcionadas por el núcleo de Chorus, revisaremos de forma breve la estructura interna del núcleo. Éste consta de cuatro partes, que se muestran en la figura 9-5. En la parte inferior está el **supervisor**, que controla el hardware y atrapa los señalamientos, las excepciones, las interrupciones y demás detalles del hardware, además de controlar el intercambio de contexto. Está escrito parcialmente en ensamblador y debe volverse a crear si Chorus se lleva a un nuevo hardware.

A continuación está el **administrador de la memoria virtual**, que controla la parte de bajo nivel del sistema de paginación. La parte más grande de éste administra los cachés de

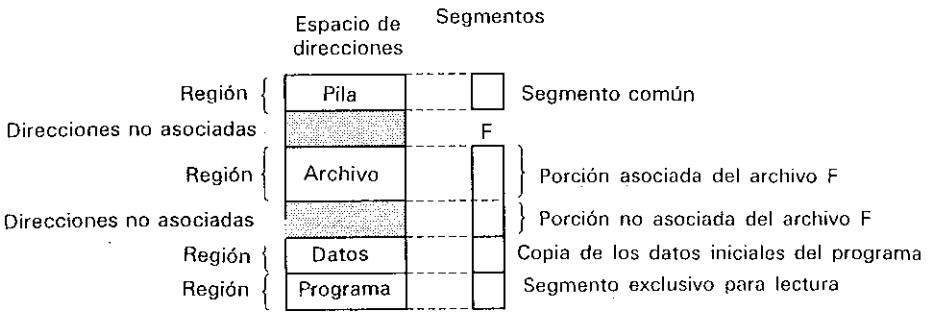


Figura 9.4. Un espacio de direcciones con cuatro regiones asociadas.

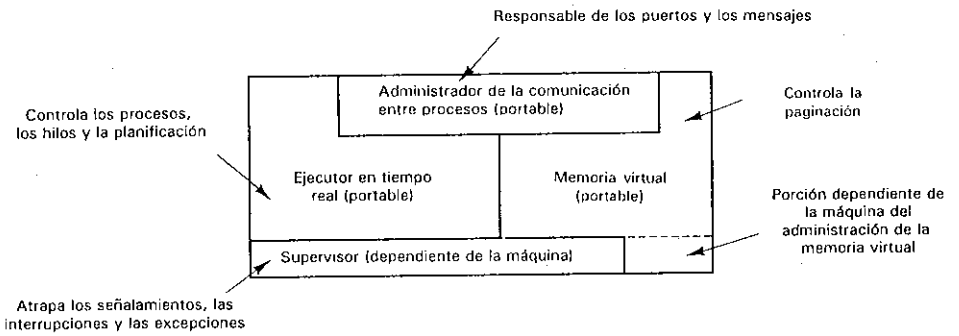


Figura 9.5. Estructura del núcleo de Chorus.

página y otros conceptos lógicos, y es independiente de la máquina. Sin embargo, una pequeña parte sabe cómo cargar y guardar los registros de MMU. Esta parte depende de la máquina y se modifica cuando Chorus se lleva a una nueva computadora.

Juntas, las dos partes del administrador de memoria virtual no realizan todo el trabajo de administración del sistema de paginación. Una tercera parte, el asociador, está fuera del núcleo y realiza la parte de mayor nivel. El protocolo de comunicación entre el administrador de la memoria virtual y el asociador está bien definido, de modo que los usuarios pueden proporcionar sus propios asociadores especializados.

La tercera parte del núcleo es el **ejecutor en tiempo real**, que es responsable de la administración de los procesos, de los hilos y de la planificación. También se encarga de ordenar la sincronización entre los hilos para la exclusión mutua y otros fines.

Por último, tenemos el **administrador de la comunicación entre procesos**, que controla los UI, los puertos, y el envío de mensajes de manera transparente. Utiliza los servicios del ejecutor en tiempo real y del administrador de la memoria virtual para realizar su trabajo.

Es por completo portable y no tiene que ser modificado en lo absoluto cuando Chorus se mueve a una nueva plataforma. Las cuatro partes del núcleo se construyen de manera modular, de modo que los cambios de una parte no afectan por lo general a las demás.

9.1.6. El subsistema UNIX

Puesto que Chorus es ahora un producto comercial, debe darle a las masas lo que desean (al menos en el mundo de las estaciones de trabajo de usuario final), la compatibilidad con UNIX. Chorus logra este objetivo proporcionando un subsistema estándar, llamado **MiX**, compatible con el System V. La compatibilidad es tanto a nivel fuente (es decir, los programas fuente de UNIX se compilan y ejecutan en Chorus) como nivel binario (es decir, los programas ejecutables compilados en un verdadero sistema UNIX para la misma arquitectura se ejecutan sin modificaciones en Chorus). Una versión anterior de MiX (3.2) era compatible con 4.2 BSD, pero en este capítulo no analizaremos esa versión.

MiX también es compatible con UNIX de varias formas más. Por ejemplo, el sistema de archivos es compatible, de modo que Chorus lee un disco de UNIX. Además, los controladores de dispositivos de Chorus son compatibles con los de UNIX en lo que se refiere a las interfaces, de modo que si existen controladores de dispositivo en UNIX para cierto dispositivo, se pueden llevar hacia Chorus con relativamente poco trabajo.

La implantación del subsistema MiX es más modular que UNIX. Consta de cuatro procesos, uno para la administración de procesos, uno para la administración de archivos, uno para la administración de dispositivos y uno para los flujos y la comunicación entre procesos. Estos procesos no comparten variables ni memoria, y se comunican de manera exclusiva mediante las llamadas a procedimientos remotos. En una sección posterior de este capítulo describiremos con detalle su funcionamiento.

9.1.7. El subsistema orientado a objetos

Como un experimento de investigación, se ha implantado un segundo subsistema, para la programación orientada a objetos. Consta de tres capas. La capa inferior realiza la administración de objetos de manera genérica y es de hecho un micronúcleo para sistemas orientados a objetos. La capa intermedia proporciona un sistema general de tiempo de ejecución. La capa superior es el sistema de tiempo de ejecución del lenguaje. Este subsistema, llamado COOL, también será analizado en una sección posterior de este capítulo.

9.2. ADMINISTRACIÓN DE PROCESOS EN CHORUS

En esta sección describiremos el funcionamiento de los procesos y los hilos en Chorus, la forma en que se manejan las excepciones y la forma en que se realiza la planificación. Concluiremos describiendo de forma breve algunas de las principales llamadas al núcleo disponibles para la administración de procesos.

9.2.1. Procesos

Un proceso en Chorus es una colección de elementos activos y pasivos que funcionan juntos para realizar cierto cálculo. Los elementos activos son los hilos. Los elementos pasivos son un espacio de direcciones (que contiene ciertas regiones) y una colección de puertos (para el envío y recepción de mensajes). Un proceso con un hilo es como proceso tradicional en UNIX. Un proceso sin hilos no puede realizar algo útil, y por lo general existe sólo durante un intervalo muy corto, mientras se crea un proceso.

Existen tres tipos de procesos, que difieren en la cantidad de privilegios y confianza que tienen, como se enumera en la figura 9-6. El privilegio se refiere a la capacidad de ejecutar la E/S y otras instrucciones protegidas. La confianza significa que se permite llamar de manera directa al núcleo.

Tipo	Confianza	Privilegio	Modo	Espacio
Usuario	No confiable	No privilegiado	Usuario	Usuario
Sistema	Confiable	No privilegiado	Usuario	Usuario
Núcleo	Confiable	Privilegiado	Núcleo	Núcleo

Figura 9.6. Los tres tipos de procesos en Chorus.

Los procesos del núcleo son los más poderosos. Se ejecutan en modo núcleo y todos comparten el mismo espacio de direcciones entre sí y con el micronúcleo. Se cargan o descargan durante la ejecución, pero fuera de ello, se pueden pensar como extensiones del propio micronúcleo. Los procesos del núcleo se comunican entre sí mediante una RPC ligera especial que no está disponible para los demás procesos.

Cada proceso del sistema se ejecuta en su propio espacio de direcciones. Los procesos del sistema no son privilegiados (es decir, se ejecutan en modo usuario) y por lo tanto no pueden ejecutar de manera directa E/S u otras instrucciones protegidas. Sin embargo, el núcleo confía en ellos para realizar llamadas al sistema, de modo que los procesos del sistema obtenga de manera directa servicios del núcleo, sin intermediarios.

Los procesos usuario no son confiables ni privilegiados. No pueden realizar E/S de manera directa, ni llamar al núcleo, excepto por aquellas llamadas que su subsistema ha decidido confiarles. Cada proceso usuario tiene dos partes: la parte regular del usuario y la parte del sistema que se llama después de un señalamiento. Esta organización es similar a la que funciona en UNIX.

Todo proceso (y puerto) tiene un **identificador de protección** asociado a él. Si el proceso se bifurca, sus hijos heredan el mismo identificador de protección. Este identificador es simplemente una cadena de bits, y no tiene una semántica asociada a él conocida por el núcleo. Los identificadores de protección proporcionan un mecanismo que se utiliza para autenticación. Por ejemplo, el subsistema UNIX podría asignar un UID (identificador de usuario) a cada proceso y utilizar los identificadores de protección para implantar el UID.

9.2.2. Hilos

Cada proceso activo en Chorus tiene uno o más hilos que ejecutan código. Cada hilo tiene su propio contexto privado (es decir, su pila, contador del programa y registros), que se guarda cuando el hilo se bloquea en espera de cierto evento y se restaura cuando se reasume de nuevo el hilo. Un hilo está unido al proceso en el que fue creado, y no se puede mover a otro proceso.

Los hilos de Chorus son conocidos por el núcleo y planificados por éste, de modo que la creación y destrucción de ellos requiere llamadas al núcleo. Una ventaja de tener hilos del núcleo (opuestos a un paquete de hilos que se ejecuta por completo en el espacio del usuario, sin conocimiento del núcleo), es que cuando un hilo se bloquee en espera de cierto evento (por ejemplo, la llegada de un mensaje), el núcleo planifica otros hilos. Otra ventaja es la capacidad de ejecutar los hilos en diferentes CPU cuando se dispone de un multiprocesador. La desventaja de los hilos del núcleo es el costo adicional necesario para administrarlos. Por supuesto, los usuarios aún son libres de implantar un paquete de hilos a nivel usuario dentro de un hilo del núcleo.

Los hilos se comunican entre sí enviando y recibiendo mensajes. No importa si el emisor y el receptor están en el mismo proceso o si están en máquinas diferentes. La semántica de la comunicación es idéntica en todos los casos. Si dos hilos están en el mismo proceso, también se comunican mediante una memoria compartida, pero entonces el sistema no se puede reconfigurar posteriormente para ejecutarse con hilos en procesos diferentes.

Se distinguen los siguientes estados, que no son mutuamente excluyentes:

1. ACTIVO: — El hilo es lógicamente capaz de ejecutarse.
2. SUSPENDIDO: — El hilo se ha suspendido de manera intencional.
3. DETENIDO: — El proceso del hilo ha sido suspendido.
4. EN ESPERA: — El hilo está esperando que ocurra cierto evento.

Un hilo en el estado ACTIVO está en ejecución o espera su turno en un CPU libre. En ambos casos, está lógicamente bloqueado y se ejecuta. Un hilo en el estado SUSPENDIDO ha sido suspendido por otro hilo (o por sí mismo) que ha realizado una llamada al sistema solicitando al núcleo que suspenda al hilo. De manera análoga, cuando se realiza una llamada al núcleo ACTIVO se colocan en el estado DETENIDO hasta que se libera al proceso. Por último, cuando un hilo realiza una operación de bloqueo que no se completa de manera inmediata, el hilo se coloca en el estado de ESPERA hasta que ocurre el evento.

Un hilo puede estar en más de un estado al mismo tiempo. Por ejemplo, un hilo en estado SUSPENDIDO entra posteriormente al estado DETENIDO si su proceso es suspendido. Desde el punto de vista conceptual, cada hilo tiene tres bits independientes asociados con él, uno para cada uno de los estados SUSPENDIDO, DETENIDO EN ESPERA. El hilo se ejecuta sólo cuando los tres bits son iguales a cero.

Los hilos se ejecutan en el modo y el espacio de direcciones correspondientes a su proceso. En otras palabras, los hilos de un proceso del núcleo se ejecutan en modo núcleo, y los hilos de un proceso usuario se ejecutan en modo usuario.

El núcleo proporciona dos mecanismos de sincronización que utilizan los hilos. El primero es el semáforo tradicional (de conteo), con operaciones UP (o V) y DOWN (o P). Estas operaciones siempre se implantan mediante llamadas al núcleo, por lo que son caras. El segundo mecanismo es el mútex, que es en esencia un semáforo cuyos valores se restringen a 0 o 1. Los mútex sólo se utilizan para la exclusión mutua. Tienen la ventaja de que las operaciones que no provocan el bloqueo del proceso que realiza la llamada se pueden realizar por completo en el espacio de éste último, lo que ahorra el costo de una llamada al núcleo.

Un problema que ocurre en todo sistema basado en hilos es la forma de administrar los datos particulares de cada hilo, como su pila. Chorus resuelve este problema asignando dos **registros de software** especiales a cada hilo. Uno de ellos contiene un apuntador a los datos particulares del hilo cuando está en modo usuario. El otro contiene un apuntador a los datos particulares cuando el hilo ha hecho un señalamiento al núcleo y ejecuta una llamada al núcleo. Ambos registros son parte del estado del hilo, y se guardan y restauran junto con los registros del hardware cuando un hilo se detiene o inicia. Al mantener un índice de estos registros, un hilo puede tener acceso a datos que (por convención) no están disponibles a otros hilos en el mismo proceso.

9.2.3. Planificación

La planificación de los CPU se realiza mediante el uso de prioridades con base en los hilos. Cada proceso tiene prioridad y cada hilo tiene prioridad relativa dentro de su proceso. La prioridad absoluta de un hilo es la suma de la prioridad de su proceso y su prioridad relativa. El núcleo mantiene un registro de la prioridad de cada hilo en estado ACTIVO y ejecuta el que tiene la máxima prioridad absoluta. En un multiprocesador con k CPU, se ejecutan los k hilos con la máxima prioridad.

Sin embargo, para adecuarse a los procesos de tiempo real, se ha agregado una característica adicional al algoritmo. Se hace una distinción entre los hilos cuya prioridad está por arriba de cierto nivel y los hilos con una prioridad por debajo de éste. Los hilos de mayor prioridad, como A y B en la figura 9-7(a), no se separan con respecto del tiempo. Una vez que un hilo de este tipo comienza su ejecución, continúa hasta que voluntariamente se mueve al estado ACTIVO como resultado de la conclusión de la E/S o de que ocurra otro evento. En particular, no se detiene sólo porque se haya ejecutado durante mucho tiempo.

En contraste, en la figura 9-7(b), se ejecuta el hilo C, pero después de que ha consumido un quantum de tiempo de CPU, será colocado en el extremo de la cola de su prioridad, y el hilo D recibirá un quantum. En ausencia de competencia por el CPU, se alternarán indefinidamente.

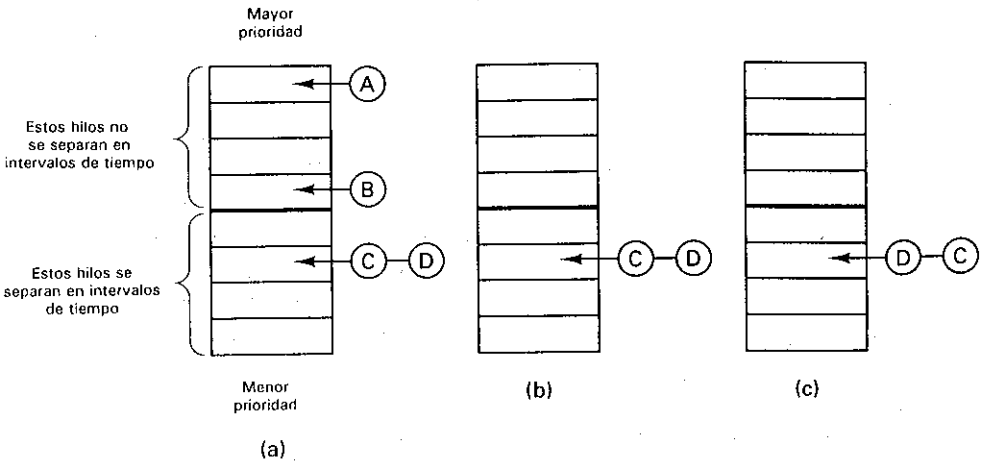


Figura 9.7. (a) El hilo *A* se ejecutará hasta terminar o bloquearse. (b)-(c) Los hilos *C* y *D* se alternarán, en modo round robin.

Este mecanismo proporciona la suficiente generalidad para la mayoría de las aplicaciones en tiempo real. Se dispone de llamadas del sistema para modificar las prioridades de los procesos y los hilos, de modo que las aplicaciones puedan decir al sistema los hilos que son más importantes y los que son menos importantes. Se dispone de algoritmos adicionales de planificación para soportar los procesos de tiempo real y de sistema, para el System V.

9.2.4. Señalamientos, excepciones e interrupciones

El software de Chorus distingue tres tipos de entradas al núcleo. Los **señalamientos** son llamadas intencionales al núcleo o a un subsistema para llamar a servicios. Los programas causan señalamientos llamando a un procedimiento de biblioteca para una llamada al sistema. El sistema soporta dos formas de manejo de señalamientos. En la primera, todos los señalamientos de un vector de señalamientos particular van a un hilo del núcleo que previamente ha anunciado su disposición a manejar ese vector. En la segunda forma, cada vector de señalamientos se une con un arreglo de hilos del núcleo, y el supervisor de Chorus utiliza el contenido de cierto registro para formar un índice del arreglo y elegir un hilo. El segundo mecanismo permite que todas las llamadas al sistema utilicen el mismo vector de señalamientos, de modo que se utilice el número de la llamada al sistema que va al registro para seleccionar un controlador.

Las **excepciones** son eventos inesperados que son causados por accidente, como la excepción de división entre cero, el desbordamiento de punto flotante, o un fallo de página. Es posible hacer que un hilo del núcleo sea llamado para manejar una excepción. Si el controlador puede terminar el procesamiento, regresa un código especial y concluye el manejo de la excepción. En caso contrario (o si no se asigna un controlador del núcleo), el núcleo suspende

el hilo que causó la excepción y envía un mensaje a un puerto especial para el manejo de excepciones asociado con el proceso del hilo. Por lo general, algún otro hilo espera un mensaje en este puerto y realizará la acción requerida por el proceso. Si no existe un puerto de excepción, el hilo fallido se elimina.

Las **interrupciones** son causadas por eventos asíncronos, como las marcas de reloj o la terminación de una solicitud de E/S. Éstas no necesariamente están relacionadas con algo realizado por el hilo activo, por lo que no es posible permitir que el proceso del hilo las maneje. En vez de ello, es posible arreglar de antemano que cuando ocurra una interrupción en cierto vector de interrupciones (es decir, un dispositivo específico), se cree un nuevo hilo del núcleo, de manera espontánea, para procesarlo. Si ocurre una segunda interrupción en el mismo vector antes de que el primero haya terminado, se crea un segundo hilo, etcétera. Todas las interrupciones de E/S, excepto el reloj, se manejan de esta manera. El reloj es revisado por el propio supervisor, pero puede configurarse para notificar a un hilo usuario si se necesita. Los hilos con interrupción pueden llamar solamente a un conjunto limitado de servicios del núcleo puesto que el sistema está en un estado desconocido cuando se inicia. Todo lo que pueden hacer, son operaciones sobre semáforos y mútex, o enviar minimensajes a minipuertos especiales.

9.2.5. Llamadas al núcleo para la administración de procesos

La mejor forma para ver lo que un núcleo o sistema operativo realiza es examinar su interfaz, es decir, las llamadas a sistema que proporciona a sus usuarios. En esta sección revisaremos las llamadas más importantes al sistema del núcleo de Chorus. Omitiremos las llamadas de menor importancia y las llamadas protegidas disponibles sólo para los hilos del núcleo.

Llamada	Descripción
actorCreate	Crea un nuevo proceso
actorDelete	Elimina un proceso
actorStop	Detiene un proceso, coloca sus hilos en el estado DETENIDO
actorStart	Reinicia un proceso desde el estado DETENIDO
actorPriority	Obtiene o configura la prioridad de un proceso
actorExcept	Obtiene o configura el puerto utilizado para el manejo de excepciones

Figura 9.8. Selección de llamadas a procesos soportadas por el núcleo de Chorus.

Comenzaremos con las llamadas a procesos, enumeradas en la figura 9-8. *ActorCreate* crea un proceso nuevo y regresa la posibilidad de ese proceso a quien hizo la llamada. El proceso nuevo hereda la prioridad, el identificador de protección y el puerto de excepción del proceso padre. Los parámetros especifican si el proceso nuevo será un proceso usuario, de sistema o del núcleo e indica el estado en el que comienza. Justo después de la creación,

el proceso nuevo está vacío, sin hilos ni regiones y con un puerto, el puerto por omisión. Observe que *actorCreate* representa un avance ortográfico con respecto de UNIX: "Create" se escribe con "e" al final.

La llamada *actorDelete* elimina un proceso. El proceso por eliminar se especifica mediante la posibilidad transferida como parámetro. *ActorStop* congela un proceso, colocando todos sus hilos en el estado DETENIDO. Los hilos sólo se ejecutan de nuevo cuando se hace una llamada *actorStart*. Un proceso se detiene a sí mismo. Estas llamadas se utilizan por lo general para depuración. Por ejemplo, si un hilo llega a un punto problemático, el depurador utiliza *actorStop* para detener los demás hilos del proceso.

La llamada *actorPriority* permite que un proceso lea la prioridad de otro proceso y, opcionalmente, que le dé un valor nuevo. Aunque Chorus es por lo general transparente con respecto de la posición, no es perfecto. Algunas llamadas, incluyendo ésta, sólo funcionan cuando el proceso objetivo está en la máquina desde la que se hizo la llamada. En otras palabras, no es posible obtener o configurar la prioridad de un proceso distante.

ActorExcept se utiliza para obtener o modificar el puerto de excepción para quien hizo la llamada o algún otro proceso para el que, quien hizo la llamada, tenga una posibilidad. También se utiliza para eliminar el puerto de excepción, en cuyo caso, si hay que enviar una excepción al proceso, éste se elimina.

El siguiente grupo de llamadas al núcleo se relaciona con los hilos, y se muestran en la figura 9-9. *ThreadCreate* y *threadDelete* crean y eliminan hilos en algún proceso (no necesariamente quien hizo la llamada), respectivamente. Los parámetros de *threadCreate* especifican el nivel de privilegio, el estado inicial, la prioridad, el punto de entrada y el apuntador a la pila.

Llamada	Descripción
threadCreate	Crea un nuevo hilo
threadDelete	Elimina un hilo
threadSuspend	Suspende un hilo
threadResume	Reinicia un hilo suspendido
threadPriority	Obtiene o configura la prioridad de un hilo
threadLoad	Obtiene el apuntador al contexto del hilo
threadStore	Configura el apuntador al contexto del hilo
threadContext	Obtiene o configura el contexto de ejecución de un hilo

Figura 9.9. Selección de llamadas a hilos soportadas por el núcleo de Chorus.

ThreadSuspend y *threadResume* detienen y reinician los hilos en el proceso objetivo. *ThreadPriority* regresa la prioridad relativa actual del hilo objetivo y opcionalmente lo configura como valor dado como parámetro.

Nuestras últimas tres llamadas se utilizan para controlar el contexto privado de un hilo. Las llamadas *threadLoad* y *threadStore* cargan y configuran el registro del contexto actual

del software, respectivamente. Este registro apunta al contexto del hilo, incluyendo sus variables particulares. La llamada *threadContext* copia de manera opcional el contexto anterior del hilo a un buffer, y opcionalmente configura el nuevo contexto de otro buffer.

Las operaciones de sincronización aparecen en la figura 9-10. Se dispone de llamadas para iniciar, adquirir y liberar mútex y semáforos. Funcionan de la manera usual.

Llamada	Descripción
mutexInit	Inicia un mútex
mutexGet	Intenta adquirir un mútex
mutexRel	Libera un mútex
semInit	Inicia un semáforo
semP	Realiza un DOWN sobre un semáforo
semV	Realiza un UP sobre un semáforo

Figura 9.10. Selección de llamadas de sincronización soportadas por el núcleo de Chorus.

9.3. ADMINISTRACIÓN DE MEMORIA EN CHORUS

La administración de la memoria en Chorus recupera varias ideas de Mach. Sin embargo, también contiene ciertas ideas no presentes en Mach. En esta sección describiremos los conceptos básicos y la forma de utilizarlos.

9.3.1. Regiones y segmentos

Los conceptos principales detrás de la administración de memoria en Chorus son las regiones y los segmentos. Una **región** es un rango adyacente de direcciones virtuales, por ejemplo de 1 024 a 6 143. En teoría, una región puede comenzar o terminar en cualquier dirección virtual, pero para hacer algo útil, una región debe estar alineada con respecto de las páginas y tener una longitud igual a cierto número entero de páginas. Todos los bytes de una región tienen las mismas características de protección (por ejemplo, exclusivo para lectura). Las regiones son una propiedad de los procesos y todos los hilos de un proceso ven las mismas regiones. Dos regiones del mismo proceso no deben traslaparse.

Un **segmento** es una colección adyacente de bytes que reciben el nombre y protección de una posibilidad. Los archivos y las áreas de intercambio son los tipos más comunes de segmentos. Los segmentos se pueden leer o escribir en ellos utilizando llamadas al sistema que proporcionen la posibilidad, el desplazamiento, el número de bytes, el buffer y la dirección de transferencia del segmento. Estas llamadas se utilizan para realizar las operaciones tradicionales de E/S sobre los archivos.

Sin embargo, otra posibilidad consiste en asociar segmentos a regiones, como se muestra en la figura 9-4. No es necesario que un segmento tenga el tamaño exacto de su región. Si el segmento es mayor que la región, sólo una parte del segmento será visible en el espacio

de direcciones, aunque se puede cambiar de porción visible mediante una reasociación. Si el segmento es menor que la región, el resultado de la lectura de una dirección no asociada es trabajo del asociador. Por ejemplo, puede ejecutar una excepción, regresar 0 o ampliar el segmento, conforme lo desee.

Los segmentos asociados son paginados por lo general según la demanda (a menos que esta característica no esté activa, por ejemplo, para los programas de tiempo real). Cuando un proceso hace referencia por vez primera a un segmento recién asociado, ocurre un fallo de página y la página del segmento correspondiente a la dirección de referencia se recupera y se reinicia la instrucción fallida. De esta forma, se puede implantar la memoria virtual ordinaria, y, además, un proceso puede hacer que uno o más archivos sean visibles en su espacio de direcciones virtuales, de modo que tenga un acceso directo a ellos en vez de tener que leerlos o escribir en ellos mediante llamadas al sistema.

El núcleo soporta segmentos especiales de E/S para al acceso a los registros de E/S de la máquina en máquinas con registros de dispositivos asociados con memoria. Al utilizar estos segmentos, los hilos del núcleo pueden realizar E/S leyendo o escribiendo de manera directa en la memoria.

9.3.2. Asociadores

Chorus soporta paginadores externos del estilo de Mách, llamados **asociadores**. Cada asociador controla uno o más segmentos que son asociados con regiones. Un segmento se asocia con varias regiones, incluso en diferentes espacios de direcciones al mismo tiempo, como se muestra en la figura 9-11. En este caso, los segmentos *S1* y *S2* se asocian ambos a los procesos *A* y *B*, en la misma máquina, pero en direcciones diferentes. Si el proceso *A* escribe en la dirección *A1* cambia la primera palabra de *S1*. Si posteriormente, el proceso *B* lee *B1* también obtiene el valor escrito por *A*. Además, si *S1* es un archivo, cuando ambos procesos terminan, el cambio se hará en el archivo en disco.

El administrador de la memoria virtual en cada núcleo mantiene un caché de página y lleva un registro de la página correspondiente a cada segmento. Las páginas en el caché local pertenecen a segmentos con nombre, como archivo, o sin nombre, como las áreas de intercambio. El núcleo lleva un registro de las páginas limpias y las sucias. Puede descartar las páginas limpias a voluntad, pero regresa las páginas sucias a los asociadores adecuados para reclamar su espacio.

Un protocolo entre el núcleo y el asociador determina el flujo de las páginas en ambas direcciones. Cuando ocurre un fallo de página, el núcleo verifica si la página necesaria está en el caché. Si no está, el núcleo envía un mensaje al asociador que controla el segmento de la página solicitando ésta (y posiblemente también las páginas adyacentes). El hilo fallido se suspende entonces, hasta que llega la página.

Cuando el asociador recibe la solicitud, verifica si la página necesaria está en su caché (en su espacio de direcciones). Si no, envía un mensaje al hilo que controla el disco para realizar una E/S y recuperar la página. Cuando llega la página (o si ya estaba presente), el

asociador informa de esto al núcleo, que entonces acepta la página, ajusta las tablas de página MMU y reasume el hilo fallido.

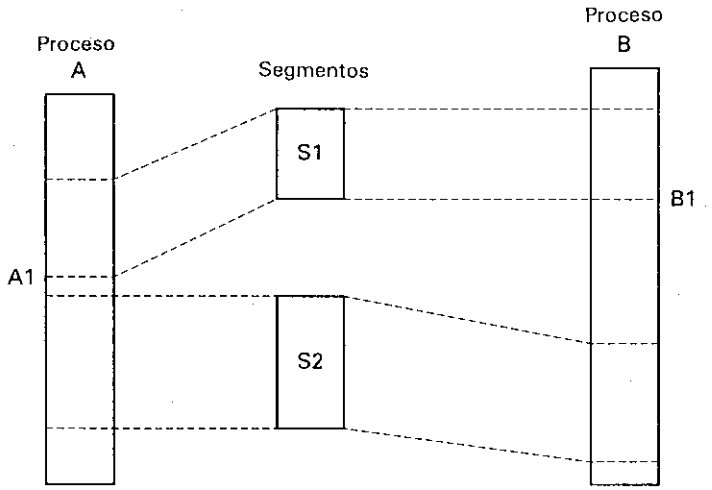


Figura 9.11. Los segmentos se pueden asociar con varios espacios de direcciones al mismo tiempo.

El asociador también toma la iniciativa y pide al núcleo que le regrese las páginas sucias. Cuando el núcleo las regresa, el asociador conserva algunas de ellas en su caché y escribe otras al disco. Se dispone de varias llamadas para que el asociador especifique las páginas que desea regresar. La mayor parte de los asociadores no llevan un registro de la cantidad de marcos de página que ocupan sus segmentos, puesto que el núcleo es libre de descartar las páginas limpias y regresar páginas sucias a sus asociadores cuando el espacio se compacta.

El mismo mecanismo de ocultamiento y administración se utiliza para las páginas que son parte de los segmentos asociados y para las páginas que son leídas y escritas utilizando instrucciones explícitas de E/S de segmentos. Este método garantiza que si un proceso modifica una página asociada y escribe en ella, de inmediato otro proceso intenta leer el archivo del que es parte, el segundo proceso obtendrá los nuevos datos, puesto que sólo existe una copia de la página en memoria.

9.3.3. Memoria compartida distribuida

Chorus soporta la memoria compartida distribuida paginada en el estilo de IVY, como analizamos en el capítulo 6. Utiliza un algoritmo descentralizado dinámico, lo que significa que diferentes administradores llevan un registro de diferentes páginas, y el administrador de una página cambia si la página se mueve a través del sistema (para páginas de escritura).

La unidad para compartir entre las diversas máquinas es el segmento. Los segmentos se dividen en fragmentos de una o más páginas. En cualquier instante, cada fragmento es

exclusivo para lectura, y en potencia está presente en varias máquinas; o bien, sirve para lectura y escritura, y sólo está presente en una máquina.

9.3.4. Llamadas al núcleo para la administración de memoria

La administración de memoria en Chorus soporta 26 llamadas al sistema diferentes, más algunas otras llamadas del núcleo a los asociadores. En esta sección describiremos de forma breve sólo las más importantes. Las llamadas que describiremos se relacionan con la administración de regiones (prefijo *rgn*), la administración de segmentos (prefijo *sg*) y las llamadas a los asociadores (prefijo *Mp*; no *mp*, que se utiliza para las llamadas a los minipuertos, descritas más adelante). Las llamadas no descritas aquí se refieren a la administración de los cachés locales (prefijo *lc*) y la memoria virtual (prefijo *vm*).

En la figura 9-12 aparece una selección de las llamadas para la administración de regiones. *RgnAllocate* especifica una región proporcionando la posibilidad de un proceso, una dirección inicial, un tamaño y varias opciones. Si no existen conflictos con otras regiones ni otros problemas, se crea la región. Las opciones incluyen la iniciación de la región con bytes nulos, la configuración de los bits lectura/escritura/ejecutable y los arreglos para que no esté paginada. *RgnFree* regresa una región previamente asignada de manera que su porción del espacio de direcciones esté libre para su asignación a otra región o regiones.

Llamada	Descripción
rgnAllocate	Asigna una región de memoria y configura sus propiedades
rgnFree	Libera una región previamente asignada
rgnInit	Asigna una región y la llena mediante un segmento dado
rgnSetInherit	Configura las propiedades hereditarias de una región
rgnSetPaging	Configura las propiedades de paginación de una región
rgnSetProtect	Configura las opciones de protección de una región
rgnStat	Obtiene las estadísticas asociadas con una región

Figura 9.12. Selección de llamadas soportadas por el núcleo de Chorus para la administración de regiones.

RgnInit es similar a *rgnAllocate*, excepto que, después de asignar la región, también se llena a partir de un segmento cuya posibilidad es un parámetro de la llamada. También existen varias otras llamadas que son similares a *rgnInit*, que llenan la región de maneras diferentes.

Las siguientes tres llamadas modifican las propiedades de una región existente de varias formas. *RgnSetInherit* se relaciona con la posibilidad de que una región pueda copiarse posteriormente, y especifica si la copia tendrá sus propias páginas o compartirá las páginas de la región original. *RgnSetPaging* es utilizada principalmente por los procesos interesados en proporcionar una respuesta en tiempo real y no pueden tolerar fallos de página o los intercambios. También indica lo que debe realizarse si el proceso intenta asignar una región

no intercambiable ni paginable que está ya configurada, pero sin que se disponga de la memoria suficiente. *RgnSetProtect* modifica los bits lectura/escritura/ejecutable asociados con una región y también puede hacer que una región sólo sea accesible al núcleo. Por último, *rgnStat* regresa al proceso que hizo la llamada el tamaño de una región y más información.

Las llamadas para E/S de segmentos aparecen en la figura 9-13. *sgRead* y *sgWrite* son las llamadas básicas de E/S que permiten a un proceso leer o escribir en un segmento. El segmento queda especificado por un descriptor. También se proporciona el desplazamiento y el número de bytes, al igual que la dirección del buffer del que se desea copiar. *SgStat* permite a quien hizo la llamada, que por lo general es un asociador, solicitar la información estadística acerca de un caché de página. *SgFlush* (y varias llamadas relacionadas con ésta) permiten a los asociadores pedir al núcleo que les envíe páginas para guardarlas en el espacio de direcciones del asociador y escribirlas de nuevo en sus segmentos del disco. Este mecanismo se necesita, por ejemplo, para garantizar que un grupo de asociadores que juntos soportan la memoria compartida distribuida puedan eliminar las páginas en las que se escriba de todas las máquinas excepto una. Las demás llamadas de este grupo se refieren al hecho de cerrar y abrir las páginas individuales en memoria y obtener los tamaños y demás información relativa al sistema de paginación.

Llamada	Descripción
sgRead	Lee datos de un segmento
sgWrite	Escribe datos a un segmento
sgStat	Solicita información relativa a un caché de página
sgFlush	Solicitud de un asociador al núcleo pidiendo páginas sucias

Figura 9.13. Selección de llamadas relacionadas con los segmentos.

Las llamadas de la figura 9-14 son llamadas al asociador, ya sea desde el núcleo o desde un programa de aplicación que solicita al asociador que realice cierta labor para quien hizo la llamada. La primera, *MpCreate*, se utiliza cuando el núcleo o un programa intercambia un segmento y necesita asignarle un espacio en disco. El asociador responde asignando un nuevo segmento en disco y regresando una posibilidad para él.

Llamada	Descripción
MpCreate	Solicitud para crear un segmento para intercambio
MpRelease	Solicitud para liberar un segmento creado con anterioridad
MpPullIn	Solicitud de una o más páginas
MpPushOu	Solicitud para que un asociador acepte una o más páginas

Figura 9.14. Llamadas a un asociador.

La llamada *MpRelease* regresa un segmento creado mediante *MpCreate*. *MpPullIn* es utilizada por el núcleo para adquirir datos de un segmento recién creado o existente. Se

pide al asociador que responda enviando un mensaje que contiene las páginas necesarias. Si se utiliza una programación MMU hábil, no es necesario copiar físicamente las páginas.

La llamada *MpPushOut* sirve para las transferencias en sentido opuesto, del núcleo al asociador, ya sea en respuesta a una llamada *sgFlush* (o alguna similar), o cuando el núcleo intercambia un segmento por su cuenta. Aunque la lista de llamadas descritas arriba no está completa, proporciona una imagen razonable del funcionamiento de la administración de la memoria en Chorus.

9.4. COMUNICACIÓN EN CHORUS

El paradigma básico de comunicación en Chorus es la transferencia de mensajes. Durante la era de la versión 1, cuando la investigación se centró en los multiprocesadores, se consideraba el uso de la memoria compartida como el paradigma de comunicación, pero se rechazó por no ser lo bastante general. En esta sección analizaremos los mensajes, los puertos y las operaciones de comunicación, concluyendo con un resumen de las llamadas al núcleo disponibles para la comunicación.

9.4.1. Mensajes

Cada mensaje contiene un encabezado (sólo para uso interno del micronúcleo), una parte fija opcional y un cuerpo opcional. El encabezado identifica la fuente y destino y contiene varios identificadores de protección y banderas. La parte fija, si está presente, siempre tiene una longitud de 64 bytes y está por completo bajo el control del usuario. El cuerpo tiene un tamaño variable, con un máximo de 64K bytes, y también está por completo bajo el control del usuario. Desde el punto de vista del núcleo, la parte fija y el cuerpo son arreglos de bytes no tipificados, en el sentido de que el núcleo no se preocupa por lo que contienen.

Cuando se envía un mensaje a un hilo en una máquina diferente, siempre se copia. Sin embargo, cuando se envía a un hilo en la misma máquina, existe una opción entre copiarlo en realidad o simplemente asociarlo con el espacio de direcciones del receptor. En el segundo caso, si el receptor escribe sobre una página asociada, se crea una copia genuina en ese punto (es decir, es el mecanismo de copiado durante la escritura de Mach). Cuando un mensaje no tiene un número entero de páginas pero el mensaje es asociado, sólo se perderán algunos datos detrás (o delante) del buffer cuando se asocie la última (o primera) página.

Otra forma de mensaje es el **minimensaje**, que sólo se utiliza entre los procesos del núcleo para los mensajes breves de sincronización, por lo general para indicar la ocurrencia de una interrupción. Los minimensajes se envían a **minipuertos** especiales de bajo costo.

9.4.2. Puertos

Los mensajes se envían a los puertos, cada uno de los cuales contiene un espacio para guardar cierto número de mensajes. Si se envía un mensaje a un puerto lleno, el emisor se

suspende hasta que se dispone del espacio suficiente. Al crear un puerto, el proceso que hizo la llamada recibe un identificador único y un identificador local. El primero se puede enviar a otros procesos de modo que éstos puedan enviar mensajes al puerto. El segundo se utiliza dentro del proceso para hacer referencia de manera directa al puerto. Sólo los hilos del proceso que posee en cierto momento al puerto pueden ser leídos de éste (los puertos pueden emigrar).

Al crear un proceso, obtiene de forma automática un puerto por omisión, utilizado por el núcleo para enviar mensajes de excepción. También crea el número de puertos adicionales necesarios. Estos puertos adicionales (no así el puerto por omisión) se mueven a otros procesos, incluso en otras máquinas. Cuando se mueve un puerto, todos los mensajes que se encuentran se mueven con él. El movimiento de los puertos es útil, por ejemplo, cuando un servidor en una máquina que se apaga por mantenimiento permite que otro servidor en una máquina diferente realice su trabajo. De esta manera, los servicios se mantienen de manera transparente, incluso cuando las máquinas servidor se apaguen y posteriormente vuelvan a trabajar.

Chorus proporciona una forma de reunir varios puertos en un **grupo de puertos**. Para esto, un proceso crea primero un grupo de puertos vacío y regresa una posibilidad para él. Por medio de esta posibilidad, puede agregar puertos al grupo, y también puede utilizarla para eliminar posteriormente puertos del grupo. Un puerto puede estar presente en varios grupos de puerto, como se muestra en la figura 9-15.

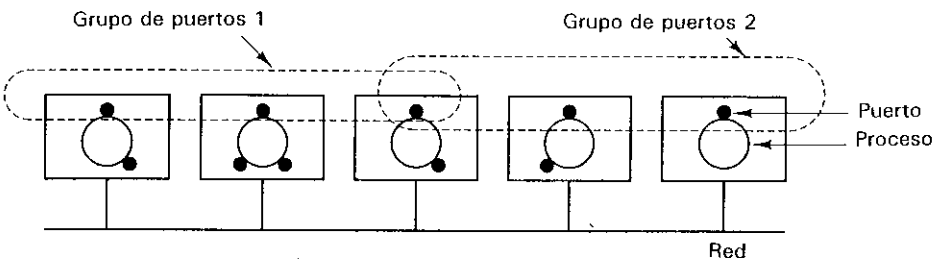


Figura 9.15. Un puerto puede ser miembro de varios grupos de puertos.

Los grupos se utilizan por lo general para proporcionar servicios reconfigurables. En un principio, cierto conjunto de servidores pertenecen al grupo, el cual proporciona cierto servicio. Los clientes envían mensajes al grupo sin tener que conocer cuáles servidores están disponibles para realizar el trabajo. Posteriormente, se pueden unir al grupo otros servidores y los anteriores salen sin perturbar el servicio y sin que los clientes tengan conciencia de que el sistema ha sido reconfigurado.

9.4.3. Operaciones de comunicación

Chorus proporciona dos tipos de operaciones de comunicación: envío asíncrono y RPC. El envío asíncrono permite que un hilo sólo envíe un mensaje a un puerto. No existe garantía

de que el mensaje llegue a su destino y no existe una notificación si algo sale mal. Ésta es la forma más pura de datagrama y permite que los usuarios construyan patrones de comunicación arbitrarios utilizando Chorus.

La otra operación de comunicación es la RPC. Cuando un proceso ejecuta una operación de RPC, se bloquea en forma automática hasta que llega la respuesta o expira el cronómetro de la RPC, en cuyo momento se elimina el bloqueo del emisor. Se garantiza que el mensaje que elimina el bloqueo del emisor es la respuesta a la solicitud. Cualquier mensaje que no incluya el identificador de transacción de la RPC se guardará en el puerto para su consumo posterior pero no despertará al emisor.

Las RPC utilizan una semántica "a lo más una vez", lo que significa que en el caso de una comunicación no recuperable o una falla de procesamiento, el sistema garantiza que una RPC regresará un código de error en vez de intentar la ejecución de una operación más de una vez.

También es posible enviar un mensaje a un grupo de puertos. Se dispone de varias opciones, como se muestra en la figura 9-16. Estas opciones determinan la cantidad de mensajes que se envían y a cuáles puertos.

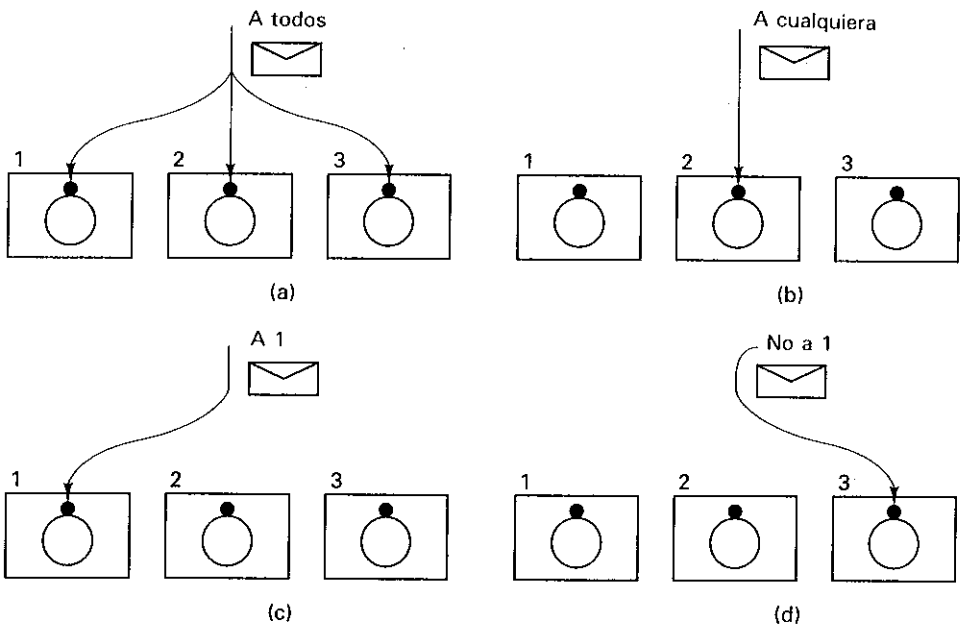


Figura 9.16. Opciones para el envío hacia un grupo de puertos. (a) Envío a todos los miembros. (b) Envío a cualquier miembro. (c) Envío a un puerto en el mismo sitio de un puerto dado. (d) Envío a un puerto que no está en un sitio específico.

La opción (a) de la figura 9-16 envía el mensaje a todos los puertos del grupo. Para un almacenamiento muy confiable, un proceso podría hacer que todos los servidores de archivos guarden ciertos datos. La opción (b) envía esto sólo a uno, pero permite que el sistema

elija cuál. Cuando un proceso sólo necesita cierto servicio, como la fecha actual, pero no se preocupa por el sitio de donde proviene, esta opción es la mejor, ya que el sistema selecciona entonces la forma más eficiente de proporcionar el servicio.

Las otras dos opciones también se envían sólo a un puerto, pero limitan las opciones del sistema. En (c), quien hace la llamada puede especificar que el puerto está en un sitio específico; por ejemplo, para balancear la carga del sistema. La opción (d) dice que se puede utilizar cualquier puerto que *no* esté en el sitio especificado. Un uso de esta opción podría obligar a la creación de una copia de respaldo de un archivo en una máquina diferente de la correspondiente a la copia primaria.

Los envíos a grupos de puertos utilizan el envío asíncrono. Los envíos de transmisión (es decir, a todos los miembros) no son controlados por el flujo. Si se requiere un control del flujo, debe ser proporcionado por el usuario.

Para recibir un mensaje, un hilo hace una llamada al núcleo indicando el puerto donde desea recibirlo. Si se dispone de un mensaje, la parte fija de éste se copia en el espacio de direcciones de quien hizo la llamada, y el cuerpo, si existe, se copia o se asocia, según las opciones. Si no se dispone de un mensaje, el hilo se suspende hasta que llega un mensaje y expira un cronómetro especificado por el usuario.

Además, un proceso especifica lo que desea recibir de alguno de los puertos que posee, sin importarle cuál. Esta opción puede refinarse aún más, deshabilitando algunos de los puertos, en cuyo caso sólo los puertos habilitados satisfacen la solicitud. Por último, los puertos tienen asignadas prioridades, lo que significa que si más de un puerto habilitado tiene un mensaje, se selecciona el puerto habilitado con la máxima prioridad. Los puertos pueden ser habilitados o inhabilitados de manera dinámica, y se modifican sus prioridades a voluntad.

9.4.4. Llamadas al núcleo para la comunicación

Las llamadas para la administración de puertos aparecen en la figura 9-17. Las primeras cuatro son directas, y permiten crear, destruir, habilitar o inhabilitar puertos. La última especifica un puerto y un proceso. Cuando concluye la llamada, el puerto ya no pertenece a su propietario original (que no tiene que ser quien hizo la llamada), sino al proceso objetivo. Sólo él podrá leer ahora los mensajes del puerto.

Llamada	Descripción
portCreate	Crea un puerto y regresa su posibilidad
portDelete	Destruye un puerto
portEnable	Habilita un puerto de modo que sus mensajes cuenten en una recepción de todos los puertos
portDisable	Inhabilita un puerto
portMigrate	Mueve un puerto a otro proceso

Figura 9.17. Selección de llamadas para la administración de puertos.

Existen tres llamadas para la administración de grupos de puertos; éstas se enumeran en la figura 9-18. La primera, *grpAllocate*, crea un nuevo grupo de puertos y regresa una posibilidad para el grupo a quien hizo la llamada. Por medio de esta posibilidad, quien hizo la llamada o cualquier otro proceso que posteriormente adquiera la posibilidad puede agregar o eliminar puertos del grupo.

Llamada	Descripción
grpAllocate	Crea un grupo de puertos
grpPortInsert	Agrega un nuevo puerto a un grupo de puertos existentes
grpPortRemove	Elimina un puerto de un grupo de puertos

Figura 9.18. Llamadas relacionadas con los grupos de puertos.

Nuestro último grupo de llamadas al núcleo maneja el envío y la recepción reales de los mensajes. Se enumeran en la figura 9-19. *IpcSend* envía un mensaje de manera asíncrona a un puerto o grupo de puertos dado. *IpcReceive* se bloquea hasta que llega un mensaje de un puerto dado. Este mensaje podría haber sido enviado de manera directa al puerto, a un grupo de puertos de los que es miembro el puerto dado, o a todos los puertos habilitados (suponiendo que el puerto especificado está habilitado). Hay que proporcionar una dirección en la cual copiar la parte fija, aunque la dirección del cuerpo es opcional, ya que el tamaño no siempre se conoce de antemano. Si no se proporciona un buffer para el cuerpo, se puede ejecutar la llamada *ipcGetData* para adquirir el cuerpo del núcleo (el tamaño ya se conoce, pues regresa con la llamada *IpcReceive*). Se puede enviar un mensaje de respuesta mediante *ipcReply*. Por último, *ipcCall* realiza una llamada a un procedimiento remoto.

Llamada	Descripción
ipcSend	Envía un mensaje de manera asíncrona
ipcReceive	Se bloquea hasta que llega un mensaje
ipcGetData	Obtiene el cuerpo del mensaje actual
ipcReply	Envía una respuesta al mensaje actual
ipcCall	Realiza una llamada a un procedimiento remoto

Figura 9.19. Selección de llamadas para comunicación.

9.5. EMULACIÓN DE UNIX EN CHORUS

Aunque parecerse a UNIX no era siquiera uno de los objetivos originales de Chorus (tenía una interfaz por completo diferente y fue escrito en UCSD Pascal interpretado), con la aparición de la versión 3, la compatibilidad con UNIX se convirtió en uno de los objetivos principales. El punto de vista adoptado en relación con esto fue hacer que el núcleo fuese neutral con respecto del sistema operativo, pero con un subsistema que permite la ejecución de los programas binarios de UNIX dentro de él sin modificarlos. Este subsistema consta

en parte de un nuevo código escrito por los implantadores de Chorus, y en parte del propio código del sistema V de UNIX, con la autorización de UNIX System Laboratories. En esta sección describiremos al UNIX de Chorus y su implantación. El subsistema UNIX se llama **MiX**, que representa **UNIX Modular**.

9.5.1. Estructura de un proceso en UNIX

Un proceso de UNIX se ejecuta como proceso usuario en Chorus, sobre el subsistema UNIX. Como tal, tiene todas las características de un proceso de Chorus, aunque los programas de UNIX existentes no utilizarán todas. Los segmentos de texto estándar, datos y de la pila son implantados mediante tres segmentos Chorus asociados.

Globalmente, el funcionamiento de Chorus no queda oculto al sistema de tiempo de ejecución y la biblioteca de UNIX (aunque está oculto al *programa*). Por ejemplo, cuando el programa desea abrir un archivo, el procedimiento de biblioteca *open* llama al subsistema, el que por último obtiene la posibilidad del archivo. Guarda esta posibilidad de manera interna y la utiliza cuando el proceso usuario llama a *read*, *write*, y a otros procedimientos que trabajan con el archivo abierto.

Los archivos abiertos no son los únicos conceptos de UNIX que se asocian a los recursos de Chorus. Los directorios abiertos (incluyendo el directorio de trabajo y el directorio raíz), los dispositivos abiertos, los entubamientos, y los segmentos en uso se representan internamente mediante posibilidades para los recursos correspondientes de Chorus. Las operaciones sobre todos ellos son realizadas por el subsistema UNIX, transfiriendo la posibilidad al servidor adecuado. Para mantener la compatibilidad binaria con UNIX, el proceso usuario nunca debe ver las posibilidades.

9.5.2. Extensiones a UNIX

Chorus proporciona procesos de UNIX con muchas extensiones, para facilitar la programación distribuida. La mayor parte de estos son sólo propiedades estándar de Chorus que se hacen visibles. Por ejemplo, los procesos de UNIX crean y destruyen nuevos hilos mediante el paquete de hilos de Chorus. Estos hilos se ejecutan de manera casi paralela (en un multiprocesador, en realidad se ejecutan en paralelo).

Puesto que los hilos de Chorus son controlados por el núcleo, cuando un hilo se bloquea, por ejemplo, en una llamada al sistema, el núcleo se planifica y ejecuta otro hilo del mismo proceso. Sin embargo, por lo general no es posible que dos hilos del mismo proceso ejecuten la mayor parte de las llamadas al sistema de manera simultánea, pues el antiguo código UNIX que controla las llamadas al sistema no es reentrante. En la mayoría de los casos, el segundo hilo se suspende de manera transparente antes de iniciar la llamada al sistema, hasta que concluye la primera.

La adición de hilos ha provocado una reflexión en cuanto al manejo de las señales. En vez de que todos los controladores de señales sean comunes a todo el proceso, los síncronos (causados por una acción del hilo) son asociados con hilos específicos, mientras que los

asíncronos (como cuando el usuario oprime la tecla DEL) son comunes a todo el proceso. Por ejemplo, la llamada al sistema ALARM hace que el hilo de la llamada se interrumpa después del número indicado de segundos. Los demás hilos no son afectados.

De manera similar, un hilo capta los desbordamientos de punto flotante y las excepciones similares, mientras que otro puede ignorarlas y un tercero puede no captarlas (lo que significa que será eliminado si ocurre alguna).

Otras señales, por naturaleza, no son específicas de un hilo. Una señal KILL enviada por otro proceso, o SIGINT o SIGQUIT del teclado se envían a todos los hilos. Cada hilo puede captarla o ignorarla, a voluntad. Si ningún hilo capta o ignora una señal, todo el proceso es eliminado.

Las señales son controladas por el propio proceso. A cada proceso de UNIX se le asocia un hilo de control dentro del subsistema UNIX que escucha al puerto de excepción. Cuando se activa una señal, este hilo por lo regular dormido se despierta. El hilo obtiene entonces el mensaje enviado al puerto de control, examina sus tablas internas y realiza la acción requerida, interrumpiendo el hilo o hilos adecuados.

Una tercera área en la que se ha extendido la funcionalidad de UNIX es la de hacerlo distribuido. Es posible que un proceso realice una llamada al sistema para indicar que los nuevos procesos no serán creados en la máquina local, sino en una máquina remota dada. Cuando un proceso nuevo se bifurca, comienza en la misma máquina que el proceso padre, pero cuando ejecuta una llamada al sistema EXEC, el proceso nuevo inicia en la máquina remota.

Los procesos usuarios que utilizan el subsistema UNIX pueden crear puertos y grupos de puertos, así como enviar y recibir mensajes, como cualquier otro proceso de Chorus. También pueden crear regiones y asociar segmentos en ellas. En general, los procesos de UNIX disponen de todas las facilidades de Chorus relacionadas con la administración de procesos, la administración de la memoria y la comunicación entre procesos.

9.5.3. Implantación de UNIX en Chorus

La implantación del subsistema UNIX se construye a partir de cuatro componentes principales: el **administrador de procesos**, el **administrador de objetos**, el **administrador de flujos** y el **administrador de la comunicación entre procesos**, como se muestra en la figura 9-20. Cada uno de ellos tiene una función específica en la emulación. El administrador de procesos captura las llamadas al sistema y realiza la administración de los procesos. El administrador de objetos controla las llamadas al sistema de archivos y también realiza la actividad de paginación. El administrador de flujos se encarga de la E/S. El administrador de la comunicación entre procesos realiza la IPC del System V. El administrador de procesos tiene un nuevo código. Los demás están tomados en gran medida del propio UNIX, para minimizar el trabajo del diseñador y maximizar la compatibilidad. Estos cuatro administradores pueden controlar varias sesiones cada uno, de modo que sólo uno de ellos esté presente en un sitio dado, sin importar el número de usuarios conectados a él.

En el diseño original, los cuatro procesos debían poder ejecutarse en modo núcleo o en modo usuario. Sin embargo, al agregarse un código con más privilegios, esto se volvió más

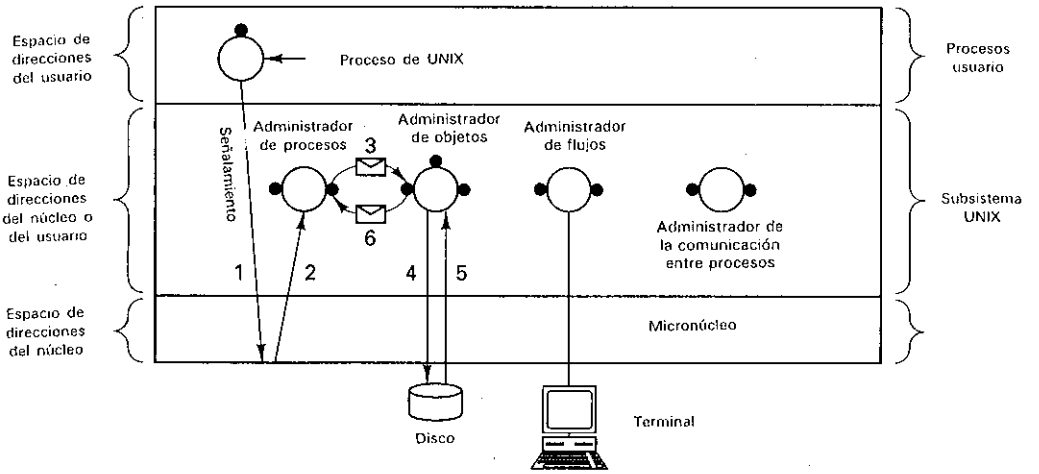


Figura 9.20. La estructura del subsistema UNIX de Chorus. Los números muestran la serie de pasos implicados en una operación de archivo común.

difícil. En la práctica, normalmente se ejecutan todos en modo núcleo, lo que también es necesario para proporcionar un rendimiento aceptable.

Para ver la relación entre las partes, examinaremos la forma en que se procesan las llamadas al sistema. En el momento de iniciación del sistema, el administrador de procesos indica al núcleo que desea manejar los números de señalamientos utilizados por el UNIX estándar de AT&T para realizar las llamadas al sistema (para obtener la compatibilidad binaria). Cuando un proceso de UNIX ejecuta después una llamada al sistema mediante un señalamiento al núcleo, como se indica en (1) en la figura 9-21, un hilo del administrador de procesos obtiene el control. Este hilo actúa como si fuera igual al hilo que hace la llamada, de manera análoga a la forma en que se realizan las llamadas al sistema en los sistemas UNIX tradicionales. Según la llamada al sistema, el administrador de procesos puede ejecutar la llamada solicitada por sí mismo; o, como se muestra en la figura 9-20, enviar un mensaje al administrador de objetos pidiéndole que realice el trabajo. Para las llamadas de E/S, se llama al administrador de flujos. Para las llamadas de IPC, se utiliza el administrador de comunicación.

En este ejemplo, el administrador de objetos realiza una operación en disco y después envía una respuesta de regreso al administrador de procesos, que configura entonces el valor adecuado de retorno y reinicia el proceso UNIX bloqueado.

El administrador de procesos

El administrador de procesos es el elemento principal en la emulación. Captura todas las llamadas al sistema y decide qué hacer con ellas. También controla la administración de procesos (incluyendo la creación y conclusión de los mismos), las señales (su generación

y recepción) y la asignación de nombres. Cuando llega una llamada al sistema que no puede manejar, el administrador de procesos realiza una RPC con el administrador de objetos o el administrador de flujos. También puede realizar llamadas al núcleo para realizar su trabajo; por ejemplo, al nacer un proceso nuevo, el núcleo realiza la mayor parte del trabajo.

Si el proceso nuevo debe crearse en una nueva máquina, se necesita un mecanismo más complejo, como se muestra en la figura 9-21. En este caso, el administrador de procesos captura la llamada al sistema, pero en vez de pedir al núcleo local que cree un proceso nuevo de Chorus, realiza una RPC con el administrador de procesos sobre la máquina objetivo, que es el paso (3) de la figura 9-21. El administrador de procesos remoto pide entonces a su núcleo que cree el proceso, como se muestra en el paso 4. Cada administrador de procesos tiene un puerto dedicado al cual se dirigen las RPC de los otros administradores de procesos.

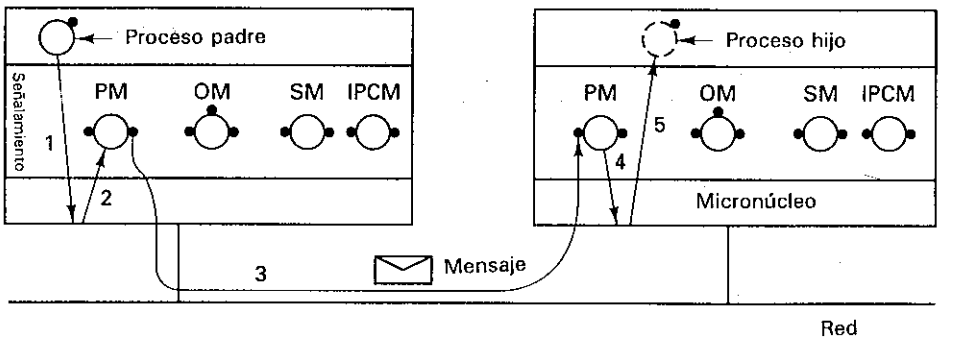


Figura 9.21. Creación de un proceso en una máquina remota. No se muestra el mensaje de respuesta.

El administrador de procesos tiene varios hilos. Por ejemplo, cuando un hilo usuario activa una alarma, un hilo del administrador de procesos se duerme hasta que el cronómetro expira. Entonces, envía un mensaje al puerto de excepción que pertenece al proceso del hilo.

El administrador de procesos también controla los identificadores únicos. Mantiene tablas que asocian los UI con los recursos correspondientes.

El administrador de objetos

El administrador de objetos controla los archivos, el espacio de intercambio, y otras formas de información tangible. También puede contener al controlador del disco. Además de su puerto de excepción, el administrador de objetos tiene un puerto para la recepción de solicitudes de paginación y un puerto para recibir solicitudes de administradores de procesos locales o remotos. Cuando llega una solicitud, se despacha un hilo para manejarla. En un momento dado pueden estar activos varios hilos del administrador de objetos.

El administrador de objetos actúa como asociador para los archivos que controla. Acepta las solicitudes por fallos de página en un puerto exclusivo, realiza la E/S necesaria al disco y envía las respuestas adecuadas.

El administrador de objetos funciona en términos de segmentos nombrados por sus posibilidades; en otras palabras, a la manera de Chorus. Cuando un proceso UNIX hace referencia a un descriptor de archivo, su sistema de tiempo de ejecución llama al administrador de procesos, el cual utiliza al descriptor de archivo como índice en una tabla para localizar la posibilidad correspondiente al segmento del archivo.

Por lógica, cuando un proceso lee de un archivo, esa solicitud es capturada por el administrador de procesos y después es enviada al administrador de objetos mediante el mecanismo normal de RPC. Sin embargo, como las lecturas y escrituras son tan importantes, se utiliza una estrategia optimizada para mejorar su desempeño. El administrador de procesos mantiene una tabla con las posibilidades de los segmentos para todos los archivos abiertos. Realiza una llamada *sgRead* al núcleo para obtener los datos requeridos. Si los datos están disponibles, el núcleo los copia de manera directa al buffer del usuario. Si no, el núcleo realiza una llamada *MpPullIn* al asociador adecuado (por lo general, el administrador de objetos). El asociador ejecuta una o más lecturas del disco, según sea necesario. Cuando se dispone de las páginas, el asociador las proporciona al núcleo, el cual copia los datos requeridos al buffer del usuario y termina la llamada al sistema.

El administrador de flujos

El administrador de flujos controla todos los flujos del System V, incluyendo al teclado, la pantalla, el ratón y los dispositivos de cinta. Durante la iniciación del sistema, el administrador de flujos envía un mensaje al administrador de objetos, anunciando su puerto e indicando los dispositivos que está preparado para controlar. Las solicitudes posteriores de E/S en estos dispositivos se pueden enviar entonces al administrador de flujos.

El administrador de flujos también controla los sockets de Berkeley y la red. De esta manera, un proceso en Chorus se puede comunicar mediante TCP/IP, así como otros protocolos de red. Los entubamientos y los entubamientos con nombre también se controlan aquí.

El administrador de la comunicación entre procesos

Este proceso controla las llamadas al sistema relacionadas con los mensajes del System V (no mensajes Chorus), los semáforos del System V (no semáforos Chorus), y la memoria compartida del System V (no la memoria compartida de Chorus). Las llamadas al sistema son poco agradables y el código se toma principalmente del propio System V. Mientras menos se hable de ellas, mejor.

Configurabilidad

La división del trabajo dentro del subsistema UNIX hace relativamente directo configurar una colección de máquinas de maneras diferentes, de modo que cada una sólo deba ejecutar el software que necesita. Los nodos de un multiprocesador también se pueden configurar de manera diferente. Todas las máquinas necesitan al administrador de procesos,

pero los otros administradores son opcionales. Los administradores necesarios dependen de la aplicación.

Analizaremos de forma breve varias configuraciones diferentes que se pueden construir mediante Chorus. La figura 9-22(a) muestra la configuración completa, que se puede utilizar en una estación de trabajo (con un disco duro) conectada a una red. Se necesitan (y están presentes) los cuatro procesos del subsistema de UNIX.

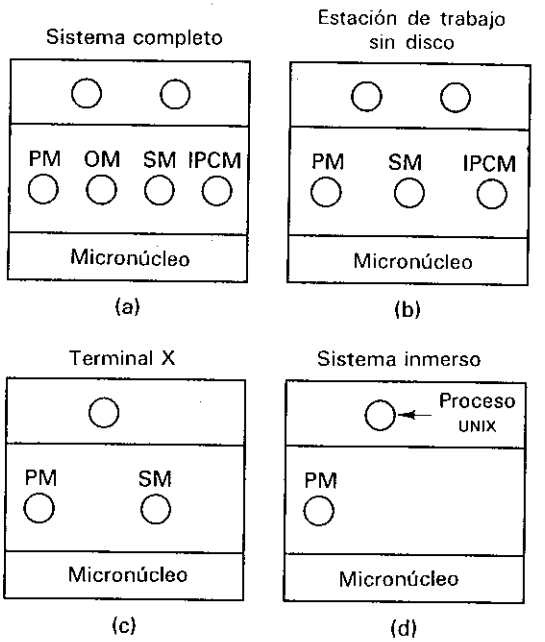


Figura 9.22. Las diferentes aplicaciones pueden controlarse mejor mediante configuraciones diferentes.

El administrador de objetos sólo se necesita en máquinas que contienen un disco, de modo que, en una estación de trabajo sin disco, la configuración de la figura 9-22(b) sería la más adecuada. Cuando un proceso lee o escribe en un archivo de esta máquina, el administrador de procesos envía la solicitud al administrador de objetos en el servidor de archivos del usuario. En principio, la máquina del usuario o el servidor de archivos realiza el ocultamiento, pero no ambos, excepto por aquellos segmentos que han sido abiertos o asociados en modo exclusivo para lectura.

Para las aplicaciones exclusivas, como una terminal X, se conocen de antemano las llamadas al sistema que podría hacer el programa y las que no. Si no se necesita un sistema local de archivos ni la IPC del System V, el administrador de objetos y el administrador de la comunicación entre procesos se pueden omitir, como en la figura 9-23(c).

Por último, para un sistema inmerso desconectado, como el controlador de un automóvil, un televisor o un muñeco parlante, sólo se necesita el administrador de procesos.

Los demás se pueden omitir para reducir la cantidad necesaria de ROM. Incluso es posible que un programa de aplicación exclusivo se ejecute de manera directa sobre el micronúcleo.

La configuración se hace de manera dinámica. Cuando el sistema se arranca, puede inspeccionar su ambiente y determinar los administradores que son necesarios. Si la máquina tiene un disco, se carga el administrador de objetos; en caso contrario, no, etc. Otra alternativa es utilizar archivos de configuración. Además, cada administrador puede crearse con un hilo. Al llegar las solicitudes, se pueden crear "al vuelo" hilos adicionales. Por ejemplo, el espacio de la tabla, la tabla de procesos, se asigna de manera dinámica mediante una pila, de modo que no es necesario tener un núcleo binario diferente para sistemas pequeños, que sólo tienen unos cuantos procesos, y para grandes, con cientos de procesos.

Aplicaciones de tiempo real

Chorus ha sido diseñado para controlar aplicaciones de tiempo real, con o sin UNIX. En particular, el planificador de Chorus refleja este objetivo. Las prioridades van de 1 a 255. Mientras menor sea el número, mayor será la prioridad, como en UNIX.

Los procesos de aplicación ordinarios de UNIX se ejecutan en las prioridades 128 a 255, como se muestra en la figura 9-23. En estos niveles de prioridad, cuando un proceso agota su quantum de CPU, se coloca al final de cola para su nivel de prioridad. Los procesos que conforman al subsistema UNIX se ejecutan en las prioridades 64 a 68. Así, se dispone de un gran número de prioridades arriba y debajo de las utilizadas por el subsistema UNIX para los procesos de tiempo real.

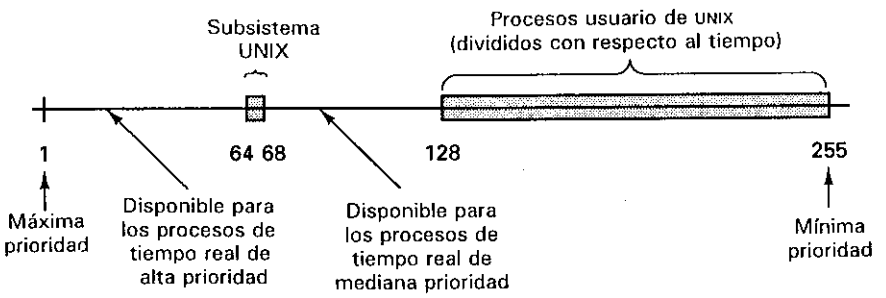


Figura 9.23. Prioridades y procesos de tiempo real.

Otra facilidad que es importante para el trabajo en tiempo real es la capacidad para reducir la cantidad de tiempo que el CPU está inactivo después de una interrupción. Lo normal es que, al ocurrir una interrupción, un hilo del administrador de objetos o del administrador de flujos realicen el procesamiento requerido de inmediato. Sin embargo, se dispone de una opción para que el hilo de la interrupción encargue a otro hilo de menor prioridad el trabajo real, de modo que la interrupción pueda concluir casi de inmediato. Al hacer esto se reduce la cantidad de tiempo muerto después de una interrupción, aunque también aumenta el costo del procesamiento de interrupciones, de modo que esta característica debe utilizarse con cuidado.

Estas facilidades se unen bien con UNIX, de modo que es posible depurar un programa en UNIX de la manera usual, pero ejecutándolo como un proceso en tiempo real cuando pasa a la etapa de producción, sólo configurando al sistema de manera diferente.

9.6. COOL: UN SUBSISTEMA ORIENTADO A OBJETOS

En realidad, el subsistema UNIX no es más que una colección de procesos de Chorus marcados como un subsistema. En consecuencia, es posible que otros subsistemas se ejecuten al mismo tiempo. Un segundo subsistema desarrollado para Chorus es **COOL** (siglas en inglés de **capa orientada a objetos de Chorus**). Fue diseñado para la investigación relativa a la programación orientada a objetos y para cubrir el hueco entre los objetos del sistema de grano grueso, como los archivos, y los objetos del lenguaje de grano fino, como las estructuras (registros). Describiremos COOL en esta sección. Para mayor información, véase (Lea *et al.*, 1991, 1993).

El trabajo en la primera versión, COOL-1, comenzó en 1988. El objeto era proporcionar el soporte a nivel de sistema de los lenguajes y aplicaciones orientados a objetos de grano fino y hacerlo de tal modo que los nuevos programas COOL y los antiguos programas UNIX se pudieran ejecutar uno al lado del otro, en la misma máquina, sin interferencias.

Cada objeto de COOL-1 consistía en dos segmentos de Chorus, uno para los datos y otro para el código. Los programas no tenían acceso directo a los segmentos de datos, sino que llamaban a procedimientos, llamados **métodos**, localizados en los segmentos de código de los objetos. De esta manera, la representación interna de los objetos que está oculta al usuario, le permitía a los escritores de objetos la libertad de usar la representación que les pareciera mejor (por ejemplo, un arreglo o una lista ligada). Esta representación incluso podría cambiarse posteriormente, sin que los programas que utilizaban los objetos lo supieran. Varios objetos de la misma clase compartían al mismo segmento de código, para ahorrar memoria.

Para abreviar la larga historia, el sistema resultante era decepcionante en términos del desempeño, el uso de recursos, etc., y el hueco entre los objetos del sistema de grano grueso y los objetos del lenguaje de grano fino no fue cubierto. En 1990, los diseñadores comenzaron de nuevo y diseñaron COOL-2. Este sistema se ejecutó uno año más tarde. Adelante describiremos su arquitectura e implantación.

9.6.1. La arquitectura COOL

Desde el punto de vista conceptual, COOL proporciona una **capa base COOL** que genera las fronteras de la máquina. Esta capa proporciona una forma de espacio de direcciones que es visible a todos los procesos COOL, sin importar el lugar donde se ejecuten, casi en la forma en que un sistema distribuido de archivos proporciona un espacio global de archivos. Sobre esta capa está el **sistema genérico de tiempo de ejecución**, que también es global al sistema. Sobre éste se encuentran los sistemas de tiempo de ejecución del lenguaje y después los programas usuarios, como se muestra en la figura 9-24.

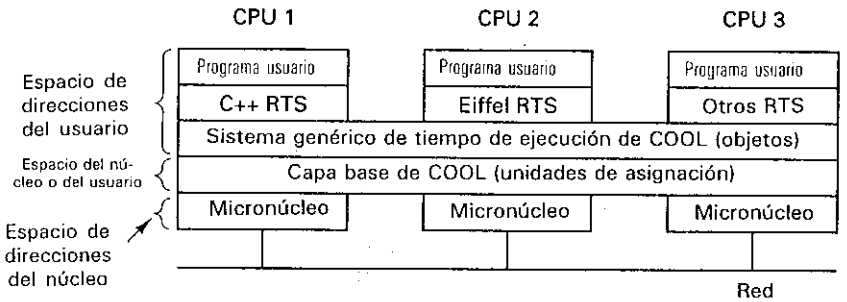


Figura 9.24. La arquitectura de COOL.

9.6.2. La capa base de COOL

La base de COOL proporciona un conjunto de servicios para los procesos usuario de COOL; en especial, para la biblioteca genérica COOL ligada a cada proceso COOL. El servicio más importante es una abstracción de la memoria, análoga a la memoria distribuida compartida, pero más dirigida hacia la programación orientada a objetos. Esta abstracción se basa en la **unidad de asignación (cluster)**, que es un conjunto de regiones de Chorus respaldadas por segmentos. Cada unidad de asignación contiene por lo general a un grupo de objetos relacionados entre sí; por ejemplo, objetos pertenecientes a la misma clase. Las capas superiores del software se encargan de determinar los objetos que van en cada unidad de asignación.

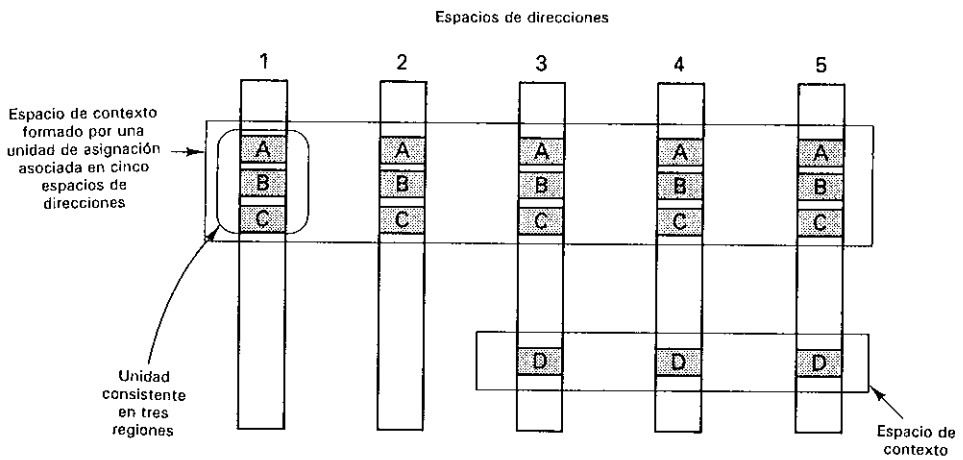


Figura 9.25. Espacios de contexto y unidades de asignación.

Una unidad de asignación se puede asociar en el espacio de direcciones de varios procesos, posiblemente en máquinas diferentes. La unidad de asignación siempre comienza en la frontera de una página, y ocupa las mismas direcciones en todos los procesos que en

la actualidad la utilizan. Las regiones de la unidad de asignación no tienen que ser adyacentes en el espacio de direcciones virtuales, por lo que una unidad de asignación podría ocupar, por ejemplo, las direcciones 1024-2047, 4096-8191 y 14336-16535 (suponiendo que el tamaño de la página es 1K).

Un segundo concepto soportado por la base de COOL es el **espacio de contexto**, que es una colección de espacios de direcciones; de nuevo, posiblemente en varias máquinas. La figura 9-25 muestra un sistema con cinco espacios de direcciones (en hasta cinco máquinas) y dos espacios de contexto. El primer espacio de contexto abarca todas las máquinas y contiene la unidad de asignación con tres regiones. Todos los hilos que viven en estos espacios de direcciones tienen acceso a los objetos en esta unidad de asignación, como si estuvieran en una memoria compartida. El segundo espacio de contexto sólo abarca tres máquinas y tiene la unidad de asignación con una región. Los hilos en los espacios de direcciones 1 y 2 no tienen acceso a los objetos en esta unidad de asignación. Las unidades correspondientes en espacios de direcciones diferentes deben estar asociadas a las mismas direcciones virtuales. La posición de los apuntadores internos debe modificarse al momento de ocurrir la asociación.

Las unidades de asignación no se duplican. Esto significa que, a pesar de que la unidad de asignación aparece en varios espacios de direcciones al mismo tiempo, sólo existe una copia física de cada unidad de asignación. Cuando un hilo usuario intenta llamar a un método sobre un objeto que está en su espacio de direcciones pero que no está físicamente en su máquina, ocurre un señalamiento a la base de COOL. La base envía entonces la solicitud a la máquina que contiene a la unidad de asignación para una llamada remota o hace que la unidad emigre a su máquina para una llamada local.

9.6.3. El sistema genérico de tiempo de ejecución de COOL

El sistema genérico de tiempo de ejecución de COOL utiliza las unidades de asignación y los espacios de contexto para controlar los objetos. Los objetos son persistentes y existen en el disco cuando ningún proceso los asocia. El sistema de tiempo de ejecución del lenguaje dispone de operaciones para crear y eliminar objetos, para asociarlos o desasociarlos del espacios de direcciones y para llamar a sus métodos. Cuando un objeto llama a un método de otro objeto localizado en su unidad de asignación, se realiza una llamada a un procedimiento local. Cuando el objeto invocado está en la unidad de asignación diferente, se utiliza el sistema genérico de tiempo de ejecución para pedir a la base de COOL que ordene la llamada remota. Debido a que el costo adicional de las llamadas dentro de las unidades de asignación es mucho menor que el de las llamadas entre las unidades, al colocar dentro de una misma unidad a objetos que se llaman entre sí con frecuencia se reduce el costo general del sistema. En COOL-1, todas las llamadas eran realmente operaciones entre las unidades, lo que demostró ser demasiado caro.

El sistema genérico de tiempo de ejecución tiene una interfaz estándar con los sistemas de tiempo de ejecución dependientes del lenguaje que lo utilizan. La interfaz se basa en

llamadas en las que el sistema genérico de tiempo de ejecución llama al sistema de tiempo de ejecución del lenguaje para determinar las propiedades de los objetos; por ejemplo, para determinar los apuntadores internos con fines de relocalización cuando se asocia un objeto.

9.6.4. El sistema de tiempo de ejecución de lenguaje

Por lo regular, cuando los programadores definen los objetos, los definen en un lenguaje especial de definición de interfaz. Estas definiciones se compilan en objetos que son llamados al tiempo de ejecución para invocar a los objetos. Después, estos objetos de interfaz deciden si las llamadas a objetos remotos se realizan de manera remota, o si los objetos emigran de manera local. Los métodos en los objetos de interfaz permiten que el programa controle la política utilizada.

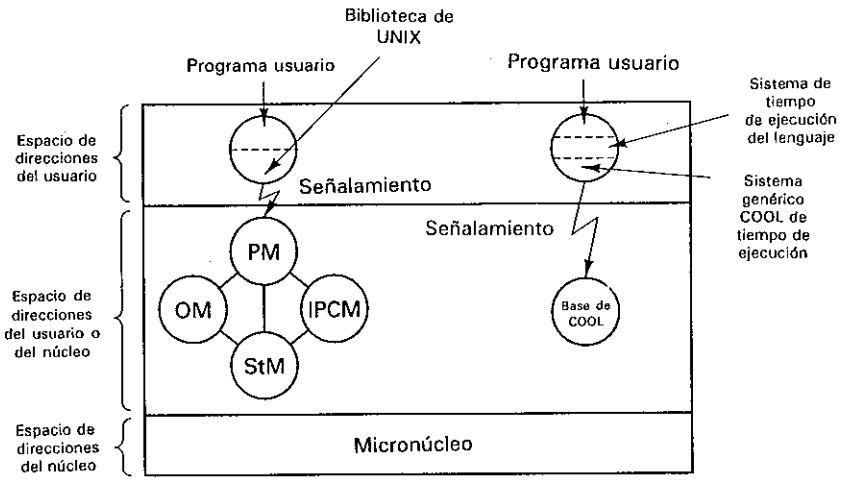


Figura 9.26. Implantación de COOL.

9.6.5. Implantación de COOL

El subsistema COOL se ejecuta sobre el micronúcleo de Chorus, al igual que el subsistema UNIX. Consta de un proceso que implanta la base COOL y un sistema genérico de tiempo de ejecución COOL ligado a cada programa COOL (junto con la biblioteca del lenguaje), como se muestra en la figura 9-26. Este diseño permite que los procesos COOL realicen llamadas al subsistema COOL, a la vez que los procesos UNIX realizan llamadas al subsistema UNIX, sin interferencias entre ellos. Esta modularidad es un resultado directo del diseño del micronúcleo, y por supuesto, también es válida para Amoeba y Mach.

9.7. COMPARACIÓN DE AMOEBA, MACH Y CHORUS

En éste y los dos capítulos anteriores analizamos tres sistemas operativos distribuidos basados en un micronúcleo, Amoeba, Mach y Chorus, con gran detalle. Aunque tienen

ciertos puntos en común, Amoeba y Mach difieren en muchos aspectos técnicos. Chorus ha retomado muchas ideas de Amoeba y Mach, y por lo tanto tiene una posición intermedia entre ambos. En esta sección analizaremos los tres sistemas a la vez para ilustrar las distintas opciones que pueden tener los diseñadores.

9.7.1. Filosofía

Amoeba, Mach y Chorus tienen una historia y filosofía diferentes. Amoeba se diseñó a partir de cero como un sistema distribuido para su uso en una colección de CPU conectados entre sí mediante una LAN. Después se añadieron los multiprocesadores y las WAN. Mach (en realidad RIG) se inició como sistema operativo en un procesador y después se añadieron los multiprocesadores y las LAN. Chorus se inició como un proyecto de investigación de sistemas operativos distribuidos, un tanto lejano del mundo de UNIX, y ha pasado tres revisiones, acercándose cada vez más a UNIX. Las consecuencias de estos diferentes fundamentos siguen siendo visibles en los sistemas.

Amoeba se basa en el modelo de la pila de procesadores. Un usuario no se conecta a una máquina particular, sino al sistema como un todo. El sistema operativo decide dónde ejecutar cada comando, con base en la carga de ese momento. Lo normal es que las solicitudes utilicen varios procesadores y lo raro es que dos comandos consecutivos se ejecuten en el mismo procesador. No existe el concepto de máquina de origen.

Por el contrario, Mach y Chorus (UNIX) se diseñaron con la idea de que el usuario entre definitivamente a una máquina específica y ejecute en ella todos sus programas de manera predefinida. No existe un intento por diseminar el trabajo de cada usuario en el mayor número de máquinas posibles (aunque en un multiprocesador el trabajo se difunde de manera automática en todos sus CPU). Aunque la ejecución remota es posible, la diferencia desde el punto de vista filosófico es que cada usuario tiene una máquina de origen (por ejemplo, una estación de trabajo) donde se lleva a cabo la mayor parte de su trabajo. Sin embargo, esta distinción se diluyó cuando Mach se llevó a Paragon de Intel, una máquina con una pila de procesadores.

Otra diferencia filosófica se relaciona con lo que es un "micronúcleo". El punto de vista de Amoeba sigue la famosa frase del aviador y escritor francés Antoine de St. Exupéry: "La perfección no se alcanza cuando ya no hay nada que agregar, sino cuando ya no hay nada que eliminar". Siempre que se hace una propuesta para añadir una característica al núcleo, la pregunta fundamental es: ¿Podemos vivir sin ella? Esta filosofía conduce a un núcleo mínimo, donde la mayor parte del código se encuentra en los servidores a nivel usuario.

Por el contrario, los diseñadores de Mach deseaban proporcionar la funcionalidad suficiente en el núcleo como para controlar el rango más amplio posible de aplicaciones. En muchas áreas, Amoeba contiene una forma de hacer algo y Mach contiene dos o tres, cada una de las cuales es más eficaz o conveniente en distintas circunstancias. En consecuencia, el núcleo de Mach es mucho mayor y tiene cinco veces el número de llamadas al sistema (incluyendo las llamadas a los hilos del núcleo) que Amoeba. Chorus ocupa una posición

intermedia entre Amoeba y Mach, pero sigue teniendo más llamadas que el BSD UNIX 4.2, que difícilmente podría llamarse micronúcleo. En la figura 9-27 se comparan estos números.

Sistema	Llamadas al sistema
Amoeba	30
Version 7 UNIX	45
4.2 BSD	84
Chorus	112
Mach	153
SunOS	165

Figura 9.27. Número de llamadas al sistema y llamadas a los hilos del núcleo en una selección de sistemas.

Otra diferencia filosófica entre Amoeba y Mach es que Amoeba se ha optimizado para el caso remoto (comunicación a través de la red) y Mach para el caso local (comunicación mediante la memoria). Amoeba tiene una RPC en extremo rápida a través de la red, mientras que el mecanismo de copiado durante la escritura de Mach proporciona una comunicación de alta velocidad en un nodo, por ejemplo. Chorus ha enfatizado principalmente los sistemas con un CPU y multiprocesadores, pero como la administración de la comunicación se realiza en el núcleo (como en Amoeba) y no en un proceso usuario (como en Mach), en potencia tiene un buen desempeño en cuanto a la RPC.

9.7.2. Objetos

Los objetos son el concepto central en Amoeba. Unos cuantos están ya integrados, como los hilos y los procesos, pero la mayoría son definidos por el usuario (por ejemplo, los archivos) y tienen un número arbitrario de operaciones sobre ellos. Cerca de una docena de operaciones genéricas (por ejemplo, obtener el estado) se definen en casi todos los objetos, además de que se definen varias operaciones específicas de cada objeto.

Por el contrario, los únicos objetos que Mach soporta de manera directa son los hilos, los procesos, los puertos y los objetos de la memoria, cada uno de ellos con un conjunto fijo de operaciones. Un software de mayor nivel utiliza estos conceptos para construir otros objetos, pero son cualitativamente distintos de los objetos ya integrados al sistema, como los objetos de memoria.

En los tres sistemas, los objetos reciben su nombre, su dirección y protección por medio de las posibilidades. En Amoeba, las posibilidades se controlan dentro del espacio del usuario y son protegidas mediante las funciones de un sentido. Las posibilidades para los objetos definidos por el sistema (por ejemplo, los procesos) y para los objetos definidos por el usuario (por ejemplo, los directorios) se trabajan de manera uniforme y aparecen en los directorios a nivel usuario para nombrar y direccionar a todos los objetos. Las posibilidades de Amoeba abarcan en principio todo su universo; es decir, un directorio puede

contener posibilidades para archivos y otros objetos localizados en cualquier otra parte. Los objetos se localizan mediante la transmisión y los resultados se guardan en un caché para su uso posterior.

Mach también tiene posibilidades, pero sólo para los puertos. Son controladas por medio del núcleo en listas de posibilidades, una para cada proceso. A diferencia de Amoeba, no existen posibilidades para los procesos o algunos otros objetos definidos por el sistema o por el usuario y no son utilizadas por lo general de manera directa por los programas de aplicación. Las posibilidades de los puertos se transfieren entre los procesos de manera controlada, de modo que Mach los pueda encontrar al buscarlos en tablas del núcleo.

Chorus soporta casi los mismos objetos integrados de Mach, pero también utiliza el sistema de posibilidades de Amoeba para permitir a los subsistemas que definan nuevos objetos protegidos. A diferencia de Amoeba, las posibilidades de Chorus no tienen un campo explícito (protegido por ciframiento) con los derechos permitidos. Como las posibilidades de Amoeba y a diferencia de Mach, las posibilidades de Chorus pasan de un proceso a otro incluyéndolas en un mensaje o escribiéndolas en un archivo compartido.

9.7.3. Procesos

Los tres sistemas soportan procesos con varios hilos por cada proceso. En los tres casos, los hilos se controlan y planifican en el núcleo, aunque se construyen paquetes a nivel usuario arriba de ellos. Amoeba no proporciona control alguno por parte del usuario sobre la planificación de los hilos, mientras que Mach y Chorus permiten que los procesos establezcan las prioridades y políticas de sus hilos en software. Mach proporciona un soporte de multiprocesador más elaborado que los otros dos sistemas.

En Amoeba y Chorus, la sincronización entre los hilos se lleva a cabo por medio de mútex y semáforos. En Mach esto se hace por medio de mútex y variables de condición. Amoeba y Mach soportan alguna forma de variables glocales. Chorus utiliza registros de software para definir el contexto particular de cada hilo.

Amoeba, Mach y Chorus funcionan en multiprocesadores, pero difieren en la forma en que controlan los hilos en esas máquinas. En Amoeba, todos los hilos de un proceso se ejecutan en el mismo CPU, en forma seudoparalela, compartiendo el tiempo con el núcleo. En Mach, los procesos tienen un control fino sobre los hilos que se ejecutan en cada CPU (mediante el concepto de conjunto de procesadores). En consecuencia, los hilos de un mismo proceso se pueden ejecutar en paralelo en distintos CPU.

En Amoeba, se puede lograr un efecto similar al hacer que varios procesos con un hilo se ejecuten en diferentes CPU y que compartan un mismo espacio de direcciones. Sin embargo, es claro que los diseñadores de Mach han dedicado más atención a los multiprocesadores que los diseñadores de Amoeba.

Chorus soporta tener varios hilos dentro de un proceso, ejecutándose en diferentes CPU al mismo tiempo, pero esto es controlado por completo por el sistema operativo. No existen primitivas visibles al usuario para administrar la asignación de cada hilo a los procesadores, pero existirán en el futuro.

Sin embargo, los diseñadores de Amoeba han puesto más empeño en el soporte del balance y heterogeneidad de la carga. Cuando el shell de Amoeba inicia un proceso, solicita al servidor de ejecución que encuentre el CPU con la carga más ligera. A menos que el usuario especifique una cierta arquitectura, el proceso se inicia en cualquier arquitectura para la cual exista un binario, sin que el usuario tenga conciencia del tipo seleccionado. Este esquema está diseñado para diseminar la carga en el mayor número posible de máquinas todo el tiempo.

En Mach, los procesos se inician por lo general en la máquina de origen del usuario. Sólo con una solicitud explícita del usuario se ejecutan los procesos de manera remota en las estaciones de trabajo inactivas, e incluso entonces tienen que hacerlo rápido si el propietario de la estación de trabajo toca el teclado. Esta diferencia se relaciona con la diferencia fundamental entre el modelo de la pila de procesadores y el modelo de la estación de trabajo.

Chorus permite iniciar un proceso en cualquier máquina. La emulación de UNIX proporciona una forma de establecer el sitio por omisión.

9.7.4. Modelo de memoria

El modelo de memoria de Amoeba se basa en segmentos de longitud variable. Un espacio de direcciones virtuales consta de cierto número de segmentos asociados a direcciones específicas. Los segmentos se asocian y desasocian a voluntad. Cada segmento es controlado por una posibilidad. Un proceso remoto que tiene la posibilidad de los segmentos de otro proceso (por ejemplo, un depurador) lee y escribe en ellos desde cualquier otra máquina Amoeba. Amoeba no soporta la paginación. Cuando un proceso se ejecuta, todos sus segmentos están dentro de la memoria. Las ideas detrás de esta decisión son la sencillez del diseño y un alto desempeño, junto con el hecho de que las memorias de gran tamaño se vuelven cada vez más comunes, incluso en las máquinas más pequeñas.

El modelo de memoria de Mach se basa en los objetos de memoria y se implanta en términos de páginas de tamaño fijo. Los objetos de memoria se pueden asociar y desasociar a voluntad. Un objeto de memoria no necesita estar por completo contenido en la memoria para ser utilizado. Cuando se hace referencia a una página ausente, ocurre un fallo de página y se envía un mensaje a un administrador externo de la memoria para que encuentre la página y la asocie de manera conveniente. Junto con el administrador predefinido de la memoria, este mecanismo soporta la memoria virtual con paginación.

Las páginas se comparten entre los distintos procesos de varias formas. Una configuración común es el copiado compartido durante la escritura, el cual se utiliza para ligar un proceso hijo a su padre. Aunque este mecanismo es una forma muy eficaz para compartir en un nodo, pierde sus ventajas en un sistema distribuido, puesto que siempre se necesita el transporte físico (si el receptor necesita leer los datos). En tal ambiente, se desperdician el código adicional y cierta complejidad. Éste es un ejemplo claro de la razón de optimizar Mach para los sistemas de un CPU o de multiprocesador, en vez de los sistemas distribuidos.

El modelo de administración de memoria de Chorus se toma principalmente de Mach. También tiene objetos de memoria (segmentos) que se pueden asociar. Se paginan según la demanda, bajo el control de un paginador externo (asociador), como en Mach.

Amoeba, Mach y Chorus soportan la memoria compartida distribuida, pero lo hacen de formas distintas. Amoeba soporta los objetos compartidos que se duplican en todas las máquinas que los utilizan. Los objetos pueden tener cualquier tamaño y soportar cualquier operación. Las lecturas se realizan en forma local y las escrituras se llevan a cabo mediante el protocolo de transmisión confiable de Amoeba.

Por el contrario, Mach y Chorus soportan una memoria compartida distribuida basada en páginas. Cuando un hilo hace referencia a una página no presente en la máquina, la página se busca en la máquina activa y se recupera. Si dos máquinas tienen demasiados accesos a una misma página donde se pueda escribir, surgen ciertos problemas de basura. Aquí aparece una contradicción entre una actualización más cara en Amoeba (debido a la duplicación de objetos donde se puede escribir) o el potencial de existencia de basura en Mach (sólo una copia de las páginas donde se puede escribir).

9.7.5. Comunicación

Amoeba soporta la RPC y la comunicación en grupo como primitivas fundamentales. Las RPC se dirigen a los puertos de envío, que son direcciones de servicio. Se protegen por ciframiento mediante las funciones de un sentido. El envío y recepción de mensajes puede hacerse en cualquier parte. La interfaz RPC es muy sencilla: sólo hay tres llamadas al sistema, las cuales no tienen opciones.

La comunicación de grupo proporciona una transmisión confiable como primitiva del usuario. Se envían mensajes a cualquier grupo con una garantía de entrega confiable. Además, todos los miembros del grupo ven que los mensajes llegan en el mismo orden.

La comunicación de bajo nivel utiliza el protocolo FLIP, que proporciona el direccionamiento a los procesos (en contraposición al direccionamiento a las máquinas). Esta característica permite la migración de los procesos y la reconfiguración de la red de manera automática, sin que el software tenga conciencia de ello. También soporta otras capacidades útiles en los sistemas distribuidos.

Por el contrario, la comunicación en Mach es de un proceso a un puerto, en vez de ser de un proceso a otro. Además, el emisor y el puerto deben estar en el mismo nodo. Por medio de un servidor de mensajes en la red o el código NORMA como representante, la comunicación se extiende a través de una red, pero esta indirección es costosa en términos del desempeño. Mach no soporta la comunicación en grupo o la transmisión confiable como primitivas básicas del núcleo.

La comunicación se lleva a cabo mediante la llamada al sistema *mach_msg*, la cual tiene siete parámetros, 10 opciones y 35 mensajes de errores potenciales. Soporta la transferencia de mensajes, tanto síncrona como asíncrona. Este método es la antítesis de la estrategia de Amoeba, "ser sencillo y hacer todo rápido". La idea aquí es proporcionar la máxima flexibilidad y el rango más amplio de soporte para las aplicaciones actuales y futuras.

Los mensajes de Mach son simples o complejos. Los mensajes simples son tan sólo bits y no se procesan de manera especial por parte del núcleo. Los mensajes complejos pueden contener posibilidades. También transfieren datos fuera de línea mediante el copiado durante la escritura, algo que no posee Amoeba. Por otro lado, esta facultad es de poco valor en un sistema distribuido, puesto que hay que buscar los datos fuera de línea mediante el servidor de mensajes de la red, combinarlos con el encabezado del mensaje y los datos en línea y por último enviarlos a través de la red. Esta optimización es para el caso local y no gana nada cuando el emisor y el receptor se encuentran en máquinas distintas.

En la red, Mach utiliza protocolos convencionales como TCP/IP. Estos tienen la ventaja de ser estables y ampliamente disponibles. FLIP, por el contrario, es nuevo, pero es más rápido para el uso típico de RPC y está en especial diseñado para las necesidades en el cómputo distribuido.

La comunicación en Chorus es filosóficamente similar a la de Mach, pero es más sencilla. Los mensajes se dirigen a los puertos y se pueden enviar de manera asíncrona o mediante RPC. El núcleo controla toda la comunicación en Chorus, como en Amoeba. No hay nada como el servidor de mensajes en la red en Chorus. Como Amoeba y a diferencia de Mach, un mensaje de Chorus tiene un encabezado fijo que contiene la información de la fuente y el destino, y un cuerpo no tipificado que es simplemente un arreglo de bytes, en lo que concierne al sistema. Como en Amoeba, las posibilidades en los mensajes no se tratan de manera especial.

Chorus soporta la transmisión a nivel del núcleo pero de manera más similar a Mach (que no la soporta) que a Amoeba (que sí la soporta). Los puertos se pueden agrupar en grupos de puertos, y los mensajes enviados a todos los miembros de un grupo de puertos (o sólo a uno). No existe una garantía de que todos los procesos vean todos los mensajes en el mismo orden, como en Amoeba.

9.7.6. Servidores

Amoeba tiene varios servidores para funciones específicas, entre las que se encuentran la administración de la memoria, la réplica de objetos y el balance de la carga. Todos se basan en los objetos y las posibilidades. Amoeba soporta los objetos duplicados mediante directorios que contienen conjuntos de posibilidades. Se dispone de una emulación de UNIX a nivel del código fuente, basada en POSIX, pero no es 100% completa. La emulación se realiza asociando los conceptos de UNIX con conceptos de Amoeba y llamando a los servidores nativos de Amoeba para que realicen el trabajo. Por ejemplo, cuando se abre un archivo, se adquiere la posibilidad del mismo y se guarda en la tabla de descripción del archivo.

Mach tiene un servidor que ejecuta el UNIX BSD como un programa de aplicación. Proporciona una emulación con una compatibilidad binaria del 100%, algo formidable para la ejecución del software existente del que sólo se dispone del código fuente. No se soporta la réplica de los objetos en general. También existen otros servidores.

Chorus proporciona una compatibilidad binaria completa con el System V de UNIX. La emulación se realiza mediante una colección de procesos (como en Amoeba) en vez de ejecutar a UNIX como un programa de aplicación (como en Mach). Sin embargo, algunos

de estos procesos contienen código real de UNIX, como en Mach y a diferencia de Amoeba. Como en Amoeba, los servidores nativos fueron diseñados a partir de cero, teniendo en mente el cómputo distribuido, de modo que la emulación traduce las construcciones de UNIX en las construcciones nativas. Al igual que en Amoeba, por ejemplo, cuando se abre un archivo, se busca una posibilidad para éste y se guarda en la tabla de descripción del archivo.

En la figura 9-28 se resumen algunos de los principales puntos analizados arriba.

Punto	Amoeba	Mach	Chorus
Diseñado para:	Sistema distribuido	Un CPU, multiprocesador	Un CPU, multiprocesador
Modelo de ejecución	Pila de procesadores	Estación de trabajo	Estación de trabajo
¿Micronúcleo?	Sí	Sí	Sí
Número de llamadas al núcleo	30	153	112
¿Balance de la carga automático	Sí	No	No
Posibilidades	General	Exclusivamente puertos	General
Posibilidades en:	Espacio del usuario	Núcleo	Espacio del usuario
Hilos controlados por:	Núcleo	Núcleo	Núcleo
¿Transparencia y heterogeneidad	Sí	No	No
¿Prioridades configurables por el usuario?	No	Sí	Sí
Soporte de multiprocesadores	Mínima	Amplia	Moderada
Objetos asociados	Segmento:	Objeto de memoria	Segmento
¿Paginación según la demanda?	No	Sí	Sí
¿Copiado durante la escritura?	No	Sí	Sí
¿Paginadores externos?	No	Sí	Sí
Memoria compartida distribuida	Basada en objetos	Basada en las páginas	Basada en las páginas
¿RPC?	Sí	Sí	Sí
Comunicación en grupo	Confiable, ordenada	Ninguna	No confiable
¿Comunicación asíncrona?	No	Sí	Sí
Mensajes entre las máquinas	Núcleo	Espacio del usuario/núcleo	Núcleo
Mensajes dirigidos a:	Proceso	Puerto	Puerto
Emulación de UNIX	Fuente	Binaria	Binaria
Compatibilidad con UNIX	POSIX (parcial)	BSD	System V
¿UNIX con un servidor?	No	Sí	No
¿UNIX con varios servidores?	Sí	No	Sí
Optimizado para:	Caso remoto	Caso local	Caso local
¿Replica automática de archivos?	Sí	No	No

Figura 9-28. Una comparación de Amoeba, Mach y Chorus.

9.8. RESUMEN

Como Amoeba y Mach, Chorus es un sistema operativo basado en un micronúcleo para su uso en los sistemas distribuidos. Proporciona una compatibilidad binaria con el System V de UNIX, un soporte para las aplicaciones de tiempo real, y la programación orientada a objetos.

Chorus consta de tres capas conceptuales: la capa del núcleo, los subsistemas, y los procesos usuario. La capa del núcleo contiene al micronúcleo propiamente dicho, así como algunos procesos del núcleo que se ejecutan en modo núcleo y comparten el espacio de direcciones del micronúcleo. La capa intermedia contiene los subsistemas, que se utilizan para proporcionar el soporte del sistema operativo a los programas usuarios, que residen en la capa superior.

El micronúcleo proporciona seis abstracciones fundamentales: los procesos, los hilos, las regiones, los mensajes, los puertos, los grupos de puertos y los identificadores únicos. Los procesos proporcionan una forma de reunir y controlar los recursos. Los hilos son los entes activos del sistema, y son planificados por el núcleo mediante un planificador basado en prioridades. Las regiones son áreas del espacio de direcciones virtuales que pueden tener segmentos asociados a ellas. Los puertos son buffers que se utilizan para guardar los mensajes recibidos que no han sido leídos. Los identificadores únicos son nombres binarios utilizados para la identificación de los recursos.

El micronúcleo y los subsistemas proporcionan juntos tres construcciones adicionales: las posibilidades, los identificadores de protección y los segmentos. Los primeros dos se utilizan para nombrar y proteger a los recursos del subsistema. La tercera es la base para la asignación de memoria, tanto dentro de un proceso en ejecución como en el disco.

En este capítulo se describieron dos subsistemas. El subsistema UNIX consta de los administradores de procesos, de objetos, de flujos y de la comunicación entre procesos, que funcionan juntos para proporcionar una emulación UNIX con compatibilidad binaria. El subsistema COOL proporciona el soporte para la programación orientada a objetos.

PROBLEMAS

1. Las posibilidades en Chorus utilizan números de época en sus UI. ¿Por qué?
2. ¿Cuál es la diferencia entre una región y un segmento?
3. El supervisor de Chorus depende de la máquina, mientras que el ejecutor en tiempo real no. Explique.
4. ¿Para qué necesita Chorus los procesos del sistema además de los procesos usuario y los procesos del núcleo?
5. ¿Cuál es la diferencia entre que un hilo quede SUSPENDIDO y quede DETENIDO? Después de todo, en ambos casos no se puede ejecutar.
6. Describa brevemente la forma en que Chorus maneja las excepciones y las interrupciones, e indique por qué se manejan en forma diferente.

7. Chorus soporta los semáforos y los mútex. ¿Es esto estrictamente necesario? ¿No bastaría con soportar únicamente los semáforos?
8. ¿Cuál es la función de un asociador?
9. Describa brevemente para qué se utilizan *MpPullIn* y *MpPushOut*.
10. Chorus soporta la RPC y un envío asíncrono. ¿Cuál es la diferencia esencial entre ambos?
11. Dé un posible uso de la migración de puertos en Chorus.
12. En Chorus, es posible enviar un mensaje a un grupo de puertos. ¿Ese mensaje va a todos los puertos del grupo, o a un puerto elegido al azar?
13. Chorus tiene llamadas explícitas para crear y eliminar puertos, pero sólo una llamada para crear grupos de puertos (*grpAllocate*). Piense por qué no existe *grpDelete*.
14. ¿Para qué se introducen los minipuertos? ¿Tienen algo que no tengan los puertos regulares?
15. ¿Por qué Chorus soporta la planificación con y sin prioridades?
16. Indique una forma en que Chorus es similar a Amoeba. Indique una forma en que Chorus es similar a Mach.
17. ¿En qué difiere el uso de los grupos de puertos en Chorus de la comunicación en grupo de Amoeba?
18. ¿Para qué extendió Chorus la semántica de las señales de UNIX?

10

Estudio 4: DCE

En los tres capítulos anteriores analizamos con cierto detalle algunos sistemas distribuidos basados en un micronúcleo. En este capítulo examinaremos un punto de vista por completo diferente, el **ambiente de cómputo distribuido, DCE** por sus siglas en inglés, de la Open System Foundation. A diferencia de los métodos basados en un micronúcleo, revolucionarios por naturaleza (desechando los sistemas operativos actuales y comenzando de nuevo), DCE adopta un punto de vista evolutivo, construyendo un ambiente de cómputo distribuido sobre los sistemas operativos existentes. En la siguiente sección introduciremos las ideas detrás de DCE, y en las siguientes, analizaremos con cierto detalle algunos de los principales componentes de DCE.

10.1. INTRODUCCIÓN A DCE

En esta sección daremos un panorama de la historia, objetivos, modelos y componentes de DCE, así como una introducción al concepto de celda, que juega un importante papel en DCE.

10.1.1. Historia de DCE

OSF fue planteada por un grupo de importantes proveedores de computadoras, entre las que estaban IBM, DEC y Hewlett-Packard, como una respuesta a la firma de un acuerdo por AT&T y Sun Microsystems para seguir desarrollando y comercializando el sistema operativo UNIX. Las demás compañías temían que este arreglo diera a Sun una ventaja competitiva sobre ellos. El objetivo inicial de OSF era desarrollar y comercializar una nueva

versión de UNIX, sobre la que ellos, y no AT&T/Sun, tuvieran el control. Este objetivo se logró mediante el surgimiento de OSF/1.

Desde un principio, fue evidente que muchos de los clientes del consorcio OSF construyen aplicaciones distribuidas sobre OSF/1 y otros sistemas UNIX. OSF respondió a esta necesidad emitiendo una "solicitud de tecnología", en la que pedían a las compañías que proporcionaran las herramientas y demás software necesarios para formar un sistema distribuido. Muchas compañías hicieron propuestas que fueron con cuidado evaluadas. OSF seleccionó entonces varias de estas ofertas, y las desarrolló aún más para producir un paquete, DCE, que se ejecuta sobre OSF/1 y también en otros sistemas. DCE es ahora uno de los principales productos de OSF. También se planeó un producto complementario, **DME** (siglas en inglés de **ambiente de administración distribuido**), para la administración de los sistemas distribuidos, pero jamás se construyó.

10.1.2. Objetivos de DCE

El objetivo principal de DCE es el de proporcionar un ambiente coherente, sin costuras que sirva como plataforma para la ejecución de aplicaciones distribuidas. A diferencia de Amoeba, Mach y Chorus, este ambiente se construye sobre los sistemas operativos existentes, en principio sobre UNIX, pero posteriormente fue llevado a VMS, WINDOWS y OS/2. La idea es que el cliente puede considerar una colección de máquinas existentes, añade el software DCE y después ejecuta las aplicaciones distribuidas, todo sin perturbar las aplicaciones existentes (no distribuidas). Aunque la mayor parte del paquete DCE se ejecuta en el espacio del usuario, en ciertas configuraciones, una parte (del sistema distribuido de archivos) debe agregarse al núcleo. La propia OSF sólo vende código fuente, que los vendedores integran a su sistema. Para simplificar la exposición, en este capítulo nos concentraremos en DCE sobre UNIX.

El ambiente ofrecido por DCE consta de un gran número de herramientas y servicios, más una infraestructura para que éstos operen. Las herramientas y servicios han sido elegidos de modo que funcionen juntos de manera integrada y faciliten el desarrollo de aplicaciones distribuidas. Por ejemplo, DCE proporciona herramientas que facilitan la escritura de aplicaciones con alta disponibilidad. Como otro ejemplo, DCE proporciona un mecanismo para la sincronización de relojes de máquinas diferentes, para obtener un concepto global de tiempo.

DCE se ejecuta en muchos tipos distintos de computadoras, sistemas operativos y redes. En consecuencia, los desarrolladores de aplicaciones producen con facilidad software portable en varias plataformas, amortizando los costos de desarrollo e incrementando el tamaño del mercado potencial.

El sistema distribuido sobre el que se ejecuta DCE puede ser un sistema heterogéneo, que conste de computadoras de diversos proveedores, cada una de las cuales tiene su sistema operativo local. La capa de software DCE sobre el sistema operativo oculta las diferencias, realizando las conversiones de manera automática entre los tipos de datos, en caso necesario. Todo esto es transparente para el programador de la aplicación.

Como consecuencia de todo lo anterior, DCE facilita la escritura de aplicaciones en la que varios usuarios en diferentes sitios trabajan juntos, colaborando en algún proyecto y compartiendo recursos de hardware y software. La seguridad es un aspecto importante de ese arreglo, de modo que DCE proporciona un conjunto amplio de herramientas para la autenticación y la protección.

Por último, DCE ha sido diseñado para trabajar en red con los estándares existentes en varias áreas. Por ejemplo, un grupo de máquinas DCE se comunican entre sí y con el mundo exterior utilizando los protocolos TCP/IP u OSI, y los recursos se pueden localizar mediante los sistemas de asignación de nombres DNS o X.500. También se utilizan los estándares POSIX.

10.1.3. Componentes de DCE

El modelo de programación subyacente en todo DCE es el modelo cliente/servidor. Los procesos usuario actúan como clientes para tener acceso a los servicios remotos proporcionados por los procesos servidor. Algunos de estos servicios son parte del propio DCE, pero otros pertenecen a las aplicaciones y son escritos por los programadores de las aplicaciones. En esta sección daremos una rápida introducción a aquellos servicios distribuidos que son proporcionados por el propio paquete DCE, que principalmente son los servicios de tiempo, directorio, seguridad y sistema de archivos.

En la mayor parte de las aplicaciones DCE, los programas clientes son programas en C más o menos normales que han sido ligados con una biblioteca especial. Los programas binarios clientes contienen entonces un pequeño número de procedimientos de biblioteca que proporcionan las interfaces con los servicios, ocultando los detalles a los programadores. Los servidores, en contraste, son por lo general grandes programas demonios. Se ejecutan todo el tiempo, esperando que lleguen las solicitudes de trabajo. Cuando éstas llegan, se procesan y se regresan las respuestas.

Además de ofrecer servicios distribuidos, DCE también tiene dos facilidades para la programación distribuida que están organizadas como servicios: los hilos y las RPC (llamadas a procedimientos remotos). La facilidad de los hilos permite la existencia de varios hilos de control en el mismo proceso, al mismo tiempo. Algunas versiones de UNIX proporcionan también los hilos, pero el uso de DCE proporciona una interfaz de hilos estándar entre los sistemas. En la medida de lo posible, DCE utiliza los hilos nativos para implantar los hilos DCE, pero cuando no existan hilos nativos, DCE proporciona un paquete de hilos a partir de cero.

La otra facilidad de DCE es la RPC, que es la base de toda la comunicación en DCE. Para tener acceso a un servicio, el proceso cliente realiza una RPC a un proceso servidor remoto. El servidor procesa la solicitud y (opcionalmente) envía una respuesta de regreso. DCE controla todo el mecanismo, incluyendo la localización del servidor, la conexión con éste y el desarrollo de la llamada.

La figura 10-1 proporciona una idea aproximada de la forma en que se entrelazan las diversas partes de DCE. En el nivel inferior está el hardware de la computadora, sobre lo

cual se ejecuta el sistema operativo nativo (con agregados de DCE). Para soportar un servidor por completo DCE, el sistema operativo debe ser UNIX u otro sistema en esencia con la misma funcionalidad que UNIX, incluyendo la multiprogramación, la comunicación local entre procesos, la administración de memoria, los cronómetros y la seguridad. Para soportar sólo un cliente DCE, se requiere menos funcionalidad, e incluso MS-DOS es suficiente.

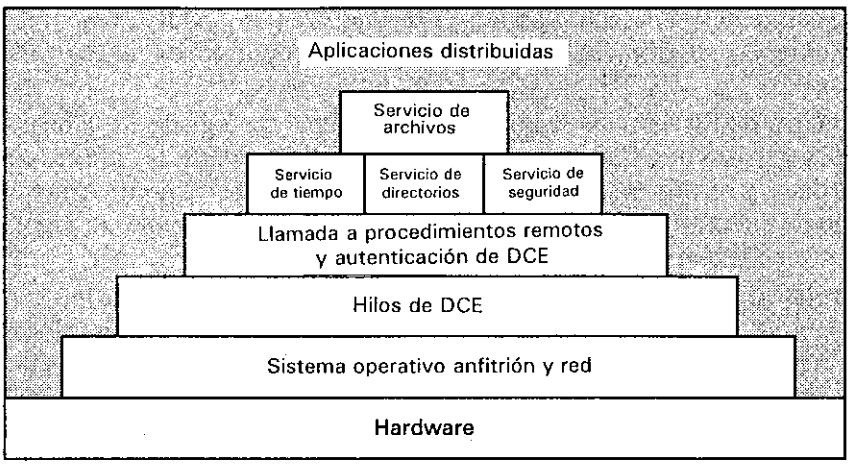


Figura 10-1. Esquema de la forma en que se relacionan las partes de DCE.

Sobre el sistema operativo está el paquete de hilos de DCE. Si el sistema operativo tiene su paquete de hilos adecuado, la biblioteca de hilos de DCE sólo sirve para convertir la interfaz al estándar DCE. Si no existe un paquete de hilos nativo, DCE proporciona un paquete de hilos que se ejecuta casi por completo en el espacio del usuario, excepto por unos cuantos cientos de líneas de código ensamblador en el núcleo para la administración de las pilas de hilos. A continuación está el paquete RPC, que, como en el caso de los hilos, es una colección de procedimientos de biblioteca. Por lógica, una parte de la seguridad está también en esta capa, puesto que se necesita para realizar una RPC autenticada.

Sobre la capa RPC vienen los diversos servicios DCE. No todo servicio se ejecuta en cada máquina. El administrador del sistema determina el lugar donde se ejecuta cada servicio. Los servicios estándar son el tiempo, el directorio y la seguridad, de modo que el servicio distribuido de archivos se ejecuta sobre ellos, como se muestra en la figura. En una configuración típica, estos servicios sólo se ejecutan en las máquinas "servidor del sistema" y no en estaciones de trabajo cliente. Daremos una breve introducción a cada uno de estos servicios.

La función general de los paquetes de hilos y RPC deben ser claros, pero sería útil una breve explicación de los servicios. Posteriormente daremos un análisis detallado de todas las facilidades y servicios. El **servicio de tiempo distribuido** es necesario ya que cada máquina en el sistema, por lo regular tiene su reloj, de modo que el concepto de lo que es tiempo es más complicado que en un sistema. Por ejemplo, el programa *make* de UNIX

examina los tiempos de creación de los archivos fuente y binario de un programa grande para determinar cuáles fuentes han sido cambiados desde que se creó por última vez el binario. Sólo hay que compilar de nuevo estos archivos fuente. Sin embargo, consideremos lo que sucede si los archivos fuente y binario se guardan en máquinas diferentes, y los relojes no están sincronizados. El hecho de que el tiempo de creación para un archivo binario sea posterior al tiempo de creación del archivo fuente correspondiente no significa que omita la recopilación. El servicio de tiempo DCE resuelve este problema, manteniendo sincronizados los relojes, para proporcionar un concepto global de tiempo.

El **servicio de directorios** se utiliza para mantener un registro de la posición de todos los recursos del sistema. Estos recursos incluyen las máquinas, las impresoras, los servidores, los datos y mucho más, y están distribuidos de manera geográfica en todo el mundo. El servicio de directorios permite que un proceso solicite un recurso sin preocuparse de su posición, a menos que el proceso se ocupe de ésta.

El **servicio de seguridad** protege los recursos de todos los tipos, de modo que el acceso se restrinja a las personas autorizadas. Por ejemplo, en un sistema de información de un hospital, la política podría ser que un doctor revise la información médica de sus propios pacientes, pero no la relativa a los pacientes de otros doctores. Las personas que trabajan en el departamento de contabilidad ven los registros de todos los pacientes, pero sólo los aspectos financieros. Si se realiza una prueba de sangre, el doctor ve los resultados médicos y el contador ve el costo (pero no los resultados médicos). El servicio de seguridad proporciona herramientas que ayudan a construir aplicaciones como ésta.

Por último, el **servicio distribuido de archivos** es un sistema mundial de archivos que proporciona una forma transparente de acceso a cualquier archivo del sistema, de la misma forma. Se construye sobre los sistemas de archivo nativos del anfitrión o utilizarse en vez de éstos.

En la parte superior de la cadena alimenticia de DCE están las aplicaciones distribuidas. Éstas utilizan (u omiten) cualquiera de las facilidades y servicios de DCE. Las aplicaciones sencillas pueden utilizar RPC de manera implícita (por medio de la biblioteca, sin darse cuenta de ello) y un poco más, mientras que las aplicaciones más complejas pueden realizar llamadas explícitas a todos los servicios.

La imagen de la figura 10-1 no es por completo precisa, pero es difícil exhibir las dependencias entre las partes, pues son recursivas. Por ejemplo, el servicio de directorios utiliza la RPC para la comunicación interna entre sus diversos servidores, pero el paquete RPC utiliza el servicio de directorios para localizar el destino. Así, no es estrictamente cierto que la capa de servicio de directorios esté sobre la capa de RPC, como se muestra en la figura. Como segundo ejemplo, para ilustrar las dependencias horizontales, el servicio de tiempo utiliza el servicio de seguridad para ver quién configura el reloj, pero el servicio de seguridad utiliza el servicio de tiempo para emitir boletos de permisos con tiempos de vida cortos. Sin embargo, en una primera aproximación, la figura 10-1 indica la estructura gruesa, de modo que la utilizaremos como nuestro modelo de aquí en adelante.

10.1.4. Celdas

Los usuarios, máquinas y otros recursos en un sistema DCE se agrupan para formar **celdas**. La asignación de nombres, la seguridad, la administración y otros aspectos de DCE se basan en estas celdas. Las fronteras entre las celdas imitan por lo general a las unidades organizativas, por ejemplo, una compañía pequeña o un departamento de una compañía grande podrían ser celdas.

Para determinar la forma de agrupar las máquinas en celdas, hay que tomar en cuenta cuatro factores:

1. Finalidad.
2. Seguridad.
3. Costo.
4. Administración.

El primer factor, la finalidad, significa que las máquinas en una celda (y sus usuarios) deben trabajar todos con un objetivo común o en un proyecto común a largo plazo (tal vez medido en años). Los usuarios se conocen entre sí y tienen más contacto entre sí que las personas fuera de la celda. Con frecuencia, los departamentos de las compañías están estructurados de esta manera. Las celdas también se organizan en torno a un servicio común ofrecido, como sistema bancario en línea, con todos los cajeros automáticos y la computadora central pertenecientes a una celda.

El segundo factor, la seguridad, tiene que ver con el hecho de que DCE funciona mejor cuando los usuarios en una celda confían entre ellos más de lo que confían en las personas fuera de la celda. La razón para esto es que las fronteras de las celdas funcionan como protección: La obtención de recursos internos es directa, pero el acceso a cualquier cosa de otra celda requiere que las dos celdas realicen una negociación para garantizar que confían entre ellas.

El tercer factor, el costo, es importante debido a que ciertas funciones DCE se optimizan para la operación dentro de una celda (por ejemplo, la seguridad). La geografía puede jugar un papel importante en este aspecto, pues si se colocan los usuarios distantes en una misma celda tendrán que comunicarse por medio de una red de área amplia. Si esta red es lenta y poco confiable, se necesitará un costo adicional para enfrentar estos problemas y compensarlos en la medida de lo posible.

Por último, toda celda necesita un administrador. Si todas las personas y las máquinas de una celda pertenecen al mismo departamento o proyecto, no habría problema para designar un administrador. Sin embargo, si pertenecen a dos departamentos muy separados, cada uno con su zar, sería difícil coincidir en una persona para que administre la celda y que tome decisiones que afecten a toda esta celda.

Con estas restricciones, es deseable tener la mínima cantidad posible de celdas para minimizar el número de operaciones que cruzan las fronteras de las celdas. Además, si un

intruso irrumpe en una celda y roba la base de datos de seguridad, establecen nuevas contraseñas entre todas las celdas. Mientras más de éstas haya, se necesitará más trabajo.

Para aclarar la idea de las celdas, consideremos dos ejemplos. El primero es un gran fabricante de equipo eléctrico, cuyos productos van de motores de avión hasta tostadores. El segundo es un editor cuyos libros abarcan todo, desde el arte hasta los zoológicos. Puesto que los productos del fabricante eléctrico son tan diversos, podría organizar sus celdas en torno de los productos, como se muestra en la figura 10-2(a), con diferentes celdas para el grupo de los tostadores y el grupo de los motores de avión. Cada celda contendría a las personas de diseño, fabricación y comercialización, suponiendo que las personas que comercializan los motores de avión necesitan mayor contacto con las personas que fabrican los motores de avión que con las personas que comercializan los tostadores.

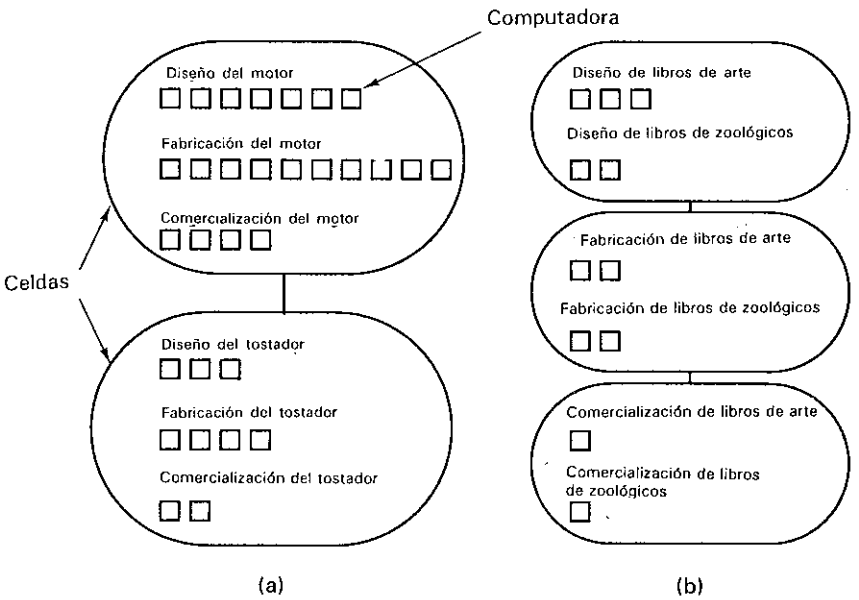


Figura 10-2. (a) Dos celdas organizadas por producto. (b) Tres celdas organizadas por función.

En contraste, el editor podría organizar las celdas como en la figura 10-2(b), puesto que la fabricación (es decir, la impresión y encuadernación) de un libro de arte es casi como fabricar un libro de zoológicos, de modo que las diferencias entre los departamentos son tal vez más significativas que las diferencias entre los productos. Por otro lado, si el editor tiene divisiones independientes para los libros infantiles, los libros de comercio y los libros de texto, las cuales eran en su origen compañías independientes con sus propias estructuras administrativas y culturas corporativas, tal vez sería mejor organizar las celdas por división y no por función.

El objetivo de estos ejemplos es hacer ver que la determinación de las fronteras de las celdas tienden a establecerse mediante consideraciones de la empresa y no por las propiedades técnicas de DCE.

El tamaño de una celda varía de manera considerable, pero todas las celdas contienen un servidor del tiempo, uno de directorios y uno de seguridad. Es posible que los tres servidores se ejecuten en una misma máquina. Además, la mayoría de las celdas contendrán una o más aplicaciones clientes (para realizar el trabajo) o uno o más servidores de aplicaciones (para los servicios). En una celda de gran tamaño, podrían existir varias instancias de los servidores de tiempo, de directorios y de seguridad, así como cientos de clientes y servidores.

10.2. HILOS

El paquete de hilos, junto con el paquete de RPC, es uno de los bloques fundamentales de construcción de DCE. En esta sección analizaremos el paquete de hilos de DCE, centrándonos en la planificación, la sincronización y las llamadas disponibles.

10.2.1. Introducción a los hilos de DCE

El paquete de hilos de DCE se basa en el estándar POSIX P1003.4a. Es una colección de procedimientos de biblioteca a nivel usuario que permiten a los procesos crear, eliminar, y manejar hilos. Sin embargo, si el sistema anfitrión tiene un paquete de hilos (del núcleo), el proveedor puede configurar a DCE para utilizarlo. Las llamadas básicas permiten crear y eliminar hilos, esperar un hilo, y sincronizar el cálculo entre hilos. Se dispone de muchas otras llamadas para manejar todos los detalles y otras funciones.

Un hilo está en uno de cuatro estados, como se muestra en la figura 10-3. Un hilo **en ejecución** es aquel que utiliza de manera activa el CPU para realizar cálculo. Un hilo **listo** está dispuesto y es capaz de ejecutarse, pero no puede hacerlo, pues el CPU está ocupado por el momento ejecutando otro hilo. En contraste, un hilo **en espera** es aquél que no se puede ejecutar de manera lógica, pues espera que ocurra algún evento (por ejemplo, que se abra un mútex). Por último, un hilo **terminado** es aquél que ha concluido pero no ha sido eliminado y cuya memoria (es decir, el espacio en la pila) no ha sido recuperado todavía. El hilo desaparecerá sólo hasta que otro hilo lo elimine de manera explícita.

En una máquina con un CPU, sólo se ejecuta un hilo en un instante dado. En un multiprocesador, ejecuta varios hilos dentro de un mismo proceso al mismo tiempo, en varios CPU (paralelismo verdadero).

El paquete de hilos ha sido diseñado para minimizar el efecto sobre el software existente, de modo que los programas diseñados para ambientes con un hilo se conviertan en procesos con varios hilos con un mínimo de trabajo. Lo ideal es que un programa con un hilo se convierta en uno con varios hilos activando sólo un parámetro que indique la posibilidad de utilizar más hilos. Sin embargo, pueden surgir problemas en tres áreas.

El primer problema se relaciona con las señales. Éstas se pueden dejar en su estado por omisión, ser ignoradas o ser captadas. Algunas señales son síncronas, causadas en especial por el hilo en ejecución. Éstas incluyen la excepción de punto flotante, que provoca una

Estado	Descripción
En ejecución	El hilo está utilizando de manera activa al CPU
Listo	El hilo espera su ejecución
En espera	El hilo está bloqueado y espera cierto evento
Terminado	El hilo ha concluido pero no ha sido destruido

Figura 10-3. Un hilo puede estar en uno de cuatro estados.

violación a la protección de la memoria y la terminación del propio cronómetro. Otras son asíncronas, causadas por algún agente externo, como cuando el usuario oprime la tecla DEL para interrumpir el proceso en ejecución.

Cuando ocurre una señal síncrona, ésta se maneja por el hilo activo, excepto que si no es ignorada ni captada, se elimina a todo el proceso. Cuando ocurre una señal asíncrona, el paquete de hilos verifica si existen hilos que la esperen. En tal caso, la señal se transfiere a todos los hilos que desean manejarla.

El segundo problema se relaciona con la biblioteca estándar, la mayoría de cuyos procedimientos no son reentrantes. Ocurre que un hilo esté asignando memoria, cuando una interrupción de reloj obliga a un intercambio de hilos. En el momento del intercambio, las estructuras de datos internos del asignador de memoria podrían ser inconsistentes, lo que provocaría problemas si el hilo recién planificado intenta asignar cierta memoria.

Este problema se resuelve proporcionando jackets en torno a alguno de los procedimientos de biblioteca (principalmente, los procedimientos de E/S) que proporcionan una exclusión mutua para las llamadas individuales. Para los demás procedimientos, un mútex global garantiza que sólo esté activo un hilo en la biblioteca a la vez. Los procedimientos de biblioteca como *read* y *fork* son todos procedimientos con jackets que controlan la exclusión mutua y que después llaman a otro procedimiento (oculto) para realizar el trabajo. Esta solución es algo así como una solución rápida; una mejor solución sería proporcionar una biblioteca adecuada reentrante.

El tercer problema tiene que ver con el hecho de que las llamadas al sistema UNIX regresan a su estado de error en una variable global, *errno*. Si un hilo realiza una llamada al sistema y, justo antes de que termine la llamada, se planifica otro hilo que, a su vez, realiza una llamada al sistema, se perderá el valor original de *errno*. Una solución consiste en proporcionar una interfaz alternativa para el manejo de errores. Ésta consta de un macro que permite al programador inspeccionar una versión de *errno* específica del hilo que se guarda y se restaura después del intercambio de hilos. Esta solución evita la necesidad de examinar la versión global de *errno*. Además, es posible que las llamadas al sistema indiquen los errores, por medio de excepciones, lo cual evitaría por completo el problema.

10.2.2. Planificación

La planificación de hilos es similar a la de procesos, excepto que es visible a la aplicación. El algoritmo de planificación determina el tiempo de ejecución de cada hilo, así como el hilo que se ejecuta a continuación. Al igual que en la planificación de procesos, se pueden construir muchos algoritmos de planificación de hilos.

Los hilos en DCE tienen prioridades y éstas son respetadas por el algoritmo de planificación. Se supone que los hilos con prioridad alta son más importantes que los hilos de prioridad baja, y por lo tanto tiene un mejor tratamiento, lo que significa que se ejecutan primero y con una mayor porción del CPU.

DCE soporta los tres algoritmos de planificación de hilos que se muestran en la figura 10.4. El primero, FIFO, busca en las colas de prioridad, de mayor a menor, para localizar la cola de máxima prioridad, con uno o más hilos en ella. Entonces se ejecuta el primer hilo de esta cola hasta que concluye, ya sea mediante un bloqueo o su conclusión. En principio, el hilo seleccionado se puede ejecutar tanto como sea necesario. Cuando ha terminado, se elimina de la cola de hilos ejecutables. Después, el planificador busca de nuevo en las colas, de mayor a menor, y considera el siguiente hilo encontrado.

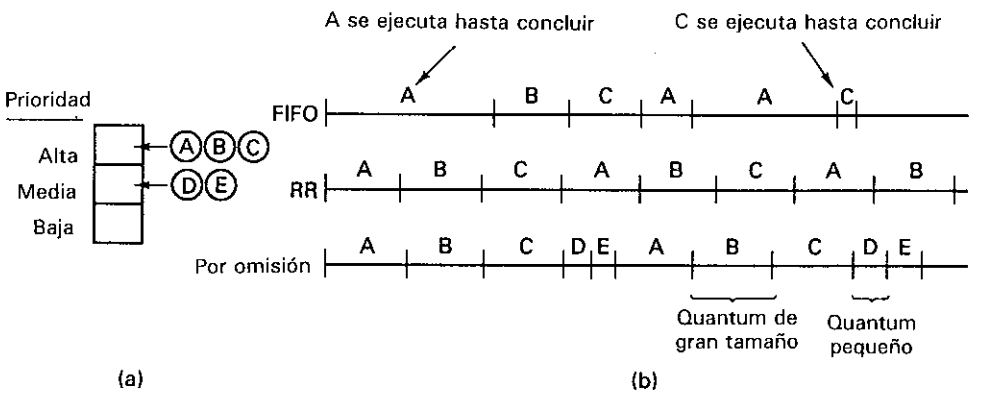


Figura 10-4. (a) Un sistema con cinco hilos y tres niveles de prioridad. (b) Tres algoritmos de planificación de hilos.

El segundo algoritmo es round robin. En este caso, el planificador localiza la cola más poblada y ejecuta cada hilo durante un quantum fijo. Si un hilo se bloquea o incluso antes de que se termine su quantum, se le elimina (de forma temporal) del sistema de colas. Si agota todo su quantum, se suspende y coloca al final de su cola. En el ejemplo intermedio de la figura 10.4(b), los hilos *A*, *B* y *C* se ejecutarán por siempre, de manera alternativa, si lo desean. Las colas de prioridad media *D* y *E* nunca tendrán una oportunidad, mientras una de las colas de máxima prioridad desee ejecutarse.

El tercer algoritmo es el definido por omisión. Ejecuta los hilos de todas las colas mediante un algoritmo de round robin con división del tiempo, pero mientras mayor sea la

prioridad, el hilo tendrá mayor quantum. De esta manera, todos los hilos se ejecutan y no existe muerte por inanición como en el segundo algoritmo.

Existe un cuarto algoritmo que tiene quanta de tamaño variable, pero con inanición. Sin embargo, éste no se define en POSIX, por lo que no es portable y debe ser evitado.

10.2.3. Sincronización

DCE proporciona dos formas de sincronizar los hilos: los mútex y las variables de condición. Los mútex se utilizan cuando es esencial evitar que varios hilos tengan acceso al mismo recurso y al mismo tiempo. Por ejemplo, al mover elementos en una lista ligada, a mitad de un movimiento, la lista estará en un estado inconsistente. Para evitar un desastre, cuando un hilo controla la lista, los demás hilos se mantienen a distancia. Al pedir que un hilo cierre primero con éxito al mútex asociado con la lista antes de tocar ésta (y abrirlo después), se garantiza una operación correcta.

Existen tres tipos de mútex, como se muestra en la figura 10-5. Difieren en la forma con que trabajan las cerraduras anidadas. Un **mútex rápido** es como una cerradura en un sistema de base de datos. Si un proceso intenta cerrar un registro no cerrado, tendrá éxito. Sin embargo, si adquiere la misma cerradura por segunda vez, se bloqueará, en espera de la liberación de la cerradura, algo que tal vez nunca ocurra. Es probable que ocurra un bloqueo.

Tipo de mútex	Propiedades
Rápido	Al cerrarlo una segunda vez causa un bloqueo
Recursivo	Se permite cerrarlo una segunda vez
No recursivo	Al cerrarlo una segunda vez se obtiene un error

Figura 10-5. Tres tipos de mútex soportados por DCE.

Un mútex **recursivo** permite que un hilo cierre un mútex ya cerrado. La idea es ésta. Supongamos que el programa principal de un hilo cierra un mútex y que después llama a un procedimiento que también cierra al mútex. Para evitar un bloqueo, se acepta la segunda cerradura. Mientras el mútex esté cerrado tantas veces como no cerrado, el anidamiento puede ser tan profundo como se desee. Como compromiso, DCE proporciona un tercer tipo de mútex, aquel en que un intento por cerrar un mútex ya cerrado no provoca un bloqueo, pero sí regresa un error.

Las **variables de condición** proporcionan un segundo mecanismo de sincronización. Éstas se utilizan junto con los mútex. Por lo general, cuando un hilo necesita cierto recurso, utiliza un mútex para obtener el acceso exclusivo a una estructura de datos que lleva un registro del estado del recurso. Si el recurso no está disponible, el hilo espera una variable

de condición, la que suspende de manera atómica al hilo y libera al mútex. Posteriormente, cuando otro hilo hace una señal a la variable de condición, se reinicia el hilo en espera.

10.2.4. Llamadas a hilos

El paquete de hilos de DCE tiene un total de 54 primitivas (procedimientos de biblioteca). Muchas de ellas no son muy necesarias, pero se proporcionan sólo por conveniencia. Este punto de vista es análogo a una calculadora de bolsillo de cuatro funciones, la cual sólo posee las teclas de +, –, × y /, sino que además tiene teclas para +1, –1, × 2, × 10, × π, /2 y /10, bajo el supuesto de que estas teclas ahorran tiempo y esfuerzo al usuario. Debido al gran número de llamadas, sólo analizaremos las más importantes (cerca de la mitad del total). Sin embargo, nuestro estudio proporciona una impresión razonable de la funcionalidad disponible.

Llamada	Descripción
Create	Crea un nuevo hilo
Exit	Un hilo la llama cuando éste termina
Join	Similar a la llamada al sistema WAIT en UNIX
Detach	Hace innecesaria la espera de un hilo padre cuando termina el proceso que hizo la llamada

Figura 10-6. Selección de llamadas a hilos de DCE para el manejo de los hijos. Todas las llamadas en esta sección tienen un prefijo *pthread* (es decir, *ptrhread_create*, no *create*) que omitimos por razones de espacio.

Para nuestro análisis, es conveniente agrupar las llamadas en siete categorías, cada una de las cuales se refiere a un aspecto distinto de los hilos y su uso. La primera categoría, enumerada en la figura 10-6, trata del manejo de los hilos. Estas llamadas permiten la creación de hilos y su salida cuando terminan su labor. Un hilo padre espera a un hijo mediante *join*, que es similar a la llamada al sistema WAIT de UNIX. Si un padre no tiene interés en un hijo y no va a esperarlo, renuncia a él mediante *detach*. En este caso, cuando el hilo hijo hace su salida, su espacio de almacenamiento se reclama de manera inmediata, en vez de esperar a que el padre llame a *join*.

El paquete DCE permite que el usuario cree, destruya y maneje **plantillas** para los hilos, mútex y variables de condición. Las plantillas se pueden configurar de modo que tengan los valores iniciales adecuados. Al crearse un objeto, uno de los parámetros de la llamada *create* es un apuntador a una plantilla. Así, por ejemplo, se crea una plantilla de hilo con el atributo (propiedad) de que el tamaño de la pila sea de 4K. Siempre que se cree un hilo con esta plantilla como parámetro, tendrá una pila de 4K. El hecho de contar con plantillas elimina la necesidad de especificar todas las opciones como parámetros inde-

pendientes. Al evolucionar el paquete, las llamadas *create* siguen siendo las mismas. En vez de esto, se pueden añadir nuevos atributos a los patrones. En la figura 10-7 se enumeran algunas de las llamadas para plantillas.

Llamada	Descripción
Attr_create	Crea una plantilla para establecer los parámetros del hilo
Attr_delete	Elimina una plantilla para hilos
Attr_setprio	Establece la prioridad de planificación por omisión en la plantilla
Attr_getprio	Lee la prioridad preestablecida de planificación de la plantilla
Attr_setstacksize	Establece el tamaño de la pila por omisión en la plantilla
Attr_getstacksize	Lee el tamaño de la pila por omisión de la plantilla
Attr_mutexattr_create	Crea una plantilla para los parámetros del mútex
Attr_mutexattr_delete	Elimina la plantilla para mútex
Attr_mutexattr_setkind_np	Determina el tipo de mútex por omisión en la plantilla
Attr_mutexattr_getkind_np	Lee el tipo de mútex por omisión de la plantilla
Attr_condattr_create	Crea una plantilla para los parámetros de las variables de condición
Attr_condattr_delete	Elimina la plantilla para las variables de condición

Figura 10-7. Algunas de las llamadas de plantilla.

Las llamadas *attr_create* y *attr_delete* crean y eliminan plantillas para hilos, respectivamente. Otras llamadas permiten a los programas leer y escribir los atributos de las plantillas, como el tamaño de la pila y los parámetros de planificación, los cuales se utilizarán cuando se creen hilos mediante la plantilla. De manera análoga, se dispone de llamadas para crear y eliminar plantillas para mútex y variables de condición. La necesidad de estas últimas no es del todo obvia, puesto que no tienen atributos ni operaciones definidas en ellas. Tal vez los diseñadores esperan que algún día alguien piense en cierto atributo.

El tercer grupo se refiere a los mútex, que se pueden crear y destruir de manera dinámica. Se definen tres operaciones en los mútex, como se muestra en la figura 10-8. Las operaciones consisten en cerrar, abrir e intentar abrir; en este último caso, se acepta un fallo en caso que no se lleve a cabo la cerradura.

Llamada	Descripción
Mutex_init	Crea un mútex
Mutex_destroy	Elimina un mútex
Mutex_lock	Intenta cerrar un mútex; si ya está cerrado, se bloquea
Mutex_trylock	Intenta cerrar un mútex; falla si ya está cerrado
Mutex_unlock	Abre un mútex

Figura 10-8. Selección de llamadas a mútex.

A continuación vienen las llamadas relativas a las variables de condición, enumeradas en la figura 10-9. Éstas también se crean y destruyen de manera dinámica. Los hilos pueden dormir debido a variables de condición, pendientes de la disponibilidad de cierto recurso necesario. Se tienen dos operaciones para despertarlos: la señalización, que despierta con exactitud un hilo; y la transmisión, que despierta a todos.

Llamada	Descripción
Cond_init	Crea una variable de condición
Cond_destroy	Elimina una variable de condición
Cond_wait	Espera en una variable de condición hasta que llega una señal o transmisión
Cond_signal	Despierta a lo más a un hilo que espera una variable de condición
Cond_broadcast	Despierta a todos los hilos que esperan una variable de condición

Figura 10-9. Selección de llamadas de variable de condición.

La figura 10-10 enumera las tres llamadas para el manejo de variables globales por cada hilo. Éstas son variables que utilizan cualquier procedimiento del hilo, pero que no utilizan procedimiento alguno fuera del hilo. El concepto de variables globales para cada hilo no es soportado por los lenguajes de programación populares, por lo que esto se debe manejar durante el tiempo de ejecución. La primera llamada crea un identificador y asigna espacio de almacenamiento, la segunda asigna un apuntador a una variable global por hilo y la tercera permite al hilo leer el valor de una variable global por hilo. Muchos científicos de la computación consideran a las variables globales dentro del mismo equipo del gran

paria de todos los tiempos, el enunciado GOTO, por lo que sin duda se alegrarían de saber la forma de dificultar su uso. (El autor intentó alguna vez diseñar un lenguaje de programación con un enunciado del estilo

SEQUEESTOESUNAIDEAESTUPIDAPERONECESITOUNGOTO ALGO;

pero fue desalentado por sus compañeros). Por otro lado, se argumenta que el hecho de que las variables globales para cada hilo utilicen llamadas a procedimientos en vez de reglas definidas por la intención del lenguaje (como locales o globales) es una medida de emergencia, la cual se utiliza simplemente porque la mayoría de los lenguajes de programación no permiten que el concepto se exprese de manera sintáctica.

Llamada	Descripción
Keycreate	Crea una variable global para este hilo
Setspecific	Asigna un valor a un apuntador a una variable global por hilo
Getspecific	Lee el valor de un apuntador a una variable global por hilo

Figura 10-10. Selección de llamadas a variables globales por hilo.

El siguiente grupo de llamadas (véase figura 10-11) trata de la eliminación de los hilos y la capacidad de resistencia de éstos. La llamada *cancel* elimina un hilo, pero a veces, esto tiene efectos devastadores, como por ejemplo, en el caso en que el hilo tenga cerrado un mútex en ese momento. Por esta razón, los hilos habilitan o inhabilitan los intentos por eliminarlos de varias formas, lo cual es un poco similar a la capacidad de los procesos en UNIX de captar o ignorar señales sin ser terminados por éstas.

Llamada	Descripción
Cancel	Intenta eliminar otro hilo
Setcancel	Habilita o inhabilita la capacidad de otros hilos para eliminar este hilo

Figura 10-11. Selección de llamadas relativas a la eliminación de hilos.

Para finalizar, nuestro último grupo (véase figura 10-12) se refiere a la planificación. El paquete permite que los hilos de un proceso se planifiquen mediante los algoritmos FIFO, round robin, con prioridad, sin prioridad y otros más. Mediante estas llamadas, se configura el algoritmo y las prioridades. El sistema funciona mejor si los hilos no eligen ser planificados con algoritmos conflictivos.

Llamada	Descripción
Setscheduler	Establece el algoritmo de planificación
Getscheduler	Lee el algoritmo de planificación actual
Setprio	Establece la prioridad de planificación
Getprio	Obtiene la prioridad de planificación actual

Figura 10-12. Selección de llamadas de planificación.

10.3. LLAMADA A PROCEDIMIENTOS REMOTOS

DCE se basa en el modelo cliente/servidor. Los clientes solicitan servicios mediante llamadas a procedimientos remotos a los servidores distantes. En esta sección describiremos la forma en que este mecanismo se presenta a ambos lados y su implantación.

10.3.1. Objetivos de la RPC de DCE

Los objetivos del sistema RPC de DCE son relativamente tradicionales. En primer lugar y principalmente, el sistema RPC permite que un cliente tenga acceso a un servicio remoto mediante una sencilla llamada a un procedimiento local. Esta interfaz permite escribir con facilidad los programas clientes (es decir, de aplicación), de manera familiar para la mayoría de los programadores. También facilita el hecho de tener grandes cantidades de códigos en ejecución en un ambiente distribuido con pocas, sino es que ninguna, modificaciones.

El sistema RPC se encarga de ocultar todos los detalles a los clientes y, en cierta medida, también a los servidores. Para comenzar, el sistema RPC puede localizar de manera automática al servidor correcto y enlazarse con él, sin que el cliente sea consciente de que esto ocurre. También controla el transporte de mensajes en ambas direcciones, fragmentando y reensamblando éstos en caso necesario (por ejemplo, si uno de los parámetros es un arreglo de gran tamaño. Por último, el sistema RPC controla de manera automática las conversiones de tipos de datos entre el cliente y el servidor, aunque se ejecuten en arquitecturas diferentes y tengan un orden distinto para los bytes.

Como consecuencia de la habilidad del sistema RPC para ocultar los detalles, los clientes y servidores son muy independientes entre sí. Un cliente puede estar escrito en C y un servidor en FORTRAN, o viceversa. Un cliente y un servidor se pueden ejecutar en diferentes plataformas de hardware y utilizar sistemas operativos diferentes. También soportan varios protocolos de red y representaciones de datos, todo sin la intervención del cliente o del servidor.

10.3.2. Escritura a un cliente y un servidor

El sistema RPC de DCE consta de varios componentes, que incluyen lenguajes, bibliotecas, demonios y programas de utilería, entre otros. Juntos permiten escribir los clientes y los servidores. En esta sección describiremos las partes y la forma en que interactúan entre sí. Todo el proceso de escritura y uso de un cliente y servidor RPC se resume en la figura 10-13.

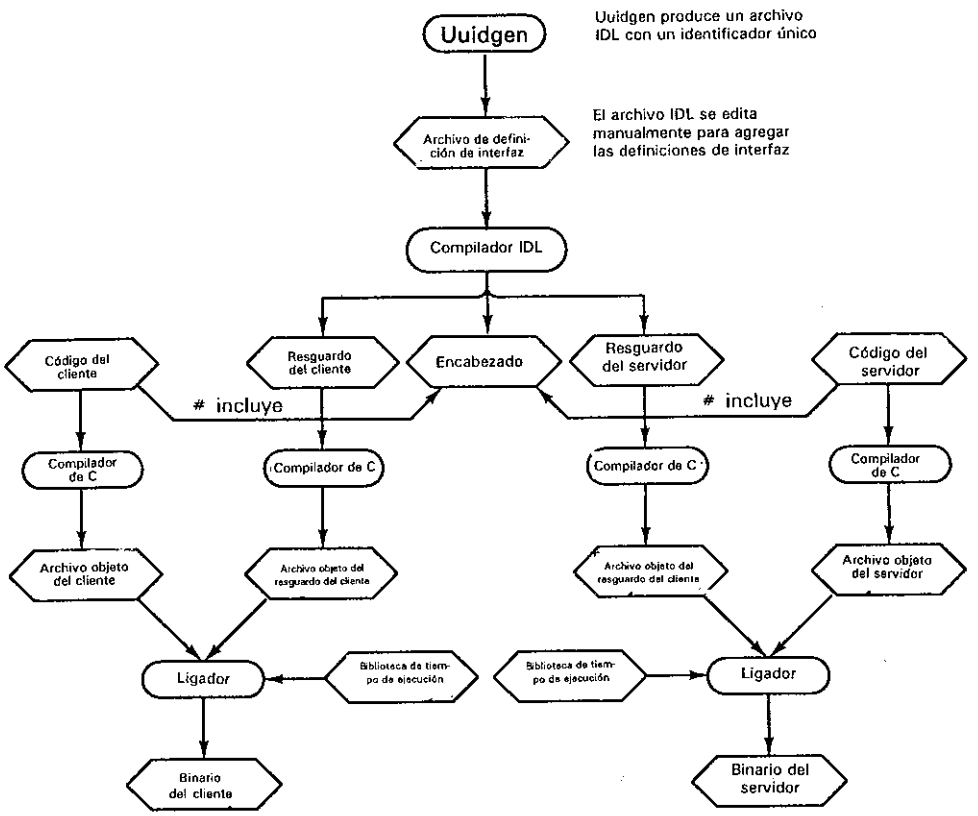


Figura 10-13. Los pasos para escribir un cliente y un servidor.

En un sistema cliente/servidor, el pegamento que une todo es la definición de la interfaz. Es en realidad un contrato entre el servidor y sus clientes, que especifica los servicios que el servidor ofrece a sus clientes.

La representación concreta de este contrato es un archivo, el archivo de definición de la interfaz, que enumera todos los procedimientos que el servidor permite que sus clientes llamen de manera remota. Cada procedimiento tiene una lista de los nombres y tipos de

sus parámetros y de su resultado. Lo ideal es que la definición de la interfaz también contenga una definición formal de lo que hacen sus procedimientos, pero tal definición está más allá del estado actual de las cosas, por lo que la definición de la interfaz sólo define la sintaxis de las llamadas y no su semántica. A lo más, el autor agrega unos cuantos comentarios describiendo lo que espera que hagan los procedimientos.

Las definiciones de interfaz se escriben en un lenguaje brillantemente llamado **lenguaje de definición de interfaz**, o **IDL**. Permite realizar declaraciones de procedimientos en una forma que recuerda en mucho a los prototipos de funciones en ANSI C. Los archivos IDL también contienen definiciones de tipos, declaraciones de constantes, y demás información necesaria para ordenar correctamente los parámetros y reordenar los resultados.

Un elemento crucial en todo archivo IDL es un identificador único en la interfaz. El cliente envía este identificador en el primer mensaje RPC y el servidor verifica que sea correcto. De esta manera, si un cliente intentase una conexión con el servidor equivocado, o incluso a una versión anterior del servidor correcto, el servidor detectará el error y no se realizará la conexión.

Las definiciones de interfaz y los identificadores únicos están muy ligados en DCE. Como se muestra en la figura 10-13, en el primer paso para escribir una aplicación cliente/servidor se llama por lo general al programa *uuidgen*, solicitando que genere un archivo IDL prototipo con el identificador de interfaz, garantizando que no utilizará de nuevo otra interfaz generada en otra parte por *uuidgen*. La unicidad queda garantizada al codificar en el archivo la posición y el instante de su creación. Consta de un número binario de 128 bits representado en el archivo IDL como cadena ASCII en hexadecimal.

El siguiente paso es editar el archivo IDL, llenando los nombres de los procedimientos remotos y sus parámetros. Es importante observar que la RPC no es por completo transparente; por ejemplo, el cliente y el servidor no comparten variables globales, pero las reglas de IDL impiden expresar construcciones no soportadas.

Cuando el archivo IDL está completo, se llama al compilador IDL para que lo procese. La salida del compilador IDL consta de tres archivos:

1. Un archivo de encabezado (por ejemplo, *interface.h*, en términos de C).
2. El resguardo del cliente.
3. El resguardo del servidor.

El archivo de encabezado contiene al identificador único, las definiciones de tipo, las definiciones constantes y los prototipos de función. Debe incluirse (mediante *#include*) en el código del cliente y del servidor.

El resguardo del cliente contiene los procedimientos reales que el programa cliente llamará. Estos procedimientos son responsables de reunir y empaquetar los parámetros en el mensaje de salida y después llamar al sistema de tiempo de ejecución para su envío. El resguardo del cliente también controla el desempaque de la respuesta y el regreso de valores al cliente.

El resguardo del servidor contiene los procedimientos llamados por el sistema de tiempo de ejecución en la máquina servidor cuando llega un mensaje. Éstos, a su vez, llaman a los procedimientos reales del servidor que hacen el trabajo.

El siguiente paso es que el autor de la aplicación escriba el código del cliente y del servidor. Después se compilan ambos, así como los dos procedimientos de resguardo. Los archivos objetos del código del cliente y del resguardo del cliente se ligan con la biblioteca del tiempo de ejecución para producir el binario ejecutable para el cliente. De manera análoga, se compilan y ligan el código del servidor y el resguardo del servidor para producir el binario del servidor. En el tiempo de ejecución, se inician el cliente y el servidor para poder ejecutar la aplicación.

10.3.3. Conexión de un cliente con un servidor

Antes de que un cliente llame a un servidor, tiene que localizarlo y conectarse a él. Los usuarios ingenuos ignoran el proceso de conexión y dejan que los resguardos se encarguen de esto de manera automática, pero la conexión ocurre a pesar de todo. Los usuarios sofisticados pueden controlarla con todo detalle; por ejemplo, para seleccionar un servidor específico en una celda distante particular. En esta sección describiremos el funcionamiento de la conexión en DCE.

El principal problema de la conexión es la forma en que el cliente localiza al servidor correcto. En teoría, la transmisión de un mensaje con el identificador único a cada proceso en cada celda, pide que todos los servidores del proceso levanten la mano (sin considerar los problemas de seguridad), pero este punto de vista es tan lento y desperdiciado que no es práctico. Además, no todas las redes soportan la transmisión.

En vez de esto, la localización del servidor se realiza en dos pasos:

1. Localizar la máquina servidor.
2. Localizar el proceso correcto en esa máquina.

Se utilizan diferentes mecanismos para cada uno de estos pasos. La necesidad de localizar la máquina servidor es obvia, pero el problema de localizar el servidor una vez conocida la máquina es más sutil. Básicamente, lo que ocurre es que para que un cliente se comunique de manera confiable y segura con un servidor, se necesita por lo general una conexión de red. Tal conexión necesita un **extremo**, una dirección numérica en la máquina servidor a la que llegan las conexiones de la red y a la cual enviar los mensajes. Es riesgoso que el servidor elija una dirección numérica permanente, puesto que otro servidor de la misma máquina podría elegir por accidente la misma. Por esta razón, los extremos se asignan de manera dinámica, y se mantiene una base de datos de entradas (servidor,extremo) en cada máquina servidor mediante un proceso llamado el **demonio RPC**, como se describe a continuación.

Los pasos implicados en la conexión se muestran en la figura 10-14. Antes de estar disponible para las solicitudes que lleguen, el servidor debe solicitar al sistema operativo

un extremo. Después registra este extremo con el demonio RPC. El demonio RPC registra esta información (incluyendo los protocolos con los que habla el servidor) en tabla de extremos para su uso futuro. El servidor también se registra con algún servidor de directorios de la celda, transfiriendo el número de su anfitrión.

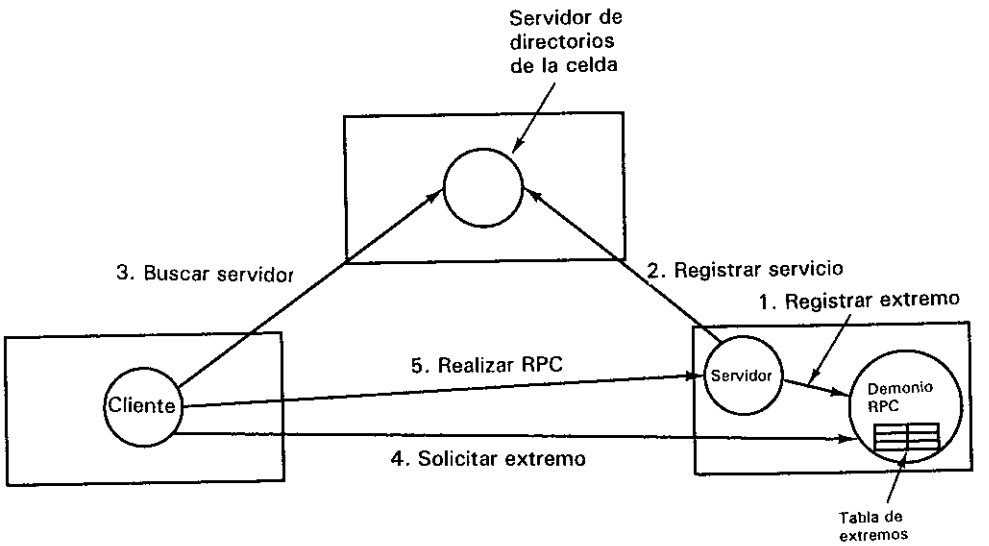


Figura 10-14. Conexión cliente-servidor en DCE.

Veamos ahora el lado del cliente. En el caso más sencillo, al momento de la primera RPC, el resguardo del cliente solicita al servidor de directorios que encuentre un anfitrión que ejecute una instancia del servidor. El cliente va entonces al demonio RPC, que tiene un extremo bien conocido, y solicita que busque el extremo (por ejemplo, el puerto TCP) en su tabla de extremos. Con esta información, se realiza ahora la RPC. En las siguientes RPC, esta búsqueda no es necesaria. DCE permite también a los clientes realizar búsquedas más complejas para un servidor adecuado, en caso necesario. La RPC autenticada también es una opción. Analizaremos la autenticación de protección en una sección posterior de este capítulo.

10.3.4. Realización de una RPC

La RPC real se realiza de manera transparente y de la forma usual. El resguardo del cliente ordena los parámetros y transfiere el buffer resultante (tal vez por partes) a la biblioteca de tiempo de ejecución para su transmisión, utilizando el protocolo elegido al momento de la conexión. Cuando llega un mensaje del lado del servidor, se rutea al servidor correcto con base en el extremo contenido en el mensaje recibido. La biblioteca de tiempo

de ejecución pasa el mensaje al resguardo del servidor, que reordena los parámetros y llama al servidor. La respuesta regresa por la ruta inversa.

DCE proporciona varias opciones de semántica. Por omisión, se utiliza la operación "a lo más una", en cuyo caso una llamada no se realiza más de una vez, incluso aunque falle el sistema. En la práctica, lo que esto significa es que si un servidor falla durante una RPC y después se recupera con rapidez, el cliente no repite la operación, por miedo a que haya sido realizada una vez.

Otra alternativa es marcar un procedimiento remoto como idempotente (en el archivo IDL), en cuyo caso se repite varias veces sin causar daño. Por ejemplo, la lectura de un bloque específico de un archivo se intenta una y otra vez hasta tener éxito. Cuando una RPC idempotente no funciona debido a una falla del servidor, el cliente espera hasta que el servidor arranque de nuevo y después intenta otra vez. En teoría, se dispone de otras semánticas (pero rara vez se utilizan), incluyendo la transmisión de la RPC a todas las máquinas en la red local.

10.4. SERVICIO DE TIEMPO

El tiempo es un concepto importante en la mayoría de los sistemas distribuidos. Para ver por qué, consideremos un programa de investigación en radioastronomía. Varios radiotelescopios diseminados en todo el mundo observan la misma fuente de radio celeste de manera simultánea, registrando con precisión los datos y el tiempo de observación. Los datos se envían a través de una red a una computadora central para su procesamiento. Para ciertos tipos de experimentos (por ejemplo, la interferometría long-baseline), es esencial para el análisis que los diversos flujos de datos se sincronicen con exactitud. Así, el experimento tiene éxito o fracasa si se tiene la habilidad para sincronizar los relojes remotos con precisión.

Como otro ejemplo, en un sistema computarizado de intercambio de acciones, podría ser importante saber quién realiza primero una oferta sobre un bloque de acciones a la venta. Si los postulantes estuvieran en Nueva York, San Francisco, Londres y Tokio, habría que marcar las ofertas con cierto tiempo para ser justos. Sin embargo, si los relojes de todos estos lugares no se sincronizan de manera adecuada, todo el esquema se colapsaría y producirían numerosos escenarios legales con expertos tratando de explicar el concepto de la velocidad de la luz a un jurado confundido.

Para evitar problemas como éstos, DCE tiene un servicio llamado **DTS (Servicio distribuido de tiempo)**. El objetivo de DTS es mantener sincronizados los relojes de las máquinas separadas. Sincronizarlos una vez no es suficiente, pues los cristales de los diferentes relojes vibran con tasas ligeramente diferentes, de modo que los relojes se apartan de forma gradual. Por ejemplo, un reloj podría tener un error relativo de una parte en un millón. Esto significa que, aunque fuera configurado perfectamente, después de una hora el reloj fallaría en 3.6 milisegundos en algún sentido. Después de un día, estaría desfasado

86 milisegundos. Después de un mes, dos relojes sincronizados precisamente podrían diferir en 5 segundos.

DTS administra los relojes en DCE. Consta de servidores de tiempo que se mantienen preguntando entre sí: "¿qué hora es?" así como otras componentes. Si DTS conoce la tasa máxima de desviación de cada reloj (que sí es conocida, ya que mide la tasa de desviación), puede realizar los cálculos de tiempo con la frecuencia necesaria para lograr la sincronización deseada. Por ejemplo, con relojes precisos hasta una parte en un millón, si ninguno de los relojes debe desfasarse más de 10 milisegundos, la resincronización se realiza al menos cada tres horas.

En realidad, DTS debe tratar dos aspectos separados:

1. Mantener los relojes mutuamente consistentes.
2. Mantener los relojes en contacto con la realidad.

El primer punto tiene que ver con la garantía de que todos los relojes regresen el mismo valor cuando se les pregunte por el tiempo (con una corrección para los husos horarios, por supuesto). El segundo punto tiene que ver con garantizar que aunque todos los relojes regresen la misma hora, ésta coincida con los relojes del mundo real. El hecho de que todos los relojes coincidan en que la hora es 12:04:00.000 es de poco consuelo si en realidad son las 12:05:30.000. A continuación describimos la forma como DTS logra estos objetivos, pero primero explicaremos que es el modelo de tiempo de DTS y la interfaz de los programas con él.

10.4.1. Modelo de tiempo DTS

A diferencia de la mayoría de los sistemas, en donde el tiempo actual es sólo un número binario, en DTS, todos los tiempos se registran como intervalos. Cuando se pregunta la hora, en vez de decir que son las 9:52, DTS podría decir que está entre 9:51 y 9:53 (exageradamente). El uso de intervalos en vez de valores permite a DTS proporcionar al usuario una especificación precisa del desfasamiento del reloj.

De manera interna, DTS mantiene un registro del tiempo mediante un número binario de 64 bits que inició con el principio del tiempo. A diferencia de UNIX, en donde el tiempo comenzó en 0000 el primero de enero de 1970, o de TAI, que comenzó en 0000 el primero de enero de 1958, el principio del tiempo en DTS es 0000 el 15 de octubre de 1582, la fecha en que el calendario gregoriano fue introducido en Italia. (Uno nunca sabe cuándo podría utilizarse un antiguo programa FORTRAN del siglo XVII.)

No se espera que las personas trabajen con la representación binaria del tiempo. Sólo se utiliza para guardar los tiempos y compararlos. Al desplegarlos, los tiempos aparecen en el formato de la figura 10-15. Esta representación se basa en el estándar internacional 8601 que resuelve el problema de si las fechas deben escribirse como mes/día/año (como en Estados Unidos) o día/mes/año (como en el resto de los países), sin adoptar ninguno de estos modelos. Utiliza un reloj de 24 horas y registra los segundos con una precisión de

0.001 segundos. También incluye de manera eficaz el huso horario, proporcionando la diferencia del tiempo con la hora media de Greenwich. Por último, el aspecto más importante es que la imprecisión está dada después de la "I" en segundos. En este ejemplo, la imprecisión es de 5.000 segundos, lo que significa que el tiempo real podría estar entre las 3:29:55 P.M. y las 3:30:05 P.M. Además de los tiempos absolutos, DTS también controla las diferencias de tiempo, incluyendo el aspecto de la imprecisión.

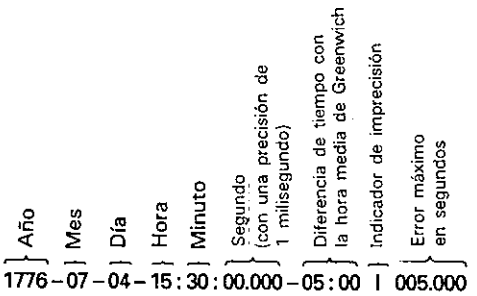


Figura 10-15. Despliegue del tiempo en DTS.

El registro de tiempos como intervalos introduce un problema que no estaba presente en otros sistemas: no siempre sirve decir esto si una hora es anterior a otra. Por ejemplo, consideremos el programa *make* de UNIX. Supongamos que un archivo fuente tiene un intervalo de tiempo de 10:35:10 a 10:35:15 y que el archivo binario correspondiente tiene el intervalo de tiempo 10:35:14 a 10:35:19. ¿Es más reciente el archivo binario? Tal vez, pero no definitivamente. El único recurso seguro para *make* es recompilar el archivo fuente.

En general, cuando un programa pide a DTS que compare dos tiempos, existen tres respuestas posibles:

1. El primer tiempo es anterior.
2. El segundo tiempo es anterior.
3. DTS no puede decir cuál es anterior.

El software que utiliza DTS debe estar preparado para enfrentar estas tres posibilidades. Para mantener una compatibilidad con software anterior, DTS también soporta una interfaz convencional donde el tiempo se representa con un valor, pero el uso a ciegas de este valor puede conducir a errores.

DTS soporta 33 llamadas (procedimientos de biblioteca) relacionadas con el tiempo. Estas llamadas se dividen en los seis grupos enumerados en la figura 10-16. Ahora mencionaremos de forma breve cada uno de éstos. El primer grupo obtiene la hora actual de DTS y la regresa. Los dos procedimientos difieren en la forma de manejo del huso horario. El segundo grupo controla la conversión del tiempo entre los valores binarios, los valores

Grupo	# llamadas	Descripción
Recuperación de tiempos	2	Obtienen la hora
Conversión de tiempos	18	Conversión binario-ASCII
Manejo de tiempos	3	Aritmética de intervalos
Comparación de tiempos	2	Compara dos tiempos
Cálculo con tiempos	5	Operaciones aritméticas con tiempos
Uso de husos horarios	3	Administración del huso horario

Figura 10-16. Grupos de llamadas relacionadas con el tiempo en DTS.

estructurados y los valores ASCII. El tercero presenta dos tiempos como entrada y obtiene un tiempo cuya imprecisión abarca todo el rango de tiempos posibles. El cuarto compara dos tiempos, utilizando o no la parte de imprecisión. El quinto grupo proporciona una forma para sumar dos tiempos, restar dos tiempos, multiplicar un tiempo relativo por una constante, etc. El último grupo controla los husos horarios.

10.4.2. Implantación de DTS

El servicio DTS consta (conceptualmente) de varias componentes. El **empleado del tiempo** es un proceso demonio que se ejecuta en las máquinas clientes y mantiene al reloj local sincronizado con los relojes remotos. El empleado lleva un registro de la incertidumbre del reloj local, que tiene un crecimiento lineal. Por ejemplo, un reloj con un error relativo de una parte en un millón ganaría o perdería tanto como 3.6 milisegundos por hora, como ya hemos analizado. Cuando el empleado del tiempo calcula que el valor posible ha rebasado el límite permitido, resincroniza.

Un empleado del tiempo resincroniza al hacer contacto con todos los **servidores de tiempo** en su LAN. Éstos son demonios cuyo trabajo es mantener el tiempo consistente y preciso dentro de límites conocidos. Por ejemplo, en la figura 10-17, un empleado del tiempo solicita y recibe la hora de cuatro servidores de tiempo. Cada uno proporciona un intervalo en donde piensa que está el UTC. El empleado calcula su nuevo valor de la hora como sigue. En primer lugar, los valores que no se intersecan con otros (como el de la fuente 4) se descartan como no confiables. Después se calcula la máxima intersección de los intervalos restantes. Por último, el empleado establece su valor de UTC como punto medio de este intervalo.

Después de resincronizar, un empleado tiene un nuevo UTC que por lo general es mayor o menor que su valor actual. Sólo podría ajustar su reloj al nuevo valor, pero por lo general es poco sabio hacer esto. En primero lugar, esto podría requerir retrasar el reloj, lo que

significa que los archivos creados después de la resincronización podrían parecer anteriores a archivos creados antes de ella. Programas como *make* se comportarían de manera incorrecta bajo estas circunstancias.

Aunque tuviera que adelantarse el reloj, es mejor no hacerlo de manera abrupta, puesto que algunos programas despliegan información y dan después al usuario cierto número de segundos para reaccionar. Si este intervalo se reduce en gran medida podría ocurrir, por ejemplo, que un sistema automatizado de exámenes desplegara una pregunta en la pantalla para un estudiante y terminara el tiempo de inmediato. Indicando al estudiante que ocupó demasiado tiempo para responderla.

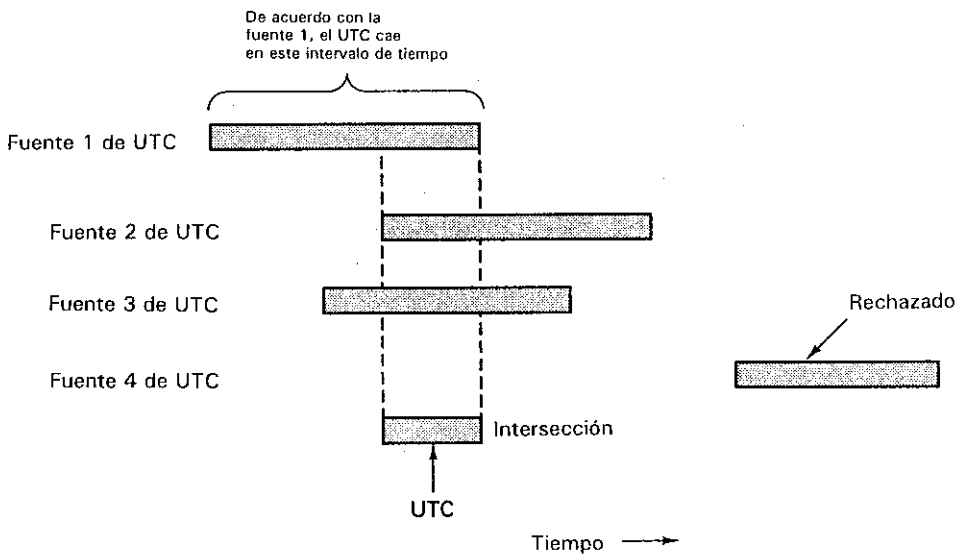


Figura 10-17. Cálculo del nuevo UTC mediante cuatro fuentes de tiempo.

En consecuencia, DTS hace la corrección de forma gradual. Por ejemplo, si el reloj está atrasado 5 segundos, en vez de sumar 10 milisegundos 100 veces por segundo, se podrían sumar 11 milisegundos en cada marca de tiempo durante los próximos 50 segundos.

Los servidores de tiempo tienen dos variantes, local y global. Los servidores locales participan en la administración del tiempo dentro de sus celdas. Los servidores locales mantienen sincronizados a los locales de tiempo de celdas diferentes. Los servidores de tiempo se comunican entre sí de manera periódica para mantener consistentes sus relojes. También utilizan el algoritmo de la figura 10-17 para elegir el nuevo UTC.

Aunque no es necesario, se obtienen mejores resultados, si uno o más servidores globales se conectan de forma directa a una fuente UTC por medio de una conexión por satélite, radio o teléfono. DTS define una interfaz especial, la **interfaz del proveedor de tiempo**, que define la forma en que DTS adquiere y distribuye el DTC a partir de fuentes externas.

10.5. SERVICIO DE DIRECTORIOS

Uno de los objetivos principales de DCE es que todos los recursos son accesibles a cualquier proceso del sistema, sin importar la posición relativa del usuario del recurso (cliente) ni la del proveedor del recurso (servidor). Estos recursos incluyen los usuarios, las máquinas, las celdas, los servidores, los archivos, los datos de seguridad y muchos otros. Para realizar este objetivo, es necesario que DCE mantenga un servicio de directorios, un registro de la posición de todos los recursos y que proporcione nombres amigables con el usuario para ellos. En esta sección describiremos este servicio y la forma en que opera.

El servicio de directorios de DCE está organizado por celda. Cada celda tiene un **servicio de directorios de celda (CDS)**, que guarda los nombres y propiedades de los recursos de la celda. Este servicio está organizado como sistema distribuido de base de datos, con réplicas, para proporcionar un buen desempeño y alta disponibilidad, incluso en presencia de fallas del servidor. Para operar, cada celda tiene al menos un servidor CDS en ejecución.

Cada recurso tiene un nombre, formado por el nombre de su celda seguido por el que utiliza dentro de ella. Para localizar un recurso, el servicio de directorios necesita una forma para localizar celdas. Se soportan dos de estos mecanismos, el **servicio global de directorios (GDS)** y el **sistema de nombres de dominio (DNS)**, GDS es el servicio "nativo" de DCE para localizar las celdas. Utiliza el estándar X.500. Sin embargo, como muchos usuarios de DCE utilizan Internet, también se soporta el sistema de asignación de nombres de Internet, DNS. Hubiera sido mejor tener un mecanismo para localizar las celdas (y una sintaxis para nombrarlas), pero consideraciones políticas hicieron esto imposible.

La relación entre estas componentes se muestra en la figura 10-18. Aquí vemos otra componente del servicio de directorios, el **agente global de directorios (GDA)**, que CDS utiliza para interactuar con GDS y DNS. Cuando CDS busca un nombre remoto, solicita a su GDA que realice el trabajo por él. Este diseño hace que CDS sea independiente de los protocolos utilizados por GDS y DNS. Como CDS y GDS, GDA se implanta como un proceso demonio que acepta solicitudes utilizando RPC y que regresa respuestas.

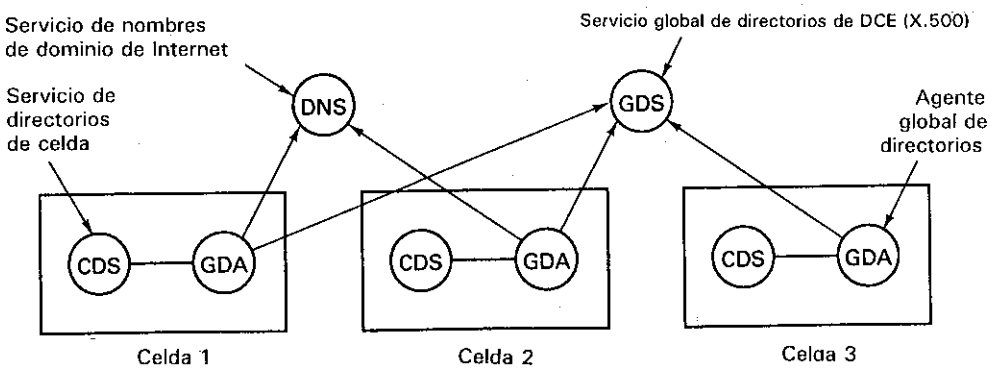


Figura 10-18. Relación entre CDS, GDS, GDA y DNS.

En la siguiente sección describiremos la formación de los nombres en DCE. Después de ello, examinaremos CDS y GDS.

10.5.1. Nombres

Cada recurso en DCE tiene un nombre. El conjunto de todos los nombres forma un espacio de nombres en DCE. Cada nombre tiene hasta cinco partes, algunas de las cuales son opcionales. Las cinco partes se muestran en la figura 10-19.

	Prefijo	Nombre de celda	Nombre	Unión	Nombre de aplicación
Ejemplos:	/...	/C=US/O=IBM/		/sec	/principals/linda
	/...	/lcs.mit.edu	/subsys		/file

Figura 10-19. Los nombres de DCE pueden tener hasta cinco partes.

La primera parte es el **prefijo**, que indica si el nombre es global a todo el espacio de nombres de DCE o si es local a la celda activa. El prefijo /.. indica un nombre global, mientras que el prefijo /.: denota un nombre local. Un nombre global contiene el nombre de la celda necesaria; un nombre local no. Cuando llega una solicitud a CDS, éste puede decir a partir del prefijo si maneja la solicitud por sí mismo o si la pasa al CDA para una búsqueda remota por GDS.

Los nombres de las celdas se especifican en notación X.500 o DNS. Ambos sistemas son muy elaborados, pero para nuestros fines bastará la siguiente introducción breve. X.500 es un estándar internacional para los nombres. Fue desarrollado dentro del mundo de las compañías telefónicas para proporcionar a los futuros clientes telefónicos un directorio telefónico electrónico. Se puede utilizar para nombrar personas, computadoras, servicios, celdas o todo aquello que necesite un nombre.

Toda entidad nombrada tiene una colección de atributos que la describen. Esto incluye su país (por ejemplo, US, GB, DE), su organización (por ejemplo, IBM, HARVARD, DOD), su departamento (por ejemplo, CS, SALES, TAX), así como otros puntos más detallados como el número de empleado, supervisor, número de oficina, número telefónico, y nombre. Cada atributo tiene un valor. Un nombre X.500 es una lista de elementos *atributo=valor* separados mediante diagonales. Por ejemplo,

/C=US/O=YALE/OU=CS/TITLE=PROF/TELEPHONE=3141/OFFICE=210/SURNAME=LIN/

podría describir al profesor Lin del departamento de ciencias de la computación de la universidad de Yale. Los atributos C, O y OU están presentes en la mayor parte de los nombres y se refieren al país, la organización y la unidad organizativa (departamento), respectivamente.

La idea detrás de X.500 es que una solicitud proporciona los atributos suficientes para que el objetivo quede especificado de manera única. En el ejemplo anterior, C, O, OU y SURNAME sirven, pero C, O, OU y OFFICE también bastarían, si el solicitante ha olvidado el nombre pero recuerda el número de oficina. No sería adecuado proporcionar todos los atributos excepto el país, en espera de que el servidor busque en todo el mundo una concordancia.

DNS es el esquema de Internet para nombrar los anfitriones y otros recursos. Divide el mundo hasta en dominios de nivel superior que constan de países y, en Estados Unidos, EDU (instituciones educativas), COM (compañías), MIL (sitios militares), y algunos otros más. Éstos, a su vez, tienen subdominios como *harvard.edu*, *princeton.edu* y *stanford.edu*, y sub-subdominios, como *cs.cmu.edu*. Se utilizan X.500 y DNS para especificar los nombres de las celdas. En la figura 10-19, las dos celdas de ejemplo podrían ser el departamento de impuestos de IBM y el laboratorio de ciencias de la computación en MIT.

El siguiente nivel del nombre es por lo general el nombre de un recurso estándar o una conexión, que es análoga a un punto de montaje de UNIX, y que provoca que la búsqueda se cambie a un sistema diferente de asignación de nombres, como el sistema de archivos o el sistema de seguridad. Por último, está el propio nombre del recurso.

10.5.2. El servicio de directorio de celda

El CDS maneja los nombres de una celda. Éstos se ordenan como una jerarquía, aunque como en UNIX, también existen los enlaces simbólicos (llamados **enlaces suaves**). Un ejemplo de la parte superior del árbol para una celda sencilla aparece en la figura 10-20.

El directorio del nivel superior contiene dos archivos de perfil que contienen la información de conexión de RPC; uno de los archivos es topológicamente independiente y uno de ellos refleja la topología de red para las aplicaciones donde es importante seleccionar un servidor en la LAN del cliente. También contiene una entrada para indicar la posición de la base de datos CDS. El directorio *hosts* enumera todas las máquinas de la celda, y cada subdirectorio en ésta tiene una entrada para el demonio RPC del anfitrión (*self*) y el perfil por omisión (*profile*), así como varias partes del sistema CDS y otras máquinas. Las conexiones proporcionan el enlace con el sistema de archivos y la base de datos de seguridad, como ya mencionamos arriba, y el directorio *subsys* contiene todas las aplicaciones usuario más la propia información administrativa de DCE.

La unidad más primitiva en el sistema de directorios es la **entrada de directorios de CDS**, que consta de un nombre y un conjunto de atributos. La entrada para un servicio contiene el nombre del servicio, la interfaz soportada y la posición del servidor.

Es importante observar que CDS sólo contiene información acerca de los recursos, pero no proporciona acceso a los propios recursos. Por ejemplo, una entrada CDS para *printer23* podría indicar una impresora láser a color, de 20 páginas por minuto, 600 puntos por pulgada, localizada en el segundo piso de Toad Hall con la dirección en la red 192.30.14.52. Esta información es utilizada por el sistema RPC para la conexión, pero para utilizar en realidad la impresora, el cliente debe hacer una RPC con ella.

Cada entrada tiene asociada una lista de los que tienen acceso a la entrada y la forma de este acceso (por ejemplo, los que podrían eliminar la entrada del directorio CDS). Esta

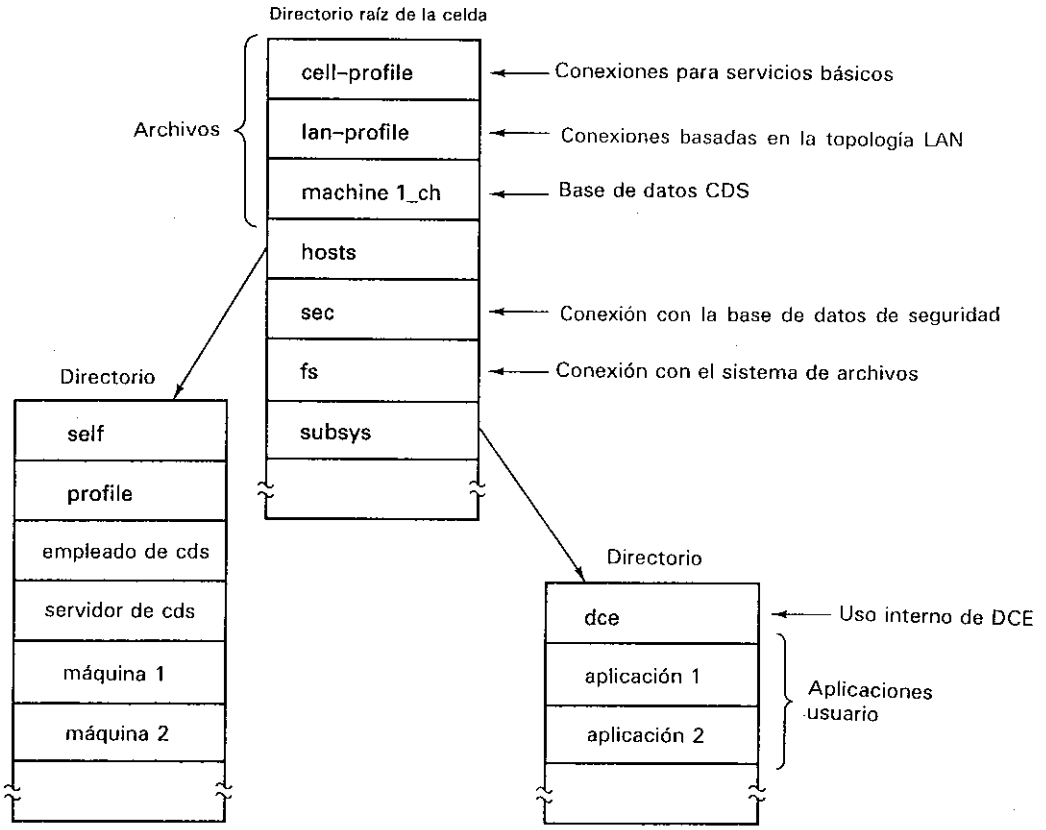


Figura 10-20. El espacio de nombres de una celda sencilla en DCE.

información de protección es controlada por el propio CDS. Al tener el acceso a la entrada GDS no se garantiza que el cliente tenga acceso al propio recurso. El servidor es el que se encarga de controlar el recurso, para decidir quién y cómo utilizarlo.

Un grupo de entradas relacionadas entre sí se agrupan en un **directorio CDS**. Por ejemplo, todas las impresoras están agrupadas en un directorio *printers*, de modo que cada entrada describa una impresora distinta o un grupo de impresoras.

CDS duplica las entradas para proporcionar una alta disponibilidad y mejor tolerancia de fallas. El directorio es la unidad de réplica, donde se duplica o no todo un directorio. Por esta razón, los directorios son un concepto más pesado que en, digamos, UNIX. Los directorios de CDS no se pueden crear y eliminar desde la interfaz usual del programador. Se utilizan programas de administración especiales.

Una colección de directorios forma un **centro de distribución**, que es una base de datos física. Una celda tiene varios centros de distribución. Cuando se duplica un directorio, aparece en dos o más centros de distribución.

Los CDS de una celda están dispersos en muchos servidores, pero el sistema ha sido diseñado de modo que la búsqueda de cualquier nombre sea iniciada por cualquier servidor. Mediante el prefijo es posible ver si el nombre es local o global. Si es global, la solicitud se pasa al GDA para continuar su procesamiento. Si es local, se busca la primera componente en el directorio raíz de la celda. Por esta razón, cada servidor CDS tiene una copia del directorio raíz. Los directorios a los que apunta la raíz son locales del servidor o están en un servidor diferente, pero en todo caso, siempre es posible continuar la búsqueda y localizar el nombre.

Con varias copias de los directorios dentro de una celda, aparece un problema: ¿cómo se realizan las actualizaciones sin causar inconsistencias? DCE toma aquí el camino fácil. Se designa una copia de cada directorio como el amo; el resto son esclavos. Las operaciones de lectura y actualización se realizan sobre el amo, pero sólo pueden realizarse lecturas en los esclavos. Cuando se actualiza el amo, se indica esto a los esclavos.

Se proporcionan dos opciones para esta propagación. Para que los datos sean consistentes todo el tiempo, los cambios se envían a todos los esclavos de manera inmediata. Para los datos menos críticos, los esclavos se pueden actualizar posteriormente. Este esquema, llamado **skulking**, envía muchas actualizaciones en mensajes con mayor tamaño y eficiencia.

CDS se implanta en principio mediante dos componentes principales. La primera, el **servidor CDS**, es un proceso demonio que se ejecuta en una máquina servidor. Acepta solicitudes, las busca en su centro local de distribución, y envía las respuestas de regreso.

La segunda, el **empleado de CDS**, es un proceso demonio que se ejecuta en la máquina cliente. Su función principal es realizar el ocultamiento del cliente. Las solicitudes clientes para buscar datos en el directorio pasan por el empleado, que entonces guarda los resultados en un caché para su uso posterior. Los empleados saben de la existencia de servidores CDS pues éstos últimos transmiten su posición de manera periódica.

Como ejemplo sencillo de la interacción entre el cliente, el empleado y el servidor, consideremos la situación de la figura 10-21(a). Para buscar un nombre, el cliente realiza una RPC con su empleado CDS local. El empleado busca entonces en su caché. Si encuentra la respuesta, responde de manera inmediata. En caso contrario, como se muestra en la figura, realiza una RPC a través de la red con el servidor CDS. En este caso, el servidor encuentra el nombre solicitado en su centro de distribución y lo envía de regreso al empleado, que entonces lo guarda en el caché para su posterior solicitud antes de regresarlo al cliente.

En la figura 10-21(b), existen dos servidores CDS en la celda, pero el empleado sólo conoce uno de ellos (el incorrecto, como se puede ver). Cuando el servidor CDS ve que no tiene la entrada de directorio necesaria, busca en el directorio raíz (recuerde que todos los servidores CDS tienen todo el directorio raíz), para encontrar el enlace suave al servidor CDS correcto. Con esta información, el empleado intenta de nuevo (mensaje 4), Como antes, el empleado guarda los resultados en un caché antes de responder.

10.5.3. El servicio de directorio global

Además de CDS, DCE tiene un segundo servicio de directorios, GDS, que se utiliza para localizar celdas remotas, pero que también se utiliza para guardar otra información

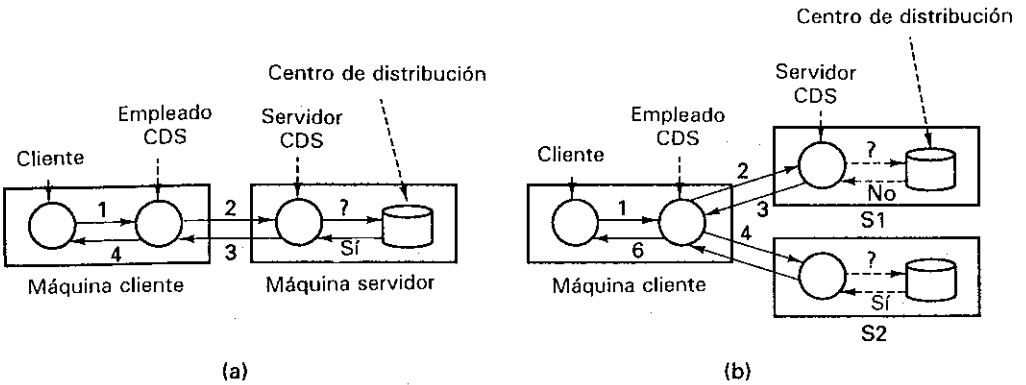


Figura 10-21. Dos ejemplos de un cliente buscando un nombre. (a) El servidor CDS que se ha contactado primero tiene el nombre. (b) El servidor CDS que se ha contactado primero no tiene el nombre.

arbitraria relativa a los directorios. GDS es importante debido a que es la implantación DCE de X.500, y como tal, se conecta mediante una red con otros servicios de directorios X.500 (no DCE). X.500 está definido en el estándar internacional ISO 9594.

X.500 utiliza un modelo de información orientado a objetos. Cada elemento guardado en un directorio X.500 es un objeto. Un objeto puede ser un país, una compañía, una ciudad, una persona, una celda, o un servidor, por ejemplo.

Cada objeto tiene uno o más atributos. Un atributo consta de un tipo y un valor (o a veces varios valores). En forma escrita, el tipo y el valor se separan con un signo de igualdad, como en *C = US* para indicar que el tipo es *country* y el valor es *United States*.

Los objetos se agrupan en clases, de modo que todos los objetos de una clase dada se refieren al mismo "tipo" de objeto. Una clase tiene atributos obligatorios, como un código postal para un objeto postal, y atributos opcionales, como un número de fax para un objeto compañía. El atributo de la clase del objeto siempre es obligatorio.

La estructura de nombre de X.500 es jerárquica. Un ejemplo sencillo aparece en la figura 10-22. En este caso mostramos sólo dos de las entradas debajo de la raíz, un país (US) y una organización (IBM). La decisión de la posición de cada objeto en el árbol no es parte de X.500, sino que es responsabilidad de la autoridad de registro. Por ejemplo, en una compañía de reciente formación, digamos, Invisible Graphics, desea registrarse justo debajo de *C=US* en el árbol a nivel mundial, por lo que debe establecer contacto con ANSI para ver si ese nombre está en uso, y si no, reclamarlo y pagar su registro.

Las rutas de acceso a través del árbol de nombres están dadas por una serie de atributos separados por diagonales, como hemos visto. En nuestro ejemplo actual, Joe, de Joes Deli en San Francisco sería

/C=US/STATE=CA/LOC=SF/O=JOES-DELI/CN=JOE/

donde *LOC* indica un lugar y *CN* es el nombre (común) del objeto. En el argot de X.500, cada componente de la ruta es el **nombre distinguido relativo (RDN)** y la ruta completa

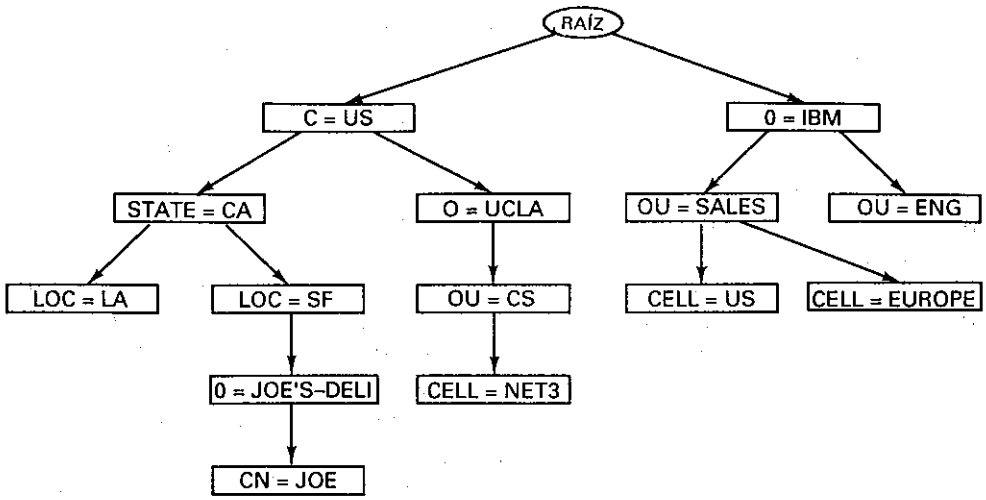


Figura 10-22. Un árbol de información de un directorio X.500.

es el **nombre distinguido (DN)**. Todos los RDN que se originan en un objeto dado deben ser distintos, pero los RDN que se originan en objetos diferentes pueden ser iguales.

Además de los objetos normales, el árbol X.500 contiene **alias**, que nombran a otros objetos. Los alias son similares a los enlaces simbólicos en un sistema de archivos.

La estructura y propiedades de un árbol de información de directorio en X.500 quedan definidos por su **esquema**, el cual consta de tres tablas:

1. Tabla de reglas de estructura:—El punto donde cada objeto pertenece al árbol.
2. Tabla de clases de objetos: —Relaciones hereditarias entre las clases.
3. Tabla de atributos: —Especifica el tamaño y tipo de cada atributo.

La tabla de reglas de estructura es básicamente una descripción del árbol en forma de tabla e indica principalmente cuál objeto es hijo de otro. La tabla de clases de objetos describe la jerarquía hereditaria de los objetos. Por ejemplo, podría existir una clase número telefónico, con subclases VOX y FAX.

Observe que el ejemplo de la figura 10-22 no contiene información alguna relativa a la jerarquía hereditaria de los objetos, pues las organizaciones aparecen ahí bajo los países, las ciudades y la raíz. La estructura que se muestra en esta figura sería reflejada en la tabla de reglas de estructura. La tabla de clases de objetos podría configurarse con la unidad organizativa como una subclase de una organización, y con una celda como subclase de la unidad organizativa, pero esta información no se obtiene de la figura. Además indica la clase de la cual se deriva una clase dada, la tabla de clases de objetos enumera su único identificador de objeto, y sus atributos obligatorios y opcionales.

La tabla de atributos indica los valores que tiene cada atributo, la cantidad de memoria que ocupa, y lo que son sus tipos (por ejemplo, enteros, booleanos o reales). Los atributos se describen en la notación ASN.1 de OSI. DCE proporciona un compilador de ASN.1 a C (MAVROS), que es similar a su compilador IDL para los resguardos de la RPC. Este compilador se utiliza en realidad para construir DCE; por lo general, los usuarios no se encontrarán con él.

Cada atributo se marca como PÚBLICO, ESTÁNDAR o SENSIBLE (PUBLIC, STANDARD o SENSITIVE). Es posible asociar a cada objeto listas de control de acceso para especificar los usuarios que lo leen y los usuarios que modifican sus atributos en cada una de estas tres categorías.

Se soporta la interfaz estándar de X.500, llamada **XOM (administración de objetos X/open)**. Sin embargo, la manera usual de acceso de los programas al sistema GDS es mediante la biblioteca **XDS (servidor de directorios X/Open)**. Cuando se llama a alguno de los procedimientos XDS, éste verifica si la entrada que se maneja es una entrada CDS o una entrada GDS. Si es CDS, sólo realiza de forma directa el trabajo. Si es GDS, hace las llamadas XOM necesarias para realizar el trabajo.

La interfaz XDS es muy pequeña, sólo 13 llamadas (contra 101 llamadas y un manual de 409 páginas para la RPC en DCE). De éstas, cinco configuran y analizan la conexión entre el cliente y el servidor de directorios. Las ocho llamadas que en realidad utilizan los objetos del directorio aparecen en la figura 10-23.

Llamada	Descripción
Add_entry	Agrega un objeto o alias al directorio
Remove_entry	Elimina un objeto o alias del directorio
List	Enumera todos los objetos que están directamente debajo de un objeto dado
Read	Lee los atributos de un objeto dado
Modify_entry	Modifica de manera atómica los atributos de un objeto dado
Compare	Compara el valor de un atributo con otro valor dado
Modify_rdn	Cambia el nombre de un objeto
Search	Busca un objeto en una parte del árbol

Figura 10-23. Las llamadas XDS para el manejo de los objetos directorio.

Las primeras dos llamadas añaden y eliminan objetos del árbol de directorios, respectivamente. Cada llamada especifica una ruta completa de modo que nunca exista una ambigüedad acerca del objeto agregado o eliminado. La llamada *list* enumera todos los objetos

que están de forma directa debajo del objeto especificado. Las entradas *read* y *modify* leen y escriben los atributos del objeto especificado, copiándolos del directorio al proceso que hizo la llamada, o viceversa. *Compare* examina un atributo particular de un objeto específico, lo compara con un valor dado, e indica si concuerdan o no. *Modify_rdn* modifica un nombre distinguido relativo; por ejemplo, cambiando de *a/b/c* a *a/x/c*. Por último, *search* parte de un objeto dado y busca en el árbol de objetos por debajo de éste (o en una parte de éste) los objetos que cumplan cierto criterio.

Todas estas llamadas operan determinando primero si se necesita CDS o GDS. Los nombres de X.500 son controlados por GDS; DNS o los nombres mixtos son controlados por CDS, como se muestra en la figura 10-24. Primero daremos el esquema de la búsqueda de un nombre en formato X.500. La biblioteca XDS ve que necesita buscar un nombre X.500, por lo que llama al **DUA (agente del usuario de directorios)**, una biblioteca ligada al código del cliente. Esto maneja el ocultamiento del GDS, de manera análoga al empleado CDS, que controla el ocultamiento del CDS. Los usuarios tienen más control sobre el ocultamiento del GDS que sobre el ocultamiento del CDS y pueden, por ejemplo, especificar los elementos que debe ocultarse en el caché. Incluso pueden pasar por encima del DUA si es en lo absoluto esencial obtener los datos más recientes.

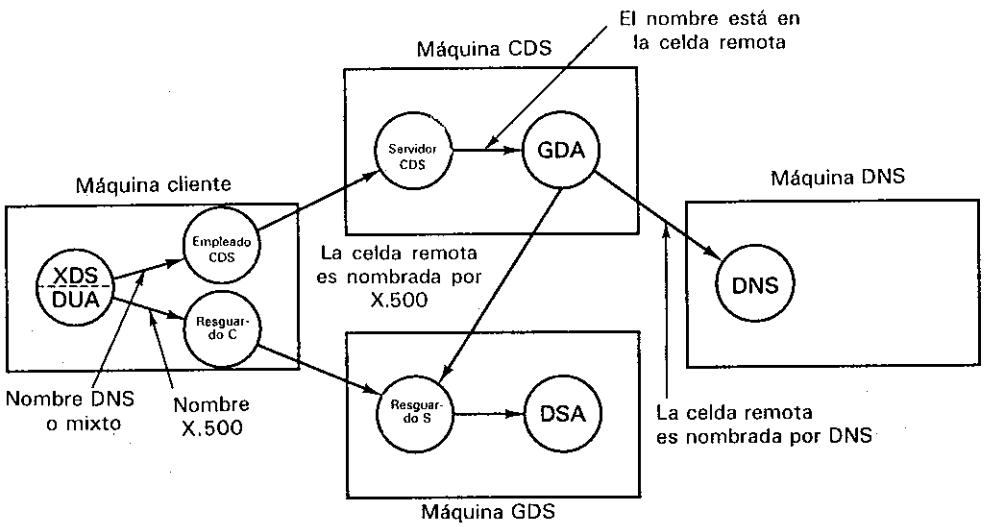


Figura 10-24. La forma en que los servidores implicados en la búsqueda de nombres se llaman entre sí.

De manera análoga al servidor CDS, existe un **DSA (agente servidor de directorios)** que controla las solicitudes recibidas de los DUA, de su celda y de las celdas remotas. Si una solicitud no se puede procesar debido a que la información no está disponible, el DSA retransmite la solicitud a la celda adecuada o indica al DUS que lo haga.

Además de los procesos DUA y DSA, también existen resguardos separados que controlan la comunicación de área amplia utilizando los protocolos ASCE de OSI y ROSE sobre los protocolos de transporte OSI.

Ahora veamos cómo se realiza la búsqueda de un DNS o un nombre mixto. XDS realiza una RPC con el empleado CDS para ver si está en un caché. Si no, se le solicita al servidor CDS. Si el servidor CDS ve que el nombre es local, lo busca. Sin embargo, si pertenece a una celda remota, pide al GDA que trabaje con él. El GDA examina el nombre de la celda remota para ver si está especificado como un nombre DNS o un nombre X.500. En el primer caso, pide al servidor DNS que encuentre un servidor CDS en la celda; en el segundo, utiliza el DSA. En resumen, la búsqueda es un asunto complejo.

10.6. SERVICIO DE SEGURIDAD

En la mayoría de los sistemas distribuidos, la seguridad es una preocupación fundamental. El administrador del sistema debe tener una idea clara de quién puede utilizar cuál recurso (por ejemplo, ningún estudiante de los primeros años de una licenciatura debería utilizar la sofisticada impresora láser de color), y muchos usuarios querrán proteger a sus archivos y buzones de los curiosos. Estos aspectos surgen también en los sistemas tradicionales de tiempo compartido, pero ahí se resuelven simplemente haciendo que el núcleo controle todos los recursos. En un sistema distribuido formado potencialmente por máquinas no confiables que se comunican a través de una red insegura, esta solución no funciona. Sin embargo, DCE proporciona una seguridad excelente. En esta sección examinaremos la forma en que esto se lleva a cabo.

Comenzaremos nuestro estudio presentando unos cuantos términos importantes. En DCE, un **principal** es un usuario o proceso que necesita comunicarse con seguridad. Los seres humanos, los servidores DCE (como CDS) y los servidores de aplicaciones (como el software de un cajero automático en un sistema bancario) pueden ser principales. Por conveniencia, los principales con los mismos derechos de acceso se pueden agrupar. Cada principal tiene un **UUID (identificador único de usuario)**, que es un número binario asociado a este único principal.

La **autenticación** es el proceso para determinar si un principal es realmente quien afirma ser. En un sistema de tiempo compartido, un usuario entra mediante su nombre y contraseña. Una sencilla verificación del archivo local de contraseña indica si el usuario miente o no. Después de que un usuario entra con éxito, el núcleo mantiene un registro de la identidad del usuario y permite o niega el acceso y otros recursos con base en ella.

En DCE, se necesita un procedimiento de autenticación diferente. Cuando un usuario entra, el programa de entrada verifica la identidad del usuario mediante un servidor de autenticación. Más adelante describiremos el protocolo, pero por el momento basta decir que *no* implica el envío de la contraseña a través de la red. El procedimiento de autenticación de DCE utiliza el sistema Kerberos desarrollado en MIT (Kohl, 1991; y Steiner *et al.*, 1988). Kerberos, a su vez, se basa en las ideas de Needham y Schroeder (1978). Para otros puntos de vista de la autenticación, véase (Lampson *et al.*, 1992; Wobber *et al.*, 1994; y Woo y Lam, 1992).

Una vez que un usuario se ha autenticado, surge la cuestión de los recursos a los cuales puede tener acceso, y la forma de éste. Este aspecto se llama la **autorización**. En DCE, la autorización se maneja mediante la asociación de una **ACL (lista de control de acceso)** a cada recurso. La ACL indica los usuarios, grupos y organizaciones que pueden tener acceso al recurso y lo que pueden hacer con él. Los recursos pueden ser tan gruesos como archivos o tan finos como entradas de bases de datos.

La protección en DCE está íntimamente ligada a la estructura de celdas. Cada celda tiene un **servicio de seguridad** en el que deben confiar los principales. El servicio de seguridad, del que es parte el servidor de autenticación, conserva las claves, las contraseñas y demás información relativa a la seguridad en una base de datos segura llamada **registro**. Puesto que las diferentes celdas pueden ser poseídas por diferentes compañías, la comunicación segura de una celda a otra requiere un protocolo complejo, y puede realizarse solamente si las dos celdas han configurado de antemano una clave secreta compartida. Para simplificar nuestra exposición, restringiremos nuestro análisis posterior al caso de una celda.

10.6.1. Modelo de seguridad

En esta sección daremos una breve revisión de los principios básicos de la **criptografía**, la ciencia del envío de mensajes secretos, y los requisitos e hipótesis de DCE en el área. Supongamos que dos partes, digamos un cliente y un servidor, desean comunicarse de manera segura a través de una red insegura. Lo que esto significa es que incluso un intruso (por ejemplo, una persona que interviene las líneas telefónicas) logra robar mensajes, no podrá interpretarlos. Aún más, si el intruso intenta personificar al cliente o si el intruso registra mensajes válidos del cliente y después los reproduce para el servidor, éste podrá notar esto y rechazar estos mensajes.

El modelo criptográfico tradicional aparece en la figura 10-25. El cliente tiene un mensaje no cifrado, P, llamado **texto plano**, que se transforma mediante un algoritmo de cifrado parametrizado por una clave, *K*. El cifrado puede ser realizado por el cliente, el sistema operativo, o un hardware especial. El mensaje resultante, *C*, llamado **texto cifrado**, es ininteligible para todos los que no posean la clave. Cuando el texto cifrado llega al servidor, se descifra mediante *K*, lo que produce el texto plano original. La notación que se utiliza por lo general para indicar el cifrado es

Texto cifrado = {Texto plano} Clave

es decir, la cadena dentro de las llaves es el texto plano, y la clave utilizada se escribe a continuación de ésta.

También existen sistemas criptográficos donde el cliente y el servidor utilizan diferentes claves (por ejemplo, criptografía de claves públicas), pero como no se utilizan en DCE, no los analizaremos.

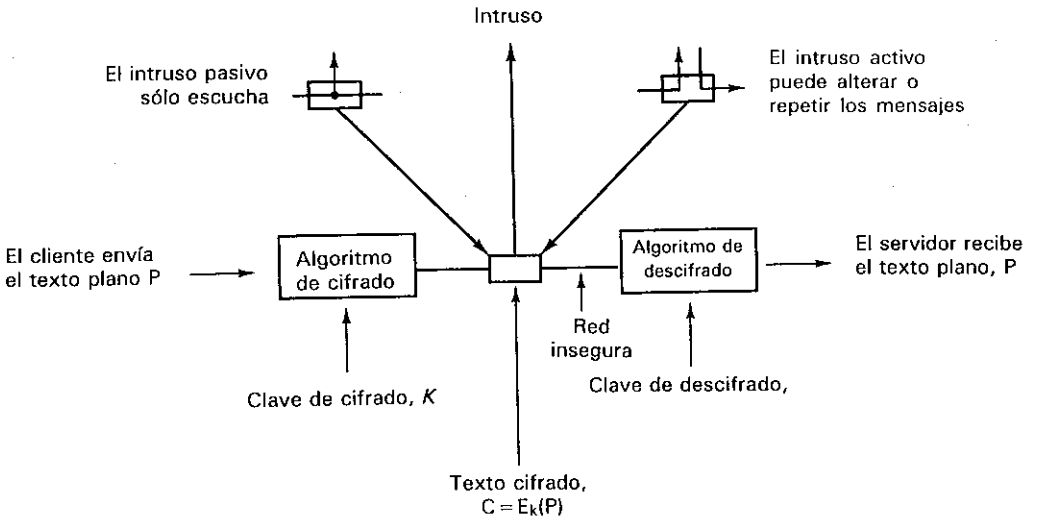


Figura 10-25. Un cliente envía un mensaje cifrado a un servidor.

Tal vez sea importante señalar de manera explícita algunas de las hipótesis subyacentes en este modelo, para evitar cualquier confusión. Suponemos que la red es por completo insegura y que cierto intruso puede capturar cualquier mensaje enviado por ella, tal vez eliminándolo. El intruso también puede introducir nuevos mensajes y repetir los mensajes anteriores a voluntad.

Aunque suponemos que la mayoría de los servidores principales son moderadamente seguros, suponemos de manera explícita que el servidor de seguridad (incluyendo sus discos) se puede colocar en un cuarto cerrado, protegido por un fiero perro de tres cabezas (Kerberos, el cancerbero de la mitología griega) y que ningún intruso puede enfrentarse con él. En consecuencia, el servidor de seguridad puede conocer la contraseña de cada usuario, aunque las contraseñas no se pueden enviar a través de la red. También se supone que los usuarios no olvidan sus contraseñas o que accidentalmente las dejan en pedazos de papel en el cuarto de las terminales. Por último, suponemos que los relojes del sistema están más o menos sincronizados, utilizando por ejemplo DTS.

Como consecuencia de este ambiente hostil, desde un principio se establecieron varios requisitos de diseño. Los más importantes son los siguientes. En primer lugar, en ningún momento pueden aparecer las contraseñas en texto plano (es decir, no cifrado) en la red o guardarse en los servidores normales. Este requisito evita hacer la autenticación enviando las contraseñas de los usuarios a un servidor de autenticación para su aprobación.

En segundo lugar, las contraseñas de los usuarios ni siquiera se pueden guardar en máquinas cliente durante más de unos cuantos microsegundos, por miedo a que puedan exponerse en un vaciado del núcleo en caso de que falle la máquina.

En tercer lugar, la autenticación debe funcionar en ambos sentidos. Es decir, no sólo el servidor debe estar convencido de la identidad del cliente, sino que el cliente debe estar convencido de la identidad del servidor. Este requisito es necesario para evitar que un intruso capture los mensajes del cliente y pretenda que es, digamos, el servidor de archivos.

Por último, el sistema debe contar con protecciones integradas. Si una clave está comprometida de algún modo (queda al descubierto), el daño causado debe ser limitado. Este requisito se puede cubrir, por ejemplo, creando claves temporales con fines específicos y con tiempos cortos de vida, utilizando estos archivos para la mayor parte del trabajo. Si una de estas claves queda a descubierto, se restringe el potencial de daño.

10.6.2. Componentes de seguridad

El sistema de seguridad de DCE consta de varios servidores y programas, los más importantes de los cuales aparecen en la figura 10-26. El **servidor de registro** controla la base de datos de seguridad, el registro, que contiene los nombres de todos los principales, grupos y organizaciones. Para cada principal, proporciona la información contable, grupos y organizaciones a los que pertenece el principal, si el principal es un cliente o un servidor, e información adicional. El registro también contiene la información por cada celda, incluyendo la longitud, el formato, y el tiempo de vida para las contraseñas y la información relacionada con ellas. El registro se puede pensar como el sucesor del archivo de contraseñas en UNIX (*/etc/passwd*). Puede ser editado por el administrador del sistema mediante el editor del registro. Se pueden agregar y eliminar principales, modificar las claves, etcétera.

EL **servidor de autenticación** se utiliza cuando un usuario entra al sistema o cuando se arranca un servidor. Verifica la identidad afirmada del principal y emite una especie de boleto (descrito más adelante) que permite al principal realizar la autenticación posterior sin tener que utilizar nuevamente la contraseña. El servidor de autenticación también es conocido como el **servidor emisor de boletos** cuando está emitiendo boletos en vez de autenticar usuarios, pero estas dos funciones residen en el mismo servidor.

El **servidor de privilegios** emite documentos llamados **PAC (certificados de atributos de privilegios)** para autenticar a los usuarios. Los PAC son mensajes cifrados que contienen la identidad del principal, la membresía de grupos, y la membresía organizativa de tal forma que los servidores se convenzan de manera inmediata sin necesidad de presentar información adicional. Los tres servidores se ejecutan en la máquina servidor de seguridad en el cuarto cerrado con el perro mutante en el interior.

La **facilidad de entrada** es un programa que pregunta a los usuarios sus nombres y contraseñas durante la secuencia de entrada. Utiliza los servidores de autenticación y de privilegios para hacer su trabajo, que es permitir la entrada del usuario al sistema y reunir todos los boletos y PAC de éste.

Una vez que un usuario ha entrado al sistema, puede iniciar un proceso cliente que pueda comunicarse de manera segura con un proceso servidor mediante RPC autenticada. Cuando llega una solicitud de RPC autenticada, el servidor utiliza el PAC para determinar la identidad del usuario, y después verifica su ACL para ver si se permite el acceso solicitado.

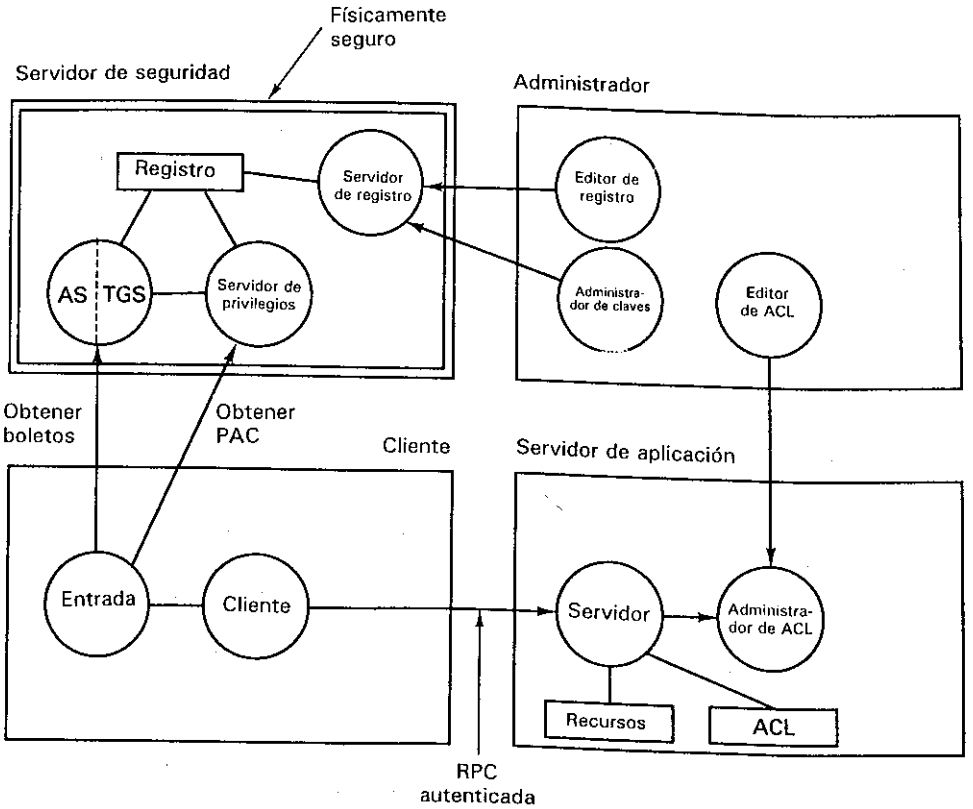


Figura 10-26. Componentes principales del sistema de seguridad de DCE para una celda.

Cada servidor tiene su propio controlador de ACL para proteger sus objetos. Los usuarios pueden sumarse o eliminarse de una ACL, recibir o eliminar permisos, etc., utilizando un programa editor de ACL.

10.6.3. Boletos y autenticadores

En esta sección describiremos el funcionamiento de los servidores de autenticación y de privilegios y la forma en que permiten la entrada de los usuarios a DCE de manera segura a través de una red insegura. La descripción apenas abarca lo esencial, e ignora la mayor parte de las variantes y las opciones disponibles.

Cada usuario tiene una clave secreta que sólo conoce él y el registro. Se calcula al pasar la contraseña del usuario por una función de un solo sentido (es decir, no invertible). Los servidores también tienen claves secretas. Para aumentar su seguridad, estas claves sólo se utilizan brevemente, cuando un usuario entra al sistema o cuando se arranca un servidor. Después de esto, la autenticación se realiza mediante los boletos y los PAC.

Un **boleto** es una estructura de datos cifrada emitida por el servidor de autenticación o emisor de boletos para demostrar a un servidor específico que el portador es un cliente con una identidad específica. Los boletos tienen muchas opciones, pero principalmente analizaremos boletos que se ven como sigue:

boleto = S,{clave de sesión, cliente, tiempo de expiración, identificador de mensaje} K_S

donde S es el servidor del que se pretende tener el boleto. La información dentro de las llaves se cifra mediante la clave particular del servidor, K_S. Los campos cifrados incluyen una clave temporal de la sesión, la identidad del cliente, el tiempo en que el boleto deja de ser válido, y un identificador de mensaje **para una ocasión**, que se utiliza para relacionar las solicitudes y respuestas. Cuando el servidor descifra el boleto con su clave particular, obtiene la clave de sesión que utiliza al hablar con el cliente. En nuestras descripciones posteriores, omitiremos todos los campos cifrados del boleto, excepto la clave de sesión y el nombre del cliente.

En ciertas situaciones, los boletos y PAC se utilizan juntos. Los boletos establecen la identidad del emisor (como una cadena ASCII), mientras que los PAC dan los valores numéricos del identificador del usuario y los identificadores de grupo asociados con un principal particular. Los boletos son generados mediante el servidor de autenticación o emisor de boletos (que son un servidor), mientras que los PAC son generados por el servidor de privilegios.

En muchas situaciones, no es esencial que los mensajes sean secretos, sino solamente que los intrusos no puedan forjarlos o modificarlos. Con este fin, se pueden añadir autenticadores a los mensajes en texto plano para evitar que los intrusos activos los modifiquen. Un **autenticador** es una estructura de datos cifrada que contiene al menos la siguiente información:

autenticador = {emisor, suma de verificación MD5, marca de tiempo} K

donde al algoritmo de suma de verificación, **MD5 (Message Digest 5)**, tiene la propiedad de que dada una suma de verificación MD5 de 128 bits, no se pueda realizar un cálculo para modificar el mensaje de modo que la suma concuerde (dicha suma está protegida mediante un cifrado). La marca de tiempo es necesario para que el receptor detecte la respuesta de un autenticador anterior.

10.6.4. RPC autenticada

La secuencia desde la entrada hasta el punto en que la primera RPC autenticada es posible requiere generalmente de cinco pasos. Cada paso consta de un mensaje del cliente a algún servidor, seguido de una respuesta del servidor al cliente. Los pasos se resumen en la figura 10-27 y se describen a continuación. Para simplificar nuestra exposición, hemos omitido la mayor parte de los campos, incluyendo las celdas, los identificadores de mensaje,

los tiempos de vida y los identificadores. Enfatizamos los aspectos fundamentales de la seguridad: las claves, los boletos, los autenticadores y los PAC.

Principales

A: Servidor de autenticación (controla la autenticación)

C: Cliente (usuario)

P: Servidor de privilegios (emite los PAC)

S: Servidor de aplicaciones (ejecuta el trabajo real)

T: Servidor emisor de boletos (emite los boletos)

Paso 1: El cliente obtiene un boleto del servidor emisor de boletos mediante una contraseña

$C \rightarrow A$:C (sólo el nombre del cliente, en texto plano)

$C \leftarrow A$:$\{K_1,\{K_1,C\}K_A\}\}K_C$

Paso 2: El cliente utiliza el boleto anterior para obtener un boleto para el servidor de privilegios

$C \rightarrow T$:$\{K_1,C\}K_A,\{C$, suma de verificación MD-5, marca de tiempo$\}K_1$

$C \leftarrow T$:$\{K_2,\{C,K_2\}K_P\}K_1$

Paso 3: El cliente solicita el PAC inicial al servidor de privilegios

$C \rightarrow P$:$\{C,K_2\}K_P,\{C$, suma de verificación MD-5, marca de tiempo$\}K_2$

$C \leftarrow P$:$\{K_3,\{PAC,K_3\}K_A\}K_2$

Paso 4: El cliente al servidor emisor de boletos un PAC utilizable por S

$C \rightarrow T$:$\{K3,PAC\}K_3\}K_4\}K_3,\{C$, suma de verificación MD-5, marca de tiempo$\}K_3$

$C \leftarrow T$:$\{K_4,\{PAC,K_4\}K_S\}K_3$

Paso 5: El cliente establece una clave con el servidor de aplicaciones

$C \rightarrow S$:$\{PAC,K4\}KS,\{C$, suma de verificación MD-5, marca de tiempo$\}K_4$

$C \leftarrow S$:$\{K_5$,marca de tiempo$\}K_4$

Figura 10-27. De la entrada a la RPC autenticada en cinco sencillos pasos.

Cuando un usuario está frente a una terminal DCE, el programa de entrada le solicita su nombre de entrada. El programa envía entonces este nombre de entrada al servidor de autenticación en texto plano. El servidor de autenticación busca en el registro y encuentra la clave secreta del usuario. Entonces, genera un número aleatorio para su uso como clave de la sesión y envía de regreso un mensaje cifrado por la clave secreta del usuario, K_C, que contiene la clave de la primera sesión, K_1, y un boleto que utilizará posteriormente con el servidor emisor de boletos. Estos mensajes se muestran en la figura 10-27, en el paso 1. Observe que esta figura muestra 10 mensajes, y para cada uno de ellos, la fuente y el destino se dan antes del mensaje, de modo que la flecha apunta de la fuente al destino.

Cuando el mensaje cifrado llega al programa de entrada, se pide la contraseña al usuario. La contraseña pasa de inmediato por la función de un solo sentido que genera las claves secretas a partir de las contraseñas. Tan pronto como la clave secreta K_C se genera a partir de la contraseña, ésta se elimina de la memoria. Este procedimiento minimiza la probabilidad de que alguien descubra la contraseña en caso de una falla del cliente. Después, se descifra el mensaje del servidor de autenticación para obtener la clave de sesión y el

boleto. Cuando esto se ha realizado, la clave secreta del cliente también se puede eliminar de la memoria.

Observe que si un intruso intercepta el mensaje de respuesta, no podrá descifrarlo y por lo tanto no podrá obtener la clave de sesión ni el boleto dentro de dicho mensaje. Si invierte mucho tiempo, el intruso podría descomponer el mensaje, pero incluso entonces, el daño sería limitado, debido a que las claves de sesión y los boletos sólo son válidos para periodos relativamente cortos.

En el paso 2 de la figura 10-27, el cliente envía el boleto al servidor emisor de boletos (de hecho, el servidor de autenticación con un nombre distinto), y solicita un boleto para hablar con el servidor de privilegios. Excepto por la autenticación inicial en el paso 1, la comunicación con un servidor de manera autenticada siempre requiere un boleto cifrado con la clave particular del servidor. Estos boletos se pueden obtener del servidor emisor de boletos, como en el paso 2.

Cuando el servidor emisor de boletos obtiene el mensaje, utiliza su propia clave particular, K_A, para descifrar el mensaje. Cuando encuentra la clave de sesión K_1, busca en el registro y verifica que éste ha asignado de manera reciente la clave al cliente C. Puesto que K_C sólo es conocida por C, el servidor emisor de boletos sabe que sólo C sería capaz de descifrar la respuesta enviada en el paso 1, y esta solicitud debe venir de C.

La solicitud también contiene un autenticador, que básicamente es una suma de verificación con marca de tiempo y fuertemente cifrada del resto del mensaje, incluyendo el nombre de la celda, la solicitud y otros campos que no aparecen en la figura. Este esquema hace imposible que un intruso modifique el mensaje sin que esto se detecte, y no requiere que el cliente cifre todo el mensaje, lo que sería caro para un mensaje de gran longitud. (En este caso, el mensaje no es tan largo, pero los autenticadores se utilizan todo el tiempo, para una mayor sencillez del modelo.) La marca de tiempo en el autenticador protege en contra de un intruso que capture el mensaje y lo transmita posteriormente debido a que el servidor de autenticación procesará una solicitud sólo si va acompañada por un autenticador fresco.

En este ejemplo, generamos y utilizamos una nueva clave de sesión en cada paso. Toda esta paranoia no es necesaria, pero el protocolo la permite y, a su vez, esto permite que cada clave tenga un periodo muy breve de vida si los relojes están bien sincronizados.

Con el boleto para el servidor de privilegios, el cliente solicita ahora un PAC. A diferencia de un boleto, que sólo contiene el nombre de entrada del usuario (en ASCII), un PAC contiene la identidad del usuario (en binario, como UUID), junto con una lista de todos los grupos a los que pertenece. Estos son importantes debido a que, con frecuencia, los recursos (por ejemplo, cierta impresora) son disponibles a todos los miembros de ciertos grupos (por ejemplo, el departamento de mercadotecnia). Un PAC es la prueba de que el usuario nombrado en él pertenece a los grupos enumerados en él.

El PAC obtenido en el paso 3 está cifrado con la clave secreta del servidor emisor de boletos/servidor de autenticación. La importancia de esta elección es que durante la sesión, el cliente podría necesitar PAC de diferentes servidores de aplicación. Para obtener un PAC y utilizarlo con un servidor de aplicaciones, el cliente ejecuta el paso 4, en el que el PAC se envía

al servidor emisor de boletos, quien los descifra en primer lugar. Puesto que sí puede descifrarlo, el servidor emisor de boletos se convence de inmediato de que el PAC es legítimo, y entonces lo vuelve a cifrar para el servidor elegido por el cliente, en este caso, *S*.

Observe que el servidor emisor de boletos no tiene que comprender el formato de un PAC. Todo lo que hace en el paso 4 es descifrar algo cifrado con su propia clave y después lo vuelve a cifrar con otra clave que obtiene del registro. Mientras hace esto, el servidor emisor de boletos introduce una nueva clave de sesión, pero esta acción es opcional. Si el cliente necesita posteriormente PAC para otros servidores, repite el paso 4 con el PAC original las veces que sea necesario, especificando cada vez el servidor deseado.

En el paso 5, el cliente envía el nuevo PAC al servidor de aplicaciones, quien lo descifra, exponiendo la clave K_4 utiliza para cifrar al autenticador, al igual que al identificador de cliente y a los grupos. El servidor responde con la última clave, conocida solamente por el cliente y el servidor. Mediante esta clave, el cliente y el servidor se pueden comunicar ahora en forma segura.

Este protocolo es complejo, pero su complejidad se debe al hecho de que ha sido diseñado para resistir una gran variedad de posibles ataques (Bellovin y Merritt, 1991). También tiene características y opciones que no hemos analizado. Por ejemplo, se puede utilizar para establecer una comunicación segura con un servidor en una celda distinta, tal vez pasando por celdas potencialmente no confiables en cada RPC, y puede verificar el servidor al cliente para evitar **spoofing** (un intruso enmascarado como un servidor confiable).

Una vez establecida la RPC autenticada, el cliente y el servidor son los encargados de determinar la cantidad de protección deseada. En algunos casos basta la autenticación del cliente o la autenticación mutua. En otros, cada paquete es autenticado contra la alteración. Por último, cuando se requiere una seguridad a nivel industrial, se puede cifrar todo el tráfico en ambas direcciones.

10.6.5. ACL

Cada recurso en DCE puede protegerse mediante una ACL (lista de control de acceso), modelada con base en la lista del estándar 1003.6 de POSIX. La ACL indica quién puede tener acceso al recurso y la forma de este acceso. Las ACL son controladas por los **controladores de ACL**, que son procedimientos de biblioteca incorporados a cada servidor. Cuando llega una solicitud al servidor que controla la ACL, éste descifra el PAC del cliente para ver su identificador y los grupos a los que pertenece y, con base en esto, la ACL y la operación deseada, se llama al controlador de ACL para tomar una decisión acerca de si otorgar o negar el acceso. Se soportan hasta 32 operaciones por recurso.

Sin embargo, la mayor parte de los servidores utilizan un controlador de ACL estándar, aunque menos inclinado a las matemáticas. Éste divide los recursos en dos categorías: recursos simples, como los archivos y las entradas de las bases de datos; y los recipientes, como los directorios y las tablas de la base de datos, que contienen recursos simples. Distingue entre los usuarios que viven en la celda y los usuarios externos, y para cada categoría los subdivide como propietario, grupo y otros. Así, es posible especificar que el propietario

puede hacer lo que desee, los miembros locales del grupo del propietario pueden hacer casi todo, los usuarios desconocidos de otras celdas no pueden hacer nada, etcétera.

Se soportan siete derechos estándar: leer, escribir, ejecutar, modificar ACL, insertar recipiente, eliminar recipiente y probar. Los primeros tres son como en UNIX. El derecho modificar ACL permite modificar la propia ACL. Los dos derechos del recipiente son útiles para controlar quiénes pueden, por ejemplo, agregar o eliminar archivos de un directorio. Por último, la prueba permite comparar un valor dado al recurso sin descubrir al propio recurso. Por ejemplo, una entrada del archivo de contraseñas podría permitir a los usuarios preguntar por la concordancia de una contraseña dada, pero sin exponer la propia entrada del archivo de contraseñas. Un ejemplo de ACL aparece en la figura 10-28.

sp_acl_data	Tipo de ACL
/.../ C=NL / O=VU / OU=CS /	Celda por omisión
user : ast : rwxcidt	Usuarios en la celda por omisión
user : bal : rwxidt	
group : staff : rwxt	Grupos en la celda por omisión
group : students : rt	
other : t	Todo lo demás en la celda por omisión
foreign_user : jennifer @ /.../ cs.nyu.edu:rt	Usuarios en otras celdas
foreign_group : staff @ /.../ cs.yale.edu:rw	

Figura 10-28. Un ejemplo de ACL.

En este ejemplo, como en las demás ACL, se especifica el tipo de ACL. Este tipo divide eficazmente las ACL en clases con base en el tipo. La celda por omisión se especifica a continuación. Después de esto vienen los permisos de dos usuarios específicos en la celda por omisión, dos grupos específicos en la celda por omisión y una especificación de lo que podrían hacer en esa celda los demás usuarios. Por último, se pueden especificar los derechos de los usuarios y grupos en otras celdas. Si un usuario que no se ajusta a cualquiera de las categorías enumeradas intentar tener un acceso, éste se le negará.

Para agregar nuevos usuarios, eliminar usuarios, o agregar, eliminar o modificar permisos, se dispone de un programa editor de ACL. Para utilizar este programa en una lista de control de acceso, el usuario debe contar con el permiso para modificar tal lista, lo que se indica en el código *c* en el ejemplo de la figura 10-28.

10.7. SISTEMA DISTRIBUIDO DE ARCHIVOS

La última componente de DCE que estudiaremos es el **sistema distribuido de archivos** (**DFS**; Kazar *et al.*, 1990). Es un sistema distribuido de archivos a nivel mundial que permite que los procesos dentro de un sistema DCE tengan acceso a todos los archivos que están autorizados a utilizar, aunque los procesos y los archivos estén en celdas distantes.

DFS tiene dos partes principales: la parte local y la parte de área amplia. La parte local es un sistema de archivos con un nodo llamado **Episode**, que es análogo a un sistema de archivos estándar de UNIX en una computadora independiente. En una configuracion DCE, cada máquina ejecutaría Episode en vez de (o además de) el sistema de archivos normal de UNIX.

La parte de área amplia de DFS es el pegamento que une todos estos sistemas de archivos individuales para formar un sistema de archivos de área amplia que abarca muchas celdas. Se deriva del sistema de archivos AFS de CMU, pero ha evolucionado en forma considerable desde entonces. Para mayor información relativa a AFS, véase (Howard *et al.*, 1988; Morris *et al.*, 1986, y Satyanarayanan *et al.*, 1985). Para el propio Episode, véase (Chutani *et al.*, 1992).

DFS es una aplicación DCE, y como tal puede utilizar todas las demás facilidades de DCE. En particular, utiliza hilos para permitir el servicio simultáneo a varios solicitudes de acceso a archivos, la RPC para la comunicación entre los clientes y los servidores, DTS para sincronizar los relojes del servidor, el sistema de directorios que permitir la localización de los servidores de archivos, y los servidores de autenticación y de privilegios para proteger los archivos.

Desde el punto de vista de DFS, cada nodo DCE es un cliente de archivo, un servidor de archivo, o ambos. Un cliente de archivo es una máquina que utiliza los sistemas de archivos de otras máquinas. Un cliente de archivos tiene un caché para guardar partes de los archivos de uso reciente, para mejorar el desempeño. Un servidor de archivos es una máquina con disco que ofrece el servicio de archivos a los procesos en esa máquina y, probablemente, a los procesos en otras máquinas.

DFS tiene varias características dignas de mención. Para comenzar, tiene un sistema uniforme de nombres, integrado con CDS, de modo que los nombres de los archivos son independientes de la posición. Los administradores pueden mover los archivos de un servidor de archivos a otro dentro de la misma celda, sin requerir cambios a los programas usuario. Los archivos también se pueden duplicar para difundir la carga de manera más uniforme y mantener la disponibilidad en caso de que falle el servidor de archivos. También existe una facilidad para distribuir de manera automática nuevas versiones de programas binarios y otros archivos exclusivos para lectura de uso intenso a los servidores (incluyendo las estaciones de trabajo del usuario).

Episode es una reescritura del sistema de archivos de UNIX y puede reemplazarlo en cualquier máquina DCE con un disco. Soporta la semántica de archivos adecuada para un sistema de UNIX, en el sentido de que cuando se realiza una escritura a un archivo e inmediatamente después se realiza una lectura, a diferencia de NFS, la lectura verá el valor recién escrito. Se adecua al estándar de llamadas al sistema POSIX 1003.1, y también soporta el control de acceso adecuado a POSIX mediante ACL, lo que da una protección flexible en sistemas de gran tamaño. También ha sido diseñado para soportar una rápida recuperación

después de una falla, eliminando la necesidad de ejecutar el programa *fsck* de UNIX para reparar el sistema de archivos.

Puesto que muchos sitios no desearían pelear con su sistema de archivos existente sólo para ejecutar DCE, DFS ha sido diseñado para proporcionar una integración sobre máquinas que ejecutan 4.3 BSD, System V, NFS, Episode y otros sistemas de archivos. Sin embargo, algunas características de DFS, como la protección mediante ACL, no estará disponible en aquellas partes del sistema de archivo soportadas por servidores distintos de Episode.

10.7.1. Interfaz DFS

La interfaz básica con DFS es (intencionalmente) muy similar a UNIX. Los archivos se pueden abrir, leer y escribir de la manera usual, y la mayor parte del software existente puede simplemente recompilarse con las bibliotecas DFS y funcionará de inmediato. También es posible el montaje de sistemas de archivos remotos.

El directorio / sigue siendo la raíz local, y los directorios como */bin*, */lib*, y */usr* siguen haciendo referencia a los directorios locales binario, de biblioteca y del usuario, como en el caso de ausencia de DFS. Una nueva entrada del directorio raíz es /..., que es la raíz global. Cada archivo de un sistema DFS (que potencialmente puede ser mundial) tiene una ruta de acceso desde la raíz global, que consta de su nombre de celda concatenado con su nombre dentro de esa celda. En la figura 10-29(a) vemos la forma de tener acceso a un archivo, *january*, mediante un nombre de celda en Internet. En la figura 10-29(b) vemos el nombre del mismo archivo con un nombre de celda X.500. Estos nombres son válidos en todas partes del sistema, sin importar la celda en que se encuentre el proceso que utiliza al archivo.

(a) Nombre global de archivo (formato Internet)

/.../cs.ucla.edu/fs/usr/ann/exams/january

(b) Nombre global de archivo (formato X.500)

/.../C = US/O = UCLA/OU = CS/fs/usr/ann/exams/january

(c) Nombre global de archivo (relativo a la celda)

/.:/fs/usr/ann/exams/janaury

(d) Nombre global de archivo (Relativo al sistema de archivos)

/:/usr/ann/exams/january

Figura 10-29. Cuatro formas de referencia al mismo archivo.

El uso de nombres globales en todas partes es un exceso, por lo que se dispone de ciertas abreviaturas. Un nombre que comienza con */.:/fs* indica un nombre en la celda actual que comienza a partir de la función *fs* (el lugar donde el sistema local de archivos se monta en el árbol DFS global), como se muestra en la figura 10-29(c). Este uso se puede abreviar aún más, como se muestra en la figura 10-29(d).

A diferencia de UNIX, la protección en DFS utiliza ACL en vez de los tres grupos de bits RWX, al menos para los archivos controlados por Episode. Cada archivo tiene una ACL que indica quién puede tener acceso al archivo y de qué forma. Además, cada directorio

tiene tres ACL, las que proporcionan el permiso de acceso para el propio directorio, para los archivos en el directorio y los directorios en el directorio, respectivamente.

Las ACL en DFS son controladas por el propio DFS, pues los directorios son objetos DFS. Una ACL para un archivo o directorio consta de una lista de entradas. Cada entrada describe al propietario, al grupo del propietario, a los otros usuarios locales, los usuarios o grupos extraños (es decir, fuera de la celda), o alguna otra categoría, como los usuarios no autenticados. Para cada entrada, las operaciones permitidas se especifican del siguiente conjunto: leer, escribir, ejecutar, insertar, eliminar y controlar. Las tres primeras son como en UNIX. Insertar y eliminar tienen sentido en directorios, y controlar tiene sentido para los dispositivos de E/S sujetos a la llamada al sistema IOCTL.

DFS soporta cuatro niveles de agregación. En el nivel inferior están los archivos individuales. Éstos se pueden agrupar en directorios de la manera usual. Los grupos de directorios se pueden agrupar en **conjuntos de archivos**. Por último, una colección de conjuntos de archivos forma una partición del disco (o **agregado** en el argot de DCE).

Un conjunto de archivos es por lo general un subárbol en el sistema de archivos. Por ejemplo, podría estar formado por todos los archivos propiedad del usuario, o todos los archivos poseídos por las personas de cierto departamento o proyecto. Un conjunto de archivos es una generalización del concepto de sistema de archivos en UNIX (es decir, la unidad creada por el programa *mkfs*). En UNIX, cada partición del disco contiene exactamente un sistema de archivos, mientras que en DFS puede contener muchos conjuntos de archivos.

El valor del concepto de conjunto de archivos se puede ver en la figura 10-30. En la figura 10-30(a), vemos dos discos (o dos particiones de disco), cada una con tres directorios vacíos. Al transcurrir el tiempo, se crean archivos en estos directorios. Ocurre que el disco 1 se llena más rápido que el disco 2, como se muestra en la figura 10-30(b). Si el disco 1 se llena y el disco 2 tiene demasiado espacio, tenemos un problema.

La solución de DFS es hacer que cada uno de los directorios *A*, *B* y *C* (y sus subdirectorios) sean un conjunto de archivos independiente. DFS permite mover los conjuntos de archivos, de modo que el administrador del sistema puede volver a balancear el espacio en disco, simplemente moviendo el directorio *A* al disco 2, como se muestra en la figura 10-30(c). Mientras ambos discos estén en la misma celda, no se cambian los nombres globales, de modo que todo sigue funcionando después del movimiento anterior.

Además de mover conjuntos de archivos, también es posible duplicarlos. Una réplica se designa como la copia maestra, y sirve para lectura y escritura. Las otras son esclavas, y sólo sirven para lectura. Los conjuntos de archivos, más que las particiones de disco, son las unidades utilizadas para movimientos, réplicas, respaldo, etcétera. Para los sistemas UNIX, cada partición de disco se considera como un agregado con un conjunto de archivos. Se dispone de varias instrucciones para que los administradores del sistema controlen los conjuntos de archivos.

10.7.2. Componentes DFS en el núcleo servidor

DFS consta de agregados a los núcleos del cliente y del servidor, así como varios procesos usuario. En esta sección y las posteriores describiremos este software y lo que hace. En la figura 10-31 se presenta un panorama de las partes.

En el servidor hemos mostrado dos sistemas de archivos, el sistema de archivos nativo de UNIX y, junto a él, el sistema local de archivos de DFS, Episode. Sobre ambos está el controlador de fichas, que controla la consistencia. Más arriba están el exportador de archivos, que controla la interacción con el mundo exterior, y la interfaz de llamadas al sistema, que controla la interacción con los procesos locales. Del lado del cliente, el principal agregado es el controlador del caché, que oculta fragmentos de archivo para mejorar el desempeño.

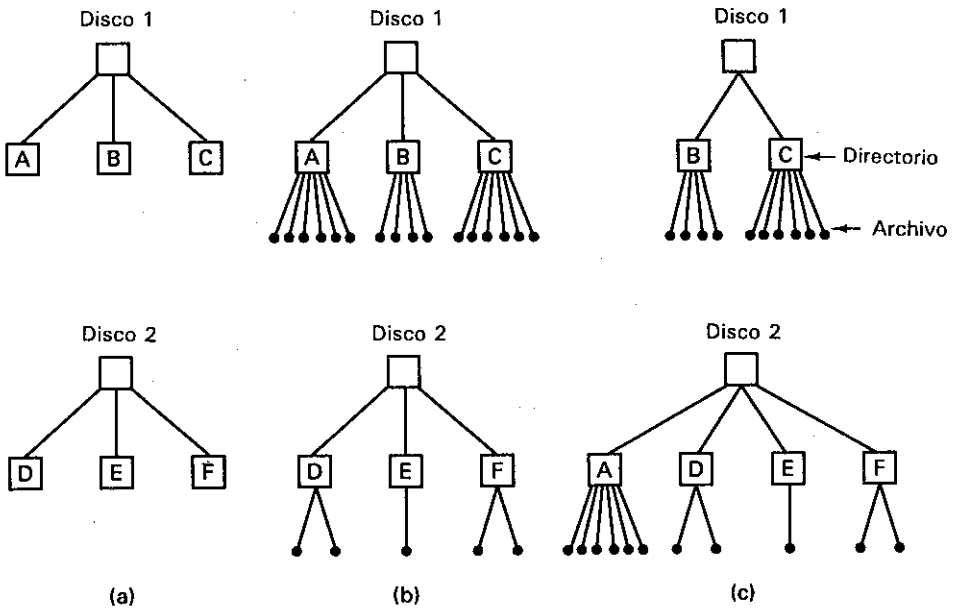


Figura 10-30. (a) Dos discos vacíos. (b) El disco 1 se llena más rápido que el disco 2. (c) La configuración después de mover un conjunto de archivos.

Ahora examinaremos Episode. Como mencionamos arriba, no es necesario ejecutar Episode, pero ofrece ciertas ventajas sobre los sistemas de archivos convencionales, que incluyen la protección con base en ACL, la réplica de conjuntos de archivos, una rápida recuperación y archivos de hasta 2^{42} bytes. Cuando se utiliza el sistema de archivos de UNIX, el software con la marca "Extensiones" en la figura 10-31(b) controla la concordancia de la interfaz del sistema de archivos de UNIX con Episode, por ejemplo, convirtiendo PAC y ACL en el modelo de protección de UNIX.

Una característica interesante de Episode es su capacidad para clonar un conjunto de archivos. Al hacer esto, se crea una copia virtual del conjunto de archivos en otra partición y la original se marca como "exclusiva para lectura". Por ejemplo, una política de celdas podría crear una instantánea exclusiva para lectura de todo el sistema de archivos a las 4 A.M., de modo que si incluso alguien eliminó un archivo de manera inadvertida, esta persona podría regresar a la versión del día anterior.

Episode crea los clones al copiar todas las estructuras de datos del sistema de archivos (nodos-i en términos de UNIX) a la nueva partición, marcando al mismo tiempo las estructuras anteriores como exclusivas para lectura. Ambos conjuntos de estructuras de datos apuntan a los mismos bloques de datos, que no son copiados. Como resultado, la clonación se puede realizar extremadamente rápido. Un intento por escribir en el sistema de archivos original es rechazado con un mensaje de error. Un intento por escribir en el nuevo sistema de archivos tiene éxito, con copias creadas a partir de los nuevos bloques.

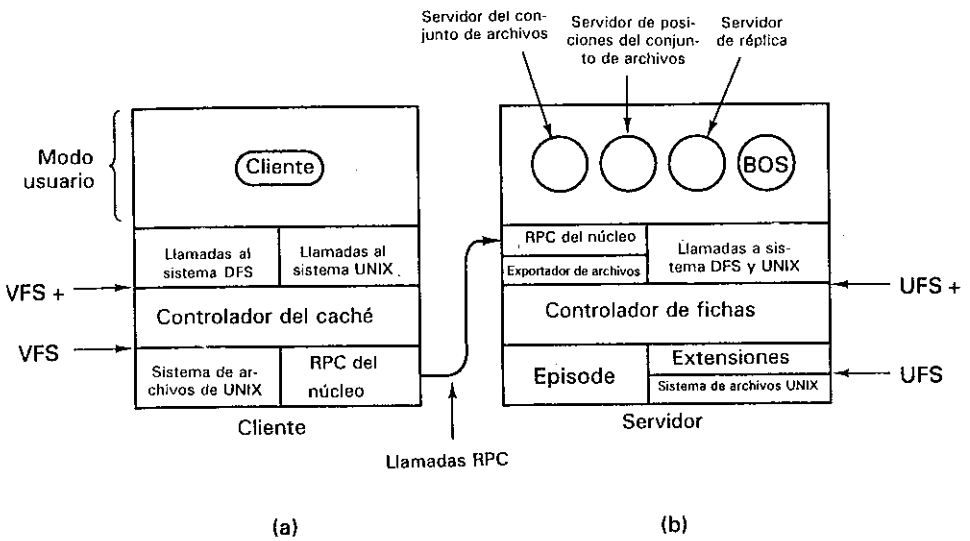


Figura 10-31. Partes de DFS (a) Máquina cliente de archivos. (b) Máquina servidor de archivos.

Episode fue diseñado para un acceso altamente concurrente. Evita que los hilos se cierren durante mucho tiempo en estructuras de datos críticas, para minimizar los conflictos entre los hilos que necesitan el acceso a las mismas tablas. También ha sido diseñado para trabajar con E/S asíncrona, proporcionando un sistema de notificación de eventos al terminar la E/S.

Los sistemas UNIX tradicionales permiten que los archivos tengan cualquier tamaño, pero limitan la mayor parte de las estructuras de datos internas a tablas de tamaño fijo. Episode, por contraste, utiliza internamente una abstracción de almacenamiento general llamada **nodo-a**. Existen nodos-a para archivos, conjuntos de archivos, ACL, mapas de bits, bitácoras, y otros elementos. Sobre la capa del nodo-a, Episode no tiene que preocuparse por el almacenamiento físico (por ejemplo, una ACL muy grande no es más problemática que un archivo de gran tamaño). Un nodo-a es una estructura de datos de 252 bytes; este número se eligió de modo que cuatro nodos-a y 16 bytes de datos administrativos entren en un bloque de disco de 1K.

Cuando se utiliza un nodo-a para una pequeña cantidad de datos (hasta 204 bytes), los datos se guardan directamente en el propio nodo-a. Los objetos pequeños, como los enlaces

simbólicos y muchas ACL caben con frecuencia en este espacio. Cuando se utiliza un nodo-a para una estructura de datos más grande, como un archivo, el nodo-a contiene las direcciones de ocho bloques con datos y cuatro bloques indirectos que apuntan a bloques de discos que contienen aun más direcciones.

Otro aspecto a destacar de Episode es la forma en que enfrenta la recuperación después de una falla. Los sistemas UNIX tradicionales tienden a escribir los cambios a los mapas de bits, los nodos-i y los directorios rápidamente en el disco, para evitar la salida del sistema de archivos en un estado inconsistente en caso de una falla. Episode, por el contrario, escribe en vez de esto una bitácora de las modificaciones en el disco. Cada partición tiene su propia bitácora. Cada entrada de la bitácora contiene el valor anterior y el nuevo valor. En caso de una falla, se lee la bitácora para ver los cambios hechos y los que no. Los cambios ya realizados (es decir, que se perdieron en la falla) se realizan en este momento. Es posible que ciertos cambios recientes al sistema de archivos sigan perdidos (si sus entradas de bitácora no se escribieron en el disco antes de la falla), pero el sistema de archivos siempre será correcto después de la recuperación.

La principal ventaja de este esquema es que al utilizarlo, el tiempo de recuperación es proporcional a la longitud de la bitácora en vez de ser proporcional al tamaño del disco, como ocurre en el caso de que el programa *fsck* de UNIX se ejecuta para reparar un disco potencialmente dañado en los sistemas tradicionales.

De regreso a la figura 10-31, la capa sobre los sistemas de archivos es el administrador de fichas. Puesto que el uso de fichas está íntimamente ligado al ocultamiento, analizaremos las fichas cuando regresemos al ocultamiento en la siguiente sección. Sobre la capa de fichas, se soporta una interfaz que es una extensión de la interfaz NFS VFS de Sun. VFS soporta las operaciones del sistema de archivos, como el montaje y el desmontaje, así como operaciones por archivo, como la lectura, escritura y asignación de nombres a archivos. Éstas y otras operaciones se soportan en VFS+. La principal diferencia entre VFS y VFS+ es la administración de las fichas.

Sobre el administrador de las fichas está el exportador de archivos, Consta de varios hilos, cuyo trabajo consiste en aceptar y procesar las RPC que deseen acceso a archivos. El exportador de archivos controla las solicitudes no únicamente para los archivos Episode, sino también para los otros sistemas de archivos presentes en el núcleo. Mantiene tablas con un registro de los distintos sistemas de archivos y particiones de disco disponibles. También controla la autenticación de los clientes, la colección de PAC y el establecimiento de canales seguros. De hecho, es el servidor de aplicaciones descrito en el paso 5 de la figura 10-27.

10.7.3. Componentes DFS en el núcleo cliente

El principal agregado al núcleo de cada máquina cliente en DCE es el controlador de caché DFS. El objetivo de este controlador es mejorar el desempeño del sistema de archivos, ocultando partes de los archivos en la memoria o en el disco, a la vez que mantiene una verdadera semántica de archivos de un sistema único UNIX. Para aclarar la naturaleza del problema, analizaremos brevemente la semántica de UNIX y por qué es tan importante este aspecto.

En UNIX (y en todos los sistemas operativos de uniprocesador) cuando un proceso escribe en un archivo y luego señala a un segundo proceso para que lo lea el valor leído *debe* ser el valor que se acaba de escribir. El obtener cualquier otro valor viola la semántica del sistema de archivo.

Este modelo de semántica se logra al tener un buffer caché dentro del sistema operativo. Cuando el primer proceso escribe en el archivo, el bloque modificado va al caché. Si el caché se llena, el bloque se puede escribir de regreso en el disco. Cuando el segundo proceso intenta leer el bloque modificado, el sistema operativo busca en primer lugar en el caché. Si no lo encuentra ahí, intenta en el disco. Puesto que sólo existe un caché, en todos los casos se regresa el bloque correcto.

Consideremos ahora el funcionamiento del ocultamiento en NFS. Varias máquinas tienen el mismo archivo abierto al mismo tiempo. Supongamos que el proceso 1 lee parte de un archivo y lo oculta. Posteriormente, el proceso 2 escribe en esa parte del archivo. La escritura no afecta el caché de la máquina donde se ejecuta el proceso 1. Si el proceso 1 vuelve ahora a leer esa parte del archivo, obtendrá un valor obsoleto, lo que viola la semántica de UNIX . Éste es el problema para cuya solución fue diseñado DFS.

En realidad, el problema es peor de lo descrito, pues los directorios también se pueden ocultar. Es posible que un proceso lea un directorio y elimine un archivo de él. Sin embargo, otro proceso de una máquina diferente podría leer posteriormente su copia del directorio, guardada antes en el caché, viendo el archivo ya eliminado. Aunque NFS intenta minimizar este problema verificando la validez con frecuencia, pueden seguir ocurriendo errores.

DFS resuelve este problema mediante fichas. Para realizar cualquier operación de archivo, un cliente hace una solicitud al administrador del caché, quien verifica primero si tiene la ficha y los datos necesarios. Si tiene la ficha y los datos, la operación puede proceder de inmediato, sin hacer contacto con el servidor de archivos. Si no tiene la ficha, el administrador del caché realiza una RPC con el servidor de archivos solicitándola (al igual que los datos). Una vez adquirida la ficha, se puede realizar la operación.

Existen fichas para abrir archivos, leer o escribir en archivos, cerrar archivos y leer o escribir en la información de estado del archivo (por ejemplo, el propietario). Los archivos se pueden abrir para lectura, escritura, lectura y escritura, ejecución, o acceso exclusivo. Las fichas de apertura y de estado se aplican a todo el archivo. Las fichas de lectura, escritura y cierre, por el contrario, sólo se refieren a cierto rango específico de bytes. Las fichas son otorgadas por el administrador de fichas en la figura 10-31(b).

La figura 10-32 muestra un ejemplo de uso de las fichas. En el mensaje 1, el cliente 1 pide una ficha para abrir cierto archivo y leerlo, y también solicita una ficha (y los datos) para la primera parte del archivo. En el mensaje 2, el servidor otorga ambas fichas y proporciona los datos. En este momento, el cliente puede guardar los datos recién recibidos en el caché y leerlos con la frecuencia que desee. La unidad normal de transferencia en DFS es de 64K bytes.

Un poco después, el cliente 2 solicita una ficha para abrir el mismo archivo, leerlo y escribir en él. y también solicita los primeros 64K de datos para escribir de manera selectiva sobre una parte de ellos. El servidor no puede otorgar estas fichas, puesto que ya no las posee. Debe enviar el mensaje 4 al cliente 1 pidiendo que las regrese.

Tan pronto como sea razonablemente conveniente, el cliente 1 debe regresar las fichas al servidor. Después de recibir de regreso las fichas, el servidor las otorga al cliente 2, quien ahora puede leer y escribir los datos a su conveniencia, sin notificar esto al servidor. Si el cliente solicita de nuevo las fichas, el servidor dará instrucciones al cliente 2 para regresar las fichas junto con los datos actualizados, de modo que estos datos se puedan enviar al cliente 1. De esta forma, se mantiene la semántica de un sistema único.

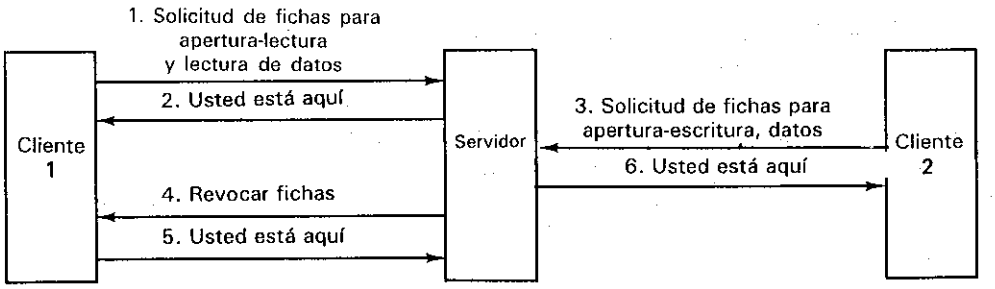


Figura 10-32. Un ejemplo de revocación de fichas.

Para maximizar el desempeño, el servidor comprende la compatibilidad de las fichas. No otorgará de manera simultánea dos fichas de escritura para los mismos datos, pero otorgará dos fichas de lectura para los mismos datos, si ambos clientes tienen el archivo abierto sólo para lectura.

Las fichas no son válidas por siempre. Cada ficha tiene un tiempo de expiración, que por lo general es de dos minutos. Si una máquina falla y no puede (o no está dispuesta a) regresar sus fichas cuando se le pide, el servidor de archivos puede simplemente esperar dos minutos y actuar después como si se le hubieran regresado.

Globalmente, el ocultamiento es transparente a los procesos usuario. Sin embargo, se dispone de llamadas para ciertas funciones de administración del caché, como una búsqueda previa de un archivo antes de utilizarlo, limpieza del caché, administración de cuotas de disco, etc. Sin embargo, el usuario promedio no las necesita.

Hay que mencionar dos diferencias entre DFS y su predecesor, AFS. En AFS se transferían archivos completos, en vez de pedazos de 64K. Esta estrategia hacía esencial la existencia de un disco local, puesto que, con frecuencia, los archivos eran demasiado grandes para ocultarlos en la memoria. Con una unidad de transferencia de 64K, ya no requieren los discos locales.

Una segunda diferencia es que en AFS, una parte del sistema de archivos estaba en el núcleo y otra en el espacio del usuario. Por desgracia, el desempeño dejaba mucho que desear, de modo que en DFS todo está en el núcleo.

10.7.4. Componentes DFS en el espacio del usuario

Ahora hemos terminado de analizar las partes de DFS que se ejecutan en los núcleos del servidor y del cliente. Ahora analizaremos brevemente las partes de él que se ejecutan

en el espacio del usuario (véase figura 10-31). El **servidor de conjuntos de archivos** controla conjuntos de archivos completos. Cada conjunto de archivos contiene uno o más directorios y sus archivos, y debe estar por completo contenido en una partición. Los conjuntos de archivos se pueden montar para formar una jerarquía. Cada conjunto de archivos tiene una cuota en disco a la que debe restringirse.

El servidor del conjunto de archivos permite al administrador del sistema crear, eliminar, mover, duplicar, clonar, respaldar o restaurar un conjunto completo de archivos con una instrucción. Cada una de estas operaciones cierra el conjunto de archivos, realiza el trabajo y libera la cerradura. Se crea un conjunto de archivos para configurar una nueva unidad administrativa para su uso posterior. Cuando ya no se necesita, se puede eliminar.

Un conjunto de archivos se puede mover de una máquina a otra para balancear la carga, en términos del espacio de almacenamiento en disco y en términos del número de solicitudes por segundo por controlar. Cuando se copia un conjunto de archivos sin eliminar al original, se crea un duplicado. Se soportan los duplicados, los que proporcionan un balance de la carga y la tolerancia de fallas. Sólo se puede escribir en una copia.

La clonación, ya descrita, simplemente copia la información de estado (nodos-a), pero no copia los datos. La duplicación crea también una nueva copia de los datos. Un clon debe estar en la misma partición de disco del original. Un duplicado puede estar en cualquier parte (incluso en una celda diferente).

El respaldo y la restauración son funciones que permiten linealizar un conjunto de archivos y copiarlo desde o hacia una cinta para su almacenamiento. Estas cintas se pueden guardar en un edificio diferente, para que sobrevivan no sólo a las fallas del disco, sino también a los incendios, las inundaciones y otros desastres.

El servidor del conjunto de archivos también puede proporcionar información relativa a los conjuntos de archivos, controlar las cuotas de los conjuntos de archivos, y realizar otras funciones de administración.

El **servidor de posiciones del conjunto de archivos** administra una base de datos duplicada a lo largo de la celda que asocia los nombres del conjunto de archivos con los nombres de los servidores que contienen a los conjuntos de archivos. Si se duplica un conjunto de archivos, se pueden encontrar todos los servidores que tienen una copia.

El servidor de posiciones del conjunto de archivos es utilizado por los administradores de caché para localizar conjuntos de archivos. Cuando un programa usuario tiene acceso a un archivo por vez primera, su administrador del caché pregunta al servidor de posiciones del conjunto de archivos la posición donde puede encontrar al conjunto de archivos. Esta información se oculta para su uso posterior.

Cada entrada de la base de datos contiene el nombre del conjunto de archivos, el tipo (lectura/escritura, exclusivo para lectura, o respaldo), el número de servidores que lo contienen, las direcciones de estos servidores, la información del propietario y el grupo del conjunto de archivos, la información relativa a los clones, la información de expiración de la ficha, y otra información administrativa.

El **servidor de réplicas** mantiene actualizadas las réplicas de los conjuntos de archivos. Cada conjunto de archivos tiene una copia maestra (es decir, para lectura/escritura) y tal

vez una o más copias esclavas (es decir, exclusivas para lectura). El servidor de réplicas se ejecuta de manera periódica, revisando cada réplica para ver cuáles archivos han sido modificados desde que se actualizó la última réplica. Estos archivos se reemplazan a partir de los archivos actuales en la copia maestra. Después de que el servidor de réplicas ha terminado, todas las réplicas están actualizadas.

El **servidor Basic Overseer** se ejecuta en cada máquina servidor. Su trabajo es garantizar que los demás servidores están vivos y en buen estado. Si descubre que algunos servidores han fallado, configura nuevas versiones. También proporciona una interfaz para que los administradores del sistema detengan o inicien los servidores en forma manual.

10.8. RESUMEN

DCE es un método de construcción de un sistema distribuido distinto al de Amoeba, Mach y Chorus. En vez de partir de cero con un nuevo sistema operativo que se ejecute sobre el metal puro, DCE proporciona una capa sobre el sistema operativo nativo que oculta las diferencias entre las máquinas individuales, y proporciona servicios y facilidades comunes que unifican a una colección de máquinas como un sistema que es transparente en algunos aspectos (pero no todos). DCE se ejecuta sobre UNIX y otros sistemas operativos.

DCE soporta dos facilidades que se utilizan intensamente, tanto dentro del propio DCE como en los programas usuario: hilos y RPC. Los hilos permiten la existencia de varios flujos de control dentro de un proceso. Cada uno tiene su propio contador de programa, pila, y registros, pero todos los hilos de un proceso comparten el mismo espacio de direcciones, descriptores de archivos y otros recursos del proceso.

RPC es el mecanismo básico de comunicación utilizado en DCE. Permite que un proceso cliente llame a un procedimiento en una máquina remota. DCE proporciona varias opciones a un cliente para enlazarse con un servidor.

DCE soporta cuatro servicios principales (y otros menores) a los que tienen acceso los clientes: los servicios de tiempo, directorios, seguridad y archivos. El servicio de tiempo intenta mantener sincronizados todos los relojes en un sistema DCE, dentro de límites conocidos. Una característica interesante del servicio de tiempo es que represente los tiempos no como valores únicos, sino como intervalos. Como resultado, es posible que al comparar dos tiempos no pueda decirse sin ambigüedad cuál ocurrió primero.

El servicio de directorios guarda los nombres y posiciones de todos los tipos de recursos y permite que los clientes los busquen. El CDS contiene nombres locales (dentro de la celda). El GDS contiene los nombres globales (fuera de la celda). Se soportan los sistemas de asignación de nombres DNS y X.500. Los nombres forman una jerarquía. El servicio de directorios es, de hecho, un sistema distribuido de base de datos.

El servicio de seguridad permite que los clientes y servidores se autentiquen entre sí y realicen una RPC autenticada. El centro del sistema de seguridad es una forma para autenticar a los clientes, y que éstos reciban PAC sin que sus contraseñas aparezcan en la red, ni siquiera en forma cifrada. Los PAC permiten que los clientes demuestren su identidad de manera conveniente y a prueba de tontos.

Por último, el sistema distribuido de archivos proporciona un espacio de nombres para todos los archivos del sistema. Un nombre global de archivo consta de un nombre de celda seguido por un nombre local. El sistema de archivos de DCE consta (opcionalmente) del sistema local de archivos de DCE, Episode, más el exportador de archivos, que hace visibles a todos los sistemas locales de archivos a través del sistema. Los archivos se ocultan mediante un esquema de fichas que mantiene la semántica tradicional de archivos de un sistema.

Aunque DCE proporciona muchas facilidades y herramientas, no es completo y tal vez nunca lo será. Algunas áreas en las que se requiere más trabajo son las técnicas y herramientas de especificación y diseño, los apoyos para depuración, la administración del tiempo de ejecución, la orientación a objetos, las transacciones atómicas y el soporte de multimedia.

PROBLEMAS

1. Una universidad está instalando DCE en todas las computadoras de su campus. Sugiera al menos dos formas de dividir las máquinas en celdas.
2. Los hilos de DCE pueden estar en uno de cuatro estados, como se describió en el texto. En dos de estos estados, listo y escritura, el hilo no se ejecuta. ¿Cuál es la diferencia entre estos estados?
3. En DCE, algunas estructuras de datos se refieren a todo el proceso, mientras que otras sólo se refieren a un hilo. ¿Piensa que las variables de ambiente de UNIX deben ser relativas a todo el proceso, o que cada hilo debe tener su propio ambiente? Defienda su punto de vista.
4. Una variable de condición en DCE siempre está asociada con un mútex. ¿Por qué?
5. Un programador acaba de escribir una parte de código con varios hilos, utilizando una estructura de datos particular. La estructura de datos no puede ser utilizada por más de un hilo a la vez. ¿Qué tipo de mútex debe utilizarse para protegerla?
6. Diga dos atributos plausibles que podría contener la plantilla de un hilo.
7. Cada archivo IDL contiene un número único (por ejemplo, producido por el programa *uuidgen*). ¿Cuál es el valor de contar con dicho número?
8. ¿Por qué IDL requiere que el programador especifique los parámetros que son de entrada y los que son de salida?
9. ¿Por qué se asignan los extremos de RPC de manera dinámica mediante el demonio de RPC y no estáticamente? Una asignación estática sería seguramente más sencilla.
10. Un programador DCE necesita el tiempo de un programa para medición del rendimiento. Llama al procedimiento de tiempo del reloj local (en su propia máquina) y ve que la hora es 15:30:00.0000. Entonces inicia el programa de medición de inmediato. Cuando éste termina, llama de nuevo al procedimiento de tiempo y ve que la hora es 15:35:00.0000. Puesto que utilizó el mismo reloj, en la misma máquina, para ambas mediciones, ¿es seguro afirmar que el programa de medición se ejecutó durante 5 minutos (± 0.1 milisegundos)? Defienda su respuesta.

11. En un sistema DCE, la incertidumbre del reloj es de ± 5 segundos. Si el archivo fuente tiene un intervalo de tiempo de 14:20:30 a 14:20:40 y su archivo binario tiene el tiempo 14:20:32 a 14:20:42, ¿qué haría *make* si se le llama a las 14:20:38? Suponga que *make* tarda un segundo en hacer su trabajo. ¿Qué ocurre si se llama a *make* una segunda vez, unos cuantos minutos más tarde?

12. Un empleado de tiempo pregunta a cuatro servidores de tiempo la hora actual. Las respuestas son

16:00:10.0750 ± 003,
16:00:10.0850 ± 003,
16:00:10.0690 ± 007,
16:00:10.0700 ± 010,

¿A qué hora pone su reloj el empleado de tiempo?

13. ¿Se puede ejecutar DTS sin una fuente UTC conectada a alguna máquina del sistema? ¿Cuáles con las consecuencias de hacer esto?

14. ¿Por qué los nombres distinguidos deben ser únicos, pero no los RDN?

15. Dé el nombre X.500 del departamento de ventas de Kodak en Rochester, Nueva York.

16. ¿Cuál es la función del empleado CDS? ¿Sería posible haber diseñado DCE sin los empleados CDS? ¿Cuáles hubieran sido las consecuencias?

17. CDS es un sistema de base de datos con réplicas. ¿Qué ocurre si dos procesos intentan modificar el mismo elemento en forma simultánea en diferentes réplicas con valores distintos?

18. Si CDS no soportara el sistema de nombres de Internet, ¿cuáles partes de la figura 10-24 podrían omitirse?

19. Durante la secuencia de autenticación, un cliente adquiere primero un boleto y posteriormente un PAC. Puesto que ambos contienen al identificador del usuario, ¿por qué el problema de adquirir un PAC una vez que se ha obtenido el boleto necesario?

20. ¿Cuál es la diferencia (si existe alguna) entre los siguientes mensajes:

$\{\{\text{mensaje}\}K_A\}K_C$ y
$\{\{\text{mensaje}\}K_C\}K_A$.

21. El protocolo de autenticación descrito en el texto permite que un intruso envíe muchos mensajes de manera arbitraria al servidor de autenticación, en un intento por obtener una respuesta que contenga una clave inicial de sesión. ¿Qué debilidad introduce esta observación al sistema? (*Sugerencia*: Puede suponer que todos los mensajes contienen una marca de tiempo e información adicional, de modo que sea sencillo decir si un mensaje es válido de un montón aleatorio.)

22. El texto sólo analiza el caso de la seguridad dentro de una sola celda. Proponga una forma de realizar la autenticación de la RPC cuando el cliente está en la celda local y el servidor está en una celda remota.

23. Las fichas pueden expirar en DFS. ¿Requiere esto relojes sincronizados que utilicen DFS?

24. Una ventaja de DFS sobre NFS es que DFS preserva la semántica de archivos de un sistema. Mencione una desventaja.

25. ¿Por qué un clon de un conjunto de archivos debe estar en la misma partición de disco?

11

Lista de lecturas y bibliografía

En los diez capítulos anteriores hemos tocado varios temas. Este capítulo intenta ayudar a los lectores interesados en continuar su estudio de los sistemas operativos. La sección 11.1 es una lista de lecturas sugeridas. La sección 11.2 es una bibliografía en orden alfabético de todos los libros y artículos citados en este libro.

Además de estas referencias, los *Proceedings* del n-ésimo *ACM Symposium on Operating Systems Principles* (SOSP) que se lleva a cabo cada dos años y los *Proceedings* de la n-ésima *International Conference on Distributed Computing Systems* (DCS) que se lleva a cabo cada año son buenos lugares para encontrar artículos recientes relativos a los sistemas operativos. Además, las *ACM Transactions on Computer Systems* y el *Operating Systems Review* son dos revistas que frecuentemente tienen artículos interesantes sobre sistemas operativos distribuidos.

11.1. SUGERENCIAS PARA LECTURA POSTERIOR

11.1.1. Introducción y obras generales

Andrews, *Concurrent Programming-Principles and Practice*

Una amplia introducción a la programación de sistemas concurrentes.

Revista *Byte*, junio 1994

Un informe especial acerca del cómputo distribuido da una perspectiva del tema, orientada al usuario. Los cinco artículos trabajan con DCE, la administración distribuida de datos, la seguridad, los sistemas de información ejecutiva y los clientes remotos.

Champine *et al.*, "Project Athena as a Distributed Computer System"

Athena es un sistema operativo de red de MIT, consistente en más de 1000 estaciones de trabajo basadas en UNIX. El proyecto ha desarrollado varios paquetes de software que se han convertido en estándares de hecho, como X (manejo de ventanas) y Kerberos (autenticación). El artículo da un panorama de todo el sistema.

Coulouris *et al.*, *Distributed Systems Concepts and Design*, segunda edición

Un buen texto general de sistemas distribuidos. Analiza los protocolos de red, RPC, sistemas operativos distribuidos, sistemas de archivos, servidores de nombres, tiempo, réplica, transacciones, control de concurrencia, tolerancia de fallas, seguridad y DSM. Se dan cuatro casos de estudio breves: Mach, Chorus, Amoeba y Clouds.

Mullender, *Distributed Systems*, segunda edición

Resultado de una escuela de verano, contiene 21 artículos de autores líderes en el campo de los sistemas distribuidos. Los temas abarcan el modelado, la especificación, la tolerancia de fallas, el tiempo real, la comunicación, la asignación de nombres, los sistemas de archivos, la planificación y la seguridad, entre otros.

11.1.2. Comunicación en sistemas distribuidos

Ballart y Ching, "SONET; Now It's the Standard Optical Network"

Comprender SONET no es para neófitos, pero este tutorial resalta las principales características de SONET de manera casi indolora. También abarca parte de la historia del proceso de estandarización.

Bershad *et al.*, "Lightweight Remote Procedure Call"

Se describe un método para realizar una RPC rápida en un procesador o multiprocesador. En ella, el cliente tiene que ejecutar un procedimiento seleccionado con anterioridad en el espacio de direcciones del servidor, con lo que se evita un cambio de contexto.

Birman y Van Renesse, "Reliable Distr. Computing with the ISIS Toolkit"

Una colección de artículos acerca de varios aspectos de ISIS. Cerca de la tercera parte de los artículos contienen un material panorámico, una tercera parte describe la base teórica de ISIS y otra tercera parte abarca las aplicaciones de ISIS.

Birrel y Nelson, "Implementing Remote Procedure Calls"

Las llamadas a procedimientos remotos se utilizan por lo general en los sistemas distribuidos para la comunicación entre procesos. Este artículo describe el diseño y la implantación del sistema original de llamada a procedimientos remotos.

Clark *et al.*, "The Aurora Gigabit Testbed"

Aurora es una de las varias redes experimentales de gigabits diseñadas para los futuros sistemas distribuidos. Este artículo describe la red, los adaptadores de anfitriones, los protocolos, las aplicaciones y la administración de la red.

De Prycker, *Asynchronous Transfer Mode*

Un libro completo de ATM. Si desea conocer toda la historia, busque aquí.

Hutchinson *et al.*, "RPC in the *x*-Kernel: Evaluating New Design Techniques"

El núcleo *x* utiliza una técnica similar a los flujos de UNIX para permitir a las pilas del protocolo que se organicen para el manejo de los protocolos por capas basados en RPC. El mecanismo es ligero y utiliza las llamadas a procedimientos entre las capas.

Le Boudec, "The Asynchronous Transfer Mode"

Una introducción sencilla a ATM. Este artículo abarca las capas física, de ATM, y de adaptación, analiza los servicios futuros basados en ATM y proporciona ciertas bases acerca de la historia de ATM.

Mullender, "Interprocess Communication"

Las redes, los protocolos y la RPC se examinan con detalle en este tutorial acerca de la comunicación entre procesos. También se analizan varios aspectos del sistema.

Tay y Ananda, "A Survey of Remote Procedure Calls"

A pesar de ciertas analogías fundamentales, los sistemas RPC difieren en varios aspectos. Este artículo da un panorama de ocho sistemas distintos de RPC, desde proyectos académicos de investigación hasta sistemas comerciales y los compara de varias formas.

11.1.3. Sincronización en sistemas distribuidos

Fidge, "Logical Time in Distributed Computing Systems"

Método para enfrentar el ordenamiento de eventos en un sistema distribuido basado en la causalidad y el ordenamiento parcial de tiempo, en vez del ordenamiento total.

Ramanathan *et al.*, "Fault-Tolerant Clock Synchronization in Distr. Systems"

Panorama de los algoritmos para la sincronización de relojes para su uso en los sistemas distribuidos. Se analizan métodos en hardware, software e híbridos.

Raynal, "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms"

Taxonomía y bibliografía de los algoritmos distribuidos de exclusión mutua. Las principales categorías definidas son "basados en los permisos" y "basados en fichas"; los algoritmos centralizados están en la intersección de las dos.

Silberschatz *et al.*, *Operating System Concepts.*

El capítulo 18 de este libro de texto trata la sincronización en los sistemas distribuidos, incluyendo el ordenamiento de eventos, la exclusión mutua, los protocolos de acuerdo y los algoritmos de elección.

Singhal, "Deadlock Detection in Distributed Systems"

Tutorial de la detección distribuida de bloqueos. Primero analiza los aspectos relacionados con el tema. Después pasa a un análisis de los algoritmos centralizados, descentralizados y jerárquicos en los sistemas distribuidos.

Weihl, "Transaction-Processing Techniques"

Una introducción a las transacciones atómicas, incluyendo las transacciones anidadas. Contiene un amplio análisis de los métodos de recuperación, para fallas en un lugar o distribuidas.

11.1.4. Procesos y procesadores en sistemas distribuidos

Anderson *et al.*, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism"

Se presenta una nueva abstracción para combinar las mejores propiedades de la administración de los hilos a nivel núcleo y a nivel usuario. La abstracción consiste en darle a cada proceso un multiprocesador virtual y utilizar las llamadas para informar a los usuarios de los eventos importantes de planificación.

Burns y Wellings, *Real-Time Systems and Their Programming Languages*

Un texto introductorio relativo a la forma de diseñar y programar sistemas de tiempo real en Ada, Modula 2 y occam 2. Los temas cubiertos comprenden la tolerancia de fallas, el manejo de excepciones, acciones atómicas, control de recursos, y aspectos de programación de bajo nivel. El libro contiene muchos fragmentos de código para ilustrar las ideas.

Cristian, "Understanding Fault-Tolerant Distributed Systems"

Una introducción a la tolerancia de fallas en los sistemas distribuidos, que incluye la clasificación, semántica y ocultamiento de las fallas. Se tratan aspectos de hardware y de software. Los ejemplos de hardware incluyen Tandem, Sequoia, la unidad de asignación de VAX y XRF de IBM. Entre los aspectos del software están la comunicación en grupo y el acuerdo global.

Marsh *et al.*, "First-Class, User-Level Threads"

Se presenta un conjunto de mecanismos y convenciones para que los hilos se puedan manejar dentro del espacio del usuario pero aprovechando el conocimiento del núcleo. La idea se basa en las llamadas.

Natarajan y Zhao, "Issues in Building Dynamic Real-Time Systems"

Un breve tutorial acerca de los sistemas de tiempo real que analiza aspectos como los requisitos, la disponibilidad, las garantías y la administración de recursos.

Nichols, "Using Idle Workstations in a Shared Computing Environment"

Descripción del sistema Butler para la búsqueda y uso de estaciones de trabajo inactivas de UNIX. Una bitácora lleva un registro de las máquinas y las asigna.

Shivarati *et al.*, "Load Distributing for Locally Distributed Systems"

En un sistema distribuido, la carga de trabajo pierde el balance, de modo que algunas máquinas estén inactivas y otras estén sobrecargadas. En este tutorial se analizan varios algoritmos para balancear la carga.

Verissimo, "Real-Time Communication"

Los sistemas distribuidos de tiempo real tienen necesidades específicas de comunicación que no están presentes en la mayor parte de los demás sistemas. Este artículo analiza algunas de ellas y la forma de conseguirlas.

11.1.5. Sistemas distribuidos de archivos

Levy y Silberschatz, "Distributed File Systems: Concepts and Examples"

La primera mitad de este artículo analiza los principios de los sistemas distribuidos de archivos. La segunda mitad analiza varios ejemplos: UNIX United, Locus, NFS, Sprite y Andrew.

Satyanarayanan, "A Survey of Distributed File Systems"

Aquí se examinan ciertos aspectos básicos del diseño de los sistemas distribuidos de archivos. También se presentan los casos de estudio de NFS, Apollo Domain, Andrew, AIX, RFS y Sprite.

Satyanarayanan, "Distributed File Systems"

En esta introducción a los sistemas distribuidos de archivos se tratan los principios y la práctica. Se analizan los mecanismos de uso común, como el ocultamiento, la transferencia en bloque, y ciertas sugerencias, así como varios sistemas existentes, AFS, Coda y NFS.

Svobodova, "File Servers for Network-Based Distributed Systems"

Panorama de los servidores de archivos utilizados en los sistemas distribuidos. Se enfatizan los servidores de archivos que proporcionan acciones y transacciones atómicas.

11.1.6. Memoria compartida distribuida

Li y Hudak, "Memory Coherence in Shared Virtual Memory Systems"
El trabajo de Li y Hudak creó el campo de la memoria compartida distribuida (DSM). Este artículo describe los sistemas de DSM basados en páginas con administradores centralizados o descentralizados.

Nitzberg y Lo, "Distr. Shared Memory: A Survey of Issues and Algorithms"
Un tutorial acerca del diseño e implantación de los sistemas DSM con centro en los modelos de consistencia. Se comparan nueve ejemplos diferentes y se citan otros ocho.

Stumm y Zhou, "Algorithms Implementing Distributed Shared Memory"
Un tutorial acerca de la memoria compartida distribuida.

Tanenbaum *et al.*, "Parallel Progr. Using Shared Objects and Broadcasting"
En contraste con las referencias anteriores, todas las cuales tratan la DSM basada en páginas, ésta muestra la forma en que se implantan los objetos en una red de máquinas con memorias locales.

11.1.7. Estudio 1: Amoeba

Douglis *et al.*, "A comparison of Two Distr. Systems: Amoeba and Sprite"
Comparación de dos sistemas distribuidos: Amoeba, que tiene un micronúcleo y utiliza el modelo de la pila de procesadores; y Sprite, que tiene un núcleo monolítico y utiliza el modelo de la estación de trabajo.

Kaashoek y Tanenbaum, "Group Communication in the Amoeba Distributed Operating System"
Introducción a la comunicación en grupo de Amoeba; en particular, el protocolo de transmisión confiable utilizado y su implantación. El artículo también analiza la tolerancia de fallas en el protocolo y la forma en que el protocolo de transmisión confiable se recupera de las fallas del secuenciador y de otras partes del sistema.

Mullender *et al.*, "Amoeba: A Distributed Operating System for the 1990s"
Panorama de Amoeba, con énfasis en los mecanismos de comunicación, objetos, seguridad, el sistema de archivos y la administración de los procesos.

Tanenbaum *et al.*, "Experiences with the Amoeba Distr. Operating System"
Otra introducción a Amoeba, la cual enfatiza los objetos, la RPC, los servidores, Amoeba de área amplia, las aplicaciones y el desempeño. Concluye con una evaluación del diseño con base en la experiencia real (lo que se hizo bien; pero más importante, lo que se hizo mal).

11.1.8. Estudio 2: Mach

Accetta *et al.*, "Mach: A New Kernel Foundation for UNIX Development"
Uno de los primeros artículos publicados acerca del sistema Mach. Describe los objetivos del sistema, las ideas básicas (como los hilos, puertos y mensajes) y la implantación.

Black, "Scheduling Support for Concurrency and Parallelism in the Mach Sys."
Aquí se describe el algoritmo de planificación de Mach para los multiprocesadores. Se analizan varias optimizaciones, como la planificación "manos fuera", y se proporcionan algunas mediciones del desempeño.

Boykin *et al.*, *Programming under Mach*
Un libro completo acerca de la forma de escribir programas que se ejecuten en Mach y utilicen sus principales facilidades. El énfasis está en la forma de utilizar Mach, y no en su funcionamiento interno.

Boykin y Langerman, "Mach/4.3BSD: A Conservative Approach to Parallelization"
Los intentos y tribulaciones para lograr que el emulador de Mach para UNIX se ejecute de manera eficiente en un multiprocesador, algo para lo que nunca se pensó. Se describen los problemas encontrados con la E/S y el sistema de archivos, al igual que algunas soluciones.

Rashid, "From RIG to Accent to Mach: The Evolution of a Network Op. Sys."
Breve historia de RIG, Accent y Mach escrita por su autor. Se describe la evolución del sistema, enfatizando los cambios ocurridos como resultado de la nueva tecnología y los nuevos objetivos.

Young *et al.*, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,"
Objetivos, diseño e implantación del sistema para la administración de la memoria en Mach, así como la forma en que éste interactúa con el sistema de comunicación. Se describe el uso de los administradores externos de la memoria.

11.1.9. Estudio 3: Chorus

Gien, "Micro-kernel Architecture: Key to Modern Operating Systems Design"
Una introducción a la filosofía y finalidades de Chorus por uno de sus diseñadores.

Gien y Grob, "Microkernel Based Operating Systems"
Un análisis de la forma en que Chorus se utiliza para emular a UNIX.

Rozier *et al.*, "Chorus Distributed Operating Systems"

Aunque ligeramente fuera de época, ésta sigue siendo la mejor introducción general a la arquitectura del micronúcleo de Chorus.

11.1.10. Estudio 4: DCE

Bever *et al.*, "Distributed Systems, OSF DCE and Beyond"

Una útil introducción a DCE enfatizando la arquitectura e interfaz de RPC. Este artículo también señala varias características que DCE no posee (por ejemplo, herramientas avanzadas, orientación a objetos, transacciones distribuidas, soporte de multimedia) y muestra la forma en que podrían quedar dentro del modelo de DCE.

Kazar *et al.*, "DEcorum File System Architectural Overview"

DEcorum es la componente del sistema distribuido de archivos de DCE y el sucesor de AFS. Aquí se proporciona un panorama, junto con un análisis de las fichas, el ocultamiento, la réplica, los bloques y otros temas.

OSF, *Introduction to OSF DCE*

El tutorial de DCE más accesible, que abarca los mismos temas que el presente libro.

Rosenberry *et al.*, *Understanding DCE*

Una introducción general a DCE que abarca todas las ideas básicas en 14 capítulos y 4 apéndices.

Shirley, *Guide to Writing DCE Applications*

Un tutorial que indica la forma de escribir programas que se ejecuten en DCE, tanto servidores como clientes, Varios fragmentos de código aparecen como ejemplos.

11.2. BIBLIOGRAFÍA EN ORDEN ALFABÉTICO

ABROSSIMOV, A., ARMAND, F., y ORTEGA, M.: "A Distributed Consistency Server for the CHORUS System," *Proc. SEDMS III Symp. on Experience with Distributed and Multiprocessor Systems.* USENIX, pp. 129-148, 1992.

ABROSSIMOV, A., ROZIER, M., y SHAPIRO, M.: "Generic Virtual Memory Management in Operating Systems Kernels, "*Proc. 12th Symp. on Operating Systems Principles* ACM, pp. 123-136, 1989.

ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., y YOUNG, M.: "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.* USENIX, pp. 93-112, 1986.

ADVE, S., y HILL, M.: "Weak Ordering: A New Definition," *Proc. 17th Ann. Int'l Symp. on Computer Architecture* ACM, pp. 2-14, 1990.

AGARWAL, A., CHAIKEN, D., D'SOUZA, G., JOHNSON, K., KRANZ, D., KUBIATOWICZ, J., KURIHARA, K., LIM, B., MAA, G., NUSSBAUM, D., PARKIN, M., y YEUNG, D.: "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor, " *Proc. Workshop on Scalable Shared Memory Multiprocessors*, Kluwer, 1991.

AGARWAL, A., y CHERIAN, M.: "Adaptive Backoff Synchronization Techniques," *Proc. 16th Ann. Int'l Symp. on Computer Architecture* ACM, pp. 396-406, 1989.

AGARWAL, A., SIMONI, R., HENNESSY, J., y HOROWITZ, M.: "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Ann. Int'l Symp. on Computer Architecture* ACM, pp. 280-289, 1988 .

AGRAWAL, D., y EL ABBADI, A: "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion," *ACM Trans. on Computer Systems* vol. 9, pp. 1-20, febrero. 1991.

AHAMAD, M., BAZZI, R.A., JOHN, R., KOHLI, P., y NEIGER, G.: "The Power of Processor Consistency," Tech. Rep. GIT-CC-92/34, College of Computing, Georgia Inst. of Technology, marzo, 1993.

AHMADI, H., y DENZEL, W.: "A Survey of Modern High-Performance Switching Techniques," *IEEE Journal of Selected Areas in Communication* vol. 7, pp. 1091-1103, septiembre. 1989.

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., y LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proc. 13th Symp. on Operating Systems Principles* ACM, pp. 95-109, 1991.

ANDERSON, T.E., OWICKI, S.S., SAXE, J.B., y THACKER, C.P.: "High-Speed Switch Scheduling for Local-Area Networks," *ACM Trans. on Computer Systems* vol. 11, pp. 319-352, noviembre. 1993.

ANDREWS, G.R.: *Concurrent Programming—Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.

ARCHIBALD, J., y BAER, J.-L.: "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems* vol. 4, pp. 273-298, noviembre. 1986.

ARMAND, F., y DEAN, R: "Data Movement in Kernelized Systems," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures,* USENIX, pp. 243-261, 1992.

ARTSY, Y., y FINKEL, R.: "Designing a Process Migration Facility," *IEEE Computer* vol. 22, pp. 47-56, septiembre. 1989.

ATTIYA, H., y FRIEDMAN, R.: "A Correctness Condition for High Performance Multiprocessors," *Proc. 24th ACM Symp. on Theory of Computing* ACM, pp. 679-690, 1992.

BAL, H.E.: *Programming Distributed Systems,* Hemel Hempstead, England: Prentice Hall Int'l, 1991.

BAL, H.E., y KAASHOEK, M.F.: "Object Distribution in Orca using Compile-Time and Run-Time Techniques," *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)* ACM, pp. 162-177, septiembre. 1993.

BAL, H.E., KAASHOEK, M.F., y TANENBAUM, A.S.: "Experience with Distributed Programming in Orca," *Proc. Int'l Conf. on Computer Languages '90,* IEEE, pp. 79-89, 1990.

BAL, H.E., KAASHOEK, M.F., y TANENBAUM, A.S.: "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Engineering,* vol. 18, pp. 190-205, marzo. 1992.

BALL, J.E., FELDMAN, J.A., LOW, J.R., RASHID, R.F., y ROVNER, P.D.: "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Trans. on Software Engineering,* vol. SE-2, pp. 321-328, diciembre. 1976.

BALLART, R., y CHING, Y.-Y.: "SONET: Now It's the Standard Optical Network," *IEEE Communications Magazine,* vol. 29, pp. 8-15, marzo. 1989.

BARBORAK, M., MALEK, M., y DAHBURA, A.: "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys,* vol. 25, pp. 171-220, junio. 1993.

BARON, R.; RASHID, R.; SIEGEL, E.; TEVANIAN, A., y YOUNG, M.: "Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications," *IEEE Software,* vol. 2, pp. 65-67, julio. 1985.

BATLIVALA, N., GLEESON, B., HAMRICK, J., LURNDAL, S., PRICE, D., SODDY, J., y ABROSSIMOV, V.: "Experience with SVR4 Over CHORUS," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures,* USENIX, pp. 223-241, 1992.

BELLOVIN, S.M., y MERRITT, M.: "Limitations of the Kerberos Authentication System," *Proc. Invierno 1991 USENIX Conf.,* USENIX, enero. 1991.

BENNETT, J.K., CARTER, J.K., y ZWAENEPOEL, W.: "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming* ACM, pp. 168-176, 1990.

BERNSTEIN, P.A., y GOODMAN, N.: "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Trans. on Database Systems,* vol. 9, pp. 596-615, diciembre. 1984.

BERSHAD, B.N., ANDERSON, T.E., LAZOWSKA, E.D., y LEVY, H.M.: "Lightweight Remote Procedure Call," *ACM Trans. on Computer Systems,* vol. 8, pp. 37-55, febrero. 1990.

BERSHAD, B.N., y ZEKAUSKAS, M.J.: "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," CMU Report CMU-CS-91-170, septiembre. 1991.

BERSHAD, B.N., ZEKAUSKAS, M.J., y SAWDON, W.A.: "The Midway Distributed Shared Memory System," *Proc. IEEE COMPCON Conf.*, IEEE, pp. 528-537, 1993.

BEVER, M., GEIHS, K., HEUSER, L., MUHLHAUSER, M., y SCHILL, A.: "Distributed Systems, OSF DCE, and Beyond," in *DCE—The OSF Distributed Computing Environment,* A. Schill (ed.), Berlin: Springer-Verlag, pp. 1-20, 1993.

BIRMAN, K.P.: "The Process Group Approach to Reliable Distributed Computing," *Commun. of the ACM,* vol. 36, pp. 36-53, diciembre. 1993.

BIRMAN, K.P., y JOSEPH, T.: "Reliable Communication in the Presence of Failures," *ACM Trans. on Computer Systems,* vol. 5, pp. 47-76, febrero. 1987a.

BIRMAN, K.P., y JOSEPH, T.: "Exploiting Virtual Synchrony in Distributed Systems," *Proc. 11th Symp. on Operating Systems Principles,* ACM, pp. 123-138, noviembre. 1987b.

BIRMAN, K.P., SCHIPER, A., y STEPHENSON, P.: "Lightweight Causal and Atomic Group Multicast," *ACM Trans. on Computer Systems,* vol. 9, pp. 272-314, agosto. 1991.

BIRMAN, K.P., y VAN RENESSE, R.: *Reliable Distributed Computing with the ISIS Toolkit,* Los Alamitos, CA: IEEE Computer Society Press, 1994.

BIRRELL, A.D., y NELSON, B.J.: "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems,* vol. 2, pp. 39-59, febrero. 1984.

BITAR, P.: "MIMD Synchronization and Coherence," Tech. Rep. 90/605, Univ. of Calif. at Berkeley, noviembre. 1990.

BJORNSON, R.D.: "Linda on Distributed Memory Multiprocessors," Ph.D. Thesis, Yale Univ., 1993.

BLACK, D.: "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer,* vol. 23, pp. 35-43, mayo. 1990.

BLACK, D.L., GOLUB, D.B., JULIN, D.P., RASHID, R.F., DRAVES, R.P., DEAN, R.W., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., y BOHMAN, D.: "Microkernel Operating System Architecture and Mach," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures,* USENIX, pp. 11-30, 1992.

BOLOSKY, W.J., FITZGERALD, R.P., y SCOTT, M.L.: "Simple but Effective Techniques for NUMA Memory Management," *Proc. 12th Symp. on Operating Systems Principles,* ACM, pp. 19-31, 1989.

BOYKIN, J., KIRSCHEN, D., LANGERMAN, A., y LoVERSO, S.: *Programming Under Mach, Reading, MA: Addison-Wesley, 1993.*

BOYKIN, J., y LANGERMAN, A.: "Mach/4.3BSD: A Conservative Approach to Parallelization," *Computing Systems,* vol. 3., pp. 69-99, invierno. 1990.

BRERETON, O.P.: "Management of Replicated Files in a UNIX Environment," *Software—Practice and Experience,* vol. 16, pp. 771-780, agosto. 1986.

BRICKER, A., GIEN, M., GUILLEMONT, M., LIPKIS, J., ORR, D., y ROZIER, M.: "A New Look at Microkernel-based UNIX operating systems: Lessons in performance and compatibility," *Proc. EurOpen Spring'91 Conf.,* EurOpen, pp. 13-32, 1991.

BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F.B., y TOUEG, S.: "The Primary—Backup Approach," in *Distributed Systems,* segunda edición, S. Mullender (ed.), Nueva York, NY: ACM Press, pp. 199-216, 1993.

BURNS, A., y WELLINGS, A.: *Real-Time Systems and Their Programming Languages,* Reading, MA: Addison-Wesley, 1990.

CARRIERO, N., y GELERNTER, D.: "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems,* vol. 4, pp. 110-129, mayo. 1986.

CARRIERO, N., y GELERNTER, D.: "Linda in Context," *Commun. of the ACM,* vol. 32, pp. 444 - 458, abril. 1989.

CARRIERO, N., GELERNTER, D., y LEICHTER, J.: "Distributed Data Structures in Linda," *Proc. ACM Symposium on Principles of Programming Languages,* ACM, 1986.

CARTER, J.B., BENNETT, J.K., y ZWAENEPOEL, W.: "Implementation and Performance of Munin," *Proc. 13th Symp. on Operating Systems Principles,* ACM, pp. 152-164, 1991.

CARTER, J.B., BENNETT, J.K., y ZWAENEPOEL, W.: "Techniques for Reducing Consistency-Related Communication in Distributed Shared memory Systems," *ACM Trans. on Computer Systems,* vol. 12, 1994.

CATLETT, C.E.: "In Search of Gigabit Applications," *IEEE Communications Magazine,* vol. 30, pp. 42-51, abril. 1992.

CHAMPINE, G.A., GEER, D.E., Jr., y RUH, W.N.: "Project Athena as a Distributed Computer System," *IEEE Computer,* vol. 23, pp. 40-51, septiembre. 1990.

CHANDY, K.M., MISRA, J., y HAAS, L.M.: "Distributed Deadlock Detection," *ACM Trans. on Computer Systems,* vol. 1, pp. 144-156, mayo. 1983.

CHANG, J., y MAXEMCHUK, N.F.: "Reliable Broadcast Protocols," *ACM Trans. on Computer Systems,* vol. 2, pp. 39-59, febrero. 1984.

CHASE, J.S., AMADOR, F.G., LAZOWSKA, E.D., LEVY, H.M., y LITTLEFIELD, R.J.: "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proc. 12th Symp. on Operating Systems Principles,* ACM, pp. 147-158, 1989.

CHEONG, H., y VEIDENBAUM, A.V.: "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. 15th Ann. Int'l Symp. on Computer Architecture,* ACM, pp. 299-307, 1988.

CHEUNG, N.K.: "The Infrastructure of Gigabit Computer Networks," *IEEE Communications Magazine,* vol. 30, pp. 60-68, abril. 1992.

CHOW, T.C.K., y ABRAHAM, J.A.: "Load Balancing in Distributed Systems," *IEEE Trans. on Software Engineering,* vol. SE-8, pp. 401-412, julio. 1982.

CHUTANI, S., ANDERSON, O.T., KAZAR, M.L., LEVERETT, B.W., MASON, W.A., y SIDEBOTHAM, R.N.: "The Episode File System," *Proc. Winter 1992 USENIX Conf.*, USENIX, pp. 43-60, 1992.

CLARK, D.D., DAVIE, B.S., FARBER, D.J., GOPAL, I.S., KADABA, B.K., SINCOSKIE, W.D., SMITH, J.M., y TENNENHOUSE, D.L.: "The Aurora Gigabit Testbed," *Computer Networks and ISDN Systems,* vol. 25, pp. 599-621, junio. 1993.

COHEN, D.: "On Holy Wars and a Plea for Peace," *IEEE Computer,* vol. 14, pp. 48-54, octubre. 1981.

COMER, D.E.: *Internetworking with TCP/IP.* Vol. 1: Principles, Protocols and Architectures, segunda edición, Englewood Cliffs, NJ: Prentice Hall, 1991.

COULOURIS, G.F., DOLLIMORE, J., y KINDBERG, T.: *Distributed Systems Concepts and Design,* segunda edición. Reading, MA: Addison-Wesley, 1994.

COX, A.L., y FOWLER, R.J.: "The Implementation of a Coherent Memory Abstraction of a NUMA Multiprocessor: Experiences with PLATINUM," *12th Symp. on Operating Systems Principles,* ACM, pp. 32-43, 1989.

CRISTIAN, F.: "Probabilistic Clock Synchronization," *Distributed Computing,* vol. 3, pp. 146-158, 1989.

CRISTIAN, F.: "Understanding Fault-Tolerant Distributed Systems," *Commun. of the ACM,* vol. 34, pp. 56-78, febrero. 1991.

DAHLGREN, F., DUBOIS, M., y STENSTROM, P.: "Combined Performance Gains of Simple Cache Protocol Extensions," *Proc. 21stAnn. Int'l Symp. on Computer Architecture,* ACM, pp. 187-195.

DAY, J.D., y ZIMMERMAN, H.: "The OSI Reference Model," *Proc. IEEE,* vol. 71, pp. 1334-1340, diciembre. 1983.

DE PRYCKER, M.: *Asynchronous Transfer Mode,* Chichester, England: Ellis Horwood, 1991.

DELP, G.S.: *The Architecture and Implementation of MemNet. An Experiment on High-Speed Memory Mapped Network Interface,* Ph.D. Thesis, Univ. of Delaware, 1988.

DELP, G.S., FARBER, D.J., MINNICH, R.G., SMITH, J.M., y TAM, M.-C.: "Memory as a Network Abstraction," *IEEE Network,* vol. 5, pp. 34 - 41, julio. 1991.

DOUGLIS, F., y OUSTERHOUT, J.: "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice and Experience,* vol. 21, pp. 757-785, agosto. 1991.

DOUGLIS, F., OUSTERHOUT, J.K., KAASHOEK, M.F., y TANENBAUM, A.S.: "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems,* vol. 4, pp. 353-384, otoño. 1991.

DRAVES, R.P.: "The Revised IPC Interface," *Proc. First USENIX Mach Symp.,* USENIX, pp. 101-121,1990.

DRAVES, R.P., BERSHAD, B.N., RASHID, R.F., y DEAN, R.W.: "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proc. 13th Symp. on Operating Systems Principles,* ACM, pp. 122-136, 1991.

DRUMMOND, R., y BABAOGLU, O.: "Low-Cost Clock Synchronization," *Distributed Computing,* vol. 6, pp. 193-203, 1993.

DUBOIS, M., SCHEURICH, C., y BRIGGS, F.A.: "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l Symp. on Computer Architecture,* ACM, pp. 434 - 442, 1986.

DUBOIS, M., SCHEURICH, C., y BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer,* vol. 21, pp. 9-21, febrero. 1988.

EAGER, D.L., LAZOWSKA, E.D., y ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Engineering,* vol. SE-12, pp. 662-675, mayo. 1986.

ECKBERG, A.E. "B-ISDN/ATM Traffic Control and Congestion," *IEEE Network,* vol. 5, pp. 28-37, septiembre. 1992.

EDLER, J., LIPKIS, J., y SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems," *Proc. USENIX Workshop on UNIX and Supercomputers,* USENIX, pp. 1-17, septiembre. 1988.

EGGERS, S.J., y KATZ, R.H.: "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *Proc. Second ASPLOS Conf.,* ACM, pp. 257-271, 1989a.

EGGERS, S.J., y KATZ, R.H.: "Evaluating the Performance of Four Snooping Cache Coherency Protocols," *Proc. 16th Ann. Int'l Symp. on Computer Architecture,* ACM, pp. 2-15, 1989b.

ESWARAN, K.P., GRAY, J.N., LORIE, J.N., y TRAIGER, I.L.: "The Notions of Consistency and Predicate Locks in a Database System," *Commun. of the ACM,* vol. 19, pp. 624-633, noviembre. 1976.

EVANS, A., KANTROWITZ, W., y WEISS, E.: "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Commun. of the ACM,* vol 17., pp. 437-442, agosto. 1974.

FERGUSON, D., YEMINI, Y., y NIKOLAOU, C.: "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems," *Proc. Eighth Int'l Conf. on Distributed Computing Systems,* IEEE, pp. 491-499,1988.

FIDGE, C.: "Logical Time in Distributed Computing Systems," *IEEE Computer,* vol. 24, pp. 28-33, agosto. 1991.

FISCHER, M., LYNCH, N., y PATERSON, M.: "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM,* vol. 32, pp. 374-382, abril. 1985.

FLEISCH, B., y POPEK, G.: "Mirage: A Coherent Distributed Shared Memory Design," *Proc. 12th Symp. on Operating Systems Principles,* ACM, pp. 211-223, 1989.

FLYNN, M.J.: "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers,* vol. C-21, pp. 948-960, septiembre. 1972.

FORIN, A., BARRERA, J., YOUNG, M., y RASHID, R.: "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," *Proc. Winter. 1989 USENIX Conf.,* USENIX, enero. 1989.

FREDRICKSON, N, y LYNCH, N.: "Electing a Leader in a Synchronous Ring," *Journal of the ACM,* vol. 34, pp. 98-115, enero. 1987.

GANTENBEIN, R.E.: "An Annotated Bibliography of Dependable Distributed Computing," *Operating Systems Review,* vol. 20, pp. 60-81, abril. 1992.

GARCÍA-MOLINA, H.: "Elections in a Distributed Computing System," *IEEE Trans. on Computers,* vol. 31, pp. 48-59, enero. 1982.

GARCÍA-MOLINA, H., y SPAUSTER, A: "Ordered and Reliable Multicast Communication," *ACM Trans. on Comp. Syst.,* vol. 9, pp. 242-271, agosto. 1991.

GELERNTER, D.: "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems,* vol. 7, pp. 80-112, enero. 1985.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., y HENNESSY, J.: "Memory Consistency and Event Ordering in Salable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. on Computer Architecture,* ACM, pp. 15-26, 1990.

GIEN, M.: "Micro-Kernel Architecture: Key to Modern Operating Systems Design," *UNIX Review,* pp. 10, noviembre. 1990

GIEN, M., y GROB, L.: "Microkernel Based Operating Systems: Moving UNIX onto Modern System Architectures," *Proc. UniForum'92 Conf.,* USENIX, pp. 43-55, 1992.

GIFFORD,D.K.: "Weighted Voting for Replicated Data," *Proc. Seventh Symp. on Operating Systems Principles,* ACM, pp. 150-162, 1979.

GOLUB, D., DEAN, R., FORIN, A., y RASHID, R.: "UNIX as an Application Program," *Proc. Summer 1990 USENIX Conf.,* USENIX, pp. 87-95, junio. 1990.

GOODMAN,J.R.: "Using Cache Memory to Reduce Processor Memory Traffic," *10th Ann. Int'l Symp. on Computer Architecture,* ACM, pp. 124-131, 1983.

GOODMAN, J.R.: "Cache Consistency and Sequential Consistency," Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GOPAL, I., GUERIN, R., JANNIELLO, J., y THEOHARAKIS, V.: "ATM Support in a Transparent Network," *IEEE Network,* vol. 6, pp. 62-68, noviembre. 1992.

GRAY, J.: "Notes on Database Operating Systems," en *Operating Systems: An Advanced Course,* R. Bayer, R.M. Graham, y G. Seegmuller (eds.), Berlin: Springer-Verlag, pp. 394 - 481,1978.

GRAY, C., y CHERITON, D.: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File System Consistency," *Proc. 11th Symp. on Operating Systems Principles,* ACM, pp. 202-210,1989.

GRAY, J.N., HOMAN, P., KORTH, H.F., y OBERMARCK, R.L.: "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System," Report RJ 3066, IBM Research Laboratory, San Jose CA, 1981.

GUSELLA, R., y ZATTI, S.: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD," *IEEE Trans. on Software Engineering,* vol. 15, pp. 847-853, julio. 1989.

HARTY, K., y CHERITON, D.: "Application-Controlled Physical Memory Using External Page-Cache Management," *Proc. Fifth ASPLOS Conf.*, ACM, pp. 187-199,1992.

HOARE, C.A.R.: "Monitors, An Operating System Structuring Concept," *Commun. of the ACM,* vol. 17, pp. 549-557, Octubre 1974; Erratum en *Commun. of the ACM,* vol. 18, p. 95, febrero. 1975.

HONG, D., y SUDA., T: "Congestion Control and Prevention in ATM Networks," *IEEE Network,* vol. 5, pp. 10-16, julio. 1991.

HOWARD, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N., y WEST, M.J.: "Scale and Performance in a Distributed File System," *ACM Trans. on Computer Systems,* vol. 6, pp. 55-81, febrero. 1988.

HUTCHINSON, N.C., PETERSON, L.L., ABBOTT, M.B., y O'MALLEY, S.: "RPC in the *x*-Kernel: Evaluating New Design Techniques," *Proc. 12th Symp. on Operating Systems Principles,* ACM, pp. 911-101, 1989.

HUTTO, P.W., y AHAMAD, M.: "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," *Proc. 10th Int'l Conf. on Distributed Computing Systems,* IEEE, pp. 302-311, 1990.

JONES, A.K., CHANSLER, R.J., Jr., DURHAM, I., FEILER, P., y SCHWANS, K.: "Software Management of CM*—A Distributed Multiprocessor," *Proc. NCC,* AFIPS, pp. 657-663, 1977.

JUL, E., LEVY, H., HUTCHINSON, N., y BLACK, A.: "Fine-Grained Mobility in the Emerald System," *ACM Trans. on Computer Systems,* vol. 6, pp. 109-133, febrero. 1988.

KAASHOEK, M.F., y TANENBAUM, A.S.: "Group Communication in the Amoeba Distributed Operating System," *Proc. 11th Int'l Conf. on Distributed Computing Systems,* IEEE, pp. 222-230,1991.

KAASHOEK, M.F., TANENBAUM, A.S., HUMMEL, S., y BAL, H.E.: "An Efficient Reliable Broadcast Protocol," *Operating Systems Review,* vol. 23, pp. 5-19, octubre. 1989.

KARLIN, A.R., LI, K., MANASSE, M.S., y OWICKI, S: "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proc. 13th Symp. on Operating Systems Principles,* ACM, pp. 41-55, 1991.

KAZAR, M.L., LEVERETT, B.W., ANDERSON, O.T., APOSTOLIDES, V., BOTTOS, B.A., CHUTANI, S., EVERHART, C.F., MASON, W.A., TU, S.-T., y ZAYAS, E.R.: "DEcorum File

System Architectural Overview," *Proc Summer 1990 USENIX Conf*, USENIX, pp 151-163, verano 1990

KELEHER, P , COX, A L , y ZWAENEPOEL, W : "Lazy Release Consistency," *Proc 19th Ann Int'l Symp on Computer Architecture,* ACM, pp 13-21, 1992

KLEIN, M H , LEHOCZKY, J P , y RAJKUMAR, R : "Rate-Monotonic Analysis for Real-Time Industrial Computing," *IEEE Computer,* vol 27, pp 24-33, enero 1994

KLEINROCK, L : *Queueing Systems* Vol 1, Nueva York John Wiley, 1974

KLEINROCK,L : "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Magazine,* vol 30, pp 36-40, abril 1992

KNAPP, E : "Deadlock Detection in Distributed Databases," *ACM Computing Surveys,* vol 19, pp 303-328, diciembre 1987

KOHL, J T : "The Evolution of the *Kerberos* Authentication Service," *Proc EurOpen Spring'91 Conf*, EurOpen, pp 295-313,1991

KOPETZ, H , DAMM, A , KOZA, C , MULAZZANI, M , SCHWABL, W , SENFT, C , y ZAINLINGER, R : "Distributed Fault-Tolerant Real-Time Systems The MARS Approach," *IEEE Micro,* vol 9, pp 25-40, febrero 1989

KOPETZ, H , y GRUNSTEIDL, G : "TTP—A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer,* vol 27, pp 14-23, enero 1994

KOPETZ, H , y OCHSENREITER, W : "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans on Computers,* vol C-36, pp 933-940, agosto 1987

KRANZ, D , JOHNSON, K , AGARWAL, A , KUBIATOWICZ, J J , y LIM, B : "Integrating Message Passing and Shared Memory Early Experiences," *Proc Fourth Symp on Principles and Practice of Parallel Programming,* ACM, pp 54-63, mayo 1993

KRISHNASWAMY, V : "A Language Based Architecture for Parallel Computing," Ph D Thesis, Yale Univ , 1991

KUNG, H T , y ROBINSON, J T : "On Optimistic Methods for Concurrency Control," *ACM Trans on Database Systems,* vol 6, pp 213-226, junio 1981

LAMPORT, L : "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun of the ACM,* vol 21, pp 558-564, julio 1978

LAMPORT, L : "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans on Computers,* vol C-28, pp 690-691, septiembre 1979

LAMPORT, L : "Concurrent Reading and Writing of Clocks," *ACM Trans on Computer Systems,* vol 8, pp 305-310, noviembre 1990

LAMPORT, L , SHOSTAK, R , y PEASE, M : "The Byzantine Generals Problem," *ACM Trans on Programming Languages and Systems,* vol 4, pp 382-401, julio 1982

LAMPSON, B W , ABADI, M , BURROWS, M , y WOBBER, E : "Authentication in Distributed Systems Theory and Practice," *ACM Trans on Computer Systems,* vol 10, pp 265-310, noviembre 1992

LAROWE, R P , y ELLIS, C S : "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Trans on Computer Systems,* vol 9, pp 319-363, noviembre 1991

LAROWE, R P , ELLIS, C S , y KAPLAN, L S : "The Robustness of NUMA Memory Management," *Proc 13th Symp on Operating Systems Principles,* ACM, pp 137-151, 1991

LE BOUDEC, J -Y : "The Asynchronous Transfer Mode A Tutorial" *Computer Networks and ISDN Systems,* vol 24, pp 279-309, abril 1992

LEA, R , AMARAL, P , y JACQUEMOT, C : "COOL-2 An Object-Orient Support Platform Built above the Chorus Microkernel," *Proc Int'l Workshop on Object-Oriented Systems,* pp 51-55, 1991

LEA, R , JACQUEMOT, C , y PILLEVESSE, E : "COOL System Support for Distributed Programming," *Commun of the ACM,* vol 36, pp 37-46, septiembre 1993

LENOSKI, D , LAUDON, J , GHARACHORLOO, K , WEBER, W -D , GUPTA, A , HENNESSY, J , HOROWITZ, M , y LAM, M : "The Stanford Dash Multiprocessor," *IEEE Computer,* vol 25, pp 63-79, marzo 1992

LEVY, E , y SILBERSCHATZ, A : "Distributed File Systems Concepts and Examples" *Computing Surveys,* vol 22, pp 321-374, diciembre 1990

LI, K : "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph D Thesis, Yale Univ , 1986

LI, K , y HUDAK, P "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans on Computer Systems,* vol 7, pp 321-359, noviembre 1989

LILJA, D J : "Cache Coherence in Large-Scale Shared-Memory Multiprocessors Issues and Comparisons," *ACM Computing Surveys,* vol 25, pp 303-338, septiembre 1993

LIPTON, R J , y SANDBERG, J S : "Pram A Scalable Shared Memory," Tech Rep CS-TR-180-88, Princeton Univ , septiembre 1988

LISKOV, B : "Practical Uses of Synchronized Clocks in Distributed Systems," *Distributed Computing,* vol 6, pp 211-219, 1993

LITZKOW, M J , LIVNY, M , y MUTKA, M W : "Condor—A Hunter of Idle Workstations," *Proc Eighth Int'l Conf on Distributed Computing Systems,* IEEE, pp 104-111, 1988

LIU, C L , y LAYLAND, J W : "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM,* vol 20, pp 46-61, enero 1973

LO, V M : "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc Fourth Int'l Conf on Distributed Computing Systems,* IEEE, pp 30-39, 1984

LUAN, S -W , y GLIGOR, V D : "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Trans on Parallel and Distributed Systems,* vol 1, pp 271-285, julio 1990

LUNDELIUS-WELCH, J , y LYNCH, N : "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation,* vol 77, pp 1-36, enero 1988

LYLES, J B , y SWINEHART, D C : "The Emerging Gigabit Environment and the Role of Local ATM," *IEEE Communications Magazine,* vol 30, pp 52-58, abril 1992

MAEKAWA, M , OLDEHOEFT, A E , y OLDEHOEFT, R R : *Operating Systems Advanced Concepts,* Menlo Park, CA Benjamin/Cummings, 1987

MALCOLM, N , y ZHAO, W : "The Timed-Token Protocol for Real-Time Communication," *IEEE Computer,* vol 27, pp 35-41, enero 1994

MARSH, B D , SCOTT, M L , LeBLANC, T J , y MARKATOS, E P : "First-Class User-Level Threads," *Proc 13th Symp on Operating Systems Principles,* ACM, pp 110-121, 1991

MELIAR-SMITH, P M , MOSER, L E , y AGRAWALA, V : "Broadcast Protocols for Distributed Systems," *IEEE Trans on Parallel and Distributed Systems,* vol 1, pp 17-25, enero 1990

MINZER, S E : "Broadband ISDN and Asynchronous Transfer Mode (ATM)," *IEEE Communications Magazine,* vol 29, pp 17-24, septiembre 1989

MORRIS, J H , SATYANARAYANAN, M , CONNER, M H , HOWARD, J H , ROSENTHAL, D S , y SMITH, F D : "Andrew A Distributed Personal Computing Environment," *Commun of the ACM,* vol 29, pp 184-201, marzo 1986

MOSBERGER, D : "Memory Consistency Models," Tech, Report TR 93/11, Dept of Computer Science, Univ of Arizona, 1993

MULLENDER, S J (ed): *Distributed Systems,* segunda edición, Nueva York, NY ACM Press, 1993

MULLENDER, S J : "Interprocess Communication," in *Distributed Systems,* segunda edición, S Mullender (ed), Nueva York, NY ACM Press, pp 217-250, 1993

MULLENDER, S J , ROSSUM, G VAN, TANENBAUM, A S , RENESSE, R VAN, y STAVEREN, H VAN: "Amoeba A Distributed Operating System for the 1990s," *IEEE Computer,* vol 23, pp 44-53, mayo 1990

MULLENDER, S J , y TANENBAUM, A S : "Immediate Files," *Software—Practice and Experience,* vol 14, pp 365-368, abril 1984

MUTKA, M W , y LIVNY, M : "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network," *Proc Seventh Int'l Conf on Distributed Computing Systems,* IEEE, pp 2-9, 1987

NATARAJAN, S , y ZHAO, W : "Issues in Building Dynamic Real-Time Systems," *IEEE Software,* vol 9, pp 16-21, septiembre 1992

NAYFEH, B A , y OLUKOTUN, K : "Exploring the Design Space for a SharedCache Multiprocessor," *Proc 21st Ann Int'l Symp on Computer Architecture,* ACM, pp 166-175, 1994

NEEDHAM, R M , y SCHROEDER, M D : "Using Encryption for Authentication in Large Networks of Computers," *Commun of the ACM,* vol 21, pp 993-999, diciembre 1978

NELSON, B J : *Remote Procedure Call,* Ph D Thesis, Carnegie-Mellon Univ , 1981

NELSON, V P: "Fault-Tolerant Computing Fundamental Concepts," *IEEE Computer*, vol 23, pp 19-25, julio 1990

NEWMAN, P: "ATM Local Area Networks," *IEEE Communications Magazine*, vol 32, pp 86-98, marzo 1994

NICHOLS, D A: "Using Idle Workstations in a Shared Computing Environment," *Proc 11th Symp on Operating Systems Principles*, ACM, pp 5-12, 1987

NIKOLAIDIS, I, y ONVURAL, R O "A Bibliography on Performance Issues in ATM Networks," *Computer Communication Review*, vol 22, pp 8-23, octubre 1992

NITZBERG, B, y LO, V: "Distributed Shared Memory A Survey of Issues and Algorithms," *IEEE Computer*, vol 24, pp 52-60, agosto 1991

OSF: *Introduction to OSF DCE*, Englewood Cliffs, NJ Prentice Hall, 1992

OUSTERHOUT, J K: "Scheduling Techniques for Concurrent Systems," *Proc Third Int'l Conf on Distributed Computing Systems*, IEEE, pp 22-30, 1982

PANZIERI, F, y SHRIVASTAVA, S K: "Rajdoot a remote procedure call mechanism with orphan detection and killing," *IEEE Trans on Software Engineering*, vol 14, pp 30-37, enero 1988

PARNAS, D: "On the Criteria to Be Used in Decomposing Systems into Modules," *Commun of the ACM*, vol 15, pp 1053-1058, diciembre 1972

PARTRIDGE, C: "Protocols for High-Speed Networks Some Questions and a Few Answers," *Computer Networks and ISDN Systems*, vol 25, pp 1019-1028, septiembre 1993

PARTRIDGE, C: *Gigabit Networking*, Reading, MA Addison-Wesley, 1994

PATTAVINA, A: "Nonblocking Architectures for ATM Switching," *IEEE Communications Magazine*, vol 31, pp 38-48, febrero 1993

PEASE, M, SHOSTAK, R, y LAMPORT, L: "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol 27, pp 228-234, abril 1980

PRZYBYLSKI, M, HOROWITZ, J, y HENNESSY, J: "Performance Tradeoffs in Cache Design," *Proc 15th Ann Int'l Symp on Computer Architecture* ACM, pp 290-298, 1988

PU, C , NOE, J D , y PROUDFOOT, A : "Regeneration of Replicated Objects A Technique and its Eden Implementation," *Proc Second Int'l Conf on Data Engineering,* pp 175-187, febrero 1986

PURDIN, T D , SCHLICHTING, R D , y ANDREWS, G R : "A File Replication Facility for Berkeley UNIX," *Software--Practice and Experience,* vol 17, pp 923-940, diciembre 1987

RAMAMRITHAM, K , STANKOVIC, J A , y SHIAH, P -F : "Efficient Scheduling Algorithms and Real-Time Multiprocessor Systems," *IEEE Trans on Parallel and Distributed Systems,* vol 1, pp 184-194, abril 1990

RAMANATHAN, J , y NI, L M : "Critical Factors in NUMA Memory Management," *Proc 11th Int'l Conf on Distributed Computing Systems,* IEEE, pp 500-507, 1991

RAMANATHAN, P , KANDLUR, D D , y SHIN, K G : "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Trans on Computers,* vol C-39, pp 514-524, abril 1990a

RAMANATHAN, P , y SHIN, K.G : "Delivery of Time-Critical Messages Using a Multiple Copy Approach," *ACM Trans on Computer Systems,* vol 10, pp 144-166, mayo 1992

RAMANATHAN, P , SHIN, K G , y BUTLER, R W : "Fault-Tolerant Clock Synchronization in Distributed Systems," *IEEE Computer,* vol 23, pp 33-42, octubre 1990b

RASHID, R F : "Threads of a New System," *Unix Review,* vol 4, pp 37-49, agosto 1986a

RASHID, R F : "From RIG to Accent to Mach The Evolution of a Network Operating System," *Fall Joint Computer Conf,* AFIPS, pp 1128-1137, 1986b

RAYNAL, M : "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms," *Operating Systems Review,* vol 25, pp 47-50, abril 1991

REED, D P : "Implementing Atomic Actions on Decentralized Data," *ACM Trans on Computer Systems,* vol 1, pp 3-23, febrero 1983

RICART, G , y AGRAWALA, A K : "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Commun of the ACM,* vol 24, pp 9-17, enero 1981

ROOHOLAMINI, R , CHERKASSKY, V , y GARVER, M : "Finding the Right ATM Switch for the Market," *IEEE Computer,* vol 27, pp 16-28, abril 1994

ROSENBERRY, W , KENNEY, D , y FISHER, G : *Understanding DCE,* Sebastopol, CA O'-Reilly, 1992

ROZIER, M , ABROSSIMOV, V , ARMAND, F , BOULE, I , GIEN, M , GUILLEMONT M , HERRMANN, F , KAISER, C , LEONARD, P , LANGLOIS, S , y NEUHAUSER, W : "Chorus Distributed Operating Systems," *Computing Systems,* vol 1, pp 305-379, octubre 1988

SANDERS, B A : "The Information Structure of Distributed Mutual Exclusion," *ACM Trans on Computer Systems,* vol 5, pp 284-299, agosto 1987

SANSOM, R D , JULIN, D P , y RASHID, R F : "Extending a Capability Based System into a Network Environment," *Proc SIGCOMM '86,* ACM, pp 265-274, 1986

SATYANARAYANAN, M : "A Study of File Sizes and Functional Lifetimes," *Proc Eighth Symp on Operating Systems Principles,* ACM, pp 96-108, 1981

SATYANARAYANAN, M : "A Survey of Distributed File Systems,' *Annual Review of Computer Science,* vol 4, pp 73-104,1990a

SATYANARAYANAN, M : "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer,* vol 23, pp 9-21, mayo 1990b

SATYANARAYANAN, M : "Distributed File Systems," en *Distributed Systems,* segunda edición, *S* Mullender (ed), Nueva York, NY ACM Press, pp 353-383, 1993

SCHEURICH, C , y DUBOIS, M : "Correct Memory Operation of Cache-Based Multiprocessors," *Proc Fourth Ann Int'l Symp on Computer Architecture,* ACM, pp 234-24, 1987

SCHNEIDER, F B : " Implementing Fault-Tolerant Services Using the State Machine Approach," *ACM Computing Surveys,* vol 22, pp 299-319, diciembre 1990

SCHROEDER, M D , y BURROWS, M : "Performance of Firefly RPC," *ACM Trans on Computer Systems,* vol 8, pp 1-17, febrero 1990

SCHWAN, K , y ZHOU, H : "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads," *IEEE Trans on Software Engineering,* vol 18, pp 736-747, agosto 1992

SCOTT, M , LeBLANC, T , y MARSH, B : "Multi-model Parallel Programming in Psyche," *Proc, Second ACM Symp on Principles and Practice of Parallel Programming,* ACM, pp 70-78, 1990

SHIN, K : "HARTS À Distributed Real-Time Architecture," *IEEE Computer*, vol 24, pp 25-35, mayo 1991

SHIRLEY, J : *Guide to Writing DCE Applications*, Sebastopol, CA O'Reilly,

SHIVARATI, N G , KRUEGER, P , y SINGHAL, M : "Load Distributing for Locally Distributed Systems," *IEEE Computer*, vol 25, pp 33-44, diciembre 1992

SILBERSCHATZ, A , y GALVIN, P : *Operating System Concepts*, Reading, MA Addison-Wesley, 1994

SINGH, S , y KUROSE, J : "Electing 'Good' Leaders," *Journal of Parallel and Distributed Computing*, vol 21, pp 184-201, mayo 1994

SINGHAL, M : "Deadlock Detection in Distributed Systems," *IEEE Computer*, vol 22, pp 37-48, noviembre 1989

SRIKANTH, T K , y TOUEG, S : "Optimal Clock Synchronization," *Journal of the ACM*, vol 34, pp 626-645, julio 1987

STANKOVIC, J A : "Misconceptions about Real-Time Computing A Serious Problem for Next-Generation Systems," *IEEE Computer*, vol 21, pp 10-19, octubre 1988

STEINER, J G , NEUMAN, B C , y SCHILLER, J I : "Kerberos An Authentication Service for Open Network Systems," *Proc Invierno 1988 USENIX Conf*, USENIX, pp 191-202, febrero 1988

STONE, H S , y BOKHARI, S H : "Control of Distributed Processes," *IEEE Computer*, vol 11, pp 97-106, julio 1978

STUMM, M , y ZHOU, S : "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, vol 23, pp 54-64, mayo 1990

SUBRAMANIAN, I : "Managing Discardable Pages with an External Pager," *Proc Second USENIX Mach Symp*, USENIX, pp 77-86, 1991

SUZUKI, T : "ATM Adaptation Layer Protocol," *IEEE Communications Magazine*, vol 32, pp 80-83, abril 1994

SVOBODOVA, L : "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, vol 16, pp 353-398, diciembre 1984

TAM, M -C , SMITH, J M , y FARBER, D J : "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," *Operating Systems Review,* vol 24, pp 40-67, julio 1990

TANENBAUM, A S : *Computer Networks,* segunda edición, Englewood Cliffs, NJ Prentice Hall, 1988

TANENBAUM, A S , KAASHOEK, M F , y BAL, H E : "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer,* vol 25, 1992

TANENBAUM, A S , MULLENDER, S J , y VAN RENESSE, R : "Using Sparse Capabilities in a Distributed Operating System," *Proc Sixth Int'l Conf on Distributed Computing Systems,* IEEE, pp 558-563, 1986

TANENBAUM, A S ,VAN RENESSE,R ,STAVEREN, H VAN,SHARP, G J , MULLENDER, S J , JANSEN, J , y ROSSUM, G VAN: "Experiences with the Amoeba Distributed Operating System," *Commun of the ACM,* vol 33, pp 46-63, diciembre 1990

TAY, B H , y ANANDA, A L : "A Survey of Remote Procedure Calls," *Operating Systems Review,* vol 24, pp 68-79, julio 1990

THEIMER, M M , LANTZ, K A , y CHERITON, D A : "Preemptable Remote Execution Facilities in the V System," *Proc 10th Symp on Operating Systems Principles,* ACM, pp 2-12,1985

THEKKATH, R , y EGGERS, S J : "Impactof Sharing-Based Thread Placement on Multithreaded Architectures," *Proc 21st Ann Int'l Symp on Computer Architecture,* ACM, pp 176-186,1994

TRAJKOVIC, L , y GOLESTANI, S J : "Congestion Control for Multimedia Services," *IEEE Network,* vol 6, pp 20-26, septiembre 1992

TSEUNG, L N : "Guaranteed, Reliable, Secure Broadcast Networks," *IEEE Network,* vol 3, pp 33-37 noviembre 1989

TUREK, J , y SHASHA, D : "The Many Faces of Consensus in Distributed Systems," *IEEE Computer,* vol 25, pp 8-17, junio 1992

ULLMAN,J : "Complexity of Sequence Problems," en *Computers and Job/Shop Scheduling Theory,* E G Coffman (ed), Nueva York Wiley, 1976

VAN RENESSE, R , y TANENBAUM, A S : "Voting with Ghosts," *Proc Eighth Int'l Conf on Distributed Computer Systems,* IEEE, 1988

VAN TILBORG, A M , y WITTIE, L D : "Wave Scheduling Distributed Allocation of Task Forces in Network Computers," *Proc Sixth Int'l Conf on Distributed Computing Systems,* IEEE, pp 337-347, 1981

VASWANI, R , y ZAHORJAN, J : "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors," *Proc 13th Symp on Operating Systems Principles,* ACM, pp 26-40, 1991

VERISSIMO, P : "Real-Time Communication," in *Distributed Systems,* segunda edición, S Mullender (ed), Nueva York, NY ACM Press, pp 447-490,1993

VERNON, M K , LAZOWSKA, E D , y ZAHORJAN, J : "Snooping CacheConsistency Protocols," *Proc 15th Ann Int'l Symp on Computer Architecture,* ACM, pp 308-317, 1988

WEBER, W , y GUPTA, A : "Analysis of Cache Invalidation Patterns in Multiprocessors," *Proc Third ASPLOS Conf,* ACM, pp 243-256, 1989

WEIHL, W : "Transaction-Processing Techniques," in *Distributed Systems,* segunda edición, S Mullender (ed), Nueva York, NY ACM Press, pp 329-352,1993

WITTIE, L D , y VAN TILBORG, A M : "MICROS, a Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Trans on Computers,* vol C-29, pp 1133-1144, diciembre 1980

WOBBER, E , ABADI, M , BURROWS, M , y LAMPSON, B : "Authentication in the Taos Operating System," *ACM Trans on Computer Systems,* vol 12, pp 332, febrero 1994

WOO, T Y C , y LAM, S S : "Authentication for Distributed Systems," *IEEE Computer,* vol 25, pp 39-52, enero 1992

YOUNG, M , TEVANIAN, A Jr , RASHID, R , GOLUB, D , EPPINGER, J , CHEW, J , BOLOSKY, W , BLACK, D , y BARON, R : "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc 11th Symp on Operating Systems Principles,* ACM, pp 63-76, noviembre 1987

ZAYAS, E R : "Attacking the Process Migration Bottleneck," *Proc 11th Symp on Operating Systems Principles,* ACM, pp 13-24, 1987

ZEGURA, E W : "Architectures for ATM Switching Systems," *IEEE Communications Magazine,* vol 31, pp 28-37, febrero 1993

Índice

A

ABCAST, 112
Abortos en cascada, 155
Accent, 432
Acceso múltiple con división del tiempo, 231-234, 238-240
Acceso no uniforme a memoria, 13
ACL (*véase* lista de control de acceso)
Activaciones del planificador, 182-183
Actor en Chorus, 479
Actuador, 224
Acuerdo en sistemas tolerantes de fallas, 217-222
Administración de memoria
  Amoeba, 392-393
  Chorus, 490-495
  Mach, 445-457
Administración de objetos X/Open, 552
Administración de procesos
  Amoeba, 388-392
  Chorus, 483-490
  Mach, 436-445
Administración del cronómetro, 94-95
Administrador ACL, 562
Administrador de memoria externa
Mach, 452-457
Agente de directorio global, 545
Agente servidor de directorio, 553
Agente usuario de directorio, 553
Agregado, DCE, 566
AIL (*véase* Lenguaje de interfaz en Amoeba)
Algoritmo
  asignación migratoria, 198
  asignación no migratoria, 198
  copia primaria, 270
  de anillo, 143
  de Cristian, 128-130
  de distribución jerárquica, 206-208
  de elección, 140-143
  de remates, 209-210
  de teoría de gráficas, 204
  del grandulón, 141-142
  del primer límite en primer lugar, 236
  del voto, 270-271
  descendente, 205
  distribución de procesadores, 203-210
  espera-muerte, 164
  herida-espera, 165
  iniciado por el emisor, 208
  iniciado por el receptor, 208-209
  mínima laxitud, 237
  monótono de tasa, 236
  NUMA, 311
  planificación en tiempo real, 237-241

sincronización de relojes de Berkeley, 130
sincronización de relojes, 124-132
Algoritmo de anillo, 143
Algoritmo de asignación de procesadores, 203-210
Algoritmo de barrido, 95
Algoritmo de consistencia del caché, 265-268
Algoritmo de Cristian, 128-130
Algoritmo de mínima laxitud, 237
Algoritmo de paginación en un multiprocesador NUMA, 311
Algoritmo de sincronización de relojes de Berkeley, 130
Algoritmo de voto, 270-271
Algoritmo del grandulón, 141-142
Algoritmo del primer límite en primer lugar, 236
Algoritmo espera-muerte, 164
Algoritmo herida-espera, 165
Algoritmo monótono de tasa, 236
Algoritmos de asignación migratoria, 198
Algoritmos de asignación no migratoria, 198
Algoritmos para asignación de procesadores
aspectos de diseño, 199-201
centralizado, 206
de remates, 209-210
de teoría de gráficas, 204
descendente, 205
implantación, 201-203
iniciado por el emisor, 208
iniciado por el receptor, 208-209
jerárquico, 206-208
Alias, 551
Almacenamiento estable, 146-147
Ambiente de computación distribuida, 520-574
CDS, 545, 547-549
celda, 525-527
centro distribuidor, 548
cliente, 536-538
componentes, 522-525
DFS, 524, 564-573
DTS, 524, 540-544
empleado de tiempo, 543
empleado, 549
Episode, 564
GDA, 545
GDS, 545, 549-554
hilo, 527-535
historia, 520-521
Kerberos, 555
lista de control de acceso, 562-563
llamadas a hilo, 531-535
mútex, 530-531
nombres, 546-547
objetivos, 521-522
plantilla, 532
proveedor de tiempo, 544
registro de seguridad
RPC autenticada, 559-562
RPC, 535-540
seguridad, 554-563
servicio de directorio global, 545, 549-554
servicio de directorio, 524, 544-554
servicio de seguridad, 524
servicio de tiempo, 540-544
servicio distribuido de archivo, 524, 564-573
servicio distribuido de tiempo, 524, 540-544
servidor de conjuntos de archivos, 571-572
servidor de directorio celda, 545, 547-549
servidor de localización de conjuntos de archivos, 572
servidor de réplica, 572
servidor de tiempo, 543
servidor supervisor básico, 572-573
servidor, 536-538
Amoeba, 376-429
administración de la memoria, 392-393
administración de los procesos, 388-392
arquitectura del sistema, 378-380
buffer de historia, 402
comparación con Mach y Chorus, 510-517
comunicación en grupo, 398-407
comunicación, 393-415
hilo, 391-392
historia, 376-377
micronúcleo, 380-382
mútex, 391-392
objetivos, 377-378
objetos, 384-388
pila de procesadores, 379-380
posibilidades, 384-388
protocolo de transmisión, 400-407
protocolo FLIP, 407-415
puerto de envío, 397, 412-414
puerto de recepción, 397
puerto, 395

redes de área amplia, 414-415
RPC, 394-398
segmento, 382, 392-393
servicio, 380
servidor de archivos, 383, 415-420
servidor de arranque, 427
servidor de directorios, 383, 420-425
servidor de ejecución, 425-427
servidor de réplicas, 425
servidor Soap, 383
servidor TCP/IP, 427-428
servidores, 415-428
tolerancia de fallas, 405-407
variables glocales, 391
Anillo de fichas, 230-231
Archivo comprometido, 417
Archivo inmutable, 247, 255, 416
Archivo inmutable, 247, 255, 416
Archivo no comprometido, 417
ARPANET, 42
Asa de archivo en NFS, 273-274
Asa en RPC, 78
Asignación de procesadores, 197-210
Asignación heurística, 200
Asociador en Chorus, 491-492
AT&T, 432-433
ATM (*véase* Modo de transferencia asíncrona)
Atomicidad, 106-107
Atributo de archivo, 246
Atributos de archivo, 246
Autenticación, 554
Autenticador, 558-559
Automontaje en NFS, 274
Autorización, 555

B

Barrera, 328
BBN butterfly, 309
Bitácora de escritura anticipada, 152-153
Bitácora de escritura anticipada, 152-153
Bloque sombra, 151
Bloqueo de encabezado, 48
Bloqueo falso, 161
Bloqueo, 158-165
Boleto de seguridad, 558-559
Buffer de historia, 402
Bus, 293-294
Buzón, 63

C

Caché
husmeador, 294
multiprocesador, 305
Caché monitor, 11, 294
Calendario gregoriano, 541
Capa de adaptación sencilla y eficiente, 47
Capa de enlace de datos, 38-40
Capa de presentación, 41-42
Capa de red, 40
Capa de sesión, 41
Capa de transporte, 40-41
Capa física, 38
Capa FLIP, 411-412
CBCAST, 112-114
CDS (*véase*, servidor de directorio de celdas)
Celda, 43-44, DCE, 525-527
Centro distribuidor de información en DCE, 548
Cerradura de dos fases, 155
Cerradura estricta de dos fases, 155
Cerradura, 154-156
Certificado de atributo de privilegio, 557
Ciphertext, 555
Cliente, 17, 51
DCE, 536-538
Co-planificación, 211-212
Coherencia de memoria, 321
Comparación de los algoritmos de planificación, 240-241
Comparación entre Amoeba, Mach y Chorus, 510-517
Comparación entre las máquinas con memoria compartida, 312-314
Compuerta inteligente de Rochester, 431-432
Comunicación de grupo, 99-114
Amoeba, 398-407
aspectos del diseño, 101-109
Chorus, 496
ISIS, 110-114
Comunicación de tiempo real, 230-234
Comunicación punto a punto, 99
Comunicación, 515-516
Amoeba, 393-415
Chorus, 495-499

Mach, 457-471
punto a punto, 99
tiempo real, 230-234
uno-muchos, 99
Confiabilidad, 27-28
Conjunto de archivo en DCE, 566
Conjunto de procesadores
en Mach, 442
Conjunto de puertos en Mach, 459-460
Conmutación ATM, 47-49
Conmutador de cruceta, 12
Consistencia
causal, 321-322
entrada, 330-331, 353-354
estricta, 315-317
PRAM, 322-325
procesador, 324-325
resumen de modelos, 331-333
secuencial, 317-321
versión impaciente, 329
versión paciente, 329
versión, 327-330, 346-348
Consistencia causal, 321-322
Consistencia de entrada, 330-331, 353-354
Consistencia de la versión, 327-330, 346-348
Consistencia de PRAM, 322-325
Consistencia débil, 325-327
Consistencia del procesador, 324-325
Consistencia estricta, 315-317
Consistencia secuencial, 317-321
Consistencia, versión paciente, 329
Consistencia, versión impaciente, 329
Control de concurrencia optimista, 156
Control de concurrencia, 154-158
optimista, 156
Control de flujo, 87
COOL (*véase* capa orientada a objetos Chorus)
Copiado durante la escritura en Mach, 451
Criptografía, 555
Cronómetro, 120
Chorus, 475-518
abstracción del núcleo, 479-481
actor, 479
administración de memoria, 490-495
administración de procesos, 483-490
administrador de flujos, 504
administrador de memoria virtual, 482
administrador de objetos, 503-504
administrador de procesos, 502-503
administrador IPC, 504
asociador, 491-492
comparación con Amoeba y Mach, 510-517
comunicación entre procesos, 482
comunicación, 495-499
configurabilidad, 504-506
Chorus (continúa)
ejecutivo en tiempo real, Chorus, 482
emulación UNIX, 483, 499-506
estructura de sistema, 478-479
estructura del núcleo, 481-483
extensiones UNIX, 500-501
hilo, 479, 485, 486
historia, 476-477
identificador de protección, 484
implantación UNIX, 501-506
llamadas a administración de memoria, 493-495
llamadas a administración de procesos, 488-490
llamadas de comunicación, 498-499
manejo de excepciones, 487-488
mensaje, 495
micronúcleo, 478
minimensaje, 495
minipuerto, 495
nucleo, 478
objetivos, 477
orientado a objetos, 483
planificación, 486-487
posibilidades, 481
proceso de sistema, 479
proceso núcleo, 478
proceso usuario, 479
puerto, 480
región, 479, 490
registro de software, 486
segmento, 481, 490
subsistema, 479
supervisor, 481-483
tiempo real, 506

D

DARPA, 432
Dash, 303-307

DCE, (*véase* Ambiente de computación distribuida)
Demonio RPC, 538-539
Descriptor de proceso, 389
Despachador, 172
Desviación del reloj, 121
Detección de bloqueo, 159-163
DFS (*véase* DCE, Sistema distribuido de archivos)
Digitalizador de página, 311
Dirección FLIP, 409
Direccionamiento de grupo, 104-105
Directorio CDS, 547-548
Disco óptico, 280-281
Dispersión/asociación, 93
Disponibilidad, 27
Dispositivo de una escritura y varias lecturas, 280
Dispositivo WORM (*véase* Dispositivo de una escritura y varias lecturas)
DN (*véase* nombre distinguido)
DNS (*véase* Sistema de nombre de dominio)
DSA (*véase* Agente servidor de directorio)
DSM (*véase* Memoria compartida distribuida)
DSM basado en objetos, 356-371
DTS (*véase* DCE, Servicio de tiempo distribuido)
DUA (*véase* Agente usuario de directorio)

E

Ejecutivo de tiempo real en Chorus, 482
Elección de algoritmo, 140-143
Empleado de tiempo, 543
Emulación de UNIX
  en Chorus, 499-506
  en Mach, 471-472
Encabezado, 36
Enlace cliente/servidor, 538-539
Enlace de tiempo real, 231
Enlace de un cliente, 538-539
Enlace dinámico, 77-80
Enlace simbólico, 252
Enlazador, 78
Entubamiento controlado por la lectura, 97
Episode, 564
Errno, 176
Error de sobre-ejecución, 87
Escalabilidad, 29-31, 109-110, 282-283
Escritura mediante el caché, 11, 265-266
Espacio múltiple, 359-361
Espera ocupada, 180
Esquema, 551
Estación de trabajo sin disco, 186-187
Estación de trabajo, 186-188
  con disco, 186-188
  de origen, 191
  inactiva, uso de, 189-193
  sin disco, 186-188
Estaciones de trabajo inactivas, 189-193
Ethernet, 230
Evento aperiódico, 224
Evento esporádico, 224
Evento periódico, 224
Eventos concurrentes, 112, 122
Eventos relacionados de manera causal, 112
Excepción, 81
Exclusión mutua en el anillo de fichas, 138-139
Exclusión mutua, 134-140
  comparación de métodos, 139-140
Exploración, 162
Extensión de archivo, 248
Exterminio de huérfanos, 84

F

Falla
  Bizantina, 214
  intermitente, 212
  permanente, 212
  transitoria, 212
Falla
  del cliente, 83-84
  del servidor, 82-83
Falla bizantina, 214
Falla de un componente, 212-213
Falla del sistema, 213-214
Falla Fail-stop, 214
Falla intermitente, 212
Falla permanente, 212
Falla silente, 213-214
Falla transitoria, 212
Fallas del cliente, 83-84
Fallas del servidor, 82-83
Fallas en RPC, 80-84
Fantasmas 271

FLIP (*véase* Protocolo Internet local rápido)
Forma canónica, 75

G

GBCAST, 112
GDA (*véase* Agente de directorio global)
GDS (*véase* Servicio de directorio global)
Generador de interfaz de Mach, 436
Grupo
  abierto, 101-102
  cerrado, 101-102
  de compañeros, 102-103
  jerárquico, 102-103
  sobrepuestos, 109
Grupo abierto, 101-102
Grupo cerrado, 101-102
Grupo de compañeros, 102-103
Grupo de puertos en Chorus, 496
Grupo jerárquico, 102-103

H

Hilo de aparición instantánea, 185
Hilo, 169-185
  Amoeba, 391-392
  Chorus, 479, Chorus, 485-486
  DCE, 527-535
  de aparición instantánea, 185
  espacio del usuario, 178-181
  interacción con RPC, 184-185
  Mach, 439-442
  núcleo, 181-182
Hilos en C, 440-442
Hipercubo, 14
Huérfano, 83-84
  expiración, 84
  exterminación, 84
  reencarnación suave, 84
  reencarnación, 84

I

Identificador de protección
  en Chorus, 484
Identificador local en Chorus, 480
Identificador único de usuario, 554
Identificador único en Chorus, 480
IDL (*véase* Lenguaje de definición de interfaz)
Imagen de un solo sistema, 19
Imperio Bizantino, 214
Incapacitar, 390
Independencia de lugar, 251
Interconexión de sistemas abiertos, 35-42
Interfaz 36
IP (*véase* Protocolo Internet)
ISIS, 110-114

J

Jacket, 180
Jerarquía digital síncrona, 44

K

Kerberos (*véase* Seguridad en DCE)

L

LAN (*véase* Red de área local)
Lecciones aprendidas, 278-279
Lectura anticipada, 277
Lenguaje de definición de interfaz, 537
Lenguaje de interfaz de Amoeba, 415
LI (*véase* Identificador local en Chorus)
Linda, 358-365
  espacio multiple, 359-361
  implantación, 361-365
  modelo del trabajador duplicado, 360-361
  plantilla, 360
Lista de control de acceso, 247, 555, 562-563
Lista de intenciones, 152
Lista de posibilidades en Mach, 460-462
Llamada a un procedimiento remoto (RPC), 68-98, 88-92
  administración del cronómetro, 94-95
  algoritmo de barrido, 95
  Amoeba, 394-398
  áreas de problemas, 95-98
  asa, 78
  costo excesivo del copiado, 92-94
  Chorus, 47
  DCE, 535-540
  enlace, 77-80
  implantación, 84-98
  interacción con hilos, 184, 185

operación básica, 68-72
ordenamiento, 72
paso de parámetros, 72-77
protocolos, 85-86
ruta crítica, 90-92
semántica, 80-84
Llamada por copiado/restauración, 70

M

Mach, 431-473
administración de memoria, 445-457
administración de procesos, 436-445
administrador de memoria externa, 452-457
cola de mensajes, 458
comparación con Amoeba y Chorus, 510-517
comunicación, 457-471
conjunto de procesadores, 442
conjunto de puertos, 459-460
copiado durante la escritura, 451
datos fuera de línea, 468
emulación de UNIX, 471-472
formatos de mensaje, 466-469
hilo, 439-442
hilos en C, 440-442
historia, 431-433
lista de posibilidades, 460-462
mecanismo de trampolín, 471
memoria compartida distribuida, 456-457
memoria compartida, 449-452
memoria virtual, 446-449
nombre de posibilidad, 461
objetivos, 433
objeto de memoria, 435, 447
paginador externo, 446
planificación de hilos, 442-445
planificación manos fuera, 445
posibilidad, 435, 460-463
primitivas de mensaje, 464-466
puerto de control, 452-453
puerto de nombre, 453
puerto de objeto, 452
puerto de proceso, 460
puerto de red, 469
puerto, 435, 457-460, 463-464
región, 447
servidor de mensajes de la red, 469-471
servidor de UNIX, 435-436
Malla de conmutadores, 47
Manejo de excepciones en Chorus, 487-488
Manejo de interrupciones en Chorus, 488
Máquina big endian, 74
Máquina little endian, 73-74
Marca de reloj, 121
Marca de tiempo, 156-68
Marco SONET, 45
Marcos, 38
MARS, 232
MD5, (*véase* Message Digest 5)
Mecanismo de trampolín en Mach, 471
Membresía de grupo, 103-104
Memnet, 298-301
Memoria coherente, 11
Memoria compartida distribuida, 289-373
basado en objetos, 356-371
basado en página, 333-345
búsqueda de las páginas, 342-343
búsqueda del propietario de página, 339-342
comparación de métodos, 371-372
Chorus, 492
diseño, 334
falsamente compartida, 336-337
granularidad, 335-336
Mach, 456-457
Munin, 346-353
reemplazo de página, 343-344
réplica, 334-335
secuencialmente consistente, 337-339
sincronización, 344-345
variable compartida, 345-355
Memoria compartida, 292-314
Memoria falsamente compartida, 336-337
Memoria virtual,
en Chorus, 490-495
en Mach, 446-449
Mensaje en Chorus, 495
Mensaje no solicitado, 267
Mensajes perdidos, 81-82
Message Digest 5, 559
Método de la máquina de estados, 215
Método, 292, 356, 366
Micronúcleo
Amoeba, 380-382
Chorus, 478
Mach, 433-435
MICROS, 206-208

Midway, 353-355
consistencia de entrada, 353-354
implantación, 355
MIG (*véase* Generador de interfaz de Mach)
Minimensaje en Chorus, 495
Minipuerto en Chorus, 495
Minitel, 29
MiX en Chorus, 483
Modelo carga/descarga, 247
Modelo cliente/servidor, 50-68
direccionamiento, 56-58
ejemplo, 52-55
implantación, 65-68
Modelo de acceso remoto, 248
Modelo de asignación de procesadores, 197-199
Modelo de conjunto de trabajo, 119-120
Modelo de entubamiento, 173
Modelo de equipo, 183
Modelo de estación de trabajo, 186-189
Modelo de memoria, 514-515
Modelo de seguridad, 555-557
Modelo del trabajador duplicado, 360-361
Modelo OSI (*véase* Interconexión de sistemas abiertos)
Modelos de consistencia, 315-333
Modo de transferencia asíncrona, 42-50
MSF, 127
Multicomputadora, 8
con base en un bus, 13-14
con conmutador, 14-15
Multiprocesador basado en anillo, 298-301
Multiprocesador de directorios, 303-305
Multiprocesador Firefly, 90
Multiprocesador NUMA, 308-311
Multiprocesador UMA, 308
Multiprocesador, 8-13
con base en un anillo, 298-301
con base en un bus, 10-12, 293-298
con base en un directorio, 303-305
con conmutador, 12-13, 301-307
con tiempo compartido, 20-22
NUMA, 308-311
UMA, 308
Multitransmisión, 100
Munin, 346-353
consistencia de la versión, 346-348
directorios, 351-352
protocolos, 348-351
sincronización, 353
Mútex, 175, 391-392
rápido, 530
recursivo, 530-531

N

NFS (*véase* Sistema de archivos en red)
NIS (*véase* Servicio de información de red)
Nivel ATM, 45-46
Nivel de adaptación ATM, 46-47
Nivel de aplicación, 42
Nivel físico ATM, 44-45
Nivel orientado a objetos de Chorus, 507-510
espacio de contexto, 508-509
implantación del espacio de contexto, 510
nivel base, 507-509
sistema de tiempo de ejecución del espacio de contexto, 509-510
unidad de asignación, 508
Nodo-a, 568
Nodo-r, 277
Nodo-v, 276
Nombre
binario, 252
DCE, 546-547
simbólico, 252
Nombre de posibilidades en Mach, 461
Nombre distinguido relativo, 550-551
Nombre distinguido, 551
Nombres binarios, 252
Nombres de dos niveles, 252
Nombres simbólicos, 252
Nomenclatura a dos niveles, 252
NORMA (*véase* Sistema sin acceso remoto)
Núcleo de Chorus, 478
NUMA (*véase* Acceso no uniforme a memoria)

O

Objeto, 292, 356, 366, 512-513
Amoeba, 384-388
operaciones, 387-388
protección, 385-387
OC-1, 45
Ocultamiento de archivos, 262-268
algoritmo de escritura a través de, 265-266

algoritmo de escritura al cierre, 266
Ocultamiento de escritura al cierre, 266
Ocultamiento de información, 356
Open Software Foundation, 432
Operación, 366
Operaciones concurrentes, 322
Orca, 365-371
administración de objetos, 368-371
lenguaje, 366-368
Ordenamiento de mensajes, 107-108
Ordenamiento de parámetros, 72
Ordenamiento según el tiempo
consistente, 108
global, 108
OSF (*véase* Open Software Foundation)
OSF DCE (*véase* Ambiente de cómputo distribuido)

P

PAC (*véase* Certificado de atributo de privilegio)
Página congelada, 311
Página preciosa, 454
Paginador externo
Chorus, 491
Mach, 446
Palabra particular, 559
Papa Gregorio, 541
Paquete de hilos, 174-178
Paquete localizador, 57
Paralelismo de grano fino, 29
Paralelismo de grano grueso, 29
Parámetro de llamada por referencia, 69
Parámetro de llamada por valor, 69
Pila de procesadores, 193-197, 379-380
Pila de protocolo, 38
Planificación de hilos en Mach, 442-445
Planificación de tiempo real, 234-241
dinámica, 236-237, 240-241
estática, 237-241
mínima laxitud, 237
monótona de tasa, 236
primer límite en primer lugar, 236
Planificación dinámica de tiempo real, 236-237, 240-241
Planificación estática de tiempo real, 237-241
Planificación, 210-212
Chorus, 486-487
DCE, 529-530
Mach, 442-445
manos fuera, 445
tiempo real dinámico, 236-237, 240-241
tiempo real estático, 237-241
tiempo real, 234-241
Planificaciones, 149
Plantilla
DCE, 532
Política de localización, 201
Política de transferencia, 200
Pollo, 36
Posibilidades, 247
Amoeba, 384-388
Chorus, 481
Mach, 435, 460-463
Predicado de direccionamiento, 105
Préstamo, 133
Prevención de bloqueo, 163-165
Primitivas (*véase* Primitivas de comunicación)
Primitivas de comunicación, 58-65
asíncrona, 59-61
bloqueo, 58-59
buzón, 63
confiable, 63-65
en buffer, 61-63
en grupo, 105-106
no confiable, 63-65
no contenida en buffer, 61-63
sin bloqueo, 59-61
síncrona, 58-59
Problema de los dos ejércitos, 219-220
Problemas generales bizantinos, 220-222
Proceso ligero (*véase* hilo)
Proceso, 513-514
Amoeba, 388-391
Chorus, 484-485
Mach, 436
Programa de tiempo real, 223
Propiedades ACID, 148
Protocolo activado por el tiempo, 232-234
Protocolo de actualización, 270-272
Protocolo de compromiso de dos fases, 153-154
Protocolo de consistencia del caché, 294
escribir a través de, 295

escribir una vez, 296
Protocolo de control de transmisión, 41
Protocolo de escritura a través del caché, 295
Protocolo de repetición selectiva, 87
Protocolo de transmisión en Amoeba, 400-407
Protocolo de una escritura, 296
Protocolo Internet local rápido, 407-415
interfaz, 409-411
Protocolo Internet, 40
Protocolo solicitud/réplica, 52
Protocolo universal de datagrama, 41
Protocolo, 35
activado por el tiempo, 232-234
compromiso de dos fases, 153-154
consistencia del caché, 294-298
de chorro, 86
detenerse y esperar, 86
multiprocesador, 305-307
NFS, 273-275
orientado a conexiones, 36
repetición selectiva, 87
RPC, 85-86
sin conexión, 36
sistemas distribuidos, 408
solicitud/réplica, 52
TCP/IP, 40-41
transmisión en Amoeba, 400-407
Protocolos por capas, 35-42
Proveedor del tiempo, 544
Puerto
Amoeba, 395
arranque de Mach, 436
control de Mach, 452-453
Chorus, 480, 495-496
excepción de Mach, 436
hilo, 439
Mach, 435, 457-460, 463-464
nombre de Mach, 453
objeto en Mach, 452
proceso en Mach, 460
red Mach, 469
registrado en Mach, 437
Puerto de arranque en Mach, 436
Puerto de envío en Amoeba, 397, 412-414
Puerto de excepción en Mach, 436
Puerto de recepción, Amoeba, 397
Puerto registrado en Mach, 437
Punto extremo en DCE, 538

Q

Quórum de escritura, 271
Quórum de lectura, 271
Quórum, 271

R

RAM con entubamiento (*véase* consistencia de PRAM)
RDN (*véase* Nombre distinguido relativo)
Recepción implícita, 185
Reconocimiento a cuestas, 401
Reconocimiento, 86-88
Red de área amplia, 2, 283-284
Red de área local, 1
Red de conmutación ATM, 47
Red omega, 12
Red óptica síncrona, 44-45
Redundancia modular triple, 215-217
Redundancia, 214-215
Región
Chorus, 479, 490
Mach, 447
Relación de ocurrencia anterior, 122
Reloj
físico, 124-127
lógico, 120-124
Reloj de cesio, 125
Reloj físico, 124-127
Reloj lógico, 120-124
Relojes sincronizados, 132-133
Réplica
activa, 215-217
paciente, 269
sistema de archivo, 268-270
Réplica activa, 215-217
Réplica de copia primaria, 270
Réplica paciente, 269, 425
Resguardo
del cliente, 70
del servidor, 70
Resguardo del cliente, 70
Resguardo del servidor, 70
Retroalimentación, 152
RFS (*véase* Sistema remoto de archivos)
RPC (*véase* Llamada a procedimiento remoto)
RPC autenticada, 559-562

Ruta crítica en RPC, 88-92
Ruteo de seguimiento de túneles, 305
Ruteo, 40

S

SDH (*véase* Jerarquía digital síncrona)
SEAL (*véase* Capa de adaptación sencilla y eficiente)
Sección amarilla, 275
Secuenciador, 369
Segmento
  Amoeba, 382, 392-393
  asociado, 393
  Chorus, 481, 490
Segundo de salto, 126
Segundo solar medio, 125
Segundo solar, 124
Seguridad
  componentes, 557-558
  DCE, 554-563
Seguridad principal, 554
Semántica
  a lo más una vez, 83
  al menos una vez, 83
  de archivos compartidos, 253-256
  de sesión, 253-254
  exactamente una vez, 83
  RPC, 80-84
  UNIX, 253-254
Semántica a lo más una vez, 83, 132-133
Semántica al menos una vez, 83
Semántica de archivos en UNIX, 253-254
Semántica de compartición de archivos, 253-256
Semántica de exactamente una vez, 83
Semántica de sesión, 253-254
Sensor, 224
Servicio de archivo, 245
  interfaz, 246-248
Servicio de directorio de celdas, 545, 547-549
Servicio de directorio global, 545, 549-554
Servicio de información de red, 275
Servicio de seguridad
  DCE, 555
Servicio de tiempo en DCE, 540-544
Servidor CDS, 549
Servidor de archivo, 17, 245
  viñeta, 415-420
Servidor de archivos
  Amoeba, 383, 415-420
  implantación, 418-120
Servidor de arranque en Amoeba, 427
Servidor de autenticación, 557
Servidor de conjunto de archivos, 571-572
Servidor de directorios
  Amoeba, 383, 420-425
  implantación, 423-425
Servidor de directorios de Amoeba, 383
Servidor de directorios X/Open, 552
Servidor de ejecución en Amoeba, 425-427
Servidor de grupo, 103
Servidor de localización de conjuntos de archivos, 572
Servidor de mensajes de red en Mach, 469-471
Servidor de nombres, 58
Servidor de privilegios, 557
Servidor de registro, 557
Servidor de réplica, 416
  Amoeba, 425
  DCE, 572
Servidor del tiempo, 543
Servidor para boletos, 557
Servidor sin estado, 274-275
Servidor SOAP (*véase* Servidor de directorios en Amoeba)
Servidor supervisor básico, 572
Servidor TCP/IP
  Amoeba, 427-428
Servidor, 51, 516-517
  Amoeba, 382-384, 415-428
  DCE, 536-538
Sincronización
  de relojes, 119-133, 124-132
  memoria compartida distribuida, 344-345
  Munin, 353
Sincronización de hilos en DCE, 530-531
Sincronización de relojes, 119-133, 226
Sistema abierto, 35
Sistema activado por el tiempo, 227
Sistema activado por eventos, 226
Sistema asegurado contra fallas, 229
Sistema asíncrono, 214
Sistema centralizado, 2
Sistema con un solo procesador, 2
Sistema de acceso, 557

Sistema de archivo, 245-286
algoritmos de réplica, 270-272
con estado, 260-262
diseño, 246-256
estructura, 258-262
implantación, 256-279
jerárquico, 248
lecciones aprendidas, 278-279
NFS, 272-278
ocultamiento, 262-268
réplicas, 268-270
sin estado, 260-262
tendencias, 279-285
tolerancia de fallas, 284-285
Sistema de archivos distribuidos (*véase* Sistema de archivo)
Sistema de archivos en red, 272-278
arquitectura, 272-273
implantación, 275-278
NIS, 275
nodo-r, 277
nodo-v, 276
protocolo, 273-275
sección amarilla, 275
Sistema de archivos sin estado, 260-262
Sistema de nombre de dominio, 545
Sistema de tiempo real
activado por el tiempo, 227
activado por eventos, 226
asegurado contra fallas, 229
Chorus, 506
diseño, 226-230
duro, 225
mitos, 225-226
predecible, 227-228
soporte de lenguaje, 229-230
suave, 225
tolerante de fallas, 228-229
Sistema de tolerancia de fallas en tiempo real, 228-229
Sistema distribuido
Amoeba, 376-429
aspectos de diseño, 22-31
Chorus, 475-518
DCE, 520-574
definición, 2
desventajas, 6-8
Mach, 430-473
objetivos, 3-8
software, 15-22
ventajas, 3-6
Sistema fuertemente acoplado, 9-10
Sistema operativo de red, 16-18
Sistema planificable, 236
Sistema remoto de archivos, 275
Sistema síncrono virtual, 111
Sistema síncrono, 214
Sistema vagamente acoplado, 9-10
Sistema vagamente síncrono, 111
Sistemas de colas, 194-196
Sistemas distribuidos de tiempo real, 223-241
Sistemas sin acceso remoto, 333
Skulking, 549
SONET (*véase* Red óptica síncrona)
Spoofing, 562
St Exupéry, Antoine de, 511
Suma de verificación, 38
Supervisor en Chorus, 481-483

T

TAI (*véase* Tiempo atómico internacional)
Tamaño de grano, 28-29
Tasa de flujo en el reloj, 128
Tasa de respuesta, 199
TCP (*véase* Protocolo de control de transmisión)
TDMA (*véase* Acceso múltiple con división del tiempo)
Tendencias en los sistemas de archivos, 279-285
Teoría de la relatividad de Einstein, 316
Texto plano, 555
Tiempo atómico internacional, 125
Tiempo de coordenadas universales, 126, 543-544
Tiempo de respuesta, 198
Tiempo medio de falla, 231
TMR (*véase* Redundancia modular triple)
Tolerancia de fallas con respaldo primario, 217-219
Tolerancia de fallas, 28, 212-222
Amoeba, 405-407
respaldo primario, 217-219
sistema de archivo, 284-285
TP0, 41

Trabajo cooperativo apoyado en una computadora, 4-5
Transacción atómica, 144-158
Transacción, 255-256
  anidada, 149-150
  atómica, 144-158
  implantación, 150-154
Transacciones anidadas, 149-150
Transmisión atómica, 106-107, 217
Transmisión confiable, 400-407
Transmisión, 100
Transparencia de la réplica, 268-269
Transparencia de localización, 251
Transparencia de nombres, 251
Transparencia, 22-25
  concurrencia, 24
  localización, 23, 251
  migración, 23
  nombre, 251
  paralelismo, 24
  réplica, 24, 268-269

U

UDP (*véase* Protocolo universal de datagrama)
UID (*véase* Identificador único en Chorus)
Uniprocesador virtual, 19
Unitransmisión, 100
Upcall, 183
Uso de archivo, 256-258
Usuarios móviles, 284
UTC (*véase* Tiempo coordenado universal)
UUID (*véase* Identificador único de usuario)

V

Variable de condición, 175-176, 530
Variable de sincronización, 325
Variable glocal, 391
Variable protegida, 328
Voto con fantasmas, 271

W

WAN (*véase* Red de área amplia)
WWV, 126-127

X

X 25, 40
XDS (*véase* Servidor de directorios X/Open)
XOM (*véase* Administración de objetos X/Open)

Sistemas Operativos Distribuidos

Andrew S. Tanenbaum

Conforme los sistemas distribuidos de computadoras cada vez son más penetrantes, aumenta la necesidad por comprender cómo se diseñan e implantan sus sistemas operativos.
Sistemas Operativos Distribuidos de Andrew S. Tanenbaum satisface esta necesidad. Representa una segunda parte revisada y muy amplia del best-seller *Sistemas operativos modernos*, y abarca el material del libro original, incluyendo la comunicación, la sincronización, los procesos y los sistemas de archivos; agrega nuevo material relativo a la memoria compartida distribuida, los sistemas distribuidos de tiempo real, los sistemas distribuidos toleranteS de fallas y las redes ATM. También contiene cuatro casos detallados: Amoeba, Mach, Chorus y OSF/DCE.
Esta obra de Tanenbaum provee a los lectores un tratamiento completo y conciso de los sistemas distribuidos.

Otras obras del autor disponibles en Prentice-Hall

-***Redes de ordenadores***, *Segunda edición*

-***Sistemas operativos modernos***

-***Sistemas operativos: Diseño e implantación***

-***Organización de computadoras. Un enfoque estructurado.*** *Tercera edición*

ISBN-968-880-627-7
90000
9 789688 806272

PHH
PRENTICE HALL