acadèmica

Concurrencia y sistemas distribuidos

Francisco Daniel Muñoz Escoí | Estefanía Argente Villaplana
Agustín Rafael Espinosa Minguet | Pablo Galdámez Saiz
Ana García-Fornes | Rubén de Juan Marín | Juan Salvador Sendra Roig

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CONCURRENCIA Y SISTEMAS DISTRIBUIDOS

Francisco Daniel Muñoz Escoí Estefanía Argente Villaplana
Agustín Rafael Espinosa Minguet Pablo Galdámez Saiz
Ana García-Fornes Rubén de Juan Marín
Juan Salvador Sendra Roig

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Colección Académica

Para referenciar esta publicación, utilice la siguiente cita: MUÑOZ ESCOÍ, F.D. [et al] (2012). *Concurrencia y sistemas distribuidos*. Valencia: Universitat Politècnica

Primera edición 2012 (versión impresa)
Primera edición 2013 (versión electrónica)

Distribución: pedidos@editorial.upv.es
Tel. 96 387 70 12 / www.editorial.upv.es / Ref. editorial: 6084

ISBN: 978-84-8363-986-3 (versión impresa)
ISBN: 978-84-9048-002-1 (versión electrónica)

Prólogo

Este libro va dirigido a los estudiantes de la asignatura de *Concurrencia y Sistemas Distribuidos* de la titulación del Grado en Ingeniería Informática de la Universitat Politècnica de València (UPV), aunque se ha estructurado de forma que cualquier persona interesada en aspectos de programación concurrente y sistemas distribuidos pueda obtener provecho de su lectura.

Los autores de este libro somos profesores de la citada asignatura y, a la hora de confeccionar una relación bibliográfica útil para nuestro alumnado, nos dimos cuenta que en el mercado existe un gran número de libros que abordan aspectos tales como concurrencia, sincronización, sistemas distribuidos, sistemas de tiempo real, administración de sistemas, pero no encontramos ningún libro que tratase todos esos contenidos en conjunto. Este libro pretende, por tanto, dar una visión integradora de las aplicaciones concurrentes y los sistemas distribuidos, ofreciendo también una primera aproximación a las tareas de administración de sistemas (necesarias en sistemas distribuidos).

Conviene destacar que se ha empleado el lenguaje Java para detallar ejemplos de programas concurrentes. La elección de dicho lenguaje de programación se ha debido a las ventajas que aporta para la implementación de programas concurrentes y a su elevado grado de utilización, tanto a nivel profesional como académico. Por ello, se ha incluido un capítulo específico sobre las utilidades que ofrece Java para la implementación de programas concurrentes.

Grado en Ingeniería Informática

El título de Graduado en Ingeniería Informática por la UPV sustituye a los títulos de Ingeniero en Informática, Ingeniero Técnico en Informática de Gestión e Ingeniero Técnico en Informática de Sistemas, como resultado de la adaptación de dichas titulaciones al marco del Espacio Europeo de Educación Superior. Desde enero de 2010 cuenta con la evaluación favorable por parte de la ANECA para su implantación, iniciada en septiembre de 2010.

El Plan de Estudios del título de Grado en Ingeniería Informática de la UPV está organizado en cuatro cursos de 60 ECTS cada uno. Cada ECTS supone 10 horas de docencia presencial y entre 15 y 20 horas para el resto del trabajo del alumno, incluida la evaluación. El Plan de Estudios se estructura en módulos y materias. Cada materia se descompone, a su vez, en una o más asignaturas con tamaños de 4.5, 6 ó 9 ECTS.

Materia de Sistemas Operativos

Dentro del Plan de Estudios del Grado de Ingeniería Informática de la UPV, la asignatura *Concurrencia y Sistemas Distribuidos* pertenece al módulo de Materias Obligatorias y, dentro de dicho módulo, a la materia de *Sistemas Operativos*. Dicha materia comprende el estudio de las características, funcionalidades y estructura de los sistemas operativos así como el estudio de los sistemas distribuidos.

El sistema operativo es un programa que actúa de interfaz entre los usuarios y el hardware del computador, abstrayendo los componentes del sistema informático. De este modo oculta la complejidad del sistema a los usuarios y aplicaciones, y se encarga de la gestión de los recursos, maximizando el rendimiento e incrementado su productividad. Por tanto, el Sistema Operativo se encarga de la gestión de los recursos hardware (CPU, memoria, almacenamiento secundario, dispositivos de entrada/salida, sistema de ficheros), y de controlar la ejecución de los procesos de usuario que necesitan acceder a los recursos hardware.

Por su parte, un sistema distribuido es una colección de ordenadores independientes que ofrecen a sus usuarios la imagen de un sistema coherente único. De forma análoga a como actúa un sistema operativo en una máquina, los sistemas distribuidos se deben encargar de ocultar al usuario las diferencias entre las máquinas y la complejidad de los mecanismos de comunicación. Los sistemas distribuidos deben facilitar el acceso de los usuarios a los recursos remotos, dar trasparencia de distribución (ocultando el hecho de que los procesos y recursos están físicamente distribuidos en diferentes ordenadores), ofrecer estándares para facilitar la interoperabilidad y portabilidad de las aplicaciones, así como ofrecer escalabilidad.

En el Plan de Estudios del Grado de Ingeniería Informática de la UPV, la materia *Sistemas Operativos* es de carácter obligatorio, con una asignación de 12 créditos ECTS y se imparte en segundo curso. Las asignaturas que componen esta materia son dos: Fundamentos de Sistemas Operativos, de 6 ECTS, que se imparte en el primer cuatrimestre de segundo curso; y Concurrencia y Sistemas Distribuidos, de 6 ECTS, que se imparte en el segundo cuatrimestre de ese mismo curso.

En la asignatura de *Fundamentos de Sistemas Operativos* (FSO) se introduce el concepto de sistema operativo, su evolución histórica, así como el funcionamiento y los servicios que proporcionan, detallando la gestión de los procesos, la gestión

de la memoria y la gestión de entrada/salida y ficheros. También se introducen los conceptos de sistemas de tiempo real y la programación de sistemas, detallando el concepto de llamada a sistema y la programación básica de sistemas.

Por su parte, la asignatura de *Concurrencia y Sistemas Distribuidos* (CSD) se centra en describir los principios y técnicas básicas de la programación concurrente, así como las características relevantes de los sistemas distribuidos. De este modo, la asignatura de Concurrencia y Sistemas Distribuidos permitirá que el alumno identifique y resuelva los problemas planteados por la ejecución de múltiples actividades concurrentes en una misma aplicación informática. Tales problemas aparecen a la hora de acceder a recursos compartidos y pueden generar resultados o estados inconsistentes en tales recursos. Para evitarlos se deben emplear ciertos mecanismos de sincronización que el alumno aprenderá a utilizar de manera adecuada. Además, en los sistemas de tiempo real aparecen otras restricciones sobre el uso de estos mecanismos que el alumno identificará y gestionará correctamente.

Por otro lado, el alumno conocerá las diferencias que comporta el diseño y desarrollo de una aplicación distribuida, así como los mecanismos necesarios para proporcionar diferentes tipos de transparencia (replicación, concurrencia, fallos, ubicación, migración, etc.) en este tipo de aplicaciones. Utilizará el modelo cliente/servidor para estructurar estas aplicaciones, por las ventajas que ello comporta a la hora de gestionar los recursos. Conocerá las ventajas que aporta el diseño de algoritmos distribuidos descentralizados, tanto por lo que respecta a la gestión de los fallos como a la escalabilidad de las aplicaciones resultantes. Por último, conocerá las limitaciones existentes a la hora de resolver problemas de sincronización en entornos distribuidos, requiriendo el uso de una ordenación lógica de los eventos en la mayoría de los casos.

Además, se iniciará al alumno en las tareas de administración de sistemas, utilizando el Directorio Activo de Windows Server para este fin.

Ampliación de los contenidos de Sistemas Operativos

Los contenidos de la materia *Sistemas Operativos* se amplían en otras asignaturas en 3^{er} y 4^{o} curso: Tecnologías de los Sistemas de Información en la Red (obligatoria), Administración de Sistemas (del Módulo de Tecnologías de la Información), Diseño de Sistemas Operativos, y Diseño y Aplicaciones de los Sistemas Distribuidos (ambas del Módulo de Ingeniería de Computadores).

Tecnologías de los Sistemas de Información en la Red (obligatoria de 3^{er} curso) introduce las tecnologías y estándares existentes para diseñar, desarrollar, desplegar y utilizar aplicaciones distribuidas. Con ello se extenderá la formación recibida en CSD sobre sistemas distribuidos, desarrollando aplicaciones distribuidas de cierta

complejidad, utilizando el modelo cliente/servidor e identificando diferentes aproximaciones para implantar cada uno de los componentes de estas aplicaciones.

Administración de Sistemas (de 3^{er} curso) desarrolla conceptos y habilidades relacionadas con las tareas de administración de sistemas más habituales en las organizaciones actuales. Entre ellas, se incluyen las habilidades de: instalación, configuración y mantenimiento de sistemas operativos; la gestión de servicios (impresión, ficheros, protocolos); el diseño, planificación e implantación de una política de sistemas (versiones, funcionalidades, licencias, etc.); la administración de dominios de sistema (usuarios, grupos, seguridad); el mantenimiento de copias de respaldo y recuperación ante desastres; la automatización (con el uso de scripts, políticas, etc.); y la formación y soporte al usuario.

Diseño de Sistemas Operativos (de 3^{er} curso) introduce los aspectos de implementación y diseño de sistemas operativos. Se trata de una asignatura fundamentalmente práctica, en las que se trabajará con un sistema operativo real, en concreto con el sistema operativo Linux, viendo con detenimiento los mecanismos que ofrece su núcleo. De este modo, se analizarán las llamadas al sistema que ofrece Linux, la gestión de interrupciones hardware y la gestión de procesos. La comprensión del funcionamiento del sistema operativo permitirá al alumno tomar decisiones de diseño (de aplicaciones) que optimicen el uso de los recursos del computador.

Por último, *Diseño y Aplicaciones de los Sistemas Distribuidos* (de 4º curso) profundiza en los aspectos de comunicación en sistemas distribuidos (modelos cliente/servidor, modelos orientados a objetos, modelos de grupos), en los conceptos de seguridad, así como en las tecnologías para la integración de aplicaciones en la web y los aspectos básicos de diseño de aplicaciones distribuidas, tales como replicación y caching.

Conocimientos recomendados

Respecto a los conocimientos recomendados para el seguimiento adecuado de la asignatura de *Concurrencia y Sistemas Distribuidos*, la primera parte de sus unidades didácticas se centra en los problemas de concurrencia y sincronización en un modelo de memoria compartida. Para ello, los alumnos deberán hacer uso de los conceptos de hilo y proceso estudiados en *Fundamentos de Sistemas Operativos*.

Por otro lado, para la realización de las prácticas, así como para el estudio, comprensión y resolución de los problemas y casos de estudio, es recomendable que los alumnos tengan un aceptable nivel de programación utilizando Java. Dicho nivel se debe haber alcanzado en las asignaturas de programación de primer curso (*Introducción a la Informática y a la Programación*; *Programación*; y *Fundamentos de Computadores*) y la correspondiente de primer cuatrimestre de segundo curso (*Lenguajes, Tecnologías y Paradigmas de la Programación*).

Índice general

Índice de figuras

Índice de tablas

Unidad 1

PROGRAMACIÓN CONCURRENTE

1.1 Introducción

Esta unidad didáctica presenta el concepto de *programación concurrente*, proporcionando su definición informal en la sección 1.2. La sección 1.3 discute las principales ventajas e inconvenientes que ofrece este tipo de programación cuando es comparada frente a la programación secuencial. Además, se presentan algunos ejemplos de aplicaciones que típicamente serán concurrentes y se discute por qué resulta ventajoso programarlas de esa manera. Por último, la sección 1.4 ofrece una primera introducción a los mecanismos incorporados en el lenguaje *Java* para soportar la programación concurrente.

1.2 Definición

Un *programa secuencial* [Dij71] es aquel en el que sus instrucciones se van ejecutando en serie, una tras otra, desde su inicio hasta el final. Por tanto, en un programa secuencial solo existe una única actividad o hilo de ejecución. Es la forma tradicional de programar, pues los primeros ordenadores solo tenían un procesador y los primeros sistemas operativos no proporcionaron soporte para múltiples actividades dentro de un mismo proceso.

Por el contrario, en un *programa concurrente* [Dij68] se llegarán a crear múltiples actividades que progresarán de manera simultánea. Cada una de esas actividades será secuencial pero ahora el programa no avanzará siguiendo una única secuencia,

pues cada actividad podrá seguir una serie de instrucciones distinta y todas las actividades avanzan de manera simultánea (o, al menos, esa es la imagen que ofrecen al usuario). Para que en un determinado proceso haya múltiples actividades, cada una de esas actividades estará soportada por un *hilo de ejecución*. Para que un conjunto de actividades constituya un programa concurrente, todas ellas deben cooperar entre sí para realizar una tarea común.

Para ilustrar el concepto de programa o proceso concurrente, podríamos citar a un procesador de textos. Existen varios de ellos actualmente (MS Word, LibreOffice Writer, Apple Pages, ...). La mayoría son capaces de realizar múltiples tareas en segundo plano, mientras se va escribiendo (para ello se necesita que haya un hilo de ejecución que se dedique a cada una de esas acciones, mientras otro atienda a la entrada proporcionada por el usuario). Algunos ejemplos de tareas realizadas por estos hilos son: revisión gramatical del contenido del documento, grabación del contenido en disco de manera periódica para evitar las pérdidas de texto en caso de corte involuntario de alimentación, actualización de la ventana en la que se muestra el texto introducido, etc. Como se observa en este ejemplo, en este programa (formado por un solo fichero ejecutable) existen múltiples hilos de ejecución y todos ellos cooperan entre sí y comparten la información manejada (el documento que estemos editando).

Presentemos seguidamente un ejemplo de una ejecución no concurrente. Asumamos que en una sesión de laboratorio abrimos varias consolas de texto en Linux para realizar las acciones que nos pida algún enunciado y probarlas. En este segundo caso, seguiríamos utilizando un único programa (un intérprete de órdenes; p. ej., el `bash`) y tendríamos múltiples actividades (es decir, habría varias actividades ejecutando ese único programa), pero esas actividades no colaborarían entre sí. Cada consola abierta la estaríamos utilizando para lanzar una serie distinta de órdenes y las órdenes lanzadas desde diferentes consolas no interactuarían entre sí. Serían múltiples instancias de un mismo programa en ejecución, generando un conjunto de procesos **independientes**. Se ejecutarían simultáneamente, pero cada uno de ellos no dependería de los demás. Eso no es una aplicación concurrente: **para que haya concurrencia debe haber cooperación entre las actividades**. Algo similar ocurriría si en lugar de lanzar múltiples consolas hubiésemos lanzado múltiples instancias del navegador. Cada ventana abierta la estaríamos utilizando para acceder a una página distinta y no tendría por qué existir ningún tipo de interacción o cooperación entre todos los procesos navegadores generados.

En esta sección se ha asumido hasta el momento que para generar una aplicación informática basta con un único programa. Esto no tiene por qué ser siempre así. De hecho, algunas aplicaciones pueden estructurarse como un conjunto de componentes y cada uno de esos componentes puede estar implantado mediante un programa distinto. En general, cuando hablemos de una *aplicación concurrente* asumiremos que está compuesta por un conjunto de actividades que cooperan entre sí para realizar una tarea común. Estas actividades podrán ser: múltiples hilos

en un mismo proceso, múltiples procesos en una misma máquina o incluso múltiples procesos en máquinas distintas; pero en todos los casos existirá cooperación entre las actividades. Por el contrario, en una *aplicación secuencial* únicamente existirá un solo proceso y en tal proceso solo habrá un único hilo de ejecución.

Para soportar la concurrencia se tendrá que ejecutar ese conjunto de actividades de manera simultánea o paralela. Para ello existen dos mecanismos:

- *Paralelismo real*: Se dará cuando tengamos disponibles múltiples procesadores, de manera que ubicaremos a cada actividad en un procesador diferente. En las arquitecturas modernas, los procesadores pueden tener múltiples núcleos. En ese caso, basta con asignar un núcleo diferente a cada actividad. Todas ellas podrán avanzar simultáneamente sin ningún problema.

  Si solo disponemos de ordenadores con un solo procesador y un único núcleo, podremos todavía lograr la concurrencia real utilizando múltiples ordenadores. En ese caso la aplicación concurrente estará formada por múltiples procesos y estos cooperarán entre sí intercambiando mensajes para comunicarse.

- *Paralelismo lógico*: Como los procesadores actuales son rápidos y las operaciones de E/S suspenden temporalmente el avance del hilo de ejecución que las haya programado, se puede intercalar la ejecución de múltiples actividades y ofrecer la imagen de que éstas progresan simultáneamente. Así, mientras se sirve una operación de E/S, el procesador queda libre y se puede elegir a otra actividad preparada para ejecutarla durante ese intervalo. Este es el principio seguido para proporcionar *multiprogramación* (técnica de multiplexación que permite la ejecución simultánea de múltiples procesos en un único procesador, de manera que parece que todos los procesos se están ejecutando a la vez, aunque hay un único proceso en el procesador en cada instante de tiempo). Por ello, el hecho de que únicamente se disponga de un solo procesador en un determinado ordenador no supone ninguna restricción que impida implantar aplicaciones concurrentes.

  El usuario será incapaz de distinguir entre un sistema informático con un solo procesador y un solo núcleo de otro que tenga múltiples procesadores, pues los sistemas operativos ocultan estos detalles. En cualquiera de los dos casos, el usuario percibe la imagen de que múltiples actividades avanzan simultáneamente.

Ambos tipos de paralelismo pueden combinarse sin ninguna dificultad. Por ello, si en un determinado ordenador tenemos un procesador con dos núcleos, el número de actividades que podremos soportar en una aplicación concurrente que ejecutemos en él no tendrá por qué limitarse a dos. Al combinar el paralelismo real con el lógico podremos utilizar tantos hilos como sea necesario en esa aplicación.

1.3 Aplicaciones concurrentes

Esta sección estudia con mayor detenimiento algunos aspectos complementarios de la programación concurrente, como son sus principales ventajas e inconvenientes, así como otros ejemplos de aplicaciones, tanto reales como académicos. Estos últimos ilustran algunas de las dificultades que encontraremos a la hora de desarrollar una aplicación concurrente.

1.3.1 Ventajas e inconvenientes

La programación concurrente proporciona las siguientes ventajas:

- *Eficiencia*: El hecho de disponer de múltiples actividades dentro de la aplicación permite que ésta pueda concluir su trabajo de manera más rápida. El hecho de que alguna actividad se suspenda al utilizar (p.ej., al acceder al disco para leer parte de un fichero) o al solicitar (p.ej., al utilizar la operación P() sobre un semáforo) algún recurso no suspende a toda la aplicación y con ello esta última podrá seguir avanzando. En una aplicación concurrente resulta sencillo que se utilicen múltiples recursos simultáneamente (ficheros, semáforos, memoria, procesador, red,...).

- *Escalabilidad*: Si una determinada aplicación puede diseñarse e implantarse como un conjunto de componentes que interactúen y colaboren entre sí, generando al menos una actividad por cada componente, se facilitará la distribución de la aplicación sobre diferentes ordenadores. Para ello bastaría con instalar cada componente en un ordenador distinto, generando una *aplicación distribuida*. De esta manera, la cantidad de recursos de cómputo que se podrán utilizar simultáneamente se incrementará. Además, si el número de usuarios de esa aplicación también se incrementara significativamente, se podrían utilizar técnicas de *replicación* [Sch90, BMST92], incrementando su capacidad de servicio. Así mejora la escalabilidad de una aplicación.

  En general, se dice que un *sistema* es *escalable* [Bon00] cuando es capaz de gestionar adecuadamente cargas crecientes de trabajo o cuando tiene la capacidad para crecer y adaptarse a tales cargas. En el caso de las aplicaciones concurrentes ambas interpretaciones se pueden cumplir combinando la distribución y replicación de componentes.

- *Gestión de las comunicaciones*: El uso eficiente de los recursos, ya comentado en la primera ventaja citada en este apartado, permite que aquellos recursos relacionados con la comunicación entre actividades sean explotados de manera sencilla en una aplicación concurrente. Si la comunicación está basada en el intercambio de mensajes a través de la red, esta comunicación no implicará que toda la aplicación se detenga esperando alguna respuesta. Por ello, el uso de mecanismos sincrónicos de comunicación entre actividades

se puede integrar de manera natural en una aplicación concurrente. En la Unidad 8 se detallarán diferentes tipos de mecanismos de comunicación.

- *Flexibilidad*: Las aplicaciones concurrentes suelen utilizar un diseño modular y estructurado, en el que se presta especial atención a qué es lo que debe hacer cada actividad, qué recursos requerirá y qué módulos del programa necesitará ejecutar. Por ello, si los requisitos de la aplicación cambiasen, resultaría relativamente sencillo identificar qué módulos tendrían que ser adaptados para incorporar esas variaciones en la especificación y a qué actividades implicarían. Como resultado de este diseño estructurado y cuidadoso, se generarán aplicaciones flexibles y adaptables.

- *Menor hueco semántico*: Un buen número de aplicaciones informáticas requieren el uso de varias actividades simultáneas (por ejemplo, en un videojuego suele utilizarse un hilo por cada elemento móvil que haya en la pantalla; en una hoja de cálculo tenemos un hilo para gestionar la entrada de datos, otro para recálculo, otro para la gestión de los menús, otro para la actualización de las celdas visibles, ...). Si estas aplicaciones se implantan como programas concurrentes resulta sencillo su diseño. Por el contrario, si se intentara implantarlas como un solo programa secuencial la coordinación entre esas diferentes funcionalidades resultaría difícil.

No obstante, la programación concurrente también presenta algunos inconvenientes que conviene tener en cuenta:

- *Programación delicada*: Durante el desarrollo de aplicaciones concurrentes pueden llegar a surgir algunos problemas. El más importante se conoce como "*condición de carrera*" (a estudiar en la unidad 2) y puede generar inconsistencias imprevistas en el valor de las variables o atributos compartidos entre las actividades, cuando éstas los modifiquen. Un segundo problema son los *interbloqueos* [CES71], que se analizarán en la Unidad 4. Por ello, deben conocerse los problemas potenciales que entraña la programación concurrente y deben tomarse las precauciones oportunas para evitar que aparezcan.

- *Depuración compleja*: Una aplicación concurrente, al estar compuesta por múltiples actividades, puede intercalar de diferentes maneras en cada ejecución las sentencias que ejecuten cada una de sus actividades. Por ello, aunque se proporcionen las mismas entradas a la aplicación, los resultados que ésta genere pueden llegar a ser distintos en diferentes ejecuciones. Si alguno de estos resultados fuese incorrecto, resultaría bastante difícil la reproducción de esa ejecución. Además, como para depurar cualquier aplicación informática se suelen utilizar herramientas especializadas (diferentes tipos de depuradores), la propia gestión del depurador podría evitar que se diera la traza que provocó ese error. Esto ilustra que las aplicaciones concurrentes no tienen una depuración sencilla y que los mecanismos empleados en sus depuradores no siempre serán idénticos a los utilizados sobre aplicaciones secuenciales.

A pesar de estos inconvenientes existe un interés creciente en el desarrollo de aplicaciones concurrentes. Una buena parte de las aplicaciones actuales se estructuran en múltiples componentes que pueden ser desplegados en ordenadores diferentes, es decir, son aplicaciones concurrentes distribuidas cuyos componentes necesitan intercomunicarse a través de la red. También están apareciendo múltiples dispositivos móviles con procesadores aceptables y con discos duros o memorias flash con suficiente capacidad de almacenamiento. Estos elementos ("*tablets*", "*netbooks*", "*ultrabooks*", teléfonos inteligentes,...) interactúan tanto con su entorno como con otros dispositivos móviles y es habitual que utilicen múltiples actividades en sus aplicaciones, tal como sugiere un modelo de programación concurrente. A su vez, no es raro que los procesadores utilizados tanto en los ordenadores personales tradicionales como en ese nuevo conjunto de ordenadores "ligeros" dispongan de múltiples núcleos. Con ello, hoy en día no supone ningún inconveniente la ejecución paralela de múltiples actividades, tanto de manera lógica como real.

1.3.2 Aplicaciones reales

Pasemos a ver seguidamente algunos ejemplos de aplicaciones concurrentes reales. Estos ejemplos ilustran los conceptos presentados hasta el momento en esta unidad, pero también proporcionan la base para entender algunos de los aspectos analizados en las unidades posteriores en las que presentamos los problemas que plantea la programación concurrente así como los mecanismos necesarios para solucionarlos:

1. *Programa de control del acelerador lineal Therac-25* [LT93]. Los aceleradores lineales se utilizan en los tratamientos de radioterapia para algunos tipos de cáncer, eliminando mediante ciertos clases de radiación (radiación de electrones, rayos X y rayos gamma, principalmente) a las células cancerígenas. El Therac-25 fue un acelerador lineal desarrollado entre 1976 y 1982, tomando como base los diseños de otros aceleradores más antiguos (el Therac-6 y el Therac-20). En el Therac-25 se podían utilizar dos tipos de radiación (de electrones y gamma) abarcando un amplio intervalo de intensidades. En los modelos anteriores, el acelerador disponía de sistemas de control por hardware que evitaban cualquier tipo de disfunción (parando el acelerador en caso de error). En el Therac-25 esa gestión se integró en el software.

   El tratamiento se realizaba en una sala aislada. El operador del sistema configuraba adecuadamente el acelerador una vez el paciente estuviese preparado. Para ello se debía especificar el tipo de radiación a utilizar, la potencia a emplear, la duración de la sesión, etc. El sistema tardaba algunos segundos en responder a esa configuración. Una vez transcurría ese tiempo, empezaba el tratamiento. El programa de control se encargaba de monitorizar todos los componentes del sistema: posición y orientación del acelerador, potencia de la radiación, posición de los escudos de atenuación (a utilizar en los tipos de

radiación más intensa), etc. Si se detectaba algún error leve (como pueda ser un ligero desenfoque), el sistema paraba y se avisaba al operador que podía reanudar el tratamiento una vez corregida la situación de error. En caso de error grave (por ejemplo, una sobredosis de radiación) el sistema paraba por completo y debía ser reiniciado.

Este programa de control era una aplicación concurrente, compuesta por múltiples actividades, gestionando cada una de ella un conjunto diferente de elementos del sistema. Entre las actividades más importantes cabría distinguir:

- *Gestión de entrada*. Esta actividad monitorizaba la consola y recogía toda la información de configuración establecida por el operador. Se dejaba indicado en una variable global si el operador había finalizado su introducción de datos.
- *Gestión de los imanes*. Los componentes internos necesarios para la generación de la radiación necesitaban aproximadamente ocho segundos para posicionarse y aceptar la nueva especificación del tipo de radiación y su intensidad. Como el Therac-25 admitía dos tipos de radiación, en uno de ellos se utilizaban escudos de atenuación y en el otro no.
- *Pausa*. Existía una tercera actividad que suspendía la gestión de entrada mientras se estuviese ejecutando la gestión de los imanes. Con ello se intentaba que la gestión de los imanes pudiera finalizar lo antes posible. Se suponía que cualquier intento de reconfigurar el equipo durante esta pausa sería atendido cuando la pausa finalizara, justo antes de iniciar el tratamiento, obligando en ese punto a reiniciar la gestión de los imanes.

Lamentablemente, existía un error de programación en la aplicación concurrente y, además de suspender temporalmente la gestión de entrada, cuando finalizaba la actividad de "*pausa*" no se llegaba a consultar si el operador había solicitado una modificación de la configuración del sistema. Así, el operador observaba que la nueva configuración aparecía en pantalla (y suponía que había sido aceptada), pero el sistema seguía teniendo sus imanes adaptados a la configuración inicialmente introducida. Se había dado una "*condición de carrera*" y el estado real del sistema no concordaba con lo que aparecía en la pantalla del operador. Esto llegó a tener consecuencias fatídicas en algunos tratamientos (en ellos se llegó a dar la radiación de mayor intensidad sin la presencia de los escudos de atenuación): entre 1985 y 1987 se dieron seis accidentes de este tipo, causando tres muertes [LT93].

Este es uno de los ejemplos más relevantes de los problemas que puede llegar a causar una aplicación concurrente con errores en su diseño.

2. *Servidor web Apache* [Apa12]. Apache es un servidor web que emplea programación concurrente. El objetivo de un servidor web es la gestión de cierto conjunto de páginas web ubicadas en ese servidor. Dichas páginas resultan accesibles utilizando el protocolo HTTP (o *Hypertext Transfer Protocol*

[FGM+99]). Los navegadores deben realizar peticiones a estos servidores para obtener el contenido de las páginas que muestran en sus ventanas.

En un servidor de este tipo interesa tener múltiples hilos de ejecución, de manera que cada uno de ellos pueda atender a un cliente distinto, paralelizando así la gestión de múltiples clientes. Para que la propia gestión de los hilos no suponga un alto esfuerzo, Apache utiliza un conjunto de hilos (o "*thread pool*") en espera, coordinados por un hilo adicional que atiende inicialmente las peticiones que vayan llegando, asociando cada una de ellas a los hilos del conjunto que queden disponibles. Al tener ese conjunto de hilos ya generados y listos para su asignación se reduce el tiempo necesario para iniciar el servicio de cada petición. Si el número de peticiones recibidas durante cierto intervalo de tiempo supera el tamaño de ese conjunto, las peticiones que no pueden servirse de inmediato se mantienen en una cola de entrada. El objetivo de este modelo de gestión es la reducción del tiempo necesario para gestionar a los hilos que sirvan las peticiones recibidas. De hecho, los hilos ya están creados antes de iniciar la atención de las peticiones y no se destruyen cuando finalizan la atención de cada petición, sino que vuelven al "*pool*" y pueden ser reutilizados posteriormente. Con ello, en lugar de crear hilos al recibir las peticiones o destruirlos al finalizar su servicio (dos operaciones que requieren bastante tiempo) basta con asignarlos o devolverlos al "pool" (operaciones mucho más rápidas que las anteriores).

3. *Videojuegos.* La mayoría de los videojuegos actuales (tanto para ordenadores personales como para algunas consolas) se estructuran como aplicaciones concurrentes compuestas por múltiples hilos de ejecución. Para ello se suele utilizar un *motor de videojuegos* que se encarga de proporcionar cierto soporte básico para la renderización, la detección de colisiones, el sonido, las animaciones, etc. Los motores permiten que múltiples hilos se encarguen de estas facetas. Así, el rendimiento de los videojuegos puede aumentar en caso de disponer de un equipo cuyo procesador disponga de múltiples núcleos.

   Estos motores también facilitan la portabilidad del juego, pues proporcionan una interfaz que no depende del procesador ni de la tarjeta gráfica. Si el motor ha llegado a implantarse sobre múltiples plataformas, los juegos desarrollados con él se pueden migrar a ellas sin excesivas dificultades. Por ejemplo, la detección de colisiones en un motor de videojuegos suele ser responsabilidad de un componente llamado *motor de física* ("*physics engine*"). En el juego FIFA de Electronic Arts, este motor (que en este caso se llama "*Player Impact Engine*") se renovó en la segunda mitad de 2011. El motor de videojuegos del que forma parte ha facilitado su portabilidad a algunas de las plataformas en las que FIFA 12 está disponible: Xbox 360, Wii, Nintendo 3DS, PC, PlayStation 3, PlayStation 2, PlayStation Portable, PlayStation Vita, Mac, iPhone, iPad, iPod y Android.

4. *Navegador web.* Cuando Google presentó su navegador *Chrome* (en diciembre de 2008), su arquitectura [McC08] era muy distinta a la del resto de navegadores web (Microsoft Internet Explorer, Mozilla Firefox, Opera, ...). En Chrome, cada pestaña abierta está soportada por un proceso independiente. De esa manera, si alguna de las pestañas falla, el resto de ellas puede seguir sin problemas. Para mejorar su rendimiento se optimizó su soporte para Javascript, compilando y manteniendo el código generado en lugar de interpretarlo cada vez. Además, en cada pestaña se utilizan múltiples hilos para obtener cada uno de los elementos de la página que deba visualizarse. Con ello, la carga de una página podía realizarse de manera mucho más rápida que en otros navegadores. Actualmente la mayor parte de los navegadores web han adoptado una arquitectura similar a la de Chrome, dadas las ventajas en rendimiento y robustez que proporciona.

1.3.3 Problemas clásicos

Existe una serie de problemas clásicos (o académicos) de la programación concurrente que ilustran una serie de situaciones en las que múltiples actividades deben seguir ciertas reglas para gestionar un recurso compartido de manera adecuada.

Aunque se incluyan en esta sección, los ejemplos que describiremos seguidamente se utilizarán en las próximas unidades para ilustrar algunos de los problemas que se plantean a la hora de desarrollar una aplicación concurrente. Por ello, no es necesario que se estudien con detenimiento dentro de esta primera unidad didáctica, pero sí que se revisen más tarde en una segunda lectura al analizar esas unidades didácticas posteriores.

Productor-Consumidor con buffer acotado

En este problema [Hoa74] se asume que existen dos tipos de actividades o procesos: los productores y los consumidores. La interacción entre ambos roles se lleva a cabo utilizando un *buffer* de capacidad limitada, en el que los procesos *productores* irán depositando elementos y de donde los procesos *consumidores* los extraerán. Este buffer gestiona sus elementos con una estrategia *FIFO* ("*First In, First Out*"), es decir, sus elementos se extraerán en el orden en que fueron insertados.

Para que el uso del buffer sea correcto, tendrá que respetar una serie de restricciones que discutiremos en función del ejemplo mostrado en la figura 1.1. En esta figura se observa un buffer con capacidad para siete elementos, construido como un vector cuyas componentes se identifican con los números naturales entre 0 y 6, ambos inclusive. Para gestionar este buffer utilizaríamos (en cualquier lenguaje de programación orientado a objetos) estos atributos:

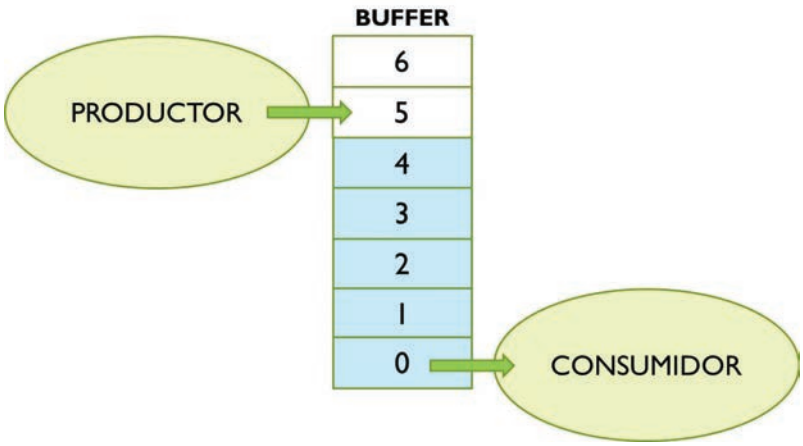

Figura 1.1: Uso de un buffer de capacidad limitada.

- `in`: Indicaría la componente en la que algún proceso productor insertaría el próximo elemento en el buffer. En el estado mostrado en la figura 1.1, su valor sería 5.
- `out`: Indicaría qué componente mantiene el elemento que algún consumidor extraerá a continuación. En la figura 1.1 su valor sería 0.
- `size`: Mantendría el tamaño del buffer. En este ejemplo sería 7.
- `numElems`: Mantendría el número actual de elementos en el buffer. En la figura 1.1 su valor sería 5.
- `store`: Vector en el que se almacenan los elementos del buffer.

Para que un productor inserte un elemento en el buffer, se tendrá que seguir un algoritmo similar al siguiente:

1. Insertar el elemento en la posición `in` del vector `store`.
2. Incrementar el valor de `in` de manera circular. Es decir: `in` ← `(in + 1) MOD size`.
3. Incrementar el número de elementos en el buffer. Es decir: `numElems++`.

Por su parte, para que un consumidor extraiga un elemento del buffer, se utilizaría este algoritmo:

1. Retornar el elemento ubicado en la posición `out` del vector `store`.

2. Incrementar el valor de `out` de manera circular. Es decir: `out` ← `(out + 1) MOD size`.

3. Decrementar el número de elementos en el buffer. Es decir: `numElems--`.

Estos algoritmos básicos podrían ocasionar problemas al ser ejecutados de manera concurrente. Veamos algunos ejemplos:

- Si en la situación mostrada en la figura 1.1 dos procesos productores iniciaran a la vez la inserción de un nuevo elemento en el buffer, ambos observarían en el primer paso de su algoritmo el mismo valor para el atributo `in`. A causa de ello, los dos insertarían sus elementos en la posición 5 y así el segundo elemento insertado sobrescribiría al primero. Posteriormente, ambos incrementarían el valor de `in` (que pasaría primero a valer 6 y después a valer 0), así como el valor de `numElems`, que llegaría a 7. Sin embargo, no habría ningún elemento en la componente 6 del vector `store`.

- De manera similar, si en esa misma situación inicial dos procesos consumidores trataran de extraer de manera simultánea un elemento del buffer, probablemente ambos obtendrían el elemento contenido en la componente 0 del vector `store`. Posteriormente se incrementaría el valor de `out` (hasta llegar a 2) y se decrementaría el valor de `numElems` que pasaría a valer 3. Cada uno de ellos estaría convencido de haber obtenido un elemento distinto, pero en lugar de ello, cada uno estaría procesando por su cuenta el mismo elemento. Por contra, el elemento mantenido en la componente 1 jamás llegaría a ser utilizado por ningún consumidor y sería sobrescrito posteriormente por algún elemento insertado por un productor.

- Si a partir de la situación mostrada en la figura 1.1 tres procesos productores llegasen a insertar elementos sin que ningún consumidor extrajera ninguno mientras tanto, la capacidad del buffer se desbordaría. Debido a ello, la tercera inserción sobrescribiría el elemento mantenido en la posición 0 del vector `store`.

- Si a partir de la situación de la figura 1.1 seis procesos consumidores extrajeran elementos del buffer, sin que ningún productor insertase algún nuevo elemento, el último de ellos encontraría un buffer vacío. Sin embargo, no sería consciente de ello y retornaría el contenido de la posición 5 del vector.

Para evitar estas situaciones incorrectas, deberían imponerse y respetarse las siguientes restricciones:

PC1 Para evitar la primera y la segunda situación de error se debería impedir que los métodos de acceso al buffer fuesen ejecutados por más de una actividad simultáneamente. Es decir, cuando alguna actividad haya iniciado la ejecución de algún método, ninguna actividad más podrá ejecutar alguno de los métodos de acceso al buffer.

PC2 Para evitar que se desbordara el buffer se debería impedir que los productores ejecutasen el método de inserción cuando el valor de los atributos `size` y `numElems` fueran iguales.

PC3 Para evitar que se extrajeran elementos de un buffer vacío se debería impedir que los consumidores ejecutasen el método de extracción cuando el atributo `numElems` ya valiese cero.

Lectores-Escritores

En este problema [CHP71] se asume que existe un recurso compartido por todas las actividades. El estado de este recurso se puede modificar. Algunas actividades utilizarán métodos para modificar el estado del recurso: se considerarán "*escritores*". Otras actividades o procesos llegarán a leer el estado del recurso, pero no lo modificarán: son "*lectores*". Un ejemplo válido de recurso de este tipo sería un fichero, que podrá ser leído o escrito por diferentes procesos.

Para garantizar un uso correcto del recurso se imponen las siguientes restricciones:

LE1 Múltiples procesos lectores pueden acceder simultáneamente al recurso.

LE2 Sólo un proceso escritor puede estar accediendo al recurso. En caso de que un escritor acceda, ningún proceso más podrá utilizar el recurso.

Así se garantiza la consistencia en las modificaciones realizadas sobre el recurso. Mientras un proceso escriba sobre él ninguna escritura más llegará a causar interferencias. También se evita que algún proceso lector pudiera leer un estado intermedio durante la escritura.

Cinco filósofos

En este problema [Dij71, Hoa85] se asume que existen cinco filósofos cuya única misión es pensar y comer. En la mesa que comparten hay cinco plazas. A estos filósofos solo les gustan los *spaghetti* y para comer necesitan dos tenedores cada uno. Sin embargo, en la mesa hay cinco platos y cinco tenedores, tal como se muestra en la figura 1.2. Por tanto, cuando un filósofo vaya a comer tendrá que coger los dos tenedores que queden a ambos lados de su plato, pero para ello ninguno de sus dos filósofos vecinos podrá estar comiendo.

Este ejemplo ilustra el problema de la escasez de recursos en un sistema, pues no hay suficientes tenedores para que todos los filósofos coman a la vez. De hecho, en el mejor de los casos solo habrá dos filósofos comiendo.

El algoritmo utilizado por cada filósofo sigue estos pasos:

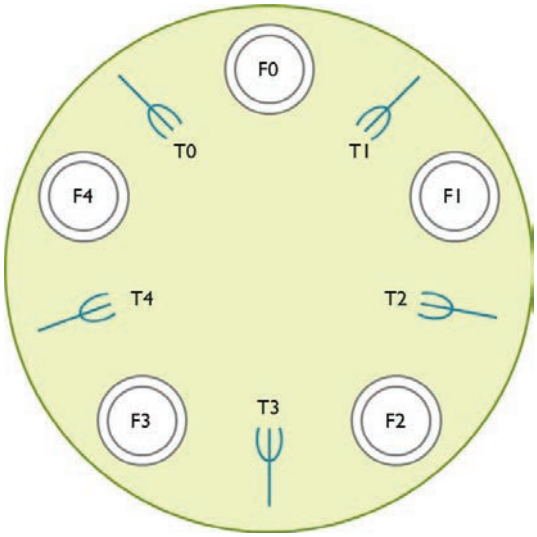

Figura 1.2: Problema de los cinco filósofos.

1. Coger el tenedor derecho.
2. Coger el tenedor izquierdo.
3. Comer.
4. Dejar ambos tenedores.
5. Pensar.
6. Volver al paso 1 cuando tenga hambre.

Los tenedores no pueden compartirse. En todo momento un tenedor estará libre o asignado a un solo filósofo, pero nunca a dos simultáneamente.

Si un filósofo observa que el tenedor que necesitaba en el paso 1 o en el paso 2 está asignado a su filósofo vecino, espera educadamente a que éste termine y lo libere.

Si todos los filósofos empezaran a la vez con sus respectivos algoritmos podrían obtener todos ellos su tenedor derecho. Sin embargo, al intentar obtener el tenedor izquierdo, ninguno de ellos lo conseguiría. Todos observarían que dicho tenedor ya está asignado a su vecino. Así, todos ellos pasarían a esperar a que tal tenedor quedara libre, pero eso nunca sucedería. Esta situación se conoce como *interbloqueo* [CES71] y se explicará con detenimiento en la unidad didáctica 4.

1.4 Tecnología Java

Para poder aprender qué es la concurrencia se necesita implantar algunos ejemplos de programas concurrentes. Para ello debe seleccionarse algún lenguaje de programación que soporte directamente el uso de múltiples hilos de ejecución en un programa. Se ha seleccionado *Java* para cubrir este objetivo por múltiples razones:

- Es un lenguaje relativamente moderno y ampliamente aceptado.
- Proporciona un soporte adecuado tanto para la programación concurrente como para la programación distribuida.
- Facilita un amplio conjunto de herramientas de sincronización que permitirán implantar de forma sencilla los conceptos que se analizan en este libro relacionados con la sincronización.
- Es independiente de la plataforma. Utiliza una *máquina virtual* [LYBB11] propia para ejecutar los programas que está soportada en la mayoría de los sistemas informáticos actuales. Con ello, el código que se obtiene al compilar un programa no depende para nada del sistema operativo utilizado en el ordenador donde ejecutaremos el programa.
- Existe abundante documentación sobre este lenguaje de programación. Buena parte de ella es gratuita. Existen guías y tutoriales [Ora12b] disponibles en la web de *Oracle* [Ora12e].

1.4.1 Concurrencia en Java

Java soporta nativamente (es decir, sin necesidad de importar ninguna biblioteca o "*package*" adicional) la creación y gestión de múltiples hilos de ejecución en todo programa que se escriba en este lenguaje. Aparte de los hilos que podamos crear explícitamente a la hora de implantar un determinado programa, todo proceso Java utilizará algunos hilos de ejecución implícitos (utilizados para gestionar la interfaz de usuario, en caso de utilizar una interfaz gráfica basada en ventanas, y para gestionar la *recolección de residuos* [WK00, Appendix A], es decir, para realizar la eliminación de aquellos objetos que ya no estén referenciados por ningún componente de la aplicación).

Se debe recordar que un *hilo de ejecución* (o "*thread*") es la unidad de planificación comúnmente utilizada en los sistemas operativos modernos. Por su parte, un *proceso* es la entidad a la que se asignan los recursos y mantiene a un programa en ejecución. Por tanto, cada proceso tendrá un *espacio de direcciones* independiente, mientras que todos los hilos de ejecución creados en una misma ejecución de un

programa determinado compartirán ese espacio de direcciones asignado al proceso que los engloba.

En el caso particular de Java, un conjunto de clases podrá definir una aplicación. En una de tales clases deberá existir un método `main()` que será el que definirá el hilo principal de ejecución de esa aplicación. Para poder ejecutar la aplicación habrá que utilizar una máquina virtual Java (o JVM) [LYBB11] y esa acción implicará la creación de un proceso soportado por el sistema operativo utilizado en ese ordenador.

1.4.2 Gestión de hilos de ejecución

La gestión básica de los hilos de ejecución en Java será analizada en la unidad 2. No obstante, se describe seguidamente cómo se pueden definir hilos en un programa Java y qué debe hacerse para asignarles un identificador y averiguar la identidad de cada hilo.

Creación de hilos

Para definir un hilo de ejecución en Java debemos definir alguna instancia de una clase que implante la interfaz `Runnable`. Java ya proporciona la clase `Thread` que implanta dicha interfaz. Por tanto, disponemos de cuatro alternativas básicas para definir hilos, tal como se muestra en la Tabla 1.1.

	Clase con nombre	**Clase anónima**

Implantando `Runnable`	public class H implements Runnable {
	  public void run() {
	  System.out.println("Ejemplo.");
	  }
	}
	Thread t = new Thread(new H());
		Runnable r = new Runnable() {
		  public void run() {
		  System.out.println("Ejemplo.");
		  }
		};
		Thread t = new Thread(r);

Extendiendo `Thread`	public class H extends Thread {
	  public void run() {
	  System.out.println("Ejemplo.");
	  }
	}
	H t = new H();
		Thread t = new Thread() {
		  public void run() {
		  System.out.println("Ejemplo.");
		  }
		};

Tabla 1.1: Variantes de la definición de hilos.

Obsérvese que el código que tendrá que ejecutar el hilo debe incluirse en el método `run()`. En la Tabla 1.1 hemos incluido una sola sentencia en dicho método: la necesaria para escribir la palabra "Ejemplo" en la salida estándar.

La columna izquierda muestra aquellas variantes en las que se llega a definir una clase (a la que se ha llamado `H`) para generar los hilos. Cuando se adopta esta

opción será posible generar múltiples hilos de esa misma clase, utilizando una sentencia "`new H()`". Por su parte la columna derecha ilustra el caso en que no se necesite generar múltiples hilos y baste con definir directamente aquél que vaya a utilizarse. En estos ejemplos la variable utilizada para ello ha sido "`t`".

A su vez, la fila superior muestra cómo debe generarse el hilo cuando se está implantando directamente la interfaz `Runnable`. Si definimos una clase (columna izquierda) habrá que añadir en su declaración un "`implements Runnable`" antes de abrir la llave que define el bloque en el que se definirán sus atributos y métodos. Por su parte, la columna derecha ilustra cómo se puede definir una instancia de dicha interfaz (utilizando para ello la variable "`r`") que después podrá utilizarse como argumento en el constructor del hilo "`t`". En este caso el código del método "`run()`" debe incluirse al definir "`r`".

La fila inferior muestra cómo debe realizarse esta gestión en caso de generar una subclase de `Thread` (si utilizamos una clase explícita, en la columna izquierda) o cómo instanciar un `Thread` (en caso de que solo interese generar un hilo, en la columna derecha) con su propio código para "`run()`".

Obsérvese que tras utilizar el código mostrado en la Tabla 1.1 en cualquiera de sus cuatro variantes se habrá generado un hilo Java, cuya referencia mantendremos en la variable "`t`". Este hilo todavía no se ejecutará, pues lo que hemos conseguido con estas definiciones es simplemente generarlo, de manera que el *runtime* de Java le habrá asignado todos los recursos necesarios para su ejecución, pero su estado de planificación será todavía "*nuevo*" en lugar de "*preparado*". Para que un hilo pase a estado "*preparado*" y así pueda iniciar su ejecución cuando el planificador del sistema operativo lo seleccione, debe utilizarse el método "`start()`".

Por ello, en alguna de las líneas que sigan a las mostradas en la Tabla 1.1 deberíamos encontrar una sentencia "`t.start();`". Esa sentencia inicia la ejecución del hilo referenciado por la variable "`t`".

Aunque la sentencia "`t.run();`" parece sugerir un efecto similar y no genera ningún error ni excepción, su resultado es muy diferente. Si dicha sentencia es ejecutada por un hilo A, en lugar de iniciarse la ejecución del hilo "`t`", lo que ocurrirá es que A será quien ejecute las sentencias del método "`run()`" y "`t`" seguirá todavía en el estado "*nuevo*".

Identificación de hilos

Al crear un hilo es posible asociarle un nombre. La clase `Thread` admite como argumento en su constructor una cadena que será la utilizada como nombre o identificador del hilo que se genere.

Si se necesitara modificar el nombre de un hilo una vez ya se haya generado éste, se puede utilizar el método `setName()`. A su vez, la clase `Thread` ofrece un método `getName()` para obtener el nombre de un hilo (aunque necesitamos una variable que lo referencie). Así, si queremos escribir el nombre del hilo actualmente en ejecución tendremos que utilizar esta sentencia:

```
System.out.println( Thread.currentThread().getName() );
```

La figura 1.3 lista un programa Java que utiliza los métodos explicados en esta unidad para lanzar 10 hilos de ejecución, que se encargan de escribir su propio nombre para finalizar de inmediato. Cada uno de esos hilos recibe como identificador la cadena "ThreadX" siendo X cada uno de los dígitos entre 0 y 9, ambos inclusive.

```
public class Sample {
    public static void main( String args[ ] ) {
        System.out.println(Thread.currentThread().getName());
        for(int i=0; i<10; i++) {
            new Thread("Thread"+i) {
                public void run() {
                    System.out.println("Executed by " +
                        Thread.currentThread().getName() );
                }
            }.start();
        }
    }
}
```

Figura 1.3: Ejemplo de creación de hilos.

Obsérvese que para asignar los nombres a los hilos se ha empleado el propio constructor de la clase `Thread` y para imprimir el nombre de cada hilo se ha utilizado el método `getName()`. Por último, ya que todos los hilos se estaban generando en una misma sentencia dentro del bucle "`for`" se ha podido utilizar la variante anónima de creación de hilos. Por ello, se ha añadido una llamada al método `start()` tras la llave que cerraba la definición de la clase.

1.5 Resumen

En esta unidad didáctica se han introducido los conceptos básicos de la programación concurrente. Un programa concurrente está constituido por diferentes actividades, que pueden ejecutarse de forma independiente, las cuales se reparten la solución del problema. Para ello, estas actividades trabajan en común, coordinándose y comunicándose entre sí empleando variables en memoria y mensajes, para así compartir datos y proporcionarse sus resultados. La programación concurrente permite mejorar las prestaciones del sistema, aumentando su eficiencia y escalabilidad, proporcionando también una gestión eficiente de las comunicaciones. Además, la programación concurrente mejora la interactividad y flexibilidad de las tareas, al utilizar diseños modulares y estructurados. Asimismo, en muchas ocasiones las aplicaciones requieren el uso de varias actividades simultáneas, por lo que resulta más directo diseñarlas a través de la programación concurrente (ya que es menor el hueco semántico que se produce), que si se intenta diseñarlas como un solo programa secuencial.

A lo largo de esta unidad didáctica se han presentado diferentes aplicaciones concurrentes reales, tales como el programa de control de acelerador lineal Therac-25 y el servidor web Apache, así como una serie de problemas clásicos (o académicos) de la programación concurrente, con los que se ilustran las características de la programación concurrente y los problemas y dificultades que se plantean a la hora de desarrollar este tipo de aplicaciones. Para dar soporte a los ejemplos de programas concurrentes, se ha seleccionado el lenguaje de programación Java, que permite la creación y gestión de múltiples hilos de ejecución. Precisamente, en esta unidad se describe cómo se definen hilos en Java y cómo asignarles un identificador.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Discriminar entre programación secuencial y programación concurrente.
- Describir las ventajas e inconvenientes que proporciona la programación concurrente.
- Enumerar distintas aplicaciones concurrentes, tanto aplicaciones reales como problemas clásicos.
- Implementar hilos en Java.
- Identificar las secciones de una aplicación que deban o puedan ser ejecutadas concurrentemente por diferentes actividades.

Unidad 2

COOPERACIÓN ENTRE HILOS

2.1 Introducción

La programación concurrente requiere que las distintas actividades que formen una determinada aplicación cooperen entre sí para llevar a cabo la tarea para la que fue diseñada tal aplicación. Esta unidad revisa los mecanismos de cooperación que podrán utilizarse para implantar una aplicación concurrente. En los casos más sencillos las actividades serán hilos de ejecución. Por ello, esta unidad revisa el concepto de hilo de ejecución en su sección 2.2.

Posteriormente se pasa a describir los mecanismos de cooperación o colaboración entre actividades, dentro de la sección 2.3. Estos mecanismos son la comunicación, para que las actividades puedan intercambiar información, y la sincronización, mediante la que se controla la ejecución de las tareas para que dichos intercambios de información no generen ningún tipo de inconsistencia.

Sin mecanismos de sincronización la ejecución concurrente de actividades puede perder su determinismo (Sección 2.5); es decir, al intercalarse de manera arbitraria la secuencia de instrucciones ejecutada por cada actividad, algunas de esas instrucciones dejarían de tener su efecto esperado. Con ello, resultaría imposible saber qué resultado generaría tal ejecución. Eso debe evitarse. Para ello se tendrá que conocer en qué partes del código de las actividades podrá haber problemas y por qué se generan tales problemas. Son las secciones críticas, que se describirán en la sección 2.6. Para proteger tales secciones críticas se utilizarán los mecanismos de sincronización básicos y la sincronización condicional, explicados en la sección 2.7.

2.2 Hilos de ejecución

Un *hilo de ejecución* es cada una de las actividades independientes que pueden ejecutarse simultáneamente (al menos de manera lógica) dentro de un proceso.

En los sistemas operativos modernos los hilos son la unidad de planificación utilizada por el sistema operativo. Es decir, el planificador toma sus decisiones en base al conjunto de hilos de ejecución que se mantengan en estado preparado en cada momento, escogiendo de entre ellos al que estará en ejecución. Cada núcleo (o "*core*") del procesador no podrá tener más de un hilo en ejecución simultáneamente.

Un hilo se suspende cuando debe esperar a que suceda algún evento. Se dice entonces que permanece en *estado suspendido*. Algunos ejemplos de eventos de este tipo son: entrada desde teclado, lectura de un bloque del disco, llegada de un paquete de información por la red, finalización de una operación de impresión sobre una multifunción, etc.

Cuando tal evento sucede, el sistema operativo reactiva al hilo que había quedado suspendido previamente. Con ello lo pasa al *estado preparado*. Dicho estado indica que el hilo está listo para ser ejecutado. Sin embargo, todavía tendrá que esperar a que el planificador lo escoja para que sea de nuevo ejecutado en algún procesador.

Para sustituir al hilo en ejecución, el sistema operativo realiza un *cambio de contexto*. Para ello:

1. Se copia el contexto del hilo actualmente en ejecución sobre su "*bloque de control*" (estructura de datos mantenida por el sistema como componente de una tabla de hilos, donde mantiene tanto el contexto como el resto de atributos de un hilo determinado).
2. Se elige a un nuevo hilo preparado para que pase a ejecución, utilizando para ello un *algoritmo de planificación* determinado.
3. Se lee el contexto del hilo seleccionado en el paso anterior y se copia sobre el procesador. Con ello, ese nuevo hilo pasará a ejecutarse, pues en dicha copia se ha modificado el valor del contador de programa del procesador.

El primero de estos pasos puede deberse a que el hilo que hasta ahora se ejecutase se haya suspendido (al programar una operación de E/S o esperar por algún otro tipo de evento) o a que se utilizara un planificador expulsivo y hubiera llegado un nuevo hilo preparado que fuese un mejor candidato según el criterio asumido por el algoritmo de planificación.

Debe recordarse que un sistema operativo es un programa dirigido por interrupciones. Por ello, el hilo actualmente en ejecución se mantendrá en dicho estado mientras no se genere alguna interrupción o excepción. Una *interrupción* es causada por:

- La finalización de una operación de E/S. En este caso se da una interrupción propiamente dicha. Ejemplos: finalización de una lectura de un bloque (o una secuencia de bloques) del disco, finalización de una escritura de un bloque (o una secuencia de bloques) del disco, pulsación en el teclado, movimiento del ratón, pulsación de un botón del ratón, llegada de un paquete de datos por la red, etc.

- La ejecución de alguna operación no permitida por parte del hilo. Este tipo particular de interrupciones se llama *excepción* y puede darse cuando se utilice una instrucción privilegiada en modo usuario o cuando, en un programa erróneo, se llegue a transferir el control a una región donde haya datos en lugar de instrucciones del procesador.

- La invocación de una llamada al sistema, mediante la generación de una "*interrupción software*". No todas las llamadas provocarán la expulsión o suspensión del hilo de ejecución actual. En caso de que el planificador no intervenga, el control retornará al mismo hilo una vez finalice la ejecución de la llamada.

- La generación de un *fallo de página*. Esto ocurre cuando el hilo accede a una *página* de su espacio virtual de direcciones que no tenga ningún *marco* asociado. En sistemas operativos que gestionen *memoria virtual*, el sistema comprobará si dicha página formaba parte de alguna región asignada al proceso en el que se ejecute el hilo. De ser así, se buscará un marco en el que ubicar el contenido de la página y se asignará a dicho proceso. Tras esto, se reintentará la ejecución de la instrucción que provocó el fallo.

Como el planificador es un componente del sistema operativo que no podrá ser controlado por los usuarios ni por los programadores, no podrá asumirse a qué velocidad relativa avanzarán los diferentes hilos de una determinada aplicación. Es decir, resultará imposible predecir en qué orden los hilos llegarán a un determinado punto de sus respectivos códigos.

2.2.1 Ciclo de vida de los hilos Java

Cuando se utilice el lenguaje de programación Java, los hilos que se generen en una aplicación seguirán el ciclo de vida mostrado en la figura 2.1. En él se pueden distinguir los siguientes estados:

- *Nuevo* ("*New*"): Es el estado en el que se encontrará el hilo cuando acabe de ser creado. Abandonará este estado cuando se invoque su método `start()`, pasando entonces al estado *preparado*.

- *Preparado* ("*Ready-to-run*"): Un hilo se encuentra preparado cuando nada le obligue a realizar una espera y pueda ser ejecutado. El único motivo por el

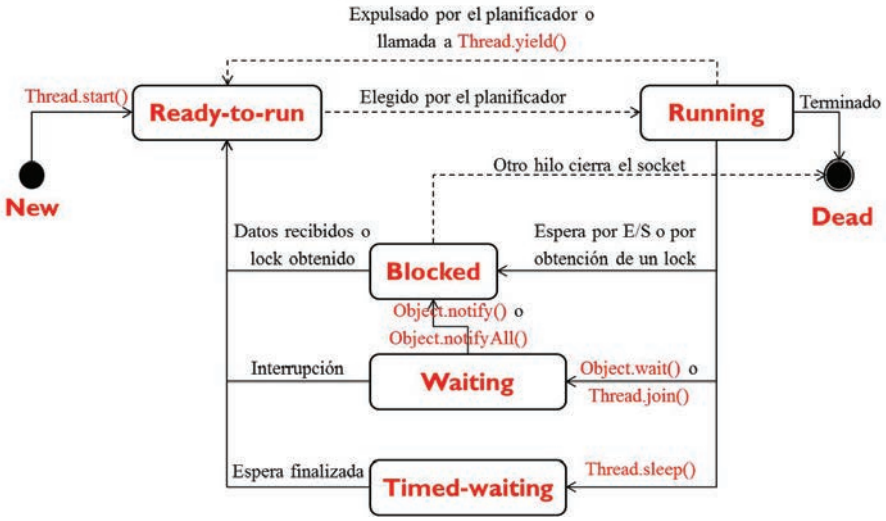

Figura 2.1: Diagrama de estados de los hilos en Java.

que no estará en ejecución es que habrá un número reducido de procesadores en el sistema y todos ellos estarán ocupados.

Si en un momento determinado existen varios hilos en estado preparado y alguno de los procesadores del sistema queda libre, el planificador seleccionará cuál de ellos pasará a ejecutarse. El criterio utilizado para realizar tal selección dependerá del algoritmo de planificación utilizado en el sistema. En general, estos algoritmos optimizan el rendimiento global o la utilización del procesador o el tiempo de respuesta u otros parámetros. En cualquier caso se intentará que el algoritmo sea lo más equitativo posible, evitando así la *inanición* (esto es, que algún hilo permanezca largo tiempo en la cola de preparados mientras otros vayan obteniendo el procesador).

- *En ejecución* ("*Running*"): Un hilo permanece en este estado mientras sus instrucciones sean ejecutadas por algún procesador.

  Los hilos abandonan el estado de ejecución cuando hayan conseguido ejecutar todas sus instrucciones (pasando entonces al estado *terminado*), cuando sean expulsados por otros hilos (en caso de que se emplee un algoritmo de planificación expulsivo, volviendo entonces al estado *preparado*), cuando utilicen el método `Thread.yield()` (que también los deja en estado *preparado*), o cuando pasen a esperar la ocurrencia de algún evento.

  En caso de que se espere por algún motivo, el hilo suspenderá su ejecución y realizará una transición a uno de estos tres estados: *Blocked*, *Waiting* o *Timed-waiting*.

- *Bloqueado* ("*Blocked*"): Estado de suspensión en el que se trata de obtener algún mecanismo de sincronización (por ejemplo, un obtener un lock o acceder a un método sincronizado de un objeto cuando hay algún otro hilo ejecutando código de alguno de los métodos sincronizados de ese mismo objeto) o bien se espera por la finalización de una operación de E/S.

  Los hilos abandonan este estado y pasan al estado *preparado* cuando termine la operación de E/S solicitada o cuando obtengan el lock o el acceso al método sincronizado por el que se habían suspendido.

- *Suspendido con temporización* ("*Timed-waiting*"): Cuando el hilo se haya suspendido debido a la utilización del método `sleep()` de la clase `Thread`. Este método admite como argumento el número de milisegundos que el hilo tendrá que permanecer suspendido.

  Los hilos abandonan este estado cuando finaliza el intervalo de suspensión que habían especificado como argumento en su llamada a `sleep()` o bien si reciben una llamada a su método `interrupt()`. En el primer caso pasan a estado *preparado*. En el segundo caso reciben una `InterruptedException`, pasando a estado *preparado* (cuando la llamada a `sleep()` estuviese protegida por un bloque `try{...} catch(...){...}` que gestionase esas excepciones) o a estado *terminado* (si no se gestionara la excepción).

- *Suspendido* ("*Waiting*"): Cuando el hilo se haya suspendido debido a la utilización del método `join()` (de la clase `Thread`, para esperar la finalización de otro hilo) o del método `wait()` (de la clase `Object`, para esperar a que se cumpla determinada condición).

  Si la suspensión fue causada por el uso del método `join()` y el hilo por el que se esperaba ha terminado, el estado pasará a *preparado*. También ocurre lo mismo (pasar a *preparado*) cuando se abandone la suspensión debido a una llamada a `interrupt()`. Por el contrario, cuando la suspensión fue originada por una llamada a `wait()` y termina debido a que otro hilo utiliza `notify()` o `notifyAll()`, el estado de hilo previamente suspendido pasa a ser *bloqueado*, pues en ese caso competirá con el hilo notificador para obtener el derecho a utilizar un método sincronizado. Se explicarán más detalles sobre esta última transición cuando se describan los monitores Java en la unidad siguiente.

- *Terminado* ("*Dead*"): Cuando el hilo finaliza la ejecución de todas sus instrucciones o es interrumpido por una excepción que es incapaz de gestionar, pasa a este estado. Con ello el hilo es eliminado del sistema.

En estas transiciones y estados se han utilizado algunos métodos que merecen cierta atención. Son los siguientes (en esta lista utilizamos el nombre de la clase y el nombre del método separados por un punto; en la práctica estos métodos no se utilizarán así, sino sobre una variable que identifique a una instancia de dicha clase. Por ejemplo, si se ha utilizado una sentencia `Thread h = new Thread();`

para definir un hilo `h`, se utilizará `h.start()` para arrancar el hilo o `h.isAlive()` para averiguar si aún no ha terminado):

- `Thread.run()`: Es el método que contiene el código a ejecutar por el hilo.
- `Thread.start()`: Método que pasa al hilo desde el estado *nuevo* al estado *preparado*. De hecho, este es el método que debe emplearse para que el hilo inicie su ejecución. Cuando el planificador lo seleccione, el hilo pasará al estado de ejecución y será entonces cuando se pase a ejecutar las instrucciones contenidas en su método `run()`. Es decir, en lugar de llamar directamente a `run()` se debe utilizar `start()`.
- `Thread.isAlive()`: Este método devuelve `true` cuando el hilo haya sido iniciado y todavía no haya terminado.
- `Thread.sleep()`: Este método suspende al hilo que lo invoque durante el número de milisegundos especificado como argumento.
- `Thread.join()`: Dados dos hilos `A` y `B` y asumiendo que `A` utilice la sentencia `B.join();`, este método suspenderá al hilo invocador (`A`, en este ejemplo) hasta que `B` finalice. Es decir, con esa sentencia `A` esperaría la terminación de `B`.
- `Thread.interrupt()`: Este método envía una `InterruptedException` al hilo especificado. Si el hilo estaba suspendido por una invocación previa a `sleep()`, `join()`, o `wait()`, abandonará dicha suspensión y tratará dicha excepción o terminará.
- `Thread.yield()`: Este método puede ser utilizado por el hilo actualmente en ejecución para abandonar voluntariamente el procesador. Para ello se utilizaría la sentencia `Thread.currentThread().yield();`. De esta manera se consigue ceder la ejecución a otro hilo preparado.

2.3 Cooperación entre hilos

Como ya se ha visto en la sección 1.2, toda aplicación concurrente estará formada por múltiples actividades que cooperarán entre sí para realizar una determinada tarea. Cuando la unidad de ejecución empleada en el sistema sean los hilos de ejecución, se necesitará que esos hilos cooperen para que lleguen a formar una aplicación concurrente. Para que dicha cooperación sea posible, se utilizarán dos mecanismos básicos: comunicación y sincronización.

La comunicación permite que los hilos intercambien información. Por su parte, las herramientas de sincronización conseguirán que el programador establezca determinado orden en casos concretos o que establezca ciertas reglas que controlen la

ejecución de los hilos. Las secciones 2.3.1 y 2.3.2 describen detenidamente estos mecanismos.

2.3.1 Comunicación

Los mecanismos de *comunicación* tienen como principal objetivo que las distintas actividades de una aplicación concurrente puedan intercambiar información entre sí. Si no hubiera comunicación entre estas actividades, pasarían a ser independientes y no se podría hablar de una aplicación concurrente.

Existen dos mecanismos básicos de comunicación:

- *Variables compartidas*: El uso de variables compartidas se da cuando las diferentes actividades de la aplicación concurrente compartan un mismo espacio de direcciones. Esto ocurre cuando en un mismo proceso se hayan llegado a utilizar múltiples hilos. En ese caso, las variables que se hayan declarado con un ámbito global dentro del programa podrán ser utilizadas por todos los hilos, siendo compartidas por ellos. De esa manera podrán intercambiar información entre sí: modificando alguna de las variables y leyendo el valor modificado por otros hilos cuando sea conveniente.

- *Intercambio de mensajes*: Una de las actividades actuará como emisora de los mensajes y la otra como receptora, en cada envío de mensaje. La información a intercambiar entre las actividades estará contenida en el mensaje. Con este mecanismo de comunicación, el emisor y el receptor podrán encontrarse en diferentes espacios de direcciones. Por ello, el intercambio de mensajes permite comunicar tanto a hilos que estén en un mismo proceso, como a procesos ubicados en una misma máquina o bien a procesos ubicados en ordenadores diferentes.

  No estudiaremos este segundo mecanismo en esta unidad, retrasando su análisis hasta la Unidad 8 una vez se haya presentado el concepto de sistema distribuido.

Cuando la comunicación se implanta mediante variables compartidas se deben utilizar mecanismos complementarios que aseguren la coordinación de los hilos a la hora de acceder a dichas variables. En general, no debería permitirse que más de una actividad modificara a la vez una misma variable, pues en ese caso no se podrá predecir cuál será el resultado de tales modificaciones. Por ello, la comunicación mediante variables compartidas deberá acompañarse con algún mecanismo de sincronización.

En el caso de los lenguajes de programación orientados a objetos como Java, C++ o C#, las variables compartidas se implantarán como objetos a los que podrán acceder múltiples hilos.

Java utiliza un modelo de memoria compartida en el que todos los hilos podrán acceder a cualquier objeto de su programa del que dispongan de alguna referencia. Para instanciar un objeto se suele hacer uso del operador `new` (como en la sentencia `MiThread h = new MiThread();`). Por ello, los objetos se instancian en *memoria dinámica* (o *heap* de la máquina virtual). Para poder utilizarlos necesitamos alguna variable que los referencie (como la variable `h` del ejemplo anterior, que permite el acceso sobre la instancia creada de la clase `MiThread`). Cualquier hilo que tenga alguna variable de este tipo podrá utilizar el objeto al que haga referencia.

Se tendrá un *objeto compartido* cuando dos o más hilos tengan una referencia a él y puedan utilizarlo.

2.3.2 Sincronización

Los mecanismos de *sincronización* tienen como objetivo el garantizar un determinado orden en la ejecución de ciertas sentencias o que se respeten ciertas restricciones en la ejecución de determinadas secciones de código.

Existen dos tipos básicos de sincronización:

- *Exclusión mutua*: Una sección de código se ejecutará en exclusión mutua cuando solamente un hilo pueda ejecutar dicha sección en cada momento. Así, si cuando un hilo A esté ejecutando la sección llegara otro hilo B que también debiera ejecutarla, B esperará a que A finalice la ejecución de dicha sección. Solo cuando A termine la sección podrá B entrar en ella.

  Las secciones de código que permitan acceder a variables compartidas tendrán que protegerse de esta manera para evitar las interferencias entre los hilos. De esta manera se garantiza la consistencia entre múltiples actualizaciones del valor de una variable compartida, así como la consistencia entre una escritura y todas las lecturas que la sigan.

- *Sincronización condicional*: Este tipo de sincronización obliga a que los hilos se suspendan mientras no se cumpla una determinada condición. Dicha condición suele depender del valor de algunas variables compartidas. Otros hilos, al modificar tales variables conseguirán que finalmente se cumpla la condición, reactivando entonces a los hilos previamente suspendidos.

Las secciones 2.4, 2.5 y 2.6 explican detenidamente cuándo y por qué se necesita una sincronización que garantice exclusión mutua. A su vez, la sección 2.7 describe la sincronización condicional.

2.4 Modelo de ejecución

Un hilo transforma su estado mediante la ejecución de sentencias. Una *sentencia* es una secuencia de acciones atómicas que realizan transformaciones indivisibles.

Una *acción atómica* es aquella que transforma el estado y no puede dividirse en acciones menores. Aquellas instrucciones máquina que no sean interrumpibles son ejemplos válidos de acciones atómicas.

En algunas notaciones se agrupa una secuencia de acciones para formar una única acción atómica empleando corchetes. Así, la secuencia $[y := x; z := y]$ sería una acción atómica y en ese caso no se podría observar ningún estado intermedio en el que $x \neq z$.

La traza de una ejecución concreta es una secuencia de acciones atómicas. La transformación de estado realizada por una acción atómica no se ve afectada por otras acciones, ya que dicha transformación no podrá ser interrumpida y cada uno de los cambios parciales que provoque no resultarán visibles. Hasta que no termine por completo dicha acción atómica la transformación de estado que ella genere no será visible.

Un *estado consistente* de un objeto es aquel que cumpla con todos los invariantes del objeto. En ese caso será *coherente* con todas las operaciones efectuadas hasta el momento sobre ese objeto. Toda transformación del estado de un objeto (realizada como resultado de la invocación de un método) se asume que es iniciada sobre un estado consistente y su objetivo será la obtención de otro estado consistente. Sin embargo, estas transformaciones estarán implantadas mediante una secuencia de sentencias (es decir, constarán de múltiples acciones atómicas), por lo que podrá haber múltiples estados intermedios que no sean coherentes.

Por tanto, para garantizar la coherencia en la ejecución de los métodos de un objeto compartido, se deberían agrupar los pasos de los que consta esa transformación de estado en una única acción atómica. De esa manera se evitaría que otros hilos puedan acceder a los diferentes estados intermedios (e incoherentes) por los que transitará el objeto.

Por ejemplo, asumamos que se ha implantado una clase Java cuyo nombre es `MyQueue` para gestionar colas de enteros, utilizando para ello el código que se muestra en la figura 2.3, basado en la clase `Node` mostrada en la figura 2.2. Esta clase `Node` mantiene un atributo público `value` de tipo entero donde se guarda el valor asociado a un nodo de la cola, y otro atributo `next` que será una referencia al siguiente nodo de la cola. De esta manera se podrá implantar una cola enlazada.

Utilizando esta clase como ejemplo para ilustrar qué son los estados intermedios incoherentes, tanto antes como después de la ejecución de un método, la clase `MyQueue` debería respetar los siguientes invariantes:

```
public class Node {
   public int value;
   public Node next;
}
```

Figura 2.2: Clase Node

IN1 El atributo `head` de esta clase es una referencia al nodo que mantiene la cabeza (es decir, el primer elemento) de la cola. Valdrá `null` cuando la cola esté vacía.

IN2 El atributo `tail` de esta clase es una referencia al último elemento de la cola. Valdrá `null` cuando la cola esté vacía.

IN3 El atributo `numElems` contabiliza el número de enteros mantenidos en la cola.

IN4 Los elementos de la cola se mantienen en memoria dinámica y están enlazados mediante sus punteros `next`, formando una lista simple iniciada en el elemento apuntado por `head` y terminando en el elemento apuntado por `tail`.

Si tras haber creado un objeto de la clase `MyQueue`, generando una cola vacía, se llamara posteriormente a `enqueue(4);`, cada una de las sentencias de ese método tendría el siguiente efecto sobre el estado del objeto:

1. `Node n=new Node();`: Generaría un nuevo nodo en el que se mantendrá el entero que se pretende insertar en la cola. Tras la ejecución de dicha sentencia se seguirían manteniendo todos los invariantes de la clase, asumiendo que la cola todavía sigue vacía. Sin embargo, la cola ya no estaba vacía por lo que el estado que se ha generado tras esta sentencia ya no es *consistente*.

2. `n.value=elem;`: Se asignaría el valor para ese nuevo nodo. Se siguen manteniendo los invariantes de una cola vacía.

3. `n.next=null;`: Se reflejaría que el nodo no tiene ningún elemento siguiente. Se siguen manteniendo los invariantes de una cola vacía.

4. `head=n;`: Se actualizaría el valor de la referencia al primer elemento de la cola. Al hacer esto dejaría de respetarse el invariante IN4. A su vez, el invariante IN1 se cumpliría si se asume que la cola ya tiene un elemento pero los invariantes IN2 e IN3 solo se cumplirían en caso de asumir una cola vacía. Esto refleja que se ha llegado a un estado inconsistente.

5. `tail=n;`: Se actualizaría el valor de la referencia al último elemento de la cola. Con ello se volvería a respetar el invariante IN4, así como el IN1 y el IN2, asumiendo en los tres casos que la cola tiene un elemento. Sin embargo,

```
public class MyQueue {
  private Node head;
  private Node tail;
  private int numElems;

  public MyQueue() {
     numElems=0;
     head=null;
     tail=null;
  }

  public void enqueue(int elem) {
     Node n = new Node();
     n.value = elem;
     n.next = null;
     if (numElems==0)
        head=n;
     else  tail.next=n;
     tail=n;
     numElems++;
  }

  public int dequeue() throws EmptyQueue{
     int elem;
     if (numElems==0)
        throw new EmptyQueue();
     elem=head.value;
     head=head.next;
     numElems--;
     return elem;
  }
}
```

Figura 2.3: Clase Queue.

no se respetaría el invariante IN3 pues el atributo `numElems` no ofrece un valor consistente con lo que se exige para una cola con un elemento.

6. `numElems++;`: El atributo `numElems` pasa a valer uno y de esta forma pasan a respetarse todos los invariantes de la clase, siendo consistentes con el estado de una cola con un único elemento. Por tanto, esta última sentencia del método pasa a generar un nuevo *estado consistente* capaz de cumplir con todos los invariantes, al igual que ocurría antes de la ejecución de este método. Ninguno de los pasos anteriores mantuvo dicha estabilidad o coherencia.

Esto demuestra que este método debería implantarse de tal manera que los estados intermedios generados entre cada una de las sentencias del método no fueran observables por el resto de hilos. Es decir, todo el método debería construirse como una única acción atómica para que no generara problemas en una ejecución concurrente.

2.5 Determinismo

Se dice que un *programa* es *determinista* cuando ante una misma combinación de datos de entrada siempre (en cada una de las ejecuciones en las que se utilice tales datos de entrada) genera una misma salida.

Generalmente los programas secuenciales (los que solo disponen de un único hilo de ejecución) son deterministas. Sin embargo, los programas concurrentes no siempre lo serán. En un programa concurrente habrá múltiples actividades ejecutándose simultáneamente. Cada una de ellas tendrá que ejecutar su propia secuencia de sentencias (y, a partir de estas sentencias, una determinada secuencia de acciones atómicas) que se intercalarán de manera arbitraria (es decir, no se puede predecir en qué orden se llegarán a "mezclar" las instrucciones máquina generadas a partir de las sentencias que deben ejecutar cada uno de los hilos del programa).

Dicho intercalado no se puede conocer a priori porque no se puede saber a qué velocidad avanzará cada hilo ya que se desconoce de qué forma el planificador del sistema llegará a organizar la ejecución de los hilos. Esto se debe a que los hilos de un mismo programa podrán ser planificados de manera distinta en dos ejecuciones diferentes, aunque ambas se realicen sobre el mismo ordenador y el mismo sistema operativo. Por ejemplo, si en una primera ejecución no hubiese otros procesos de usuario dentro de ese ordenador, todo el tiempo de procesador que quedara disponible sería aprovechado por los hilos del proceso en cuestión. Sin embargo, en una segunda ejecución podríamos tener muchos más procesos iniciados y en ese caso, el procesador debería repartirse entre todos ellos. Además, si la planificación estuviera basada en prioridades y hubiese algún hilo H1 poco prioritario en ese proceso junto a otros (H2 y H3) con la prioridad más alta para hilos de usuario, en la primera ejecución H1 no habría observado problemas para avanzar, pues tendría

pocos competidores (solo H2 y H3, pero mientras ellos estuvieran suspendidos, H1 avanzaría sin dificultad). Sin embargo, si los procesos adicionales que han intervenido en la segunda ejecución tuvieran hilos cuya prioridad fuera superior a la de H1 pero inferior a la de H2 y H3, dichos hilos evitarían que H1 obtuviera el procesador, pero no afectarían a H2 y H3. Con ello, las diferencias en la velocidad de avance entre H1, H2 y H3 serían mucho más altas en la segunda ejecución que en la primera.

Aunque se desconozca cómo se intercalarán las acciones atómicas ejecutadas por cada hilo, ése no es motivo suficiente para asegurar que se perderá el determinismo en la ejecución de un programa concurrente. La pérdida de determinismo se dará en caso de que varios de esos hilos accedan a un mismo objeto compartido. En ese caso sí que será problemático el hecho de que se desconozca el orden concreto en el que van a ejecutarse esas instrucciones, pues se podría provocar que alguna actualización de ese objeto compartido no se aplicara de manera correcta, generando inconsistencias. Este problema ya se había anticipado al discutir el concepto de *estado consistente*: si el hilo abandona el procesador cuando el estado del objeto no fuera consistente y otros hilos llegan a observar tal estado, las acciones que tomarán como resultado de esa observación podrían ser incorrectas (es decir, ya no serían deterministas).

Además, en los programas concurrentes cada actividad o hilo tendrá que utilizar los registros del procesador que ejecute sus instrucciones para mantener temporalmente copias de los objetos compartidos que deba utilizar dicha actividad. Por ello, al compilar el programa puede que se generen más instrucciones máquina de lo que inicialmente pudiera sospecharse. Instrucciones aparentemente sencillas, tales como una asignación, pueden necesitar múltiples instrucciones máquina para trasladar el valor entre los registros y la memoria principal, generando así múltiples estados intermedios que otros hilos no deberían observar.

```
public class Counter {
  private long count = 0;
  public void add(long x) {
    count += x;
  }
  public long getCount() {
    return count;
  }
}
...
Counter c = new Counter();
```

Figura 2.4: Contador no determinista.

Por ejemplo, la figura 2.4 muestra un ejemplo de código Java que no será determinista si se ejecuta en un programa concurrente. Si se crea un objeto `c` de la clase `Counter` como se observa en la figura y el programa correspondiente tiene más de un hilo (por ejemplo, asumamos que tenga dos hilos A y B), se podrá encontrar alguna ejecución que genere problemas. Para ello debemos recordar que el objeto `c` se mantiene en el "*heap*" pero para incrementar el valor de su atributo `count` se necesitará utilizar una variable local en cada uno de los hilos que utilice su método `add()`. Así, si A utiliza la sentencia `c.add(2)` y B la sentencia `c.add(3)` tras crearse el objeto `c` (con valor inicial cero para su atributo `count`) podría darse el siguiente intercalado:

1. A inicia la ejecución del método `c.add()` y carga el valor actual de `c.count` en una variable local ($\texttt{local}_A$). Dicho valor es cero.
2. El planificador expulsa a A y cede el procesador a B.
3. B inicia la ejecución del método `c.add()` y carga el valor actual de `c.count` en una variable local ($\texttt{local}_B$). Dicho valor es cero.
4. B incrementa en tres unidades su variable local ($\texttt{local}_B$=3).
5. B pasa el valor de su variable local al atributo `c.count`, `c.count=`$\texttt{local}_B$, con lo que `c.count=3`.
6. Tras cierto intervalo, el planificador reasigna el procesador a A.
7. A incrementa en dos unidades su variable local. Como esa variable no tiene nada que ver con la gestionada por B, seguía teniendo el valor cero obtenido en el primer paso de esta traza. Por ello, ahora pasa a tener valor 2 ($\texttt{local}_A$=2).
8. A pasa el valor de su variable local al atributo `c.count`, `c.count=`$\texttt{local}_A$, con lo que `c.count=2`.

Es decir, se habría ejecutado la secuencia formada por los dos métodos que incrementaban el contador. El valor esperado como resultado era 5, pero en lugar de él se ha obtenido un 2. Esto se debe a que el hilo B ejecutó el método cuando el estado del objeto compartido no era consistente. Debido a ello, sus instrucciones no tuvieron el efecto esperado y el incremento que debía aplicar se perdió. El resultado de esta ejecución no ha sido determinista. Una ejecución determinista de esas dos sentencias siempre habría retornado un 5 como valor final para el atributo `c.count`. En este ejemplo, el intercalado de las instrucciones máquina utilizado ha roto el determinismo, debido a la intervención del planificador expulsando a A cuando se encontraba en un estado inconsistente. Cuando se da una situación como ésta en la que se pierde el determinismo y se generan resultados incorrectos se dice que ha habido una *condición de carrera* o una *interferencia* entre las ejecuciones de los diferentes hilos.

La falta de determinismo (esto es, la existencia de condiciones de carrera) debería evitarse. Por ello, se debe tener especial cuidado a la hora de acceder y utilizar un objeto compartido en un programa concurrente.

Como consecuencia de esto, para garantizar el determinismo en un programa concurrente se debería asegurar que:

- Las acciones ejecutadas por un hilo en cada ráfaga de procesador siempre dejen a todos los objetos compartidos que hayan sido utilizados durante el intervalo en un estado consistente.

  Con ello, los estados intermedios de estos objetos compartidos no serían observados por otros hilos.

Pero esto no puede garantizarlo el sistema operativo. Debe ser el programador quien lo fuerce, utilizando para ello algunos mecanismos de sincronización. Esos mecanismos de sincronización deberían garantizar que cualquier intercalado de las operaciones a ejecutar por los diferentes hilos del programa fuera correcto.

Para considerar que un programa concurrente es correcto se suelen exigir dos tipos de propiedades:

- *Seguridad*[Lam77]: No puede ocurrir nada "malo" (es decir, erróneo o incorrecto) durante la ejecución. Para que se respete la seguridad, se suelen exigir dos condiciones:
  - *Exclusión mutua*[Dij65]: Cuando un hilo o actividad esté ejecutando el código de un objeto compartido (es decir, alguno de sus métodos), ningún otro hilo o actividad podrá estar ejecutando un método de ese mismo objeto.
  - *Ausencia de interbloqueos*[CES71]: Los hilos que accedan a los recursos de una aplicación concurrente (o del sistema en que ésta se ejecute) no podrán quedar esperándose mutuamente evitando así el avance de todos ellos.
- *Vivacidad* [Lam77, OL82]: En algún momento ocurrirá algo "bueno"; o lo que es lo mismo, en algún momento el programa proporcionará una salida y dicha salida respetará la especificación del problema a resolver.

  Para que se dé la propiedad de vivacidad se suele asumir que se dan estas dos condiciones:
  - *Progreso*: Todo servicio solicitado se completa en algún momento. Es decir, no podrá quedar ninguna petición o método iniciados pendiente de su compleción durante un tiempo ilimitado.

- *Equidad*: Todo hilo preparado pasará en algún momento a ejecución. Esto dependerá del planificador utilizado en el sistema, pero se asumirá que éste será equitativo, evitando así que se retrase indefinidamente la entrada de un hilo en algún procesador. Por ejemplo, los planificadores Round-Robin son equitativos y la mayoría de los sistemas operativos modernos están basados en ese algoritmo.

Ya que la equidad depende del planificador utilizado por el sistema operativo y se asumirá que está garantizada, y que el progreso se dará como resultado de la ausencia de interbloqueos (y esta última será analizada en la unidad 4, por lo que no incidiremos en ella en este momento), la única condición que queda pendiente es la garantía de exclusión mutua en el acceso a recursos compartidos. Para proporcionar tal garantía se tendrá que utilizar mecanismos de sincronización de tal manera que se evite que los efectos de un método cuyas instrucciones generen estados intermedios inconsistentes puedan ser observados por otros hilos concurrentes. Cada una de las secuencias de instrucciones de este tipo (que generen estados intermedios inconsistentes) será una *sección crítica* y su estudio se analiza seguidamente.

2.6 Sección crítica

Se dice que una *sección* de código es *crítica* cuando su uso pueda generar condiciones de carrera. Debe recordarse que una *condición de carrera* se da cuando un hilo genere estados inconsistentes que puedan ser observados por otros hilos provocando una pérdida de determinismo o, lo que es lo mismo, un resultado incorrecto que no sea coherente con todas las sentencias ejecutadas hasta ese momento por dicho conjunto de hilos. De manera general, debería considerarse como sección crítica toda secuencia de sentencias en la que se acceda a variables u objetos compartidos por más de un hilo.

Obsérvese que el acceso a objetos locales que solo pueden ser utilizados por su hilo propietario jamás podría ocasionar una pérdida de determinismo. De igual manera, cuando se esté utilizando algún objeto compartido por múltiples hilos pero cuyo valor actual solo pueda ser leído por todos ellos (objetos inmutables) tampoco se llegará a ocasionar ningún problema. Por tanto, en estos dos casos no habrá una sección crítica.

Para proteger adecuadamente una sección crítica se tendrá que utilizar algún mecanismo de sincronización que garantice que toda la sección pueda considerarse una única acción atómica. Para ello, cuando algún hilo inicie su ejecución se evitará que ningún otro pueda ejecutar esa misma sección crítica. Con ello se está garantizando la condición de seguridad conocida como exclusión mutua y así se evitará que haya condiciones de carrera.

Como resultado se obtendrá un código que será "*thread-safe*" (esto es, seguro aunque haya múltiples hilos): la propiedad de seguridad estará garantizada aunque el programa inicie múltiples hilos y todos ellos se ejecuten activamente. Los mecanismos de sincronización ya se encargarán de asegurar que no haya problemas en esas situaciones, suspendiendo temporalmente a aquellos hilos que pudieran romper la exclusión mutua.

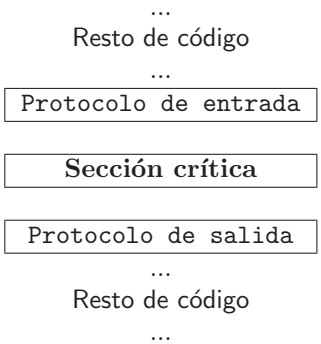

Figura 2.5: Protocolos de entrada y salida a una sección crítica.

Para que un determinado fragmento de código que sea una sección crítica pueda protegerse de manera adecuada mediante mecanismos de sincronización, se debe estructurar encerrando tal sección crítica entre un *protocolo de entrada* y un *protocolo de salida* tal como muestra la figura 2.5. El *protocolo de entrada* será un fragmento de código ubicado justo antes de la sección crítica en el que se regulará qué hilo podrá pasar en cada momento para iniciar la ejecución de la sección crítica. En caso de que la sección ya esté ocupada por algún hilo este protocolo de entrada evitará que otros candidatos puedan superar dicho protocolo de entrada, obligándoles a suspenderse o a que ejecuten de manera cíclica las instrucciones del protocolo. A su vez, el *protocolo de salida* es un segundo fragmento de código ubicado tras la sección crítica en la que se registrará la salida del hilo que esté ejecutando la sección permitiendo entonces que alguno de los demás hilos candidatos que hayan llegado al protocolo de entrada pueda entrar en la sección. En caso de que no hubiese ningún hilo en el protocolo de entrada, éste recordaría que la sección crítica está actualmente vacía permitiendo así la entrada del primer hilo que llegue posteriormente a solicitar la entrada en la sección.

Toda solución correcta al problema de la sección crítica (es decir, que garantice ejecuciones correctas de este tipo de secciones) constará de los protocolos de entrada y salida necesarios para cumplir estas tres propiedades:

- *Exclusión mutua*: Es una propiedad de seguridad que fuerza a que no haya más de un hilo ejecutando simultáneamente la sección crítica. Obsérvese que

las incoherencias surgen precisamente cuando hay dos o más hilos ejecutando a la vez un acceso sobre un objeto compartido. Si se fuerza a que solo haya un hilo como máximo ejecutando la sección, dicho problema desaparecerá.

- *Progreso*: Cuando la sección crítica quede libre, la decisión sobre qué nuevo hilo podrá ejecutar la sección no deberá retrasarse indefinidamente y dicha decisión solo podrá considerar a los hilos que hayan llegado al correspondiente protocolo de entrada.
- *Espera limitada*: Es una propiedad de equidad. Si un hilo A llega al protocolo de entrada, deberá acotarse el número de veces que otros hilos accedan a la sección mientras A permanezca esperando en dicho protocolo.

Se han llegado a dar múltiples soluciones que cumplen esas tres propiedades. Para clasificarlas se suele utilizar como criterio qué elemento del sistema proporciona el soporte necesario para implantar el mecanismo de sincronización utilizado en tales protocolos. Existen tres posibles fuentes para dicho soporte:

- El hardware: Un ejemplo de este tipo se da cuando el mecanismo de sincronización utilizado está basado en la inhabilitación de interrupciones. De esta manera no se pueden generar temporalmente las interrupciones hardware (hasta que se vuelvan a habilitar explícitamente), garantizando que cierta secuencia de código no pueda interrumpirse y de ese modo ningún otro hilo podrá "expulsar" al que ejecute tal secuencia. En este caso, la instrucción necesaria para inhabilitar las interrupciones formará el protocolo de entrada, mientras que la instrucción que las vuelve a habilitar será el protocolo de salida. Debe observarse que ambas instrucciones son privilegiadas y, por tanto, solo pueden ser utilizadas dentro del núcleo del sistema operativo. Por ello, esta solución únicamente es válida para proteger secciones críticas existentes dentro del código del núcleo del sistema operativo pero no para programas que deban ser ejecutados a nivel de usuario.

  Otras soluciones basadas en el hardware implantan los protocolos de entrada y salida mediante instrucciones que intercambian el contenido de diferentes posiciones de la memoria principal de manera atómica. No vamos a entrar a explicar estas variantes, pues no son sencillas y no siempre consiguen cumplir las tres propiedades de corrección mencionadas anteriormente.

- El sistema operativo: El propio sistema operativo puede facilitar cierto conjunto de llamadas al sistema para manejar ciertos mecanismos de sincronización. Los semáforos serían un buen ejemplo. Un *semáforo* [Dij68] es un contador entero cuyo valor puede ser modificado mediante dos operaciones. La operación `P()` permite decrementar en una unidad el valor del contador, suspendiendo al hilo invocador cuando el resultado sea negativo. Por su parte, la operación `V()` incrementa en una unidad el valor del contador, reactivando a uno de los hilos suspendidos en caso de que el resultado sea

negativo o cero. Con esta herramienta, el protocolo de entrada consistiría en una invocación sobre la operación `P()` de un semáforo cuyo valor inicial fuera 1, mientras que el protocolo de salida estaría formado por una invocación sobre la operación `V()` de ese mismo semáforo.

Existen otros mecanismos de sincronización similares gestionados por el sistema operativo. Algunos ejemplos son los *mutex* y las *condiciones* de la interfaz POSIX.

- El lenguaje de programación: En algunos casos el lenguaje de programación puede construir herramientas de mayor nivel de abstracción que proporcionan una gestión mucho más sencilla al programador, combinando de manera adecuada los mecanismos proporcionados por el sistema operativo. Un ejemplo clásico de este tipo son los *monitores* [Bri73] capaces de garantizar la exclusión mutua en el acceso a cualquier método de una determinada clase, así como sincronización condicional. En caso de utilizar una herramienta de este tipo, el programador ya no tendrá que preocuparse por la implantación de los protocolos de entrada y salida de una sección crítica pues el propio lenguaje de programación se encarga de gestionar dichos protocolos de manera transparente, sin que el programador realice ningún esfuerzo ni adopte ninguna precaución. En la Unidad 3 se estudiará el uso de monitores.

Un mecanismo de sincronización que puede estar soportado por un sistema operativo o por un lenguaje de programación es el "*lock*". Mediante él se puede dar una buena solución al problema de la sección crítica, como se explica en la próxima sección.

2.6.1 Gestión mediante *locks*

Un "*lock*" es un objeto con dos posibles estados (abierto o cerrado) y dos operaciones (abrir o cerrar el lock). Al crear el lock, éste se encontrará inicialmente abierto. Las operaciones de cierre y apertura tienen el siguiente comportamiento:

- *Cerrar*: Si el lock se encontraba abierto, el hilo invocador lo cerrará y podrá seguir ejecutándose sin realizar ninguna espera. Si el lock estuviese cerrado y fue cerrado por el propio hilo que invoque a esta operación, la operación de cierre no tendrá ningún efecto. Sin embargo, si el lock se encontrase cerrado pero hubiese sido cerrado por otro hilo, entonces el hilo invocador se suspenderá (en el caso de Java, pasaría a estado "*Blocked*").

- *Abrir*: Si el lock ya estuviese abierto o estuviera cerrado pero dicho cierre hubiese sido provocado por otro hilo, esta llamada no tendría ningún efecto. Por el contrario, si el lock estuviese cerrado por el propio hilo invocador, entonces:

- Si no hubiese ningún hilo suspendido por una operación de cierre efectuada cuando el lock ya estaba cerrado, el lock quedaría abierto.
- Si hubiese algún hilo suspendido, se selecciona a uno de ellos y se reactiva, quedando el lock cerrado por tal hilo.

  La decisión sobre qué hilo será reactivado depende del criterio que adopte quien haya implantado estos locks, pero normalmente cumplirá la propiedad de *espera limitada* explicada anteriormente.

A la hora de utilizar un lock para gestionar una sección crítica, el protocolo de entrada se reducirá al uso de la operación de cierre sobre el lock, mientras que el protocolo de salida será una invocación de la operación de apertura del lock. Así, ambos protocolos tienen una implantación muy sencilla, aunque todavía no es automática pues el programador debe recordar dónde deben implantarse los protocolos de entrada y salida y debe llevar cuidado para no confundir qué operación debe emplear en cada protocolo.

Con esta solución se logra convertir una sección crítica en una acción atómica. Aunque el planificador expulsara al hilo dentro de la sección crítica, ningún otro hilo podría iniciar la ejecución de esa misma sección, pues la operación de cierre del lock que implanta su protocolo de entrada evitaría que otros hilos consiguieran entrar en la sección. Por ello, la sección crítica se estaría ejecutando en exclusión mutua, respetando el principio de atomicidad: toda la sección se considerará indivisible y cuando un hilo inicie la ejecución de la sección ningún otro hilo podrá ejecutar esa misma sección al mismo tiempo. Mientras él no finalice su sección crítica ningún otro hilo podrá acceder a ella.

De esta manera se garantiza la actualización fiable de objetos compartidos, pues:

- La sección donde se realice la actualización será una sección crítica y estará libre de condiciones de carrera y de la corrupción de dicho estado compartido.
- Sólo resultarán visibles aquellos estados que sean coherentes. Mientras no lleguen a serlo, la exclusión mutua evitará que resulten visibles. Esto es lo que evita la corrupción del estado del objeto compartido.
- Se asegura así que todo hilo acceda al valor más reciente de cada objeto compartido.

En Java, todos los objetos poseen un lock asociado implícito. Por ser implícito no será necesario declararlo (no es otro objeto ubicado dentro del que deseemos proteger, es él mismo) y además el programador tampoco tendrá que preocuparse por utilizar explícitamente sus operaciones de apertura y cierre. Estas son automáticas.

No obstante, el lenguaje Java sí que permite que el programador decida qué métodos de un objeto formarán parte de la sección crítica asociada a dicho objeto. Para

que un método forme parte de la sección crítica deberá calificarse como "sincronizado", utilizando para ello la palabra reservada "`synchronized`" antes de indicar el tipo o clase del valor retornado por el método y después del calificador "`public`" o "`protected`".

Así, al iniciar la ejecución de un método sincronizado se cerrará automáticamente el lock asociado al objeto al que pertenezca dicho método. La operación de apertura también se invocará automáticamente al terminar la ejecución de cada método sincronizado. Con ello, todo método sincronizado se comportará como una acción atómica. Además, para un determinado objeto, todos sus métodos etiquetados con "*synchronized*" se ejecutan en exclusión mutua entre sí (pueden considerarse como múltiples partes en una misma sección crítica).

En general, convendrá calificar como sincronizados todos aquellos métodos de una determinada clase en los que se lea o modifique alguno de los atributos de la clase que puedan cambiar su valor durante la ejecución del programa. Es decir, los únicos métodos que no sería necesario proteger de esta manera serían aquellos que únicamente consultaran (y retornaran) el valor de algún atributo constante.

Con esta ayuda, el contador no determinista presentado en la figura 2.4 podría transformarse en un contador determinista calificando sus dos métodos como sincronizados, tal como se observa en la figura 2.6. Con este último código sería imposible que se diera alguna condición de carrera por lo que ante cualquier secuencia de incrementos siempre se obtendría un valor coherente como resultado. Jamás se perderá una actualización.

```
public class Counter {
  private long count = 0;
  public void synchronized add(long x) {
    count += x;
  }
  public long synchronized getCount() {
    return count;
  }
}
...
Counter c = new Counter();
```

Figura 2.6: Contador determinista.

2.7 Sincronización condicional

El concepto de sección crítica evita interferencias en el acceso a objetos compartidos por múltiples hilos. La utilización de mecanismos que garanticen una solución correcta a la hora de implantar secciones críticas y que además admitan una utilización sencilla (como los locks implícitos de Java) es una excelente ayuda a la hora de desarrollar aplicaciones concurrentes. Sin embargo, esto no será siempre suficiente. En ocasiones se tendrá que suspender durante un determinado intervalo de tiempo a un hilo dentro de una sección crítica hasta que alguna condición relacionada con el valor de las variables protegidas por dicha sección se llegue a cumplir. En esos casos, no bastará con solucionar la sección crítica, sino habrá que implantar también soluciones para la sincronización condicional dentro de una sección crítica.

Al igual que ha ocurrido con las secciones críticas, interesaría diseñar algún mecanismo que pudiera utilizarse de manera implícita, permitiendo que el programador se despreocupe de los detalles necesarios para el desarrollo de una solución correcta para este problema. Así se podría simplificar en gran medida el desarrollo de aplicaciones concurrentes.

En la próxima unidad se describirá la gestión de los monitores, mediante los cuales es posible solucionar ambos problemas (sección crítica y sincronización condicional) simultáneamente de una manera transparente para el programador. Debido a esto no se describirá ninguna otra solución para realizar sincronización condicional.

2.8 Resumen

La programación concurrente implica que las distintas actividades que conforman una determina aplicación cooperen entre sí para resolver la tarea para la que fue diseñada dicha aplicación. Los tipos de actividades que podemos considerar son los procesos y los hilos de ejecución. Un proceso multihilo contiene varios flujos de control diferentes dentro del mismo espacio de direcciones. A modo de ejemplo, en esta unidad se ha analizado el ciclo de vida de los hilos Java, detallando los diferentes estados en los que pueden estar los hilos, así como los métodos que permiten transitar entre estados.

La cooperación entre actividades requiere de mecanismos de comunicación y de sincronización. Para la comunicación, existen dos mecanismos básicos: el uso de variables compartidas, en donde las actividades de la aplicación concurrente comparten un mismo espacio de direcciones; y el intercambio de mensajes, que permite comunicar tanto a hilos de un mismo proceso como a procesos distintos en una misma máquina o en máquinas diferentes. Respecto a la sincronización, existen dos tipos básicos: la exclusión mutua, en el que los grupos de sentencias que acce-

den a variables u objetos compartidos no pueden ser ejecutados simultáneamente por múltiples actividades; y la sincronización condicional, en la cual un hilo debe retrasar su ejecución hasta que se cumpla determinada condición.

Un hilo transforma su estado mediante secuencias de acciones atómicas que realizan transformaciones indivisibles. Toda transformación del estado de un objeto se iniciará desde un estado consistente y su objetivo será la obtención de otro estado consistente, aunque dicha transformación del estado puede requerir de una secuencia de sentencias (es decir, de múltiples acciones atómicas), dando lugar a múltiples estados intermedios que no sean coherentes.

Los programas concurrentes no siempre serán deterministas, pues habrá múltiples actividades ejecutándose simultáneamente y cada una con su propia secuencia de sentencias, que podrá ser intercalada por el planificador del sistema de manera arbitraria. Por ello, el acceso concurrente a datos compartidos puede dar lugar a incoherencia de los datos (interferencias y condiciones de carrera). Se produce una condición de carrera cuando la ejecución de un conjunto de operaciones concurrentes sobre una variable compartida deja a la variable en un estado inconsistente. Las interferencias entre hilos ocurren cuando dos operaciones (compuestas de múltiples pasos), ejecutándose en hilos diferentes, pero actuando sobre el mismo dato, se intercalan, de modo que las secuencias de estos pasos se entrecruzan, produciéndose un resultado no esperado.

Para considerar que un programa concurrente es correcto se exige que cumpla las propiedades de seguridad (dándose las condiciones de exclusión mutua y ausencia de interbloqueos), y de vivacidad (dándose las condiciones de progreso y equidad). Toda solución al problema de la sección crítica constará de los protocolos de entrada y salida necesarios que cumplan las propiedades de exclusión mutua, progreso y espera limitada. Para ello, existen tres fuentes que dan el soporte necesario a los mecanismos de sincronización requeridos: el hardware (ej., con la inhabilitación de interrupciones); el sistema operativo (que proporciona semáforos, mutex, etc.), y los lenguajes de programación concurrente (que proporcionan herramientas de mayor nivel de abstracción, como los monitores).

Los mecanismos analizados en esta unidad permiten resolver el problema de la exclusión mutua, pero no resuelven la sincronización condicional, en la que se requiere que los hilos se suspendan hasta que se cumpla una determinada condición. En la próxima unidad se analizará el concepto de monitor, que permite solucionar tanto la exclusión mutua como la sincronización condicional.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar las secciones de una aplicación que deban o puedan ser ejecutadas concurrentemente por diferentes hilos.

- Enumerar los problemas que pueden darse al acceder a recursos compartidos de manera concurrente.
- Describir el ciclo de vida de los hilos Java.
- Distinguir los estados consistentes de un objeto y sus acciones atómicas.
- Identificar los diferentes mecanismos de sincronización de hilos que proporcionan los sistemas operativos modernos.
- Identificar los problemas que puede originar un uso incorrecto de los mecanismos de sincronización.
- Describir las propiedades que garantizan la corrección de un programa concurrente.
- Distinguir las secciones críticas de una aplicación y protegerlas mediante los mecanismos adecuados. En concreto, haciendo uso de locks, a nivel general, y del lock implícito que ofrece Java.

Unidad 3

PRIMITIVAS DE SINCRONIZACIÓN

3.1 Introducción

Como ya se ha explicado en la unidad anterior, la programación concurrente se da cuando múltiples actividades cooperan dentro de una misma aplicación. Para que dicha cooperación sea posible, esas actividades tendrán que comunicarse y sincronizarse. Por tanto, habrá que diseñar algunos mecanismos que resuelvan tales necesidades de comunicación y sincronización. En el caso particular de la programación concurrente basada en hilos de ejecución dentro de un mismo proceso, cabría distinguir los siguientes niveles de soporte para tales mecanismos:

- Sin la participación del sistema operativo.

  Si el procesador fue diseñado sin tener en cuenta las necesidades de la programación concurrente, solo se podrán utilizar modelos de programación basados en *espera activa*. La espera activa consiste en la utilización de bucles en los que se comprueba cierta condición, iterando continuamente (el cuerpo del bucle no tiene por qué contener instrucciones) mientras dicha condición no se satisfaga. Con esta construcción se implanta la sincronización, pues la actividad no prosigue mientras la condición no se cumpla, así como la comunicación, pues la condición está construida utilizando variables compartidas con otros hilos y no se cumplirá mientras esos hilos adicionales no modifiquen el valor de tales variables.

  La espera activa no resulta eficiente pues implanta la sincronización obligando a que los hilos involucrados en ella estén activos (es decir, preparados

o en ejecución). De esa manera se malgasta el tiempo de procesador, pues aquellos hilos que incumplan sus respectivas condiciones estarán ejecutando sus bucles, impidiendo en algunos casos que los hilos que podrían reactivarlos avancen.

Un segundo tipo de mecanismo de sincronización consiste en la inhabilitación de interrupciones mientras un hilo esté ejecutando su sección crítica. De esa manera se garantiza que otros hilos no interrumpan la ejecución de dicha sección, pudiendo así ejecutarla completamente en una misma ráfaga de procesador sin que otros hilos intervengan entre tanto. Este mecanismo no se puede utilizar en los programas de usuario, ya que las instrucciones necesarias como protocolo de entrada (inhabilitación de interrupciones) y como protocolo de salida (habilitación de interrupciones) para acceder a dicha sección crítica serían privilegiadas y solo se podrían ejecutar dentro del núcleo del sistema operativo. Además, para que un sistema funcione de manera aceptable las interrupciones no se podrán inhabilitar durante intervalos arbitrariamente largos. Por ello, esta solución no es útil pues ningún sistema operativo ofrecerá las llamadas necesarias para emplearla en los programas de usuario. Solo tiene sentido para proteger algunas secciones críticas (formadas por unas pocas instrucciones, requiriendo así un intervalo muy breve para ejecutarlas por completo) en el núcleo del propio sistema operativo.

- Soporte proporcionado directamente por el sistema operativo.

  Para eliminar la espera activa se necesita que el sistema operativo facilite algún mecanismo capaz de suspender a aquellos hilos que no puedan o no deban avanzar mientras no se cumpla una determinada condición. Cuando finalmente tal condición llegue a cumplirse, deberá utilizarse otra llamada que reactive a los hilos que estuviesen esperando por tal condición.

  Una primera solución de este tipo está basada en el uso de *semáforos* (ya explicada en la unidad anterior). Los *locks* también ofrecen un soporte similar.

  Los semáforos no requieren espera activa. Cuando un hilo deba acceder a una determinada sección crítica, utilizará la operación `P()` para ello. Si la sección ya está siendo ejecutada por otro hilo, dicha invocación suspenderá al hilo invocador. Cuando el hilo que esté dentro de la sección salga de ella, lo hará llamando a la operación `V()` del mismo semáforo (que es la que implanta el protocolo de salida). Como resultado de dicha invocación, uno de los hilos previamente suspendidos al llamar a `P()` se reactivará. En lugar de iterar en un bucle que consulta continuamente una condición construida con variables compartidas, el semáforo suspende o reactiva a los hilos según el estado de la sección crítica que protejan. Con ello, el resultado será mucho más eficiente pues únicamente permanecerán activos aquellos hilos que puedan avanzar en su ejecución.

A pesar de eliminar el uso de esperas activas, los mecanismos de sincronización facilitados directamente por el sistema operativo todavía plantean el problema de que no automatizan el uso de tales mecanismos. El programador sigue siendo el responsable del diseño de las soluciones, decidiendo cuántos y qué semáforos (o locks, u otros mecanismos) habrá que utilizar y dónde hacerlo. Si no se sigue una estrategia clara de diseño que permita identificar fácilmente cuáles son las secciones críticas que habrá que proteger en cada hilo, será relativamente fácil equivocarse y dejar alguna sin protección. Por ello, debería buscarse otra solución que automatice la sincronización. Para llegar a una solución de ese tipo se requiere que el propio lenguaje de programación realice gran parte de la gestión relacionada con dicha sincronización.

- Soporte proporcionado por el lenguaje de programación.

  Algunos lenguajes de programación están diseñados asumiendo que con ellos se programarán aplicaciones concurrentes. Por ello, disponen de algunas construcciones (disponibles como tipos o clases específicos) que automatizan el uso de herramientas de sincronización a la hora de proteger el acceso sobre objetos o variables compartidos entre múltiples hilos de ejecución. Uno de los primeros lenguajes de programación de este tipo fue *Pascal concurrente* (*Concurrent Pascal*) en el que la construcción especialmente diseñada para automatizar la sincronización se llamó *monitor*. Hoy en día un buen número de lenguajes soportan programación concurrente: Java, C#, ...

  Estos lenguajes facilitan algunos tipos, clases o sentencias especiales mediante los que se automatizará la protección de secciones críticas, proporcionando exclusión mutua, progreso y espera limitada en el acceso a tales secciones. En algunos casos, también se ofrecerá soporte para sincronización condicional dentro de las secciones críticas.

  De manera general, un lenguaje de programación con soporte para concurrencia permite:

  - Crear múltiples hilos de ejecución dentro de un mismo proceso. Es el caso de la clase `Thread` en Java o de las tareas (tasks) en Ada.
  - Automatizar el uso de herramientas de sincronización a la hora de proteger una sección crítica. Ni siquiera se necesita especificar qué secciones de código tendrán esa consideración. Basta con declarar los objetos o variables que se vayan a compartir entre múltiples hilos de cierta manera. Así el programador ya no debe preocuparse por decidir dónde tendría que declarar y utilizar los *locks* o los *semáforos* para proteger el acceso sobre esos recursos compartidos.

    Por ejemplo, en Java esa especificación especial para los objetos compartidos consiste en añadir el calificador `synchronized` en todos los métodos de aquellas clases puedan ser utilizadas por múltiples hilos de ejecución.

3.2 Monitores

Un *monitor* es una estructura de datos (en la que podrán definirse todos los atributos privados que se necesite, constituyendo su estado) que ofrece un conjunto de operaciones públicas (similares a los métodos que constituyen la interfaz de una clase en un lenguaje de programación orientado a objetos). El monitor garantiza que todas sus operaciones se ejecutarán en exclusión mutua: si algún hilo inicia la ejecución de alguna operación no podrá haber ningún otro hilo activo ejecutando código de ese mismo monitor. Además de la ejecución en exclusión mutua, los monitores ofrecen una segunda construcción interna: las condiciones. Cada *condición* permite suspender a hilos dentro del monitor liberando el acceso a éste. Los hilos suspendidos podrán reactivarse cuando otros hilos modifiquen el estado del monitor y les notifiquen. No obstante, dicha reactivación debe seguir respetando la exclusión mutua: solo uno de los hilos activos podrá ejecutarse, los demás tendrán que esperar hasta que ése salga del monitor.

3.2.1 Motivación

Aunque ya se ha explicado de manera general qué es un monitor, merece la pena estudiar con más detenimiento por qué fue necesario el diseño y uso de monitores. Para ello se utilizará un ejemplo sencillo: una aplicación que simula el comportamiento de una colonia de hormigas. Este ejemplo permitirá identificar los problemas que surgen al utilizar algunas herramientas de sincronización como los *locks*, obligando al programador a gestionar de manera explícita la sincronización que garantice un buen comportamiento de cierto conjunto de hilos.

En este modelo de simulación se define un territorio como una matriz de celdas. Cada celda podrá estar ocupada o libre. Cada hormiga se modela como un hilo de ejecución, siguiendo ciertas pautas a la hora de definir el recorrido que la hormiga realizará. El territorio debe cumplir como restricción que solo haya una hormiga como máximo en cada celda.

El desplazamiento de las hormigas se realiza eligiendo una de las celdas vecinas a la que actualmente esté ocupando la hormiga. El programa comprobará que la celda escogida esté libre. En caso de que sea así, la hormiga se desplaza. Por el contrario, si la celda ya estuviese ocupada por otra hormiga, la hormiga que pretendía desplazarse esperará a que la celda quede libre.

Para asegurar que no haya condiciones de carrera, se podría sincronizar el desplazamiento de las hormigas utilizando un lock global. Con ello, se podría modelar el territorio como:

```
boolean ocupada[N][N];
```

y de esa manera, el pseudocódigo necesario para desplazar una hormiga de la posición (x,y) a la posición (x',y') sería:

```
cerrar lock;
// Si ocupada[x'][y'], entonces habrá que esperar
// a que dicha posición quede libre.
ocupada[x'][y'] = true;
ocupada[x][y] = false;
abrir lock;
```

Con ello el lock garantiza que las modificaciones sobre la matriz `ocupada` se realicen en exclusión mutua. Es una buena solución para proteger una sección crítica y esas dos sentencias incluidas entre las operaciones del lock lo son. Obsérvese que múltiples hormigas podrían iniciar sus desplazamientos a la vez y debe protegerse el contenido de la matriz para que no quede en un estado incoherente al realizar múltiples modificaciones concurrentes. Sin embargo, en este ejemplo hay un detalle adicional que no está todavía resuelto utilizando el lock: si la celda a la que pretendía trasladarse esa hormiga estuviese ocupada, el desplazamiento debería suspenderse hasta que la celda destino quedase libre. Eso se menciona en el comentario incluido en el pseudocódigo anterior, pero no está implantado.

Un primer boceto de código que aparentemente implanta dicha espera sería el siguiente:

```
cerrar lock;
while ( ocupada[x'][y'] ) {};
ocupada[x'][y'] = true;
ocupada[x][y] = false;
abrir lock;
```

En ese fragmento se utiliza espera activa: se comprueba si la celda destino está ocupada y, mientras lo esté, se seguirá iterando. Solo se abandonará dicho bucle cuando la celda destino esté libre.

Dicho fragmento consigue bloquear a la hormiga que ha iniciado el desplazamiento. Eso es parte de lo que se pretendía y sí que llega a implantarse. Sin embargo, esta solución no es eficiente y tampoco correcta, por los siguientes motivos:

- Resulta ineficiente porque la hormiga que inicia su desplazamiento no se suspende al ver que la celda destino está ocupada. En lugar de ello, sigue comprobando el estado de dicha celda, esperando que cambie. Es una *espera activa* y ya se dijo anteriormente que ese tipo de sincronización rara vez será recomendable.

- Es incorrecta porque dicha espera activa se realiza dentro de una sección crítica. Al haberlo implantado de esa manera, la sección crítica jamás queda libre y ninguna otra hormiga podrá entrar en su respectiva sección crítica para modificar el estado de la celda destino. De hecho, ninguna otra hormiga podrá llegar a desplazarse, tanto si para ello necesita utilizar alguna de las dos celdas implicadas en el desplazamiento de la hormiga que realiza la espera activa como si necesitase utilizar cualquier otra celda. La hormiga que está iterando en su espera activa ha cerrado el lock y ninguna otra hormiga podrá superar dicha sentencia de cierre. Todas las que lleguen posteriormente a esa sección de código se suspenderán tratando de cerrar el lock.

  Como resultado de todo esto, el simulador que utilizase ese código se quedaría "colgado" tan pronto como alguna de las hormigas iniciara una espera activa. Obviamente, eso no es correcto.

Está claro que habrá que buscar otros tipos de soluciones. La combinación de espera activa para realizar una sincronización condicional y el uso de locks para garantizar exclusión mutua no funciona.

Convendría tener una primitiva que pudiera combinar ambos tipos de sincronización: exclusión mutua y sincronización condicional. Además, interesaría que dicha primitiva se utilizase de manera implícita al llamar a los métodos de los objetos utilizados por múltiples hilos (como es el caso de la matriz `ocupada` en este ejemplo). Eso es precisamente un monitor. Con los monitores el programador ya no tendrá que preocuparse por este tipo de problemas.

La programación orientada a objetos es una herramienta válida para lograr el uso implícito de los mecanismos de sincronización. Su soporte se implanta sobre los servicios ofrecidos por el sistema operativo y puede utilizar los mecanismos de sincronización facilitados por éste, como por ejemplo, los semáforos o los locks.

Un lenguaje de programación orientado a objetos (como Java o C#) ofrece *clases*. La clase separa la interfaz de su implantación: quien desarrolle la clase tendrá que escribir y conocer tanto los atributos que definen el estado de cada instancia (objeto) de esa clase como el código de cada una de las operaciones que aparezcan en su interfaz, pero quien deba utilizar dicha interfaz no tendrá que preocuparse por cómo esté implantada. No le importará qué código se ha empleado ni qué atributos definen su estado. Basta con que conozca la interfaz y la funcionalidad de cada uno de los componentes que aparezcan en tal interfaz.

La programación orientada a objetos ha demostrado su utilidad:

- Permite realizar un diseño más cercano al problema que se pretende resolver. Basta con identificar qué elementos intervienen y modelarlos como clases que después habrá que desarrollar. El diseño resultante es así más fácil de seguir pues permite una representación más cercana a la realidad.

- El diseño y el programa resultantes son más fáciles de mantener. Resulta sencillo identificar qué clases habrá que modificar cuando varíe ligeramente la especificación del problema.
- Resulta más fácil documentar el código resultante y generar código claro y legible, pues tanto las clases como sus métodos podrán utilizar identificadores que se ajusten fielmente al campo del problema que se deba resolver.
- Como consecuencia de los dos puntos anteriores la depuración también se simplifica pues el programador identificará rápidamente qué objetos y métodos pueden ser la causa de algún comportamiento no esperado en el programa desarrollado.

Por todo esto, resulta conveniente integrar la gestión de la sincronización en un lenguaje de programación orientado a objetos. El *run-time* del lenguaje seleccionado permitirá utilizar los mecanismos de sincronización necesarios para garantizar exclusión mutua en el acceso a objetos compartidos y el soporte necesario para la sincronización condicional, requiriendo un esfuerzo mínimo por parte del programador.

3.2.2 Definición

Un *monitor* (en un lenguaje de programación orientado a objetos) es una clase que permite definir objetos que se compartirán de manera segura entre múltiples hilos. Estas clases:

- Garantizan que sus métodos se ejecuten en exclusión mutua.

  Disponen de una cola de entrada donde quedarán esperando aquellos hilos que intenten entrar en el monitor cuando ya haya otro hilo ejecutando alguno de sus métodos. Al proporcionar esa exclusión mutua se impide que haya condiciones de carrera dentro del monitor.

- Resuelven la sincronización condicional.

  Para ello se permite definir colas de espera dentro del monitor, utilizando el tipo `Condition`.

  Por ejemplo, si el monitor modelara un buffer de capacidad limitada, se podría declarar:

  ```
  Condition noVacío, noLleno;
  ```

  para definir dos condiciones en las que suspender a los hilos consumidores mientras el buffer estuviese vacío (y esperan en la condición `noVacío`) o a los hilos productores mientras el buffer estuviese lleno (esperando en la condición `noLleno`).

Así, si un hilo consumidor invocara el método necesario para extraer un elemento del buffer y éste se encontrara sin elementos, el hilo debería utilizar el método `wait()` sobre la condición `noVacío`:

```
noVacío.wait();
```

y con ello se suspendería en esa sentencia, dejando libre el monitor.

Más pronto o más tarde llegaría algún hilo productor que insertaría un nuevo elemento en el buffer. Cuando eso ocurriera, el productor debería invocar la sentencia que sigue para reactivar al consumidor previamente suspendido:

```
noVacío.notify();
```

Es decir, se reactivaría al consumidor notificándole que la condición por la que esperaba (que el buffer dejase de estar vacío) se está dando en el estado actual del monitor.

En caso de que no hubiese llegado en primer lugar un hilo consumidor sino un productor, cuando el productor invocase a `noVacío.notify()` no habría ningún hilo suspendido en dicha condición. En ese caso el `notify()` no tiene ningún efecto. Es decir, esa notificación se pierde y no permitirá que quien posteriormente llame a `noVacío.wait()` supere tal invocación sin suspenderse. Esto se debe a que las notificaciones sobre condiciones vacías no son recordadas.

Como ya se ha comentado previamente, la gestión de la exclusión mutua se realiza de manera implícita, evitando que el programador deba preocuparse de ese problema. En la sincronización condicional no se puede llegar a ese mismo nivel de automatización, pues el programador deberá decidir dónde y cómo se utilizan las condiciones dentro de los métodos del monitor. Sin embargo, sí que se consigue integrar de manera cómoda la gestión de las condiciones con la garantía de exclusión mutua, como se verá posteriormente.

3.2.3 Ejemplos

Para ilustrar cómo puede implantarse un monitor, la figura 3.1 muestra un primer ejemplo. Su código emplea una notación que no se corresponde con la de ningún lenguaje de programación real, aunque sigue una sintaxis inspirada en Java. Representa un esquema básico para implantar un buffer de capacidad limitada que pueda ser utilizado concurrentemente por hilos productores y consumidores de sus elementos. Aquellos hilos que invoquen el método `put()` tendrán el rol de productores e insertarán un elemento en el buffer con cada invocación de dicho método. Por su parte, quienes invoquen `get()` serán consumidores y extraerán un elemento en cada invocación.

Para llegar a ese esquema básico se debe seguir esta guía de diseño:

```
Monitor Buffer {
  ...          // Atributos para implantar el buffer.
  Condition noLleno, noVacío;
  int elems = 0;

  entry void put(int x) {
    if (elems==N)
      noLleno.wait();
    ... // Código para insertar el elemento.
    elems++;
    noVacío.notify();
  }

  entry int get() {
    if (elems==0)
      noVacío.wait();
    ... // Código para extraer el elemento.
    elems--;
    noLleno.notify();
    return elem;
  }

  entry int numElems() {
    return elems;
  }
}
```

Figura 3.1: Ejemplo de monitor.

1. Definir la interfaz (métodos) e implantación (código básico y atributos) como en cualquier otra clase.

2. Rellenar una tabla en la que se especifique para cada método de la interfaz:
   - Su perfil: identificador del método, argumentos y tipo de retorno.
   - En qué casos (estados del monitor) el hilo invocador tendrá que esperar.
   - En caso de que ese método modifique el estado del monitor, a qué hilo o hilos que se encuentren en espera habrá que notificar.

   Para el ejemplo de la figura 3.1, el contenido de esa tabla sería el que se muestra en la tabla 3.1.

Perfil	Espera cuando	Notifica a
`int get()`	Buffer vacío	Quien espere por buffer lleno
`void put(int e)`	Buffer lleno	Quien espere por buffer vacío
`int numElems()`	—	—

Tabla 3.1: Tabla de métodos.

3. Definir una cola de espera (variable de tipo `Condition`) por cada caso de espera que aparezca en las filas de la tabla del paso anterior.

   En este ejemplo, se necesitarían dos colas de este tipo. Una para mantener a los hilos consumidores que encuentren un buffer vacío (y deben esperar en la cola de la condición `noVacío` a que el buffer no esté vacío) y otra para mantener a los hilos productores que encuentren un buffer lleno (y por tanto se quedan esperando a que el buffer no esté lleno en la cola de la condición `noLleno`).

El código resultante es el que se ha mostrado en la figura 3.1. En él se ha utilizado el calificador `entry` para indicar qué métodos del monitor serán públicos y garantizarán exclusión mutua en su ejecución. Esa sintaxis fue una de las utilizadas en los primeros monitores, integrados en el lenguaje Pascal concurrente.

Como segundo ejemplo se revisará el simulador de la colonia de hormigas presentado en la sección 3.2.1. El monitor que se necesitaría para implantar dicho simulador ofrecería una sola operación `desplaza()` mediante la cual una hormiga se desplazaría desde una posición `(x,y)` a otra posición contigua `(x',y')`. Para diseñar dicho monitor habrá que rellenar la tabla correspondiente, generando el resultado que se muestra en la tabla 3.2.

Perfil	**Espera cuando**	**Notifica a**
`void desplaza(int x, int y, int x', int y')`	Posición (x',y') ocupada.	Quien espere por posición ocupada.

Tabla 3.2: Tabla de métodos del monitor `Hormigas`.

En función de dicha tabla, el monitor necesitaría al menos un atributo de tipo `Condition` para suspender a aquellos hilos que encuentren su celda destino ocupada. Para conocer el estado de cada celda se necesitaría una matriz de booleanos manteniendo tal estado. Con ello, el monitor resultante sería el que aparece en la figura 3.2.

Sin embargo, este monitor todavía no garantiza el comportamiento esperado. Podría darse el caso de que hubiese múltiples hilos esperando en la condición `libre` y el `notify()` con el que finaliza el método `desplaza()` no liberaría a todos ellos. Simplemente escogería uno al azar.

Para refinar esta solución se debería utilizar una matriz de colas de espera, con una componente por cada celda del territorio en lugar de una sola condición donde suspender a todos los hilos que encuentren alguna celda ocupada. La versión resultante se muestra en la figura 3.3. De esta manera solo se llegará a notificar a quien estuviera esperando por la liberación de la celda origen de nuestro desplazamiento. Con ello, el comportamiento ya será el esperado.

```
Monitor Hormigas {
  boolean ocupada[N][N]; // Estado de cada celda (inicialmente "false").
  Condition libre;

  entry void desplaza(int x, int y, int x', int y') {
    while (ocupada[x'][y'])
      libre.wait();      // Celda destino ocupada. Hay que esperar.
    ocupada[x][y]=false; // Liberamos la celda origen.
    ocupada[x'][y']=true;// Ocupamos la celda destino.
    libre.notify();      // Notificamos a los que esperen.
  }
}
```

Figura 3.2: Monitor del simulador.

```
Monitor Hormigas {
  boolean ocupada[N][N]; // Estado de cada celda (inicialmente "false").
  Condition libre[N][N];

  entry void desplaza(int x, int y, int x', int y') {
    while (ocupada[x'][y'])
      libre[x'][y'].wait();// Celda destino ocupada. Hay que esperar.
    ocupada[x][y]=false; // Liberamos la celda origen.
    ocupada[x'][y']=true;// Ocupamos la celda destino.
    libre[x][y].notify();// Notificamos a quien espere en la celda origen.
  }
}
```

Figura 3.3: Segundo monitor del simulador.

3.2.4 Monitores en Java

En los ejemplos de monitor que se han presentado hasta el momento, se ha utilizado una notación que no concuerda con ninguno de los lenguajes de programación existentes. En esta sección se presentará el soporte que ofrece el lenguaje Java para utilizar monitores.

En Java todo objeto posee una cola de entrada y una cola de espera (similar a las `Condition`) implícitas. Esta cola de espera implícita es la principal diferencia respecto a los monitores explicados hasta ahora. En un monitor "tradicional" habría tantas colas de espera como atributos de tipo `Condition` haya declarado el programador. En un monitor Java únicamente podrá haber una sola cola de espera (el programador no podrá declarar más) y, por ser implícita, no resulta necesario declararla.

Por su parte, la cola de entrada es equivalente a mantener un objeto *lock* que garantice la exclusión mutua en el uso del monitor. Aunque dicha cola de entrada sea implícita, el programador podrá decidir a qué métodos del objeto afectará, pues la exclusión mutua solo regulará el acceso sobre aquellos métodos que estén calificados como "`synchronized`". Esto es, el uso de los métodos calificados como sincronizados se traducirá en el cierre del lock implícito antes de su primera instrucción y la apertura de ese lock tras su última instrucción.

A su vez, los métodos siguientes permiten operar con la cola de espera:

- `wait()`: Suspende el hilo invocador en la cola de espera implícita del objeto/monitor. Con ello, el hilo espera (porque los atributos que mantienen el estado del monitor no cumplen la condición que necesitaba tal hilo) hasta que la condición se cumpla y otro hilo notifique tal hecho.
- `notify()`: El hilo actualmente activo en el monitor notifica (reactivándolo) a uno de los hilos que estaba esperando.
- `notifyAll()`: El hilo actualmente activo en el monitor notifica a todos los hilos que estaban esperando. Esto implica que todos ellos se reactivan de manera lógica, pero como el monitor debe garantizar exclusión mutua, la reactivación práctica se irá produciendo secuencialmente (a medida que los hilos ya reactivados hayan abandonado el monitor se irá reactivando el siguiente).

Java no obliga a que todos los objetos que se instancien en un determinado programa se comporten como monitores. Cualquier objeto Java tendrá su cola de entrada y cola de espera implícitas, pero puede ocurrir que el código de esa clase no respete las restricciones que imponen los monitores Java. Dichas restricciones son:

- Todos los atributos que definan el estado del objeto tendrán que declararse como privados.
- Todos los métodos públicos definidos en esa clase deberán calificarse como sincronizados (utilizando la palabra reservada `synchronized` tras el calificador `public` y antes del tipo retornado por el método).
- Utilizar los métodos implícitos (ofrecidos por la clase `Object`, de la que heredan todas las demás clases) `wait()`, `notify()` y `notifyAll()` para implantar la sincronización condicional.

Esta sincronización condicional planteará algunos problemas en los monitores Java. Esto se debe a que en cualquier monitor tradicional resultaba posible declarar tantos atributos `Condition` como fuese necesario. Sin embargo, en Java solo habrá una cola de espera implícita y, en principio, no se podrán declarar más.

Por ejemplo, en el monitor `Buffer` que aparecía en la figura 3.1 y que seguía la especificación tradicional se declararon dos atributos de tipo `Condition`. Uno de ellos permitía a los hilos consumidores esperar hasta que el buffer no estuviese vacío (`noVacío`), mientras que el otro permitía a los hilos productores esperar hasta que el buffer no estuviese lleno (`noLleno`). Si se debiera implantar un monitor con la misma funcionalidad empleando el lenguaje Java solo habría una única cola de espera y tanto los hilos productores como los hilos consumidores deberían suspenderse en ella. Por ello, cuando variase el estado del monitor no convendría utilizar el método `notify()` pues al reactivar a un solo hilo podría ocurrir que el hilo despertado no fuese del tipo que se pretendía reactivar (por ejemplo, en caso de querer reactivar a un productor podría reactivarse involuntariamente a un consumidor). Así, se tendría que utilizar el método `notifyAll()` para reactivarlos a todos. Además, se tendrían que reevaluar la condiciones que condujeron a la suspensión del hilo, para comprobar si el nuevo estado del monitor permite el avance del hilo reactivado o bien obliga a suspenderlo de nuevo. Una primera plantilla para implantar ese mismo monitor en Java se muestra en la figura 3.4.

Por tanto, de manera general las diferencias que plantea un monitor Java frente al modelo tradicional de monitor explicado hasta el momento para implantar la sincronización condicional serían:

- En Java solo habrá una cola de espera en cada monitor. En un monitor tradicional se podían declarar tantas colas de espera como fuese necesario.
- Dicha cola de espera es implícita en Java. En los monitores tradicionales las declaraciones de las colas de espera debían ser explícitas, utilizando para ello el tipo `Condition`.
- En Java las notificaciones deberían reactivar a todos los hilos (con el método `notifyAll()` que estén en espera, pues todos ellos comparten una misma cola de espera y no se puede prever a quién se despertaría en caso de utilizar el método `notify()`.
- En Java debe reevaluarse la condición que condujo a la espera en caso de que un hilo sea reactivado. Esto se debe a que las notificaciones se realizan sobre todos los hilos en espera y en la mayoría de los casos solo algunos de ellos podrían continuar.

Si se aplican estas reglas sobre el ejemplo del monitor que simulaba el comportamiento de una colonia de hormigas, ilustrado en las figuras 3.2 y 3.3 el resultado obtenido sería el que aparece en la figura 3.5. En él se observa que se pasa a utilizar el método `notifyAll()` para reactivar a los hilos suspendidos y que las condiciones de espera están incluidas como guardas de un bucle `while` en lugar de una sola sentencia `if`.

```
public class Buffer {
  ...          // Atributos para implantar el buffer.
  // Sólo habrá una "Condition" implícita, que tendrá que simular
  // a las Condition noLleno y noVacío originales.
  private int elems;

  public Buffer() { elems=0; }

  public synchronized void put(int x) {
    while (elems==N)
      try { wait();
      } catch (Exception e) { };
    ... // Código para insertar el elemento.
    elems++;
        // Reactivar a todos.
    notifyAll();
  }

  public synchronized int get() {
    while (elems==0)
      try { wait();
      } catch (Exception e) { };
    ... // Código para extraer el elemento.
    elems--;
        // Reactivar a todos.
    notifyAll();
    return elem;
  }

  public synchronized int numElems() {
    return elems;
  }
}
```

Figura 3.4: Monitor Buffer en Java.

3.3 Variantes

La sección 3.2.2 de esta unidad describió un esqueleto básico de lo que sería la sintaxis genérica de un monitor. Las piezas clave de este modelo genérico son:

- Los métodos públicos de un monitor están calificados con la palabra reservada `entry` y su uso siempre garantizará exclusión mutua. No puede haber ningún método público del monitor que no esté calificado como `entry`. No se puede romper la exclusión mutua al ejecutar el código de un monitor.
- Para implantar la sincronización condicional se podrán declarar múltiples atributos de tipo `Condition`. Sobre cada uno de estos atributos se podrán utilizar los métodos `wait()` y `notify()`.

```
public class Territorio {
  private boolean[][] ocupada; // Estado de cada celda.

  public Territorio(int N) {
    ocupada=new boolean[N][N];
    for (int i=0;i<N;i++)
      for (int j=0;j<N;j++)
        ocupada[i][j]=false; // Celdas inicializadas como libres.
  }

  public synchronized void desplaza(int x0, int y0, int x, int y) {
    while (ocupada[x][y])
      try {
        wait();          // Celda destino ocupada. Hay que esperar.
      }catch(Exception e) {};
    ocupada[x0][y0]=false;// Liberamos la celda origen.
    ocupada[x][y]=true;  // Ocupamos la celda destino.
    notifyAll();         // Notificamos a los que esperen. Solo deberían
                         // continuar aquellos hilos que esperasen a que la
                         // celda (x0,y0) quedara libre.
  }
}
```

Figura 3.5: Monitor Java para el simulador de la colonia de hormigas.

No existe ningún método `notifyAll()` en un monitor general. Ninguno de los lenguajes clásicos de la programación concurrente utilizó alguna operación de este tipo. Si en algún método de un monitor se necesitaba despertar a todos los hilos suspendidos en una determinada condición, entonces se empleaba una *reactivación en cascada*. Para ilustrar su uso se presenta un ejemplo en la figura 3.6.

```
monitor Barrier {
  Condition c;
  entry void await() {
    c.wait();
    c.notify();
  }
  entry void open() {
    c.notify();
  }
}
```

Figura 3.6: Ejemplo de monitor con reactivación en cascada.

Este ejemplo muestra un monitor `Barrier` que modela una barrera que suele estar cerrada, bloqueando el paso de los hilos que llamen a su método `await()`. Cuando otro hilo llame al método `open()`, la barrera se abre momentáneamente y reactiva a todos sus hilos suspendidos. Hecho esto, volverá a permanecer cerrada hasta

la siguiente llamada a `open()`. Para que se reactiven todos los hilos bloqueados basta con una sola llamada a `c.notify()` en la operación `open()`. Por su parte, cada uno de los hilos que sean reactivados por dicho `notify()` realizará otro `notify()` justo antes de abandonar el monitor. Con ello se reactiva al siguiente hilo suspendido, que a su vez reactivará al siguiente, y así sucesivamente hasta que se reactiven todos. No resulta necesario el uso de algún método `notifyAll()` en las condiciones de un monitor general.

Este esquema se puede extender fácilmente para incluir en un mismo monitor métodos que realicen reactivación en cascada y otros que solo reactiven a un único hilo. Para ello basta con utilizar algún atributo booleano que distinga entre ambos tipos de reactivación, como se ilustra en la figura 3.7.

```
monitor Barrier {
  Condition c;
  boolean onlyOne;

  entry void await() {
    c.wait();
    if (!onlyOne)   // Solo se notifica al siguiente en caso de
      c.notify();   // utilizar reactivación en cascada.
  }

  entry void open() {
    onlyOne=false;  // Se utilizará reactivación en cascada.
    c.notify();
  }

  entry void openOne() {
    onlyOne=true;   // Sólo se reactivará un hilo.
    c.notify();
  }
}
```

Figura 3.7: Ejemplo de monitor con ambos tipos de reactivación.

Por otro lado, al utilizar el método `notify()` de alguna de las condiciones de un monitor surge un problema, pues se pasaría a tener dos hilos activos dentro del mismo monitor. Esto rompería la exclusión mutua que los monitores garantizan. Por ello, todo monitor debe mantener suspendido temporalmente (hasta que el otro se suspenda o abandone el monitor) a uno de los dos hilos involucrados en los efectos del método `notify()`: el invocador (o notificador) o bien el que debía ser reactivado.

Existen diferentes clases de monitores en función de cómo resuelvan esta situación [BFC95]. Las variantes de Brinch Hansen [Bri73, Bri75] y Hoare [Hoa74] otorgan prioridad al hilo que debía ser reactivado, mientras que la variante de Lampso-

n/Redell [LR80] (que es la utilizada en los *POSIX threads* [IEE08] y en Java) da mayor prioridad al hilo notificador. Se describirá seguidamente cada una de las variantes, cuyas características se resumen en la tabla 3.3.

	Variantes		
Características	Brinch Hansen	Hoare	Lampson/Redell
El hilo notificador...			
sale del monitor.	X		
pasa a una cola especial.		X	
continúa.			X
El hilo notificado...			
pasa a la cola de entrada.			X
continúa.	X	X	

Tabla 3.3: Características de las variantes de monitor.

3.3.1 Variante de Brinch Hansen

Esta variante [Bri75] obliga a que el método `notify()` de las condiciones, en caso de utilizarse, sea la última sentencia del método en que aparezca. El código de los dos ejemplos del monitor `Barrier` presentados en las figuras 3.6 y 3.7 cumple con esa restricción. No todos los métodos de un monitor están obligados a utilizar `notify()`.

Al garantizarse que la sentencia `notify()` sea la última en aquellos métodos donde aparezca, se obtiene como resultado que el hilo notificador abandone el monitor al efectuar esa invocación. Con ello, el hilo reactivado no encuentra ningún "competidor activo" y la propiedad de exclusión mutua se garantiza sin problemas.

El ejemplo de la figura 3.8 ilustra cómo se podría definir un monitor que implantara un semáforo (concepto que ya se comentó en la sección 2.6).

Este ejemplo muestra que no todos los métodos de un monitor necesitan usar `notify()`. Además, también ilustra que no todos los monitores necesitarán utilizar la reactivación en cascada.

Lo que no se ha llegado a ilustrar en ninguno de los ejemplos de esta sección es la necesidad de emplear más de una condición en un mismo monitor. La figura 3.9 muestra un ejemplo donde podrá apreciarse tal necesidad. Es el monitor `BoundedBuffer`, desarrollado según la variante de Brinch Hansen, que se utilizaría para dar soporte al problema del "Productor-Consumidor con Buffer Acotado" refinando así la plantilla mostrada en la figura 3.1.

```
monitor Semaphore {
    int counter;
    Condition c;

    public Semaphore(int value) {
        counter = value;
    }

    entry void P() {
        counter--;
        if (counter < 0)
            c.wait();
    }

    entry void V() {
        counter++;
        if (counter < 1)
            c.notify();
    }
}
```

Figura 3.8: Monitor que implanta el tipo o clase `Semaphore`.

Obsérvese que utilizando la variante de Brinch Hansen no es necesario evaluar de nuevo la expresión condicional que obligó a suspender a un hilo en un `wait()`. Si otros hilos le notifican (y el monitor está correctamente implantado) seguro que la situación que le obligó a suspenderse ya no se dará. Por tanto, se podrá desarrollar el código de estos monitores utilizando sentencias `if` que conduzcan a la suspensión. No es necesario utilizar bucles `while` que reevalúen tales expresiones.

3.3.2 Variante de Hoare

En algunos casos no resulta sencillo ubicar la llamada a `notify()` como última sentencia de un método. La variante de Hoare elimina dicha restricción y, como resultado, se ve obligada a utilizar una cola especial de re-entrada al monitor donde depositará temporalmente al hilo notificador. Como ya se había dicho anteriormente, en esta variante tiene prioridad el hilo reactivado y ello obliga a que el notificador deba permanecer suspendido hasta que el hilo reactivado se suspenda de nuevo en otra condición o abandone el monitor. El hilo notificador permanece entre tanto en esa cola especial, y podrá reanudar antes la ejecución del código del monitor que aquellos hilos que se hayan suspendido en la cola de entrada al invocar un método `entry` y haber encontrado el monitor ocupado.

Observe el código del monitor `BoundedBuffer` de la figura 3.9. En él se ha tenido que utilizar un argumento de salida en su método `get()`, empleando una sintaxis

```
monitor BoundedBuffer {
    int buffer[];
    int numElem;
    int capacity;
    int first, last;
    condition notEmpty, notFull;

    public BoundedBuffer(int size) {
        buffer = new int[size];
        numElem = first = last = 0;
        capacity = size;
    }

    entry void put(int elem) {
        if (numElem == capacity)
            notFull.wait();
        buffer[last] = elem;
        numElem++;
        last = (last + 1) % capacity;
        notEmpty.notify();
    }

    entry void get(int *elem) {
        if (numElem == 0)
            notEmpty.wait();
        *elem = buffer[first];
        first = (first + 1) % capacity;
        numElem--;
        notFull.notify();
    }
}
```

Figura 3.9: Monitor `BoundedBuffer` (variante de Brinch Hansen).

similar a la del lenguaje C. Algunos lenguajes de programación no admiten ese tipo de argumentos. En ese caso, deberíamos terminar el método `get()` con una sentencia similar a "`return elem;`" que estaría detrás del "`notFull.notify();`". Obviamente, eso no respetaría la variante de Brinch Hansen y explica por qué surgió la variante de Hoare.

El código que se muestra en la figura 3.10 ilustra cómo se implantaría el monitor `BoundedBuffer` bajo la variante de Hoare. En esta variante, al igual que en la anterior, basta con evaluar una sola vez la expresión condicional a la hora de decidir si un hilo debe suspenderse o no. No son necesarios los bucles `while`.

Las trazas no son tan sencillas como en la variante anterior. Si a la hora de utilizar este monitor `BoundedBuffer`, se hubiera creado con capacidad para dos enteros (realizando un "`b = new BoundedBuffer(2)`") y se hubiera dado la siguiente secuencia de invocaciones por parte de cinco hilos diferentes: C1:`b.get()`,

```
monitor BoundedBuffer {
    int buffer[];
    int numElem;
    int capacity;
    int first, last;
    condition notEmpty, notFull;

    public BoundedBuffer(int size) {
        buffer = new int[size];
        numElem = first = last = 0;
        capacity = size;
    }

    entry void put(int elem) {
        if (numElem == capacity)
            notFull.wait();
        buffer[last] = elem;
        numElem++;
        last = (last + 1) % capacity;
        notEmpty.notify();
    }

    entry int get() {
      int item;
        if (numElem == 0)
            notEmpty.wait();
        item = buffer[first];
        first = (first + 1) % capacity;
        numElem--;
        notFull.notify();
        return item;
    }
}
```

Figura 3.10: Monitor `BoundedBuffer` (variante de Hoare).

P1:`b.put(2)`, P2:`b.put(3)`, P3:`b.put(4)`, P4:`b.put(5)`, P1:`b.put(6)`, C1:`b.get()`; el resultado habría sido el que se ilustra en la tabla 3.4.

Algunas de las casillas de esta tabla merecen algún comentario adicional. Así, en la cola especial, durante la sentencia `b.put(2)` invocada por el hilo P1 (tercera fila de la traza), el hilo P1 llega a estar en esa cola cuando invoca el `notEmpty.notify()`, pero posteriormente la abandona cuando el hilo C1 finaliza la ejecución del método `get()` del monitor. Por su parte, en esa misma sentencia el valor del atributo `numElem` llega a ser 1 temporalmente tras haber insertado P1 el elemento 2 en el buffer, pero después vuelve a ser cero cuando el hilo C1 completa la ejecución del método `get()`. Esto se ha representado en la tabla utilizando asteriscos como superíndices en el valor utilizado como contenido en esas casillas.

En cada fila se ha mostrado cuál era el resultado tras haber completado la invocación del método presentado en la primera columna de esa fila. En el tercer paso de

Evento	Cola especial	buffer[]	numElem	first	last	Cola notFull	Cola notEmpty
Constructor	vacía	{?,?}	0	0	0	vacía	vacía
C1:b.get()	vacía	{?,?}	0	0	0	vacía	C1
P1:b.put(2)	vacía*	{2,?}	0*	1	1	vacía	vacía
P2:b.put(3)	vacía	{2,3}	1	1	0	vacía	vacía
P3:b.put(4)	vacía	{4,3}	2	1	1	vacía	vacía
P4:b.put(5)	vacía	{4,3}	2	1	1	P4	vacía
P1:b.put(6)	vacía	{4,3}	2	1	1	P4,P1	vacía
C1:b.get() (notify)	vacía → C1	{4,3}	1	0	1	P4,P1 → P1	vacía
(P4 activo y sale)	C1	{4,5}	2	0	0	P1	vacía
(C1 retorna valor 3)	vacía	{4,5}	2	0	0	P1	vacía

Tabla 3.4: Traza con la variante de Hoare.

la traza, cuando el hilo P1 realiza el `b.put(2)`, su `notEmpty.notify()` reactiva a C1 que es capaz de ejecutar todo lo que le faltaba en su método `b.get()`, encontrando el valor 2 que acaba de insertar P1. Con ello, C1 abandona el monitor justo antes de que lo haga P1 que, durante ese lapso, ha permanecido suspendido en la cola especial del monitor.

El segundo `b.get()` realizado por el hilo C1 también merece una consideración especial. La primera de las filas asignadas a esa invocación ilustra los cambios realizados hasta que C1 invoca `notFull.notify()`. En ese punto resulta reactivado P4 (que desaparece de la cola asociada a esa condición) y C1 pasa a la cola especial del monitor. A continuación prosigue P4 y los efectos de su ejecución se muestran en la penúltima fila de la traza. Ese hilo consigue insertar el valor 5 en el buffer y actualizar los atributos que reflejan dicha inserción. Con ello, P4 termina la ejecución del `b.put(5)` que inició en el sexto paso de la traza. Antes de concluir ha llamado a `notEmpty.notify()` pero no ha tenido ningún efecto porque no había consumidores suspendidos en esa `Condition`. Con ello, C1 será el que ahora se reactive y, gracias a ello, retornará el valor 3 como resultado de su `b.get()`. Ahí finalizaría esta traza, en espera de que otros productores o consumidores invoquen los métodos del monitor.

3.3.3 Variante de Lampson y Redell

Esta es la variante que otorga prioridad al hilo notificador. Con ello, el hilo que teóricamente debía reactivarse, en lugar de hacerlo, es trasladado a la cola de entrada al monitor, compitiendo con el resto de hilos que deseen acceder al monitor.

El ejemplo del `BoundedBuffer` que acabamos de presentar para la variante de Hoare funcionaría también correctamente bajo la variante de Lampson/Redell. Sin

```
monitor SynchronousLink {
   Condition OKsender, OKreceiver;
   int senders_waiting, receivers_waiting;
   Message msg;

   public SynchronousLink() {
        senders_waiting = receivers_waiting = 0;
        msg = null;
   }

   entry void send(Message m) {
        if (receivers_waiting > 0) {
            msg = m;
            OKreceiver.notify();
        } else {
            senders_waiting++;
            OKsender.wait();
            senders_waiting--;
            msg = m;
        }
    }

    entry Message receive() {
        if (senders_waiting > 0) {
            OKsender.notify();
        } else {
            receivers_waiting++;
            OKreceiver.wait();
            receivers_waiting--;
        }
        return msg;
    }
}
```

Figura 3.11: Monitor `SynchronousLink` (variante de Hoare).

embargo, esto no se cumplirá siempre. Por ejemplo, si se revisa con detenimiento el código del monitor `SynchronousLink` que se presenta en la figura 3.11, se observa que para una traza en la que llegue antes el emisor que el receptor (es decir, se invoque antes a `send(msg)` que a `receive()`), el comportamiento será correcto con la variante de Hoare, pero no así con la de Lampson/Redell. En esta última el receptor prosigue tras el `OKsender.notify()` y realiza un `return msg;` antes de que sobre el atributo `msg` se haya asignado el mensaje que quería transmitir el emisor.

Por otra parte, si el notificador realiza más de un `notify()` sobre diferentes atributos `Condition` y llega a reactivar de manera lógica a múltiples hilos, sería posible que la ejecución de los que hayan reanudado antes su actividad modifique el entorno que asumía el notificador para los demás. Por ello, en lugar de una construcción del tipo:

`if (`*expresión lógica*`) cond.wait();`

en esta variante resulta mucho más recomendable otra del tipo:

`while (`*expresión lógica*`) cond.wait();`

y así comprobar de nuevo que el estado del monitor permita el avance de cada uno de los hilos reactivados. En las dos variantes anteriores, al tener prioridad el hilo reactivado, estaba garantizado que el estado que observaría dicho hilo reactivado coincidía con el que había generado el notificador. Ningún otro hilo podía intervenir entre la notificación y la reactivación. En la variante de Lampson y Redell esta garantía no existe, ya que el notificador puede reactivar a múltiples hilos en sucesivas llamadas a `notify()` y cuando después estos se vayan ejecutando cada uno de ellos podrá modificar el estado que asumió el notificador para los demás.

Como se ha dicho previamente, el lenguaje de programación Java adopta esta variante de Lampson y Redell. Hasta el momento solo se han considerado las implicaciones que ofrece esta variante general. Para el caso particular de Java, debemos recordar que existen restricciones adicionales:

- No se pueden declarar múltiples atributos de tipo `Condition` en un mismo monitor Java. Por ello, no acaba de cumplirse al pie de la letra lo que uno esperaría en una variante clásica de monitor. En todas las variantes clásicas (Brinch Hansen, Hoare, Lampson/Redell) resultaba viable tener múltiples atributos de tipo `Condition`. Con el soporte básico del lenguaje Java esto no es posible.
- No existe ninguna clase `Monitor` en Java capaz de garantizar exclusión mutua y de la cual heredar. Debemos simularlo utilizando el calificador `synchronized` en todos los métodos públicos.

El monitor `BoundedBuffer` visto en los ejemplos anteriores tendría el código que muestra la figura 3.12 en Java.

Como puede observarse, en este caso no tiene sentido utilizar simplemente `notify()` al tratar de reactivar a otros hilos, combinado con el uso de `if` en lugar de `while`. Si solo reactiváramos a uno, no habría manera de prever cuál se reactivaría.

Por ejemplo, si nuestro `BoundedBuffer` tuviera capacidad para un solo elemento y se hubiera dado esta traza (siendo C1 y C2 los identificadores de dos hilos consumidores y P1 el identificador de un hilo productor): C1:`b.get()`, C2:`b.get()`, P1:`b.put(3)`, ... un código como el de la figura 3.13 no funcionaría. Esto se debe a que cuando se ejecutase el P1:`b.put(3)`, se reactivaría a C1 o a C2 (asumamos que sea C1), que ahora podría avanzar, encontrando el elemento 3 en el buffer y retornándolo correctamente. Pero ese consumidor realizaría un `notify()` justo antes de salir del monitor. Con ello, se reactivaría C2 sin advertir que el buffer estaba vacío en ese momento, retornando un valor erróneo. Por ello, es crítico que

```
public class BoundedBuffer {
    private int buffer[];
    private int numElem;
    private int capacity;
    private int first, last;

    public BoundedBuffer(int size) {
        buffer = new int[size];
        numElem = first = last = 0;
        capacity = size;
    }

    public synchronized void put(int elem) {
        while (numElem == capacity)
          try {
            wait();
          } catch (Exception e) {};
        buffer[last] = elem;
        numElem++;
        last = (last + 1) % capacity;
        notifyAll();
    }

    public synchronized int get() {
      int item;
        while (numElem == 0)
          try {
            wait();
          } catch (Exception e) {};
        item = buffer[first];
        first = (first + 1) % capacity;
        numElem--;
        notifyAll();
        return item;
    }
}
```

Figura 3.12: Monitor `BoundedBuffer` en Java.

se utilicen bucles para reevaluar las condiciones y que se realice un `notifyAll()` para que todos los posibles candidatos examinen si el estado del monitor les permitirá progresar o no.

```
public class IncorrectBoundedBuffer {
    private int buffer[];
    private int numElem;
    private int capacity;
    private int first, last;

    public IncorrectBoundedBuffer(int size) {
        buffer = new int[size];
        numElem = first = last = 0;
        capacity = size;
    }

    public synchronized void put(int elem) {
        if (numElem == capacity)
          try {
            wait();
          } catch (Exception e) {};
        buffer[last] = elem;
        numElem++;
        last = (last + 1) % capacity;
        notify();
    }

    public synchronized int get() {
      int item;
        if (numElem == 0)
          try {
            wait();
          } catch (Exception e) {};
        item = buffer[first];
        first = (first + 1) % capacity;
        numElem--;
        notify();
        return item;
    }
}
```

Figura 3.13: Monitor `BoundedBuffer` incorrecto en Java.

3.4 Invocaciones anidadas

Los monitores son una herramienta de sincronización excelente al combinar de manera sencilla la exclusión mutua con la sincronización condicional. Sin embargo, también puede plantear problemas en caso de que desde el código de alguno de los métodos de un monitor A se necesite utilizar algún método de otro monitor B. Si eso sucediera el hilo H1 que siguiera esa secuencia de llamadas seguiría estando activo dentro del monitor A, impidiendo que ningún otro hilo pueda ejecutar sus métodos (por la exclusión mutua que ofrece el monitor). A su vez, también estaría activo dentro del monitor B. Si al ejecutar el código del monitor B llegara a suspenderse en alguna de sus condiciones, nadie podría ejecutar el monitor A, pues aunque se habría liberado la entrada del monitor B (al quedar suspendido el hilo

en una condición de B), la entrada del monitor A seguiría cerrada (ya que el hilo no está suspendido en ninguna condición de A). Todos aquellos hilos que llegasen a intentar ejecutar el monitor A quedarían suspendidos en su cola de entrada.

Ante una situación como ésta, lo único que puede decirse es que no basta con disponer de buenas herramientas de sincronización para garantizar un alto nivel de concurrencia en la ejecución de aplicaciones complejas. Se sigue necesitando cierto cuidado en el diseño de estas aplicaciones. Uno de los mayores problemas que se podrá encontrar en dicho diseño será la aparición de situaciones de interbloqueo, como se describirá en la unidad siguiente.

3.5 Resumen

En esta unidad se ha analizado una herramienta de sincronización que combina de manera sencilla la exclusión mutua con la sincronización condicional: el monitor. Un monitor es una estructura de datos que ofrece un conjunto de operaciones públicas que se ejecutarán en exclusión mutua de forma automática. De este modo, si algún hilo inicia la ejecución de alguna operación del monitor, no podrá haber ningún otro hilo activo ejecutando código de ese mismo monitor. Además, los monitores proporcionan un mecanismo general de suspensión, denominado condición, que permite suspender a hilos dentro del monitor, liberando el acceso a éste. Los hilos suspendidos podrán reactivarse cuando otros hilos modifiquen el estado del monitor y les notifiquen. Para el diseño apropiado de monitores se debe seguir la guía de diseño explicada en esta unidad, que permite identificar las distintas colas de espera (variables de tipo `Condition`) que serán necesarias para el funcionamiento adecuado del monitor. A modo de ejemplo, se ha presentado el soporte que ofrece el lenguaje Java para utilizar monitores. En este caso, no existe una clase denominada *Monitor* específica, sino que todo objeto de Java puede actuar como un monitor (si se siguen las restricciones adecuadas), ya que posee una cola de entrada y una cola de espera implícitas. A diferencia del modelo tradicional de monitor, en Java solo hay una cola de espera (implícita) en cada monitor y por ello las notificaciones deberían reactivar a todos los hilos, los cuales deben reevaluar la condición (para así volver a suspenderse si la condición por la que esperaban todavía no se cumple).

Tanto en el monitor Java como en el monitor general, cuando se reactiva algún hilo suspendido en una condición surge un problema, ya que se pasaría a tener dos hilos activos dentro del mismo monitor: el invocador (o notificador) y el que se ha reactivado. Esto rompería la exclusión mutua que los monitores garantizan. Por ello, todo monitor debe mantener suspendido temporalmente a uno de esos dos hilos, hasta que el otro se suspenda o abandone el monitor. En esta unidad hemos analizado las tres variantes de monitor existentes que permiten resolver esta situación: las variantes de Brinch Hansen y Hoare, que otorgan prioridad al hilo

que debía ser reactivado, y la variante de Lampson/Redell (usada en Java), que da mayor prioridad al hilo notificador. En la variante de Brinch Hansen se obliga a que el método `notify()` sea la última sentencia del método donde aparezca. De este modo el hilo notificador abandona el monitor tras efectuar esa invocación, por lo que el hilo reactivado no encuentra ningún otro hilo ejecutando código del monitor y se garantiza la exclusión mutua. En la variante de Hoare dicha obligación no existe y, en este caso, el hilo notificador pasa a una cola especial de re-entrada tras ejecutar el `notify()`, que tiene prioridad sobre la cola de entrada. Así, el notificador permanece suspendido en esa cola especial hasta que el hilo reactivado se suspenda en otra condición o abandone el monitor. Finalmente, en la variante de Lampson/Redell, se otorga prioridad al hilo notificador, por lo que el hilo que teóricamente debía reactivarse es trasladado a la cola de entrada al monitor, compitiendo con el resto de hilos que deseen acceder al monitor.

Los monitores pueden presentar problemas si se realizan invocaciones anidadas. Así, si desde el código de alguno de los métodos de un monitor A se necesita utilizar algún método de otro monitor B y el hilo que lo ejecuta llegara a quedarse suspendido en alguna de las condiciones del monitor B, ningún otro hilo podría ejecutar el monitor A, pudiéndose producir situaciones de interbloqueo. Este tipo de situaciones se explicarán en la siguiente unidad.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Proteger las secciones críticas de una aplicación mediante los mecanismos adecuados. En concreto, se emplearán los monitores como mecanismos de sincronización.
- Describir y comparar las diferentes variantes de monitor existentes.
- Identificar los problemas que puede originar un uso incorrecto de los mecanismos de sincronización. En concreto, los problemas que surgen de un mal diseño de los monitores.
- Identificar la problemática existente en las invocaciones anidadas de los monitores.
- Utilizar el soporte que ofrece el lenguaje Java para implementar monitores.
- Distinguir las diferencias que plantea un monitor Java frente al modelo tradicional de monitor.

Unidad 4

INTERBLOQUEOS

4.1 Introducción

Cuando múltiples actividades concurrentes necesiten acceder a cierto conjunto de recursos que no admitan un uso compartido existe el riesgo de que haya un interbloqueo [Dij68]. Es decir, que tales procesos o hilos lleguen a suspenderse esperando la ocurrencia de algún evento que solo podría ser generado por otro proceso o hilo que también pertenece a ese mismo conjunto de actividades bloqueadas. En ese caso, ninguna de esas actividades podrá continuar y si el usuario o el administrador no lo remedian, esta situación problemática se mantendrá indefinidamente.

En esta unidad se estudiará ese problema, identificando qué condiciones deberán presentar tales recursos y las actividades que soliciten su uso para que tales situaciones de interbloqueo puedan ocurrir. Una vez conozcamos esas condiciones necesarias [CES71] se podrá diseñar estrategias que prevengan la ocurrencia de este problema así como algoritmos que eviten su aparición comprobando ante cada petición si su atención y asignación entraña algún riesgo. Otro tipo de soluciones se basa en permitir que el interbloqueo aparezca, utilizando periódicamente algún algoritmo que compruebe el estado de asignación de los recursos y las relaciones de espera entre procesos para detectar tales interbloqueos. En caso de que sean detectados, se eliminará alguna de las actividades bloqueadas, liberando así los recursos que tenía asignados, con lo que otras actividades bloqueadas lograrán reactivarse, desapareciendo el interbloqueo.

A pesar de que haya un buen número de estrategias para gestionar la existencia de interbloqueos, no todos los sistemas actuales utilizan alguna de ellas. La mayor parte de los sistemas operativos actuales (diseñados para ordenadores personales) asumen que será raro que un conjunto de procesos independientes lleguen a inter-

bloquearse y, por ello, no hacen nada para evitar tal problema ni para detectarlo en caso de que ocurra. Por tanto, conviene que los usuarios de tales sistemas sepan por qué aparecen los interbloqueos y cómo pueden resolverse (tal como se explicará en esta unidad). En otros entornos, como las bases de datos relacionales, la ocurrencia de interbloqueos no es nada rara y sus propios sistemas gestores ya resuelven este problema, sin que sus usuarios deban preocuparse por ello.

4.2 Definición de interbloqueo

Existen muchas definiciones posibles para una situación de *interbloqueo* en un sistema concurrente. Una de ellas aparece en [SS94] (extendiendo la que se dio en [Hol72]) y es la siguiente:

> "*Se da un interbloqueo cuando un conjunto de procesos (o hilos) en un sistema está bloqueado esperando algún evento que jamás llegará a darse. Estos procesos, mientras mantienen algunos recursos, están solicitando otros recursos mantenidos por otros procesos del mismo conjunto. Esto es, los procesos están involucrados en una espera circular.*"

Si se analiza con detenimiento esa definición se observa que el motivo de que ese conjunto de procesos esté "interbloqueado" es el uso de cierto conjunto de recursos. Cada proceso se bloquea porque necesita algunos recursos más que habrá solicitado pero que ya estarán otorgados a otros procesos del mismo conjunto. Por ello, el sistema operativo (o cualquier otro componente del sistema que gestione recursos) bloquea a cada solicitante mientras no pueda asignarle aquello que haya solicitado. A su vez, cada proceso bloqueado habrá sido capaz de recibir previamente algún recurso que estará utilizando y que obligará a otros procesos a que también se bloqueen si llegan a solicitarlo. De esta manera se van creando "ciclos de espera" y ninguno de los procesos interbloqueados será capaz de continuar con su ejecución pues aquello que esperan solo podrá liberarlo algún otro proceso que también está bloqueado.

En esa definición se está hablando de "recursos". Conviene aclarar a qué se refiere tal concepto. Un *recurso* es cualquier elemento físico (como un dispositivo de E/S: disco, impresora, teclado, ratón,...) o lógico (como pueda ser un semáforo, fichero, región de un espacio lógico de memoria,...) de un sistema informático que pueda ser utilizado por un hilo o proceso.

En un sistema determinado puede que existan múltiples *instancias* de un determinado recurso. En ese caso, cuando un hilo o proceso solicite el uso de dicho recurso se le podrá asignar una cualquiera de tales instancias. Por ejemplo, si en los laboratorios en los que se realizan las prácticas de una determinada asignatura

se solicita imprimir algún documento, el recurso necesario para completar dicha tarea será una impresora. Supongamos que en dicho laboratorio hayan varias impresoras disponibles (todas ellas impresoras láser monocromas) gestionadas por un servidor. Bastará con utilizar una cualquiera de ellas. Cada una de esas impresoras es una instancia distinta del recurso "impresora" y todas ellas son equivalentes (al ser todas del mismo tipo, es decir, láser monocromas). Si hubiese alguna característica que diferenciase a dichos elementos del sistema y fuese relevante a la hora de solicitar su uso, pasarían a ser recursos distintos. Por ejemplo, si alguna de las impresoras permitiera imprimir en color y algunos documentos debieran imprimirse en color, se distinguiría entre ambas clases de recurso: impresora monocroma e impresora en color. Habría un número determinado de impresoras de cada tipo. Con ello, habría dos recursos capaces de imprimir documentos, cada uno con cierto número de instancias.

Los hilos o procesos usan los recursos siguiendo este protocolo:

1. *Petición.* El hilo tendrá que solicitar el uso del recurso a su gestor. En función del estado del recurso, el gestor se lo podrá otorgar o no. La mayoría de los recursos exigen un uso en exclusión mutua: solo un proceso o hilo puede utilizarlos en cada momento y mientras no finalice su actividad, todos los demás solicitantes tendrán que esperar.

2. *Uso.* Una vez el gestor ha asignado el recurso a cierto hilo o proceso, éste podrá utilizarlo. Como ya se ha dicho en el punto anterior, esa instancia asignada solo será utilizada por un hilo o proceso en cada momento.

3. *Liberación.* Cuando finalice el uso del recurso, el hilo o proceso indicará al gestor que ya no lo necesita y así el gestor podrá asignarlo a otro solicitante.

Como resultado de este protocolo, un hilo A espera a otro hilo B cuando A solicita un recurso que ahora mismo esté asignado a B. Mientras B no libere dicho recurso, A permanecerá esperando tal liberación. Cuando cierto conjunto de hilos no puedan progresar debido a que haya un ciclo de esperas entre ellos, se dará un interbloqueo.

El problema de interbloqueos puede parecer similar al problema de inanición que se puede llegar a producir durante la planificación del procesador en cualquier sistema operativo. Se da una situación de *inanición* cuando un proceso espera por un recurso que está habitualmente disponible pero que jamás recibe debido a que continuamente aparecen otros procesos que son considerados mejores candidatos para la asignación de tal recurso. Tanto en las situaciones de inanición como en las de interbloqueo, el proceso está suspendido y no puede avanzar. No obstante, cuando se dé inanición los receptores del recurso solicitado siguen ejecutándose, mientras que en una situación de interbloqueo ninguno de los receptores de los recursos solicitados puede continuar. A su vez, el propio recurso estará en uso en una

situación de inanición, mientras que en una situación de interbloqueo los recursos están asignados, pero ninguno de ellos llega a utilizarse, ya que sus propietarios actuales están suspendidos intentando obtener otros recursos.

4.2.1 Condiciones de Coffman

Coffman *et al.* [CES71] justificaron la existencia de cuatro condiciones necesarias para que se dé una situación de interbloqueo. Son las siguientes:

- *Exclusión mutua.* Mientras una instancia de un recurso esté asignada a un proceso, otros procesos no podrán obtenerla.
- *Retención y espera.* Como los recursos se solicitan a medida que se necesitan, un proceso podrá tener algún recurso asignado y solicitar otro que en ese momento no esté disponible, obligándole a suspenderse.
- *No expulsión.* Un recurso que ya esté asignado solo podrá liberarlo su dueño. El gestor de recursos no podrá expropiarlo.
- *Espera circular.* En el grupo de procesos interbloqueados, cada uno está esperando un recurso asignado a otro proceso del grupo. Con ello se cierra un ciclo de esperas y ninguno de los procesos puede proseguir con su ejecución.

Si estas cuatro condiciones se cumplen simultáneamente en un sistema, existirá riesgo de interbloqueo. Sin embargo, al no ser condiciones suficientes, no existe la garantía de que tal interbloqueo exista. Si en un sistema llega a darse un interbloqueo, seguro que las cuatro condiciones se estarán cumpliendo. A su vez, por ser condiciones necesarias, bastará con que se incumpla alguna de ellas para garantizar que en el sistema no haya interbloqueos.

4.2.2 Ejemplos

Tomando como base los ejemplos de aplicaciones concurrentes explicados en las unidades anteriores, veamos seguidamente algunas situaciones en las que se daría una situación de interbloqueo, analizando por qué se cumplen en ellas todas las condiciones de Coffman.

Hormigas

En la sección 3.2.1 se presentó como ejemplo de aplicación concurrente un programa que simulaba el comportamiento de una colonia de hormigas. Cada hilo de ese programa representaba a una hormiga de la colonia y se utilizó un monitor para regular el uso del territorio por parte de ese grupo de hormigas. Si en ese programa

tenemos dos hormigas (A y B) que ocupan celdas contiguas del territorio y cada una de ellas desea moverse a la posición que ocupa la otra, se dará un interbloqueo.

Comprobemos que se cumplan todas las condiciones de Coffman. Para ello consideraremos que los recursos a utilizar por estos hilos son cada una de las celdas que modelan el territorio por el que pueden moverse las hormigas. De esta manera, las condiciones son:

- *Exclusión mutua*. Se cumple, pues una misma celda no puede estar ocupada a la vez por más de una hormiga. Los monitores presentados en las figuras 3.2, 3.3 y 3.5 garantizaban esta condición.
- *Retención y espera*. También se cumplirá. Se ha asumido que las hormigas A y B ocupaban posiciones contiguas en el territorio. Imaginemos que tales posiciones son (x_A, y_A) para la hormiga A y (x_B, y_B) para la hormiga B. Cada una de ellas pretende trasladarse a la posición que ocupa la otra. Por tanto, la hormiga A está reteniendo el recurso (x_A, y_A) y solicita el recurso (x_B, y_B) que hasta el momento está asignado a B. A su vez, la hormiga B está reteniendo el recurso (x_B, y_B) y solicita el recurso (x_A, y_A) que está asignado a A. Con ello, la condición de retención y espera se cumple en los dos hilos que están interbloqueados.
- *No expulsión*. El programa que simula el comportamiento de las hormigas no puede "expulsar" a una hormiga de una celda determinada. A su vez, una hormiga no puede expulsar a otra de la celda que esta última ocupe. La única manera en que se libera una celda del territorio se da cuando el hilo que representa a la hormiga que hasta el momento ocupaba esa posición decida trasladarse a otra celda contigua. Por tanto, esta tercera condición también se cumple.
- *Espera circular*. También se cumple. El hilo A espera al B y el B también espera al A. Por tanto, se daría un ciclo dirigido con dos aristas si representáramos mediante un grafo estas relaciones de espera.

Se observa que todas las condiciones de Coffman se están cumpliendo en este ejemplo. Esto certifica que habrá riesgo de interbloqueo. Además, se comprueba que ninguno de los hilos involucrados en el ciclo de esperas puede continuar (ni siquiera con la ayuda de otros hilos que ahora mismo no estuvieran en dicho ciclo de esperas y pudieran liberar recursos). Por ello, hay interbloqueo.

Cinco filósofos

El problema de los *cinco filósofos* [Dij71, Hoa85] ya fue descrito en la página 12. Si se implantara algún programa para resolver este problema y en alguna de sus ejecuciones se sentaran a la vez todos los filósofos y pudieran coger su primer tenedor, todos ellos quedarían bloqueados al tratar de obtener el segundo tenedor. Con ello se daría una situación de interbloqueo y se cumplirían todas las condiciones de Coffman, como se justifica seguidamente:

- *Exclusión mutua*. En este problema los recursos a utilizar son los cinco tenedores ubicados en la mesa. Un tenedor determinado no puede ser utilizado a la vez por dos filósofos diferentes. Por tanto, el uso de los recursos debe realizarse respetando esta condición de exclusión mutua.
- *Retención y espera*. También se cumplirá. Cada filósofo habrá sido capaz de obtener (y retener) su primer tenedor, pero se bloqueará al solicitar el segundo tenedor pues estará retenido por su filósofo vecino.
- *No expulsión*. Los filósofos no pueden quitar un tenedor a sus vecinos. Aquel que lo haya obtenido será su propietario y no lo liberará hasta que termine de comer y vuelva a "pensar". Por tanto, también se cumple la condición de "no expulsión".
- *Espera circular*. También se cumple. Los cinco filósofos describen un ciclo de cinco esperas.

4.3 Representación gráfica

A la hora de analizar si en un sistema se está dando una situación de interbloqueo resulta interesante el uso de una representación del estado actual de dicho sistema mediante algún tipo de grafo. Un *grafo de asignación de recursos* (GAR; propuesto inicialmente en [Hol72] como "grafo general de recursos") cumple este objetivo. La figura 4.1 muestra un ejemplo de grafo de este tipo, que debe interpretarse de la siguiente manera:

- Los procesos y los recursos constituyen el conjunto de nodos de este grafo, aunque se representarán de diferente manera. En esta unidad utilizaremos círculos para denotar procesos o hilos y rectángulos para representar recursos.

  En el ejemplo de la figura 4.1 tenemos tres procesos (P1, P2 y P3) y tres recursos (R1, R2 y R3).
- Aquellos nodos que modelen a los recursos tendrán tantos puntos internos como instancias tengan tales recursos. En la figura 4.1 tanto el recurso R1 como el R2 tienen una única instancia mientras R3 tiene dos instancias.

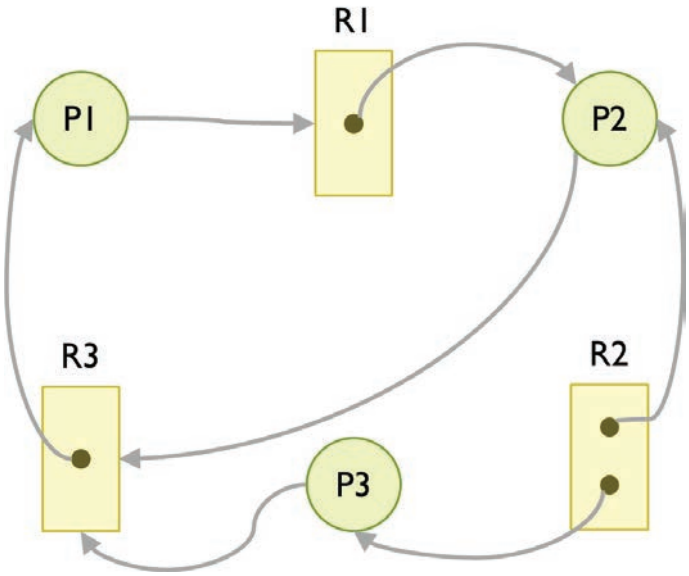

Figura 4.1: Grafo de asignación de recursos.

- Existen dos tipos de aristas (dirigidas) en el grafo:
  - *Arista de petición*: Es la que parte de un proceso P_i y llega a un recurso R_j. Representa la solicitud de una instancia de R_j por parte del proceso P_i e indica que dicha solicitud no ha podido ser atendida de manera inmediata (por no haber suficientes instancias libres), quedando el proceso P_i bloqueado al realizar tal petición.

    La flecha que represente a una arista de este tipo finaliza en el rectángulo que represente al recurso solicitado (no llega a ninguna instancia concreta, sino a todo el recurso). En la figura 4.1, las aristas $(P1, R1)$, $(P2, R2)$ y $(P3, R2)$ son de este tipo.

    Todo proceso del que parta alguna arista de este tipo se entiende que está bloqueado y que no podrá reanudar su ejecución hasta que dicha petición haya podido ser atendida, transformando entonces tales aristas de petición en aristas de asignación.
  - *Arista de asignación*: Es la que parte de una instancia concreta del recurso R_j y llega a un proceso P_i. Indica que esa instancia del recurso R_j se ha podido asignar al proceso P_i y éste todavía la utiliza. Cuando finalice su uso, el proceso liberará esa instancia y la arista de asignación correspondiente se eliminará del grafo.

    En la figura 4.1 hay cuatro aristas de este tipo: $(R1, P2)$, $(R2, P2)$, $(R2, P3)$ y $(R3, P1)$.

Un proceso o hilo podría en una misma petición solicitar más de un recurso o instancia. En tal caso [Hol72], si todas las instancias solicitadas estuviesen libres, se pasaría a representar el estado resultante mediante aristas de asignación, mostrando que la petición ha podido ser atendida y el proceso sigue activo. En caso contrario (es decir, si no hay suficientes instancias libres en cada uno de los recursos solicitados en esa acción del proceso), la solicitud efectuada quedará registrada mediante el conjunto necesario de aristas de petición. Esto implica que el sistema gestor asignará de una sola vez todas las instancias solicitadas, sin permitir asignaciones parciales.

4.3.1 Algoritmo de reducción de grafos

Cuando en un sistema haya un interbloqueo, seguro que su GAR presentará al menos un ciclo dirigido de aristas. Sin embargo, no todos los GAR que mantengan algún ciclo dirigido de aristas corresponden a situaciones de interbloqueo. Por tanto, no podemos tomar como único criterio para decidir si en un sistema hay un interbloqueo el hecho de que su GAR tenga o no ciclos dirigidos. Se necesita algo más. Holt [Hol72] demostró que utilizando el algoritmo de reducción de grafos mostrado en la figura 4.2 se puede decidir si un sistema presenta una situación de interbloqueo o no.

Mientras exista algún proceso P_i no bloqueado con aristas de asignación:
1.- Eliminar todas las aristas de asignación que terminen en P_i.
2.- Reasignar las instancias liberadas en el paso anterior a otros procesos que se bloquearon al solicitar tales recursos.
  2.1.- En caso de que haya múltiples solicitantes, asignar las instancias al proceso que tenga menos instancias pendientes y que pueda reactivarse con tal reasignación.

Figura 4.2: Algoritmo de reducción de grafos.

Este algoritmo asume que los procesos actualmente presentes en el sistema no realizarán más peticiones y que, con el transcurso del tiempo, irán liberando los recursos que tienen asignados. La secuencia de procesos P_i utilizada por el algoritmo sería una posible *secuencia de finalización*. Al finalizar la ejecución del algoritmo basta con comprobar si sigue habiendo aristas en el grafo. De ser así, el sistema presenta una situación de interbloqueo y en tal interbloqueo intervienen los procesos y recursos relacionados por tales aristas restantes. En caso contrario (esto es, cuando el algoritmo consiga eliminar todas las aristas existentes en el grafo), el estado actual del sistema no presenta ningún interbloqueo y la secuencia de finalización contendría a todos los procesos que formaban el grafo.

Si se intenta aplicar este algoritmo al sistema mostrado en la figura 4.1 se observa que no es posible realizar ni una sola iteración. Es decir, no se puede dar ningún

paso de reducción y el grafo resultante seguirá teniendo todas las aristas. Por tanto, dicho sistema presentaba un interbloqueo y en él estaban involucrados todos los procesos.

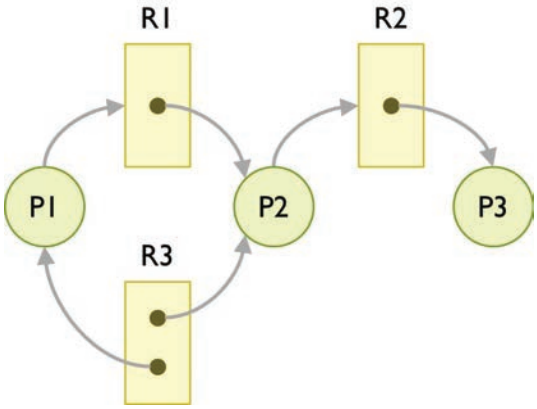

Figura 4.3: Traza del algoritmo de reducción (inicial).

Apliquemos ahora el algoritmo al grafo de asignación de recursos de la figura 4.3. Se realizarán los siguientes pasos:

1. En la situación inicial ilustrada en la figura 4.3 se observa que el proceso $P1$ está bloqueado solicitando una instancia del recurso $R1$ (que no tiene ninguna instancia libre) y que el proceso $P2$ también está bloqueado al pedir una instancia del recurso $R2$ (que tampoco tiene instancias libres). Por su parte el proceso $P3$ no está bloqueado y tiene asignada una instancia del recurso $R2$.

2. Al aplicar el algoritmo sobre dicho estado inicial, se elegirá al proceso $P3$ para aplicar el primer paso de reducción. Como resultado de éste, $P3$ se queda sin aristas de asignación y la instancia de $R2$ que queda momentáneamente libre se asigna a $P2$. La figura 4.4 muestra el estado resultante.

3. Al terminar esa iteración se detecta que $P2$ ya no está bloqueado y mantiene algunas aristas de asignación. Por ello, la segunda iteración elimina tales aristas para realizar la segunda reducción. $P2$ tenía asignada una instancia de cada uno de los tres recursos. Tras esta reducción esas tres instancias quedan libres y se permite así que $P1$ reciba la instancia de $R1$ que había solicitado. La figura 4.5 muestra el estado resultante.

   De momento, la secuencia de finalización que se está construyendo ya contiene a $< P3, P2 >$, en ese orden.

4. Al finalizar esa iteración se detecta que $P1$ ya no está bloqueado y mantiene las aristas de asignación que quedan en el grafo. Así, esta última iteración

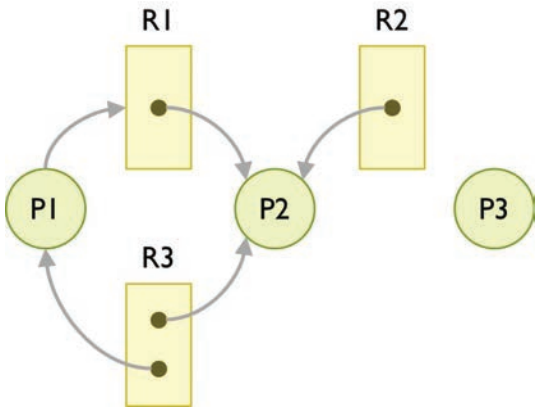

Figura 4.4: Traza del algoritmo de reducción (iteración 1).

eliminará todas esas aristas y dejará como resultado un grafo "limpio", sin ninguna arista, como muestra la figura 4.6. Eso demuestra que en la situación inicial no había ningún interbloqueo.

La secuencia de finalización $< P3, P2, P1 >$ es la que se obtiene en esta traza. Al contener a todos los procesos del grafo confirma que no hubo interbloqueo.

Veamos ahora qué ocurre cuando se aplica el algoritmo de reducción de grafos en otro sistema cuyo estado es el que se muestra en la figura 4.7:

1. Al iniciar la ejecución del algoritmo, el sistema está formado por tres procesos y tres recursos. $P1$ tiene asignada la única instancia del recurso $R2$ y está bloqueado solicitando la única instancia del recurso $R1$. $P2$ tiene asignada la única instancia de $R1$ y una de las dos instancias de $R3$, pero también está bloqueado al solicitar una instancia de $R2$. Por su parte, $P3$ no está bloqueado y tiene asignada una instancia de $R3$.

2. El algoritmo puede realizar una iteración de reducción, ya que el estado de $P3$ admite tal reducción. Esto tendrá como efecto la eliminación de la arista de asignación $(R3, P3)$. Tras esto no se puede hacer nada más, ya que la instancia que se acaba de liberar no había sido solicitada por ningún proceso bloqueado.

   Como resultado, ya no se puede aplicar ninguna iteración más del algoritmo, pues no existe ningún proceso que no esté bloqueado y tenga alguna arista de asignación dirigida hacia él.

Se puede observar en la figura 4.8 que en este estado final sigue habiendo aristas en el GAR y ello indica que existe un interbloqueo en el estado del sistema sobre

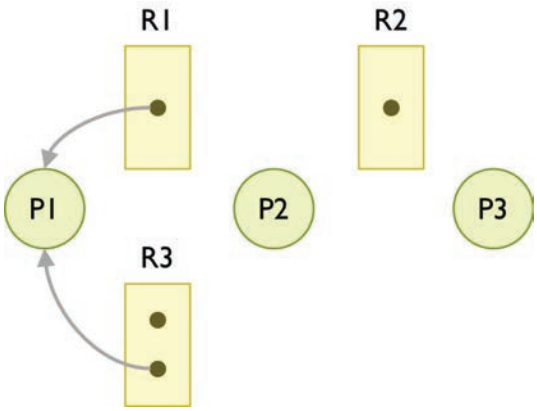

Figura 4.5: Traza del algoritmo de reducción (iteración 2).

el que se aplicó el algoritmo y en dicho interbloqueo intervienen los procesos $P1$ y $P2$. El algoritmo no ha sido capaz de construir ninguna posible secuencia de finalización de todos los procesos existentes.

Este ejemplo ilustra que aunque el algoritmo sea inicialmente aplicable, se podrá llegar a un punto en el que ya no sea posible efectuar más pasos de reducción y en el que todavía queden aristas. Es decir, habrá situaciones en las que solo algunos procesos del sistema estén interbloqueados.

4.4 Soluciones

Existen cuatro estrategias posibles para afrontar las situaciones de interbloqueo en un sistema (Coffman *et al.* [CES71] identificaron las tres primeras, mientras que Holt [Hol72] añadió la última):

1. *Prevención*. Diseñar el sistema de tal manera que se rompa alguna de las condiciones de Coffman. Los gestores de recursos que haya en el sistema deben implantarse de forma que al menos una de las cuatro condiciones necesarias no llegue a darse nunca. Así, jamás podrá darse ningún interbloqueo.

2. *Evitación*. El sistema debe conocer en esta estrategia cuál será la cantidad máxima de instancias que cada proceso solicitará de cada recurso. Con esta información, se monitoriza cada una de las peticiones realizadas por los procesos y se evalúa si resulta seguro asignar las instancias solicitadas en esa petición. Si tal asignación introdujera el riesgo de un interbloqueo futuro, adoptando una previsión pesimista en la que ninguno de los procesos participantes libere ninguna de sus instancias mientras no haya terminado su

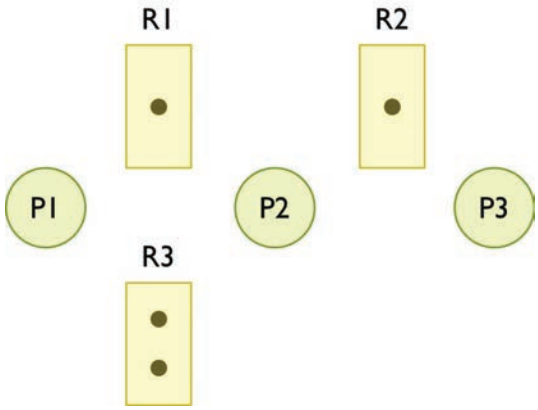

Figura 4.6: Traza del algoritmo de reducción (iteración 3).

ejecución y cada uno de ellos necesite su máximo de instancias, entonces la petición se rechazaría.

3. *Detección y recuperación*. El sistema monitoriza periódicamente su estado. Por ejemplo, cada vez que un proceso realice una petición de recursos y deba ser bloqueado porque alguno de los recursos solicitados no tenga instancias disponibles. Esta monitorización emplea cierto algoritmo de detección (el de reducción de grafos explicado en la sección 4.3.1 es un buen ejemplo). En caso de que el algoritmo detecte que ya hay un interbloqueo, se pasa a utilizar alguna estrategia de recuperación, consistente en la eliminación de alguno de los procesos que cierran el ciclo de esperas, liberando los recursos que tenía asignados y repartiéndolos entre el resto de procesos, asegurando así su avance.

4. *Ignorar el problema*. El sistema no se preocupa por la aparición de interbloqueos y no realiza ninguna gestión relacionada con tal problema. De esta manera se delega en el operador o en el usuario la tarea de detectar la ocurrencia de esta situación y resolverla como pueda.

   Esta es la estrategia utilizada en la mayoría de los sistemas operativos para ordenadores personales. No es difícil para el usuario advertir cuándo alguna aplicación se ha "colgado". El usuario observa que la aplicación no responde y está bloqueada. Quizá algunas más también se encuentren en una situación similar. Si el sistema operativo es capaz de responder, aceptando nuevas órdenes o gestionando adecuadamente el entorno gráfico, el usuario podrá solicitar la terminación de la aplicación bloqueada. Con ello ya se resolvería el problema. En caso de que todo el sistema esté bloqueado, no queda otro remedio que reiniciar el equipo.

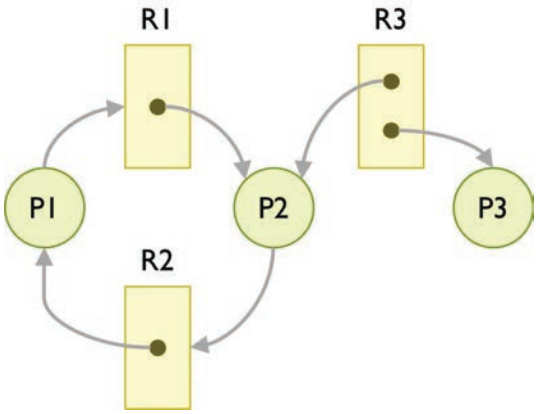

Figura 4.7: Segunda traza de reducción (estado inicial).

A continuación se revisará cada una de estas estrategias detalladamente.

4.4.1 Prevención

Como ya se ha comentado previamente, las estrategias de prevención [CES71] consisten en lograr que alguna de las cuatro condiciones de Coffman no se pueda cumplir. De esta manera al romper al menos una de las condiciones necesarias, el interbloqueo no llegaría a darse. Analicemos si es posible romper cada condición y de qué manera podría hacerse:

- *Exclusión mutua.* La condición de exclusión mutua viene fijada por la propia naturaleza del recurso. Habrá recursos que se podrán compartir y otros que no. Los que puedan ser compartidos jamás provocarán un interbloqueo pues cualquier proceso o hilo podrá acceder a ellos sin necesidad de bloquearse o esperar. Aquellos que no puedan compartirse requieren que su uso se realice en exclusión mutua. Esa característica es la que obliga a emplear un protocolo de tres pasos (solicitud, uso y liberación) para utilizar el recurso.

  Por ejemplo, de nada nos serviría que el servicio de impresión proporcionado por un sistema operativo permitiera que en una determinada impresora se mezclara el contenido de dos documentos que habían solicitado imprimir dos usuarios distintos, rompiendo así la condición de exclusión mutua en el acceso a tal recurso. En cada página impresa aparecerían líneas de uno y otro documento y tal resultado no le serviría ni a uno ni a otro usuario. Por tanto, esta condición se tendrá que mantener siempre. No hay ninguna manera de romperla.

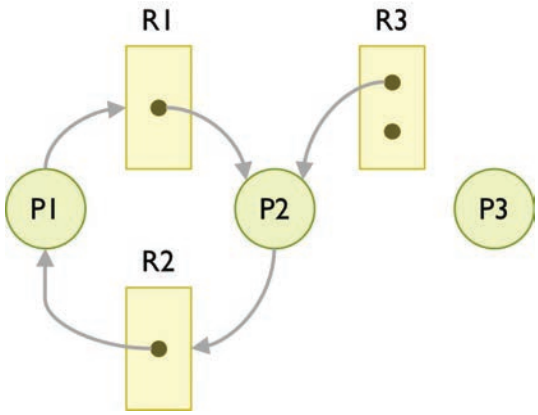

Figura 4.8: Segunda traza de reducción (iteración 1).

- *Retención y espera.* La condición de retención y espera es una consecuencia de la forma habitual en la que los procesos utilizan lo recursos: se van solicitando a medida que resulten necesarios. Como resultado, puede que todavía se necesite un recurso A, obtenido y utilizado previamente con lo que se retiene la posibilidad de acceder a él y ahora se necesite utilizar otro recurso B y la petición correspondiente se bloquea al no encontrar ninguna instancia disponible de tal recurso B. Para evitar que estas situaciones ocurran, bastará con liberar todos los recursos mantenidos cada vez que se necesite algún recurso nuevo. Tras dicha liberación se pedirán todos aquellos recursos que sean necesarios durante esa nueva fase de la ejecución del proceso. Puede que algunos de esos recursos ya estuvieran asignados previamente, pero han tenido que liberarse para evitar que se cumpla la condición de retención y espera.

  En lugar de adoptar esa estrategia, excesivamente restrictiva, se consideraron dos opciones más sencillas:

  1. Solicitar todos los recursos antes de iniciar la ejecución del proceso. Se realiza una única petición conjunta que bloquea el inicio del proceso en caso de que alguna instancia solicitada no esté disponible. Si dicha petición tiene éxito el proceso recibe todos los recursos que necesitará durante su ejecución y los mantendrá todos hasta que termine.

     Para implantar esta estrategia, el programa correspondiente debe indicar qué recursos (y cuántas instancias de cada uno) necesitará durante su ejecución. Esa información no suele estar disponible en los sistemas operativos modernos pues los recursos a utilizar suelen depender de la entrada facilitada al programa y ésta a su vez depende de lo que decida el usuario en cada ejecución. No obstante, en los primeros sistemas

operativos (sistemas de procesamiento por lotes en los que una secuencia o conjunto de programas definían un trabajo y eran ejecutados en un ordenador sin requerir ninguna intervención por parte del operador del sistema) la definición del lote de programas a ejecutar requería que se especificase qué recursos necesitaría cada programa, indicándolo en algunas tarjetas de control que procesaba el sistema antes de iniciar la ejecución de tales programas. En aquellos primeros sistemas multiprogramados sí que fue viable la implantación de esta estrategia.

2. Modificar las llamadas al sistema necesarias para obtener los recursos, de manera que no sean bloqueantes. En lugar de bloquear al proceso cuando el recurso esté ocupado, el sistema retornará el control inmediatamente devolviendo algún resultado indicando que no hay instancias disponibles. Con este soporte, en caso de que una solicitud de recurso devuelva un error (recurso ocupado), el proceso solicitante estará obligado a liberar todos los recursos que tenía asignados. Tras cierto intervalo de espera, el proceso tendrá que solicitar de nuevo todos los recursos, siguiendo ese mismo procedimiento (es decir, consultando el resultado de tales peticiones). Sólo cuando puedan obtenerse todos, el proceso continuará.

   Al emplear esta estrategia seguirá habiendo intervalos de espera, pero durante tales esperas los procesos no retendrán ningún recurso y así se romperá esta condición.

A pesar de que existan estas dos estrategias alternativas, ninguna de ellas consigue eliminar los problemas que supone la rotura de la condición de retención y espera: poca concurrencia, baja utilización de los recursos y largos intervalos de espera para los procesos.

El principal problema de la primera estrategia es que los recursos se asignan a los procesos durante todo el tiempo que estos últimos permanecen en el sistema. Ya que los recursos respetan la condición de exclusión mutua, eso implica que la utilización de cada recurso será muy baja pues durante todo ese intervalo el proceso no habrá utilizado únicamente un recurso sino que habrá tenido que repartir su tiempo entre todos ellos.

En la segunda estrategia no mejora excesivamente ese problema. Cada vez que un proceso se tope con un recurso ya otorgado a otros procesos tendrá que liberar todo lo que ya tenía asignado, reintentado posteriormente su obtención. Desde la liberación a la posterior reasignación, el recurso ha podido quedar libre con lo que su utilización será menor.

Como consecuencia de esto, el grado de concurrencia real en el sistema también baja: hay pocos procesos que estén avanzando a la vez. La mayor parte de ellos permanecen suspendidos tratando de obtener alguno de los recursos que necesitan.

Por último, ambas estrategias perjudican gravemente a aquellos hilos o procesos que necesiten varios recursos solicitados con frecuencia por otros hilos o procesos. Al tener que obtenerlos todos a la vez, la probabilidad de que todos esos recursos estén libres será muy baja y tales procesos sufrirán el problema de *inanición*.

- *No expulsión*. Para romper esta condición se podría permitir que un proceso A expropie los recursos mantenidos por otro proceso B en caso de que A solicite un recurso retenido por B y B ya esté bloqueado porque no ha podido obtener ni expropiar otro recurso que necesitara. La estrategia 2 descrita anteriormente para romper la retención y espera también sería otro ejemplo de estrategia basada en *expropiación*, pues el sistema obligaba a que un solicitante liberase todos sus recursos asignados en caso de no poder avanzar en alguna petición.

  Cualquiera de las dos soluciones plantea el problema de que aquellos recursos que hayan sido expropiados tendrán que ser obtenidos de nuevo. Esto solo tendrá sentido si el recurso en cuestión puede "recordar" el estado en que se encontraba cuando fue expropiado y retomar tal estado cuando sea reasignado al mismo proceso algo más tarde. No todos los recursos admiten este tipo de gestión. Por ejemplo, una impresora no lo admite: no tiene sentido dejar una página a mitad de su impresión para pasar a imprimir otro documento distinto. Sin embargo, el procesador sí que puede gestionarse de esa manera (una operación de *cambio de contexto* guarda y restaura copias del estado del procesador en los PCB de los procesos que participen en dicho cambio) y los algoritmos de planificación expulsivos serían un ejemplo válido del uso de esta estrategia.

  El uso de una estrategia basada en expropiación debe estar asociado a cierto criterio que ordene de alguna manera a los procesos (por ejemplo, asignándoles prioridades) y que siga dicha ordenación a la hora de decidir qué proceso puede expropiar los recursos de otro. En caso de que no se dé tal ordenación podría suceder que varios procesos no puedan avanzar porque se estén expropiando los recursos mutuamente. Eso no sería un interbloqueo (pues los procesos no se bloquean al solicitar recursos, sino que los expropian y obtienen inmediatamente), sino un "*livelock*": los procesos de ese conjunto están todos ellos activos pero ninguno llega a avanzar pues antes de tener todos los recursos necesarios otros procesos han expropiado alguno de los que ya estaban asignados.

- *Espera circular*. La misma idea de ordenación que se ha comentado para evitar los "*livelocks*" proporcionará la base para romper la espera circular. En este caso la ordenación se realiza sobre los identificadores de recurso, generando así un orden total entre todos los recursos del sistema. Los procesos o hilos estarán obligados a pedir sus recursos en un determinado orden (creciente o decreciente, pero todos los procesos del sistema deben utilizar

el mismo). Solicitando los recursos de esta manera se puede demostrar fácilmente que será imposible cerrar un ciclo de esperas.

Esta es la condición que resulta más fácil de romper. Sin embargo, la imposición de ese orden en las peticiones obliga a que los procesos soliciten algunos recursos antes de que éstos se necesiten. En la práctica, será un orden artificial que pocas veces coincidirá con el orden en que el proceso debía utilizar tales recursos. Esto provocará que la utilización de los recursos baje (al pedirlos antes de tiempo) y prolongará los intervalos de espera de otros procesos que los hayan pedido algo más tarde y sí que los necesiten.

Una aproximación similar consiste en estructurar una aplicación concurrente en niveles [Dij71]. Los procesos ubicados en un determinado nivel solo pueden solicitar los recursos gestionados por el nivel situado justo debajo. De esta manera también se evita que se pueda cerrar un ciclo de esperas. Se podría llegar a encadenar cierta secuencia de esperas, siguiendo un orden descendente dentro de los niveles de la arquitectura. Sin embargo, dicha secuencia pararía en el nivel más bajo y jamás se establecería una espera hacia arriba cerrando algún ciclo.

4.4.2 Evitación

Las estrategias de prevención impiden que el interbloqueo pueda darse en un sistema y lo consiguen rompiendo al menos una de las condiciones de Coffman. Sin embargo, siguen planteando problemas que también impiden recomendar dicha aproximación de manera general. Los más graves son la baja utilización de los recursos (si se rompe retención y espera o la no expulsión) y el tener que solicitar los recursos en un orden artificial (al romper la espera circular). Además, esto último también conduce a una baja utilización para aquellos recursos que se soliciten antes de que el proceso los necesite.

Una solución mejor consistiría en permitir que los procesos soliciten los recursos cuando los necesiten, verificando entonces si la atención de tal petición entrañará algún riesgo para que en un futuro próximo se cierre un ciclo de esperas que genere un interbloqueo. Esa es la aproximación seguida en las estrategias de *evitación*. Sin embargo, para que sea factible la implantación de estas técnicas se necesita que todos los procesos declaren cuáles serán sus necesidades máximas de cada recurso. Dicha información debe ser facilitada al sistema cuando cada proceso empiece su ejecución y el sistema la gestionará para decidir si el estado resultante tras atender cada petición de recursos será seguro o no.

En las estrategias de evitación se habla de un *estado seguro*[Hab69] cuando se pueda encontrar una secuencia de atención de procesos tal que las necesidades completas de recursos de un proceso P_i no superen las instancias disponibles ac-

tualmente de tales recursos más las que liberarán los procesos P_j que precedan a P_i en tal secuencia.

Esta definición de estado seguro asume que los procesos podrán obtener todos los recursos que necesiten y que, tras ello, conseguirán terminar, liberando entonces todo lo que tenían asignado. En ese momento, esas instancias liberadas quedarán disponibles para los siguientes procesos de la secuencia, que también se comportarán de esa manera (liberándolo todo en su terminación y dejándolo disponible para los demás).

La secuencia de atención de procesos que se utiliza en la definición de estado seguro se conoce como *secuencia segura*[Hab69].

Existen algunos algoritmos que verifican si la atención de una petición de recursos seguirá permitiendo que haya alguna secuencia segura. De ser así, la petición se atiende y el proceso obtiene el recurso o recursos solicitados. En caso contrario, cuando el estado resultante tras la atención de esa petición dejara al sistema sin ninguna secuencia segura posible, la petición es rechazada y el proceso tendrá que reintentarla más tarde. Cuando no haya ninguna secuencia segura, el sistema no presentará un estado seguro. Cuando un sistema no ofrece un estado seguro, existe riesgo de que se genere un interbloqueo. El objetivo de los algoritmos de evitación es garantizar que el sistema siempre se mantenga en un estado seguro. Con ello se evita el interbloqueo. Un ejemplo de este tipo de algoritmos es el *algoritmo del banquero*, propuesto por Dijkstra [Dij68] para un solo recurso y refinado por Habermann [Hab69] para múltiples recursos. Posteriormente hubo un buen número de algoritmos de este tipo [IK82, Min82], variando ligeramente sus requisitos.

No vamos a revisar aquí tales algoritmos, ya que dependen de que los procesos comuniquen sus necesidades máximas de recursos y eso ya no se exige en los sistemas actuales de propósito general. No obstante, esta solución tuvo sentido en los sistemas multiprogramados por lotes.

4.4.3 Detección y recuperación

Si las estrategias de prevención no son aconsejables, pues reducen la utilización de los recursos y las estrategias de evitación tampoco son convenientes por requerir que cada proceso comunique al sistema qué cantidad máxima de recursos llegará a necesitar, parece obvio que la única estrategia que tendrá sentido será una que esté basada en la detección y recuperación de interbloqueos.

Los sistemas operativos modernos no suelen preocuparse por la gestión de los interbloqueos. Por tanto, optan por la cuarta alternativa posible: no hacer nada, asumiendo que el usuario será quien se encargue de detectarlos y recuperarlos cuando ocurran. Sin embargo, aparte de los sistemas operativos hay otros gestores de recursos que sí llegan a gestionar las situaciones de interbloqueo y que optan por

una estrategia de detección. Son los *sistemas gestores de bases de datos* (SGBD). Un SGBD mantiene un conjunto de tablas en un dispositivo de almacenamiento secundario y gestiona transacciones para consultar y modificar los datos mantenidos en tales tablas. Necesita algún mecanismo de control de concurrencia para mantener la consistencia de esa información cuando sea accedida por múltiples transacciones simultáneamente. Dicho control de concurrencia suele estar basado en *locks*, que se utilizan implícitamente al ejecutar las sentencias contenidas en las transacciones. Los interbloqueos pueden darse fácilmente en un sistema de este tipo y la estrategia clásica para gestionarlos ha sido la detección y recuperación [ABC+76]. Los algoritmos de detección suelen emplear principios similares a los que ya se han descrito en la sección 4.3.1, en la que se explicó el algoritmo de reducción de grafos. El objetivo es detectar el interbloqueo y eliminarlo. Para ello, habrá que seleccionar aquella transacción (o proceso) que esté en el ciclo de esperas y haya sido la última en solicitar su lock (pues lleva menos tiempo bloqueada) o haya sido la última en ser iniciada (pues al ser la más reciente, se supone que habrá realizado menos actualizaciones) para abortarla. Al abortar tal transacción se liberarán los recursos que tuviera asignados y con ello se romperá el ciclo de esperas, eliminando el interbloqueo.

Estos sistemas usan el algoritmo de detección iniciándolo de manera periódica o cada vez que una transacción se bloquee tratando de obtener algún *lock*.

Los SGBD actuales siguen utilizando esta aproximación, como puede observarse en su documentación. Algunos ejemplos son: Microsoft SQL Server 2008 R2 [Mic10], Oracle Database 11g Release 2 [Ora11], PostgreSQL 9.2.0 [Pos12], ...

4.4.4 Ejemplos de soluciones

Cuando se utilicen las estrategias de detección y evitación, bastará con utilizar los algoritmos correspondientes para gestionar las posibles situaciones de interbloqueo. La forma en que se utilicen tales algoritmos no depende de los procesos o del entorno que pueda provocar tal interbloqueo. Sin embargo, cuando se quiera utilizar alguna estrategia de prevención, la solución que deba adoptarse sí que dependerá del programa o aplicación que se pretenda ejecutar en el sistema. El objetivo de tales soluciones será romper alguna de las cuatro condiciones de Coffman. Como ejemplo, se analizará a continuación una serie de soluciones para el problema de los *cinco filósofos* [Dij71], ya introducido en la sección 1.3.3:

- Forzar a que cada filósofo solicite a la vez ambos tenedores. De esta manera, la petición solo tendrá éxito cuando el proceso solicitante obtenga simultáneamente ambos recursos. Si no están los dos libres, el filósofo correspondiente permanecerá bloqueado sin retener ninguno de los dos recursos que necesita.

  Esta solución rompe la condición de retención y espera. Fue descrita en [Dij71], donde también se observó que podía conducir a situaciones de ina-

nición, dependiendo de cómo fueran atendidas las peticiones múltiples por parte del gestor de recursos.

- Numerar los tenedores y exigir un orden global de petición. Este es el principio que ya se explicó en la sección 4.4.1. Para el caso particular de los cinco filósofos, se puede asumir que existirán cinco hilos de ejecución (F_0, F_1, F_2, F_3, F_4) que representarán a los filósofos y cinco recursos (T_0, T_1, T_2, T_3, T_4) que representarán a los tenedores. Se asume que la ordenación es creciente en base al identificador de cada recurso. Esto es $T_i < T_j$ si $i < j$. Se exige también que los filósofos soliciten los tenedores en orden creciente.

  La figura 4.9 muestra cómo se ubicarían los filósofos y los tenedores en la mesa. Para cada filósofo se ha identificado qué plato utilizaría. Con ello también se está indicando qué tenedores tendrá que usar.

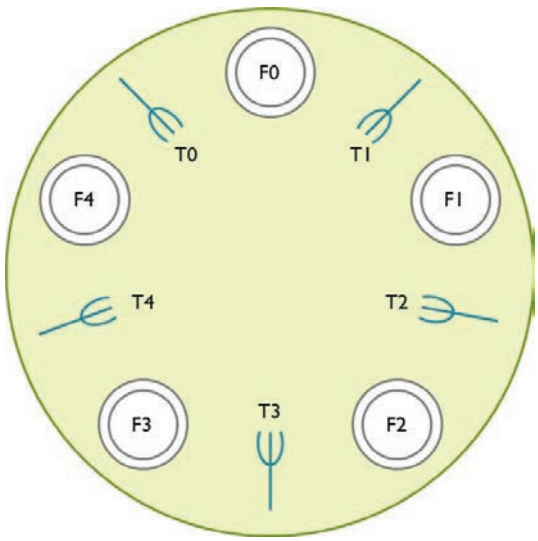

Figura 4.9: Problema de los cinco filósofos.

De este modo, F_0 necesitará los tenedores T_0 y T_1, F_1 el T_1 y el T_2, F_2 el T_2 y el T_3, F_3 el T_3 y el T_4, mientras que F_4 requiere el T_0 y T_4. Además, deben solicitarlos en el orden que acabamos de utilizar. Dicho orden de peticiones romperá la espera circular pues resultará imposible que todos los procesos lleguen simultáneamente a la mesa y todos ellos consigan su primer tenedor a la vez. Obsérvese que tanto F_0 como F_4 necesitan obtener en primer lugar el mismo tenedor (T_0). Sólo uno de ellos lo conseguirá. El otro se quedará esperando sin retener ningún recurso. Por ello, no llega a cerrarse el ciclo de esperas cuando cada uno de los filósofos no bloqueados pase a pedir su segundo tenedor.

- Asimetría en las peticiones. El interbloqueo podía llegar a darse en este problema de los cinco filósofos cuando todos ellos sigan el mismo procedimiento a la hora de solicitar sus tenedores. Por ejemplo, si todos piden en primer lugar su tenedor derecho y después el izquierdo, podrá haber interbloqueo, como ya vimos en la página 76. Si seguimos un orden creciente a la hora de pedir los tenedores, el filósofo F_4 rompe esta simetría y con ello rompe también la condición de espera circular. Otra forma romper la simetría es obligar a que los filósofos pares y los impares pidan los tenedores de forma distinta. Por ejemplo, los pares seguirán pidiendo primero su tenedor derecho para después pedir el izquierdo, mientras que los impares pedirán primero su tenedor izquierdo y después el derecho. Así se romperá tanto la propiedad de retención y espera como la de espera circular.
- Limitar el grado de concurrencia. Se puede programar la solución a este problema de manera que no pueda haber cinco filósofos simultáneamente en la mesa. Para ello se podría asumir que la mesa en la que discurre este problema está ubicada en cierto comedor y que los filósofos no permanecen en él para pensar, sino que salen a una sala de estar. En el hipotético caso de que los cinco filósofos tuvieran hambre a la vez, el último de ellos que pretenda entrar en el comedor no podrá hacerlo. Para ello se asumirá que existe un mayordomo que cierra la puerta del comedor tan pronto como haya cuatro filósofos sentados, evitando que el quinto entre en el comedor. Así, también se consigue romper la condición de espera circular.

4.5 Resumen

Se dice que un conjunto de procesos (o hilos) se encuentra en un estado de interbloqueo cuando todos ellos se encuentran bloqueados esperando algún recurso que mantiene retenido otro proceso (o hilo) del grupo. Un interbloqueo se produce solo si se dan simultáneamente en el sistema las cuatro condiciones necesarias (Condiciones de Coffman): exclusión mutua, retención y espera, no expulsión y espera circular. Con la utilización de un grafo de asignación de recursos (GAR), que muestra las peticiones de recursos por parte de los procesos (o hilos) y las asignaciones realizadas, se puede detectar si en un momento determinado existe o puede existir un interbloqueo.

Existen cuatro estrategias posibles para afrontar las situaciones de interbloqueo en un sistema: (i) prevención, diseñando el sistema de modo que se rompa alguna de las condiciones de Coffman, asegurando así que el sistema nunca entrará en un estado de interbloqueo; (ii) evitación, monitorizando las peticiones de recursos y evaluando si resulta seguro asignar las instancias solicitadas; (iii) detección y recuperación, permitiendo que el sistema entre en un estado de interbloqueo, pero detectándolo y luego recuperándose de él, terminando alguno de los procesos

interbloqueados o desalojando recursos que tengan ocupados; y (iv) ignorar el problema, actuando como si los interbloqueos nunca fueran a producirse en el sistema. Esta última opción es la adoptada por la mayoría de los sistemas operativos actuales, que asumen que será el diseñador quien se encargue de aplicar estrategias de prevención o bien el usuario quien detecte los interbloqueos y se recupere de ellos cuando ocurran. En esta unidad se han revisado diferentes ejemplos de soluciones para las diferentes estrategias existentes.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar los problemas que puede originar un uso incorrecto de los mecanismos de sincronización. En concreto, se identificará el problema de interbloqueo.
- Caracterizar las situaciones de interbloqueo.
- Conocer las técnicas de gestión de interbloqueos y decidir qué aproximación es la más adecuada para cada tipo de aplicación y entorno.

Unidad 5

BIBLIOTECA java.util.concurrent

5.1 Introducción

Java ha soportado la generación de múltiples hilos de ejecución en un mismo programa desde su primera versión. Sin embargo, hasta la versión 1.5.0 no existía ninguna biblioteca de interfaces y clases que proporcionaran ayuda para diseñar o utilizar herramientas complementarias de sincronización de alto nivel. El programador debía conformarse con el uso de monitores, utilizando la variante ya descrita en la unidad 3, con importantes limitaciones al compararla con las variantes tradicionales. Es cierto que con estos monitores ya se podía implantar otras herramientas; por ejemplo, es relativamente fácil el diseño y desarrollo de semáforos y locks, utilizando como base un monitor Java. Sin embargo, esto obligaba a que cada programador ideara sus propias soluciones y que el código resultante no siempre fuera óptimo en cuanto a consumo de recursos y eficiencia. Por ello, en la versión 1.5.0 de Java (que finalmente fue presentada como Java 2 Platform Standard Edition 5.0) se incluyó una biblioteca `java.util.concurrent` en la que era posible encontrar un buen número de herramientas de sincronización para facilitar el desarrollo de aplicaciones concurrentes en este lenguaje.

Esta unidad se centra en la descripción de las principales interfaces y clases existentes en `java.util.concurrent`, proporcionando algunos ejemplos sencillos que ilustran cómo pueden utilizarse y qué problemas pretenden resolver. Esta descripción puede complementarse con el tutorial sobre programación concurrente en Java incluido en la documentación oficial de este lenguaje [Ora12c], así como en las páginas de dicha documentación dedicadas a cada una de las interfaces y clases que se irán comentando en las secciones siguientes.

5.2 Inconvenientes de las primitivas básicas de Java

El lenguaje de programación Java facilita de manera implícita un buen número de herramientas de sincronización. Entre ellas sobresale su soporte para monitores, capaz de garantizar exclusión mutua entre todos aquellos métodos calificados como "`synchronized`" para un determinado objeto. Además, esto se combina con la declaración implícita de una condición interna para dicho objeto, sobre la que se podrán utilizar los métodos `wait()`, `notify()` y `notifyAll()` implantando una forma limitada de la variante de Lampson y Redell [LR80].

Utilizar dicho soporte no requiere ningún esfuerzo adicional: basta con diseñar nuestros propios monitores e implantarlos utilizando este lenguaje. No obstante, estos monitores básicos de Java presentan las siguientes limitaciones, algunas de ellas ya comentadas en la unidad 3:

- Limitaciones relacionadas con la exclusión mutua:
  1. En caso de encontrar el monitor ocupado, la espera a la que se ve forzado el hilo para acceder al código del monitor no puede ser interrumpida voluntariamente. Es decir, no se puede establecer un plazo máximo de espera a la hora de "solicitar" la entrada.
  2. En ocasiones interesaría preguntar por el estado del monitor (o la sección crítica protegida por un *lock*) antes de solicitar el acceso a tal método o sección. De esta manera, en caso de encontrarlo ocupado se podría pasar a ejecutar otras acciones, reintentándolo posteriormente. Esta es una buena estrategia para prevenir los interbloqueos (ya que con ella se podría romper la condición de retención y espera, como ya se explicó en la sección 4.4.1). Esta estrategia no está disponible en los monitores básicos de Java.
  3. Las herramientas que garantizan exclusión mutua están orientadas a bloques. El calificador `synchronized` puede aplicarse tanto a métodos completos como a secciones de código más pequeñas, definidas como bloques constituidos por múltiples sentencias comprendidas entre un par (llave-abierta {, llave-cerrada }). En ambos casos no resultará posible cerrar un *lock* en un determinado método y abrirlo posteriormente en otro método.
  4. No podemos extender su semántica. Por ejemplo, no podremos utilizar estas construcciones para resolver el problema de los lectores-escritores [CHP71], en el que la exclusión mutua debe darse entre múltiples hilos escritores o entre hilos escritores y lectores, pero no entre múltiples hilos lectores. Es decir, en Java básico no dispondremos de ningún tipo particular de *lock* con tal semántica y tendremos que implantar nosotros mismos algún monitor que proporcione tal soporte, en caso de que lo necesitemos en nuestra aplicación concurrente.

- Limitaciones relacionadas con la sincronización condicional:
  1. Solo podrá existir una única condición en cada monitor. Esto obliga a que, independientemente del motivo por el que se suspendan o de lo que esperen para reactivarse, todos los hilos que deban suspenderse dentro de un monitor vayan a parar a una misma cola.
  2. Se utiliza la variante de Lampson y Redell. Con ello, cuando un hilo sea reactivado, los valores que encontrará en los atributos del monitor quizá no sean los que esperaba y debido a esto tendrá que volverse a suspender en esa misma condición implícita. Por esa misma razón el programador estará obligado a utilizar una estructura del tipo:

     `while (`*expresión lógica*`) wait();`

     para consultar el estado del monitor y suspenderse, en lugar de la más simple:

     `if (`*expresión lógica*`) wait();`

     que encontrábamos en las variantes de Brinch Hansen y Hoare. En estas últimas también se admitía la definición de múltiples atributos de tipo `Condition` en un mismo monitor.

5.3 La biblioteca java.util.concurrent

Las limitaciones que se han comentado en la sección anterior pudieron eliminarse con la incorporación de la biblioteca `java.util.concurrent`[1] en J2SE 5.0. Este `package` proporcionó un buen número de clases e interfaces que facilitan el desarrollo de aplicaciones concurrentes. Las próximas secciones estudiarán detalladamente algunas de las herramientas proporcionadas. También es recomendable consultar el tutorial sobre esta biblioteca incluido en la documentación de Java [Ora12c].

5.3.1 Locks

El "*package*" `java.util.concurrent.locks` proporciona diferentes clases e interfaces para la gestión o desarrollo de múltiples tipos de *locks*. Entre las características que cabe resaltar encontramos las siguientes:

- Los constructores de las clases presentan un argumento "`boolean fair`" que permite especificar si se requiere o no una gestión equitativa de la cola de espera mantenida por el *lock*. En caso de que se solicite la gestión equitativa, la cola de espera se tratará en orden FIFO. Este argumento es opcional; es decir, existe otra versión del constructor que no requiere ningún argumento y que no utilizará una gestión equitativa. La documentación indica que esta segunda variante permite una gestión mucho más eficiente del *lock*.

[1] No es una biblioteca propiamente dicha sino un `package` Java.

- Se facilitan varios tipos de *locks*, con semántica diferente. Por ejemplo, hay tanto *locks* orientados a exclusión mutua (como pueda ser la clase `ReentrantLock`) como otros que resuelven el problema de lectores-escritores (como es el caso de `ReentrantReadWriteLock`).

  La clase `ReentrantLock` implanta un *lock* reentrante. De manera general, se dice que una función, procedimiento o método es *reentrante* cuando su ejecución puede ser interrumpida y durante esa interrupción se llega a ejecutar esa misma función, procedimiento o método sin causar problemas en la ejecución interrumpida. Esto último puede suceder en un sistema monoprogramado si dentro del código que maneja tal interrupción se invoca a la función, procedimiento o método que había sido interrumpido. Por su parte, en los sistemas multiprogramados puede suceder más fácilmente: basta con que múltiples hilos ejecuten simultáneamente esa función, procedimiento o método. En general, para que un fragmento de código sea reentrante debe evitarse que lea y modifique variables globales.

  Cuando se hable de un lock (o de cualquier otra herramienta de sincronización) reentrante, esto implicará que dentro de la sección de código protegida por esa herramienta de sincronización se podrá volver a utilizar dicha herramienta sin que haya problemas de bloqueo. Por ejemplo, un lock se considera reentrante si se puede invocar sucesivas veces a su método de cierre sin que esto implique el bloqueo perpetuo del hilo que lo haga.

- Existe un método `tryLock()` que no suspende al invocador en caso de que el *lock* ya esté cerrado por otro hilo, retornando un valor `false` en ese caso sin realizar ninguna espera. Esto permite romper la condición de retención y espera con el objetivo de prevenir las situaciones de interbloqueo, como ya se describió en la sección 4.4.1.

Como ya se ha comentado, una de las clases proporcionadas es `ReentrantLock` cuyos métodos se listan en la tabla 5.1 y están descritos en profundidad en la documentación del lenguaje Java. Esta clase rompe algunas de las limitaciones que planteaban los monitores básicos de Java de la siguiente manera:

- Se permite especificar un plazo máximo de espera para obtener el *lock*. Para ello debe utilizarse el método `tryLock()` en su variante con dos argumentos.

- Se pueden definir tantas condiciones como sea necesario en un determinado monitor. Esto se consigue mediante el método `newCondition()`, que crea una nueva condición asociada al lock sobre el que se invoque tal método. Dicho método puede invocarse tantas veces como sea necesario, creando una condición distinta en cada una de las invocaciones.

- Permite cerrar y abrir los *locks* en diferentes métodos de la aplicación. Es decir, no restringe el uso de estos objetos a la construcción de monitores.

Métodos de la clase `ReentrantLock`
public ReentrantLock()
public ReentrantLock(boolean fair)
public void lock()
public void lockInterruptibly() throws InterruptedException
public boolean tryLock()
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException
public void unlock()
public Condition newCondition()
public int getHoldCount()
public boolean isHeldByCurrentThread()
public boolean isLocked()
public final boolean isFair()
protected Thread getOwner()
public final boolean hasQueuedThreads()
public final boolean hasQueuedThread(Thread thread)
public final int getQueueLength()
protected Collection<Thread> getQueuedThreads()
public boolean hasWaiters(Condition condition)
public int getWaitQueueLength(Condition condition)
protected Collection<Thread> getWaitingThreads(Condition condition)
public String toString()

Tabla 5.1: Métodos de la clase `ReentrantLock`.

Se pueden utilizar para proteger el acceso a cualquier recurso. Por tanto, es factible que un hilo cierre un `ReentrantLock` en un método en el que obtenga acceso a un determinado recurso y que después lo abra desde otro método de su código en el que se libere tal recurso.

- En caso de utilizar el método `tryLock()` (en su variante con argumentos) o `lockInterruptibly()`, si el hilo invocador debe suspenderse por encontrar el *lock* ya cerrado, tal espera podrá ser interrumpida, utilizando para ello el método `Thread.interrupt()` ya explicado en la sección 2.2.1.

A pesar de que todas estas características son ventajas del `ReentrantLock` frente al calificador `synchronized` de los monitores básicos de Java, también existen algunos inconvenientes que vale la pena resaltar. Son estos:

- Cuando se utilice un monitor básico de Java, la gestión de los *locks* es implícita. Por ello, el programador no debe preocuparse del cierre y apertura de los *locks*. El propio *runtime* de Java se encargará de estas labores.

  Si nos decidimos a utilizar `ReentrantLock`, tendremos que revisar cuidadosamente el código para asegurarnos de que, una vez se haya liberado un recurso que debía utilizarse en exclusión mutua, el *lock* correspondiente se abra.

- Debe prestarse especial atención a la sentencias que puedan generar excepciones y queden dentro de una sección crítica protegida con un `ReentrantLock`. Cuando se genere una excepción así, el código correspondiente debe asegurar que se ejecute el método `unlock()` sobre dicho `ReentrantLock`.

  Para ello se recomienda estructurar los protocolos de entrada y salida de dicha sección crítica tal como se muestra en la figura 5.1.

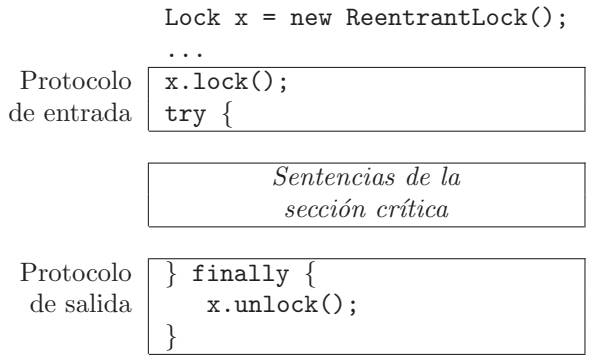

Figura 5.1: Sección crítica protegida por un `ReentrantLock`.

Así, habrá que abrir un bloque "`try { ... } finally {...}`" justo después de cerrar el *lock*. En dicha construcción se dejará dentro del `finally` la invocación al método `unlock()` del `ReentrantLock`. Con ello se garantiza que el `unlock()` siempre se invoque, pues el fragmento de código asociado a `finally` siempre se ejecutará al terminar el bloque protegido con el `try`, tanto cuando se dé una excepción como cuando no se genere ninguna.

Obsérvese que no es necesario dejar la sentencia "`x.lock();`" dentro del bloque asociado a `try`, pues el método `lock()` no puede ser interrumpido.

5.3.2 Condition y monitores

Una de las limitaciones más serias de los monitores básicos de Java residía en el hecho de que solo hubiese una cola de espera interna (implícita) en cada monitor, mientras que las variantes clásicas de monitor permitían declarar múltiples atributos de tipo `Condition` para definir múltiples colas de espera. Como ya hemos visto en la sección 5.3.1 esta limitación desaparecerá cuando se utilicen objetos de la clase `ReentrantLock` para simular un monitor. Su método `newCondition()` permite generar todas las colas de espera que resulten necesarias. Dicho método devuelve un objeto que implanta la interfaz `Condition`, que a su vez ofrece los métodos listados en la tabla 5.2.

Métodos de la interfaz `Condition`
void await() throws InterruptedException
void awaitUninterruptibly()
long awaitNanos(long nanosTimeout) throws InterruptedException
boolean await(long time, TimeUnit unit) throws InterruptedException
boolean awaitUntil(Date deadline) throws InterruptedException
void signal()
void signalAll()

Tabla 5.2: Métodos de la interfaz `Condition`.

Como puede observarse, sus métodos pueden dividirse en dos categorías. En la primera entrarían todos los métodos utilizados para suspender a un hilo en la condición. En ella se incluye el método `await()` con todas sus variantes. Obsérvese la diferencia respecto al método utilizado en los monitores básicos (en estos últimos se llama `wait()`, mientras que en la interfaz `Condition` pasa a llamarse `await()` para evitar confusiones). En la segunda categoría tendríamos los métodos necesarios para notificar la ocurrencia del evento esperado. Los métodos correspondientes se llaman `signal()` y `signalAll()` (recuérdese que en los monitores básicos los métodos que tenían esta misma funcionalidad se llamaron `notify()` y `notifyAll()`, respectivamente).

Empleando este soporte, se podría reescribir el monitor de la figura 3.12 (véase la página 66) utilizando el código que aparece en la figura 5.2 para implantar un buffer de capacidad limitada. En ese código se puede observar cómo debe ser utilizado el `ReentrantLock` para garantizar la exclusión mutua en la ejecución de todos los métodos y cómo pueden usarse dos condiciones distintas en un mismo monitor. La condición `notFull` sirve para suspender a los hilos productores que

```
public class BoundedBuffer {
    private int buffer[];
    private int numElem;
    private int capacity;
    private int first, last;
    private ReentrantLock l;
    private Condition notFull, notEmpty;

    public BoundedBuffer(int size) {
        buffer = new int[size];
        numElem = first = last = 0;
        capacity = size;
        l = new ReentrantLock();
        notFull = l.newCondition();
        notEmpty = l.newCondition();
    }

    public void put(int elem) {
      l.lock();
      try {
        while (numElem == capacity)
            notFull.await();
        buffer[last] = elem;
        numElem++;
        last = (last + 1) % capacity;
        notEmpty.signalAll();
      } finally {
        l.unlock();
      }
    }

    public int get() {
      int item;
      l.lock();
      try {
        while (numElem == 0)
            notEmpty.await();
        item = buffer[first];
        first = (first + 1) % capacity;
        numElem--;
        notFull.signalAll();
        return item;
      } finally {
        l.unlock();
      }
    }
}
```

Figura 5.2: Monitor `BoundedBuffer`.

encuentren el buffer lleno cuando intenten insertar un nuevo elemento. Por su parte, la condición `notEmpty` suspende a los hilos consumidores si encuentran el buffer vacío. Estos monitores siguen respetando la variante de Lampson y Redell por lo que conviene utilizar un bucle `while` a la hora de comprobar si la condición que obligó a suspender al hilo se sigue cumpliendo cuando éste es reactivado.

5.3.3 Colecciones concurrentes

En algunos casos un programa necesita utilizar alguna colección de objetos. En las bibliotecas del lenguaje Java existe un buen número de estructuras de datos que permiten manejar colecciones de objetos. Algunos ejemplos son:

- Los vectores asociativos, que permiten indexar objetos, asociándoles claves, permitiendo así realizar búsquedas rápidas en función del valor de la clave asignada. Estos vectores respetarán la interfaz `Map`. Existe un buen número de clases que implantan tal interfaz: `EnumMap`, `HashMap`, `TreeMap`, `Attributes`, `Hashtable`, `Properties`, `TreeMap`, etc.
- Secuencias ordenadas de objetos, que permiten elegir en qué posición se insertará un nuevo objeto, generando así una lista dinámica (esto es, con un tamaño que no está fijado durante su definición y que se irá incrementando a medida que se vayan insertando nuevos elementos). Estas secuencias respetan la interfaz `List`. Hay varias clases Java que implantan tal interfaz: `ArrayList`, `LinkedList`, `Stack`, `Vector`, etc.
- Colas de objetos, que son un tipo particular de listas que siguen una política FIFO a la hora de extraer sus elementos. Estas estructuras de datos respetan la interfaz `Queue`. Un ejemplo de clase que implanta tal interfaz es `PriorityQueue`.

Las clases que se han citado hasta el momento como ejemplos de estos tipos de colecciones no son *thread-safe*. Esto implica que su comportamiento no será siempre el esperado en caso de que sean accedidas simultáneamente por múltiples hilos.

El *package* `java.util.concurrent` incluye otras clases que implantan esos tipos de colecciones y sí que son *thread-safe*. Esas clases son `ConcurrentHashMap` y `ConcurrentSkipListMap`, implantando la interfaz `Map`. `CopyOnWriteArrayList` es la utilizada para implantar la interfaz `List`. Por último, `ArrayBlockingQueue`, `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, `DelayQueue` y `SynchronousQueue` son las que implantan la interfaz `Queue`.

La documentación oficial de Java incluye una descripción de cada una de estas clases e interfaces. Para ilustrar su funcionamiento estudiaremos la interfaz `BlockingQueue`, que extiende a `Queue` y que podrá utilizarse para resolver el problema de los productores-consumidores, implantando de manera inmediata el buffer de capacidad limitada.

Para ello, la tabla 5.3 lista los métodos que ofrece la interfaz `BlockingQueue`, cuya funcionalidad se explica seguidamente:

Métodos de la interfaz `BlockingQueue`
boolean add(E e)
boolean offer(E e)
boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException
void put(E e) throws InterruptedException
E take() throws InterruptedException
E poll()
E poll(long timeout, TimeUnit unit) throws InterruptedException
boolean remove(Object o)
E remove()
E peek()
int remainingCapacity()
boolean contains(Object o)
int drainTo(Collection<? super E> c)
int drainTo(Collection<? super E> c, int maxElements)

Tabla 5.3: Métodos de la interfaz `BlockingQueue<E>`.

- `add()`: Inserta el elemento especificado en la cola, retornando `true` en ese caso. Si no hay espacio en la cola, se genera una `IllegalStateException`.
- `offer()`: Inserta el elemento especificado en la cola, retornando `true` en ese caso. Si no quedase espacio en la cola, se retorna el valor `false`. En la variante con dos argumentos, si la cola no tiene espacio suficiente para insertar el elemento, el hilo invocador se bloquea durante un intervalo máximo especificado en tales argumentos, desbloqueándose en caso de que otros hilos extraigan algún elemento.
- `put()`: Inserta el elemento especificado en la cola, esperando hasta que la cola tenga suficiente espacio libre.
- `take()`: Recupera y elimina el primer elemento (o cabeza) de la cola, esperando si fuera necesario hasta que haya algún elemento que extraer.
- `poll()`: Hace lo mismo que el método anterior, pero en caso de que no haya elementos en la cola, se esperará como máximo el intervalo especificado a que haya algún elemento. En caso de que se utilice la variante sin argumentos y no haya ningún elemento en la cola, se retorna de inmediato un valor `null`.

- `remove()`: En la variante con un argumento, elimina de la cola una instancia del objeto pasado como argumento devolviendo `true` en caso de encontrarlo. La variante sin argumentos elimina y retorna el primer elemento de la cola o devuelve `null` en caso de que la cola esté vacía.
- `peek()`: Retorna el primer elemento de la cola, sin extraerlo de ella, o devuelve `null` en caso de que la cola esté vacía.
- `remainingCapacity()`: Devuelve el número de elementos que todavía se pueden insertar en la cola. En caso de que la cola tenga capacidad ilimitada devolverá `Integer.MAX_VALUE`.
- `contains()`: Devuelve `true` si la cola contiene el objeto facilitado como argumento.
- `drainTo()`: En la variante con un solo argumento, elimina todos los elementos de esa cola y los añade a la colección facilitada como argumento. En la variante de dos argumentos, traslada a la colección como máximo el número de elementos especificado en el segundo argumento.

Para resolver el problema de los productores-consumidores que acceden a un buffer de capacidad limitada se podría utilizar cualquiera de las clases que implanten esta interfaz y que acoten la capacidad de la cola. Un ejemplo es `ArrayBlockingQueue`. En su constructor se puede pasar como único argumento cuál será la capacidad de la cola. Si asumimos que los hilos productores y consumidores gestionan elementos de tipo `Integer` y la capacidad del buffer está limitada a cinco elementos, se podría utilizar el código que se muestra en la figura 5.3.

En ese ejemplo se crean tres productores (que generan una secuencia de números positivos ordenada e iniciada en cero) y un solo consumidor (que imprime cada uno de los elementos extraídos del buffer), que es iniciado en primer lugar y empezará encontrando el buffer vacío. Posteriormente los consumidores llenarán el buffer y se irán bloqueando, pero no se perderá ninguno de los elementos que se pretendía insertar.

Para comprobar que este código funciona, copie el código en un fichero llamado `Setup.java`, compílelo y ejecútelo. Redirija su salida hacia algún fichero e interrumpa la ejecución del proceso pulsando `[Ctrl]+[C]` en la terminal, pues todos los hilos ejecutan un bucle infinito y no terminarían nunca. Observará que la traza obtenida no pierde ninguno de los elementos de la secuencia generada por cada productor.

```
import java.util.concurrent.*;

class Producer implements Runnable {
  private final BlockingQueue<Integer> queue;
  private int i;
  Producer(BlockingQueue<Integer> q) { queue=q; i=0; }
  public void run() {
    try {
      while(true) { queue.put(produce()); }
    } catch (InterruptedException ex) {}
  }
  Integer produce() { return new Integer(i++); }
}

class Consumer implements Runnable {
  private final BlockingQueue<Integer> queue;
  Consumer(BlockingQueue<Integer> q) { queue = q; }
  public void run() {
    try {
      while(true) { consume(queue.take()); }
    } catch (InterruptedException ex) {}
  }
  void consume(Integer x) { System.out.println("Elem: "+x); }
}

class Setup {
  static public void main(String args[]) {
    BlockingQueue<Integer> q = new ArrayBlockingQueue<Integer>(5);
    Producer p1 = new Producer(q); Producer p2 = new Producer(q);
    Producer p3 = new Producer(q); Consumer c = new Consumer(q);
    new Thread(c).start();
    new Thread(p1).start();
    new Thread(p2).start();
    new Thread(p3).start();
  }
}
```

Figura 5.3: Gestión de un buffer de capacidad limitada.

5.3.4 Variables atómicas

La especificación del lenguaje Java admite que cuando un hilo lea repetidas veces el valor de un determinado atributo, solo en su primera lectura acceda al valor mantenido en memoria principal. El resto de los accesos pueden consultar una copia del valor mantenida en una especie de `caché` asociada a cada hilo. Esto puede plantear problemas pues, durante ese intervalo, otros hilos han podido modificar el valor de ese atributo y el hilo lector no se daría cuenta de que tales cambios han sucedido.

Una primera solución para este problema consiste en usar el calificador `volatile`. Al utilizarlo se está exigiendo que cuando deba accederse a tal atributo no se

utilizarán nunca las cachés de los hilos. Todos los accesos sobre ese atributo, tanto de lectura como de escritura se realizarán directamente sobre la posición que ocupe en memoria principal. Esto evita que se lean involuntariamente valores ya obsoletos de los atributos en cuestión. Sin embargo, no basta con esas medidas para garantizar que los programas concurrentes siempre se comporten correctamente. En concreto, cuando haya modificaciones concurrentes sobre el atributo, nada garantiza que tales operaciones obtengan el valor esperado. Por ejemplo, una sentencia tan sencilla como "`i++;`" sobre un atributo "`i`" de tipo "`int`" se traduce en varias instrucciones de la máquina virtual Java. El planificador podría expulsar a un hilo que ejecute "`i++;`" en mitad de esa secuencia de instrucciones. Eso puede tener graves consecuencias (pérdida de actualizaciones) si otros hilos actualizan esa misma variable entre tanto. Esto se debe a que una operación de actualización (suma, resta, multiplicación, ...) es una sección crítica y tal sección crítica debe protegerse garantizando su ejecución en exclusión mutua. Tal garantía está fuera del ámbito del calificador `volatile`.

Para asegurar una ejecución correcta en estos casos habría que usar `synchronized` para proteger el método o bloque de código en el que se realice la modificación del atributo, o bien emplear la clase `ReentrantLock` (u otra similar) para garantizar exclusión mutua. Ambas soluciones son correctas, pero costosas (es decir, ineficientes) pues exigen un buen número de sentencias de bajo nivel para implantar tales "construcciones". Lo ideal sería disponer de alguna instrucción de bajo nivel que asegurase exclusión mutua. En algunos procesadores existen tales instrucciones. Por ello, Java utiliza algunas clases "atómicas" que facilitan una serie de métodos que en tales procesadores aprovecharán esas instrucciones existentes para facilitar la exclusión mutua en las operaciones de modificación y consulta de un valor, asegurando una *consistencia atómica* [Lam86]. En un modelo de consistencia atómica se garantiza que una vez que algún hilo o proceso haya sido capaz de leer un determinado valor de una variable, ningún otro hilo o proceso llegará a leer más tarde un valor más antiguo. En la práctica, esto implica que cuando haya escrituras concurrentes, éstas se realizarán de manera indivisible (primero una y luego otra, sin mezclar las instrucciones que se necesite para implantarlas) y todos estarán de acuerdo sobre qué secuencia se ha empleado para ordenarlas (incluso si afectan a diferentes variables).

Estas clases atómicas están en el *package* `java.util.concurrent.atomic` y son las que aparecen listadas en la tabla 5.4.

El objetivo de estas clases es garantizar la ejecución no interrumpible (para lograr así exclusión mutua) de ciertas secuencias de acciones: incrementar, decrementar, consultar e incrementar, incrementar y consultar, etc. Para ilustrar los conjuntos de métodos que ofrecen, se utilizará como ejemplo la clase `AtomicInteger`, cuyos métodos aparecen en la tabla 5.5 y se explican seguidamente:

Clases mantenidas en `java.util.concurrent.atomic`	
AtomicBoolean	Booleano con actualización atómica.
AtomicInteger	Entero con actualización atómica.
AtomicIntegerArray	Vector de enteros con actualización atómica.
AtomicIntegerFieldUpdater<T>	Permite acceso a atributos `volatile int`.
AtomicLong	Entero `long` con actualización atómica.
AtomicLongArray	Vector de enteros `long` con actualización atómica.
AtomicLongFieldUpdater<T>	Permite acceso a atributos `volatile long`.
AtomicMarkableReference<V>	Referencia con marca y con actualización atómica.
AtomicReference<V>	Referencia con actualización atómica.
AtomicReferenceArray<E>	Vector de referencias con actualización atómica.
AtomicReferenceFieldUpdater<T,V>	Permite acceso a referencias para atributos `volatile`.
AtomicStampedReference<V>	Referencia con "*timestamp*" y actualización atómica.

Tabla 5.4: Clases del *package* `java.util.concurrent.atomic`.

- Existen dos constructores. Uno no tiene argumentos e inicializa el valor del objeto a cero. El segundo tiene un argumento y utiliza su valor para fijar el valor inicial del `AtomicInteger`.
- `get()`. Retorna el valor actual del objeto, convirtiéndolo a `int`.
- `set()`. Fija el valor actual del objeto.
- `lazySet()`. Hace lo mismo que el método anterior (fijar el valor del objeto), pero esta escritura puede reordenarse tras otras escrituras concurrentes solicitadas tras su invocación que no sean atómicas.
- `getAndSet()`. Fija como valor del objeto aquel que se facilite como argumento, retornando como resultado su valor previo.
- `compareAndSet()`. Compara el valor actual del objeto con el facilitado como primer argumento. Si son iguales entonces lo actualiza al valor pasado como segundo argumento. Retorna `true` si ambos valores fueron idénticos y se realizó el cambio. Retorna `false` si los valores no coincidían.
- `weakCompareAndSet()`. Hace lo mismo que el método anterior pero no asegura consistencia atómica respecto a otras variables que no hayan intervenido en este método.

Métodos de la clase `AtomicInteger`
public AtomicInteger()
public AtomicInteger(int initialValue)
public final int get()
public final void set(int newValue)
public final void lazySet(int newValue)
public final int getAndSet(int newValue)
public final boolean compareAndSet(int expect, int update)
public final boolean weakCompareAndSet(int expect, int update)
public final int getAndIncrement()
public final int getAndDecrement()
public final int getAndAdd(int delta)
public final int incrementAndGet()
public final int decrementAndGet()
public final int addAndGet(int delta)
public String toString()
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()

Tabla 5.5: Métodos de la clase `AtomicInteger`.

- `getAndIncrement()`. Devuelve el valor actual del objeto y posteriormente incrementa su valor en una unidad. Esta secuencia de dos pasos se realiza de manera atómica.
- `getAndAdd()`. Como el anterior, pero la modificación implica sumar el valor del argumento recibido, en lugar de incrementar en una unidad.
- `incrementAndGet()`. Como `getAndIncrement()`, pero invirtiendo el orden de esos dos pasos. En este método se realiza en primer lugar el incremento en una unidad y posteriormente se devuelve el valor resultante.
- `decrementAndGet()`. Decrementa en una unidad el valor del objeto y después retorna el valor resultante.
- `addAndGet()`. Suma la cantidad entera especificada como argumento al valor del objeto y después retorna el valor resultante.
- `toString()`. Retorna la representación como `String` del valor actual del objeto.
- `intValue()`. Retorna, convertido a `int`, el valor actual del objeto.

- `longValue()`. Retorna, convertido a `long`, el valor actual del objeto.
- `floatValue()`. Retorna, convertido a `float`, el valor actual del objeto.
- `doubleValue()`. Retorna, convertido a `double`, el valor actual del objeto.

Utilizando esta clase se podrá implantar de manera mucho más eficiente cualquier código que deba manipular una variable entera mediante incrementos y decrementos. Por ejemplo, para gestionar un contador y asegurar que no haya condiciones de carrera en los incrementos que deba registrar. El código necesario para ello se muestra en la figura 5.4, mostrando dos variantes: la primera utiliza un monitor convencional, la segunda un entero atómico.

Monitor

```
public class Counter {
  private int value;
  public Counter() {
    value=0;
  }
  public synchronized int next() {
    return ++value;
  }
  public synchronized int get() {
    return value;
  }
}
```

AtomicInteger

```
public class Counter {
  private AtomicInteger value;
  public Counter() {
    value = new AtomicInteger();
  }
  public int next() {
    return value.incrementAndGet();
  }
  public int get() {
    return value.get();
  }
}
```

Figura 5.4: Contador concurrente.

El número de sentencias que se necesitan en ambos casos es similar. Sin embargo, la gestión de los métodos sincronizados exige el uso de algún *lock* interno, que llegará a suspender a los hilos que encuentren el monitor ocupado. Por su parte, el uso de los métodos proporcionados por las clases atómicas puede que no necesite tales *locks*. Un buen número de procesadores modernos pueden implantar cada uno de esos métodos con una sola instrucción máquina. Por ello, garantizan que tales métodos no podrán ser interrumpidos y así evitan sin dificultades las condiciones de carrera. Es preferible utilizar esta segunda alternativa, pues es mucho más eficiente. A su vez, puede observarse que la variante inspirada en `AtomicInteger` lo único que hace es renombrar dos de sus métodos. Si el programador ya conociera tal clase, ni siquiera debería encapsularla dentro de la clase `Counter`; bastaría con haber definido instancias de `AtomicInteger` utilizando sus métodos `incrementAndGet()` y `get()` allí donde fuera necesario.

5.3.5 Semáforos

La clase `Semaphore` permite utilizar *semáforos* [Dij68] en Java. Los semáforos tradicionales son contadores enteros que ofrecen dos operaciones básicas [Dij64]: `P()` (de "*prolaag*", un neologismo propuesto por Dijkstra cuyo significado es *comprobar y decrementar*: "*probeer te verlagen*") y `V()` (de "*verhoog*": incrementar). La operación `P()` comprueba el valor actual del semáforo. En caso de ser positivo, lo decrementa y continúa, pero si ya es negativo o cero, el proceso o hilo invocador lo decrementará y quedará suspendido. Por su parte, la operación `V()` incrementa en una unidad el valor del semáforo. Si tras ese incremento el valor sigue siendo negativo o cero, se reactiva a uno de los hilos o procesos suspendidos previamente al invocar la correspondiente operación `P()`.

En Java los métodos principales de la clase `Semaphore` pasan a llamarse `acquire()` (en lugar de `P()`) y `release()` (en lugar de `V()`). El valor inicial del semáforo debe facilitarse como primer argumento en su constructor. Existe otra variante del constructor que admite un segundo argumento de tipo `boolean` para especificar si debe utilizarse una gestión equitativa (FIFO) de su cola de espera (cuando el valor facilitado sea `true`) o no. Aparte del constructor y los dos métodos principales, los semáforos Java ofrecen un buen número de métodos adicionales que se describen detalladamente en la documentación oficial de este lenguaje.

Los semáforos han sido (y siguen siendo) una herramienta de sincronización potente, flexible y de amplia aceptación. Con ellos es posible implantar una gran variedad de utilidades de sincronización. Por ejemplo:

- Si se inicializan a uno permiten garantizar exclusión mutua en el acceso a una determinada sección de código. Para ello, se utilizará el método `acquire()` antes de que empiece la sección a proteger y el método `release()` justo después de la sección protegida. De esta manera, cuando llegue el primer hilo que intente ejecutar dicha sección, encontrará el valor del semáforo a uno y podrá continuar, dejándolo a cero. Si antes de que ese hilo salga de la sección llegase otro hilo, invocaría el método `acquire()` pero al encontrar su contador a cero, quedaría bloqueado. Si llegasen otros entre tanto, también les ocurriría lo mismo: quedarían bloqueados en ese método `acquire()`.

  Finalmente, el hilo que estuviese activo dentro de la sección, completaría la ejecución de ésta e invocaría el método `release()` del semáforo. Al existir otros hilos suspendidos en el semáforo, se reactivaría a uno de ellos. Si el semáforo fue creado con gestión equitativa, se reactivaría al primero que se suspendió y entonces se cumpliría con los tres requisitos de una solución válida para el problema de la sección crítica (exclusión mutua, progreso y espera limitada), que ya fueron analizadas en la sección 2.6.

- Si se inicializan a un valor positivo superior a uno y se utilizan como se ha descrito en el punto anterior, podrán utilizarse para limitar el grado de con-

currencia en la ejecución de una determinada sección de código. Por ejemplo, en la sección 4.4.4 se listaron algunas estrategias de prevención de interbloqueos. Una de ellas consistía precisamente en limitar el grado de concurrencia para romper así la condición de espera circular. Para el caso particular del problema de los cinco filósofos debía garantizarse que el grado máximo de concurrencia fuera cuatro. Por ello, se podría utilizar un semáforo inicializado a dicho valor para prevenir los interbloqueos. Así se modelaría la existencia de un comedor en el que únicamente pudieran sentarse cuatro filósofos simultáneamente, asegurando que al menos uno de ellos obtenga ambos tenedores.

- Si se inicializan a cero garantizan cierto orden de ejecución entre dos o más hilos. Por ejemplo, si se quiere garantizar que un hilo H1 ejecute la sentencia S1 después de que el hilo H2 ejecute la sentencia S2, bastará con utilizar un semáforo `A` inicializado a cero, ubicando la sentencia `A.acquire()` justo antes de S1 y la sentencia `A.release()` justo después de S2. Así, si H1 llegase antes a su sentencia S1 que H2 a su sentencia S2, encontraría en ese punto su `A.acquire()`. Como el valor inicial de `A` es cero, al tratar de ejecutar ese `A.acquire()`, H1 se suspenderá (y todavía no ha podido llegar a S1). Tiempo después H2 ejecutará la sentencia S2. Cuando la haya completado realizará el `A.release()` y reactivará a H1. Cuando el planificador lo decida, H1 reanudará su ejecución y ejecutará S1, respetando el orden "*S2 antes que S1*" que se quería imponer.

  Con ese mismo código, si es H2 el hilo que llega antes a S2, dejará el semáforo `A` a 1 como resultado de su `A.release()` ejecutado tras S2. Con ello, cuando H1 trate de ejecutar S1, encontrará en primer lugar la sentencia `A.acquire()`. Afortunadamente, como el valor del semáforo ya es 1, esa sentencia no bloquea a H1 y las sentencias S1 y S2 se siguen ejecutando en el orden que debía respetarse: "*S2 antes que S1*".

Para ilustrar esta funcionalidad, la figura 5.5 muestra una clase Java que implanta una solución al problema de los productores-consumidores con buffer de capacidad limitada.

En esta clase solo se utilizan atributos estáticos (no es necesario instanciar objetos de la clase `ProdCons`) y se crea un hilo productor y uno consumidor dentro del método `main()`. La sincronización condicional relacionada con el buffer de capacidad limitada se implanta mediante dos semáforos. El primero se llama `item` y representa el número de elementos que existen actualmente en el buffer. En caso de que su valor sea cero, servirá para bloquear a los consumidores que intenten extraer elementos del buffer. El segundo semáforo se llama `slot` y representa el número de componentes vacías en el buffer. Cuando su valor sea cero indicará que el buffer está lleno. En ese caso servirá para bloquear a los productores que intenten insertar elementos en el buffer. El buffer propiamente dicho es un vector

```
import java.util.concurrent.Semaphore;

class ProdCons {
  static final int N=6; // buffer size
  static int head=0, tail=0;
  static int[] data= new int[N];
  static Semaphore item= new Semaphore(0,true);
  static Semaphore slot= new Semaphore(N,true);
  static Semaphore mutex= new Semaphore(1,true);

  public static void main(String[] args) {
    new Thread (new Runnable () { // producer
      public void run() {
        for (int i=0; i<100; i++) {
           put(i);
           System.out.println("Producer: " + i);
        }
      }
    }).start();
    new Thread (new Runnable () {// consumer
      public void run() {
        for (int i=0; i<100; i++) {
           System.out.println("Consumer: " + get());
        }
      }
    }).start();
  }

  public static int get() {
    try {item.acquire();} catch (InterruptedException i) {}
    try {mutex.acquire();} catch (InterruptedException i) {}
    int x=data[head]; head=(head+1)%N;
    mutex.release();
    slot.release();
    return x;
  }

  public static void put(int x) {
    try {slot.acquire();} catch (InterruptedException i) {}
    try {mutex.acquire();} catch (InterruptedException i) {}
    data[tail]=x; tail=(tail+1)%N;
    mutex.release();
    item.release();
  }
}
```

Figura 5.5: Solución con semáforos al problema de productores-consumidores.

de enteros. Su número de componentes está fijado al valor del atributo `N`, que en la figura 5.5 es 6. Tal valor también se utiliza para inicializar el semáforo `slot`. Los atributos `head` y `tail` mantienen las posiciones desde las que el consumidor extraerá y en la que el productor insertará el siguiente elemento, respectivamente. Inicialmente ambos valen cero. Se incluye también un tercer semáforo llamado `mutex` e inicializado a 1. Con él se garantizará exclusión mutua entre los métodos que modifican los atributos utilizados para gestionar el buffer.

El método `main()` se encarga de crear los dos hilos. Cada uno de ellos ejecutará un bucle de cien iteraciones. En cada iteración se inserta (productor) o extrae (consumidor) un elemento del buffer. En ambos casos se escribe un mensaje en la salida estándar describiendo la acción realizada.

Aparte de `main`, hay dos métodos que permiten modificar el buffer. Son los siguientes:

- `put()`. Es utilizado por el productor para insertar un elemento en el buffer. Su primera acción consiste en tratar de decrementar (utilizando el método `acquire()`) el semáforo `slot`. Si el buffer estuviera lleno, `slot` tendría valor cero y eso bloquearía al productor. Obsérvese que al utilizar el método `acquire()` se necesita encerrar su invocación en un bloque `try{...} catch(...)`, pues en caso de bloquear al hilo invocador dicha suspensión podría ser interrumpida, generando en ese caso una excepción que debe ser tratada.

  Una vez se haya superado esta comprobación, el hilo productor sabe que hay espacio en el buffer para insertar el nuevo elemento. En ese caso habrá que decrementar el semáforo `mutex` para asegurar la exclusión mutua a la hora de modificar el buffer (evitando que otros productores o consumidores puedan realizar otras actualizaciones simultáneamente). Superado ese segundo `acquire()` ya se pasará a insertar el elemento en la posición `tail` del vector `data` y a incrementar el valor de `tail`. Dicho incremento se realiza mediante la sentencia `tail=(tail+1)%N` para realizar una gestión "circular" del vector `data`. Es decir, si el incremento generase un valor igual a la capacidad del buffer (`N` en este ejemplo), esa sentencia proporciona cero como resultado.

  Tras completar estas modificaciones se incrementa el valor del semáforo `mutex` (para liberar la sección crítica) y el del semáforo `item` (indicando que hay un elemento más en el buffer, pudiendo desbloquear a un consumidor que encontrase el buffer vacío previamente).

- `get()`. Es utilizado por el consumidor para extraer un elemento del buffer. Su primera acción consiste en tratar de decrementar (utilizando el método `acquire()`) el semáforo `item`. Si el buffer estuviera vacío, `item` valdría cero y eso bloquearía al consumidor.

Una vez se haya superado esta comprobación, el hilo consumidor sabe que hay elementos en el buffer que pueden ser extraídos. En ese caso habrá que decrementar el semáforo `mutex` para asegurar exclusión mutua a la hora de modificar el buffer (evitando que otros productores o consumidores puedan realizar otras actualizaciones simultáneamente). Superado ese segundo `acquire()` ya se pasará a extraer el elemento de la posición `head` del vector `data` y a incrementar el valor de `head`. Dicho incremento se realiza mediante la sentencia `head=(head+1)%N` para realizar una gestión "circular" del vector `data` tal como ya se ha explicado anteriormente al describir la modificación realizada sobre `tail`.

Tras completar estas modificaciones se incrementa el valor del semáforo `mutex` (para liberar la sección crítica) y el del semáforo `slot` (indicando que hay un hueco más en el buffer, pudiendo desbloquear a un productor que encontrase el buffer lleno previamente).

5.3.6 Barreras

En `java.util.concurrent` encontramos dos tipos diferentes de barreras capaces de sincronizar a múltiples hilos de ejecución. Son las clases `CyclicBarrier` y `CountDownLatch`.

CyclicBarrier

La clase `CyclicBarrier` se utiliza para suspender a un grupo de hilos (utilizando para ello el método `await()`) hasta que el último de esos hilos invoque a `await()`. En ese instante, todos los hilos del grupo se reactivarán y podrán continuar. El número de hilos que forman ese grupo se facilitará como primer argumento en el constructor de la `CyclicBarrier`. Así, se puede decir que la barrera se crea "cerrada" y permanece en ese estado bloqueante hasta que todos los hilos del grupo han utilizado `await()`. En ese momento "se abre" y con ello se reactivan todos esos hilos, pero vuelve a quedar cerrada de inmediato. Esto último justifica el adjetivo "`Cyclic`" utilizado en el nombre de esta clase: tiene un comportamiento cíclico. Tan pronto como libere a los hilos que estaban esperando vuelve a quedar en el mismo estado en que fue creada y pasará a comportarse de igual manera hasta que se bloqueen todos los hilos del grupo, liberándolos y "reiniciándose" de nuevo.

Opcionalmente se puede facilitar un segundo argumento de tipo `Runnable` en el constructor que contendrá cierto código a ejecutar antes de la reactivación de los hilos. Existen algunos métodos más en esta clase, cuya descripción puede consultarse en la documentación oficial del lenguaje Java.

```
import java.util.concurrent.*;

class Task extends Thread {
  private CyclicBarrier cb;
  private int id;

  public Task(int i, CyclicBarrier c) {
    id = i; cb = c;
  }

  public void run() {
    for (int i=0; i<200; i++) {
      System.out.println("ID: " + getName() +
                         " iteration: " + i);
      if ((i+1) % 50 == 0) {
        try {cb.await();} catch(Exception e) {}
        if (id==0)
          System.out.println("Synchronized!");
      }
    }
  }
}

public class Synchro {
  private final static int N=3;
  private static CyclicBarrier bar = new CyclicBarrier(N);

  public static void main(String args[]) {
    for (int i=0; i<N; i++) {
      new Task(i,bar).start();
    }
  }
}
```

Figura 5.6: Sincronización con `CyclicBarrier` (incorrecta).

Para ilustrar el funcionamiento de `CyclicBarrier` asumamos que en cierto programa se crean tres hilos de una misma clase y estos ejecutan una secuencia de 200 iteraciones en la que cada uno escribe su identificador y el número de iteración. Queremos que estos hilos se sincronicen cada 50 iteraciones, esperándose mutuamente y escribiendo uno de ellos el mensaje "`Synchronized!`" cuando todos hayan llegado a ese punto antes de que empiecen a escribir el primer mensaje de las 50 iteraciones siguientes. Veamos un primer ejemplo de cómo se podría implantar un programa que hiciera eso. En la figura 5.6 se muestra un primer ejemplo pero todavía no resuelve este problema de manera precisa.

Obsérvese que el mensaje "`Synchronized!`" se escribe en salida estándar justo después de haber superado la espera en el método `await()`. Sin embargo, nada garantiza que cuando el hilo escriba dicho mensaje, los demás hilos no hayan escrito ya los mensajes de algunas de las siguientes iteraciones. Dependerá de lo que decida

el planificador del procesador. En la mayoría de las ejecuciones será raro que se respete la restricción que imponía el enunciado.

Para resolver esa situación tenemos el segundo argumento del constructor de `CyclicBarrier`. El código que establezcamos en ese `Runnable` seguro que se ejecuta cuando todos hayan invocado a `await()` pero antes de que alguno de los hilos haya sido reactivado. Esa era precisamente la restricción que se imponía en este ejemplo. Veamos una solución de este tipo en la figura 5.7.

```
import java.util.concurrent.*;

class Task extends Thread {
  private CyclicBarrier cb;

  public Task(CyclicBarrier c) {
    cb = c;
  }

  public void run() {
    for (int i=0; i<200; i++) {
      System.out.println("ID: " + getName() +
                         " iteration: " + i);
      if ((i+1) % 50 == 0) {
        try {cb.await();} catch(Exception e) {}
      }
    }
  }
}

public class Synchro2 {
  private final static int N=3;
  private static CyclicBarrier bar = new CyclicBarrier(N, new Runnable() {
    public void run() {
      System.out.println("Synchronized!");
    }});

  public static void main(String args[]) {
    for (int i=0; i<N; i++) {
      new Task(bar).start();
    }
  }
}
```

Figura 5.7: Sincronización con `CyclicBarrier` (correcta).

CountDownLatch

En ocasiones se necesitará suspender a un grupo de hilos quedando éstos a la espera de que suceda algún evento que debe ser generado por un hilo ajeno a tal grupo. La clase `CyclicBarrier` no puede gestionar tales situaciones pues únicamente dispone del método `await()` que sirve tanto para iniciar las esperas como para generar el evento que reactiva a todos los hilos. Sería necesario diseñar una nueva clase que proporcionara un método para bloquear a los hilos del grupo y otro método distinto para generar los eventos que reactiven a los hilos bloqueados. Ese es el objetivo de la clase `CountDownLatch`.

En este segundo tipo de barreras se mantiene un contador de eventos. Al crear un objeto `CountDownLatch` se especificará en su constructor (como único parámetro) el valor inicial asignado a tal contador. Esta barrera se crea inicialmente cerrada. Mediante el método `await()` los hilos se bloquearán en la barrera mientras esta permanezca cerrada, cosa que ocurre mientras su contador sea superior a cero. Existe un método `countDown()` que decrementa en una unidad el valor del contador, si éste era superior a cero. En caso de que el contador pase a valer cero, entonces la barrera se abre, liberando a todos los hilos bloqueados. Una vez abierta, la barrera permanecerá en ese estado. No hay manera de volverla a cerrar. Si alguno de los hilos utiliza `await()` una vez la barrera ya esté abierta, no llegará a bloquearse.

Como ejemplo de su uso, se resolverá el mismo problema citado en el apartado anterior, pero siendo ahora un hilo externo el responsable de escribir el mensaje "`Synchronized!`". Para ello utilizaremos dos `CountDownLatch`, uno para que el hilo externo sepa cuándo los hilos del grupo han llegado al punto de sincronización y otro para que ese hilo externo pueda reactivar a los hilos del grupo. Como los `CountDownLatch` no son reutilizables, el mensaje no se imprimirá cada 50 iteraciones, sino solo una vez: cuando hayan completado todos su iteración 100, pero antes de iniciar la 101. El código resultante se muestra en la figura 5.8.

El primer `CountDownLatch` es utilizado por el hilo externo (implantado mediante la clase `External` en la figura 5.8) para esperar hasta que todos los hilos del grupo (de la clase `Task`) hayan llegado a completar su iteración 100. Para ello, el hilo `External` utiliza un `first.await()` mientras que los hilos `Task` utilizan un `first.countDown()` y ese `CountDownLatch` se inicializa al valor `N`, que es el número de hilos de la clase `Task`.

Por su parte, el segundo `CountDownLatch` se utiliza al revés: para bloquear a los hilos de clase `Task` y que el hilo `External` pueda reactivarlos una vez haya escrito el mensaje "`Synchronized!`" en su salida estándar. Para ello bastará con inicializarlo a uno, pues se utilizará un único `second.countDown()` tras haber escrito el mensaje.

```
import java.util.concurrent.*;

class Task extends Thread {
  private CountDownLatch first,second;

  public Task(CountDownLatch one, CountDownLatch two) {
    first = one; second = two;
  }

  public void run() {
    for (int i=1; i<201; i++) {
      System.out.println("ID: " + getName() +
                         " iteration: " + i);
      if (i == 100) {
        first.countDown();
        try {second.await();} catch(Exception e) {}
      }
    }
  }
}

class External extends Thread {
  private CountDownLatch first,second;

  public External(CountDownLatch one, CountDownLatch two) {
    first=one; second=two;
  }

  public void run() {
    try {first.await();} catch(Exception e) {}
    System.out.println("Synchronized!");
    second.countDown();
  }
}

public class SynchroCount {
  private final static int N=3;
  private static CountDownLatch step1 = new CountDownLatch(N);
  private static CountDownLatch step2 = new CountDownLatch(1);

  public static void main(String args[]) {
    for (int i=0; i<N; i++) {
      new Task(step1,step2).start();
    }
    new External(step1,step2).start();
  }
}
```

Figura 5.8: Sincronización con `CountDownLatch`.

5.3.7 Ejecución de hilos

En la sección 1.4.2 ya se explicó que para crear hilos de ejecución en Java existían diferentes alternativas, basadas en las variantes admitidas para desarrollarlos: crear subclases de `Thread` o escribir clases que implanten la interfaz `Runnable`. En la biblioteca `java.util.concurrent` se diseñaron diferentes mecanismos para lanzar tales hilos. Esto se debe a que la creación de un hilo es una operación costosa, que requiere un buen número de recursos (memoria, entradas en las tablas manejadas por el sistema operativo, ...) y que, por tanto, suele resultar lenta. Por ello se han definido diferentes interfaces (`Executor`, `ExecutorService`, `ScheduledExecutorService`,...) que permiten gestionar dicha generación de hilos. La documentación oficial del lenguaje Java describe detalladamente estas interfaces. Obsérvese que `ScheduledExecutorService` extiende la interfaz `ExecutorService` y, a su vez, esta última extiende la interfaz `Executor`.

Una técnica sencilla para agilizar la generación de hilos consiste en mantener un conjunto de hilos ya creados, reciclándolos para que pasen a ejecutar los objetos `Runnable` que mantengan el código de la nueva clase de hilos que deba ejecutarse. Así la generación resulta más rápida y, una vez terminada la ejecución de tal código, el hilo pueda ser retornado a ese conjunto (conocido como "*Thread pool*") para ser reutilizado posteriormente utilizando cualquier `Runnable` que se facilite.

La pieza básica de todo este soporte es la interfaz `Executor`. Esta interfaz proporciona un único método: `execute()` con el que se podrá sustituir código del tipo:

```
Runnable r = ...;

(new Thread(r)).start();
```

por una sentencia como la siguiente: "`e.execute(r);`", siendo `e` un objeto que implante la interfaz `Executor`. Dependiendo del tipo de `Executor` utilizado, se podrá reaprovechar un `Thread` creado previamente, crear uno nuevo a propósito o comprobar si existe algún `Thread` disponible y decidir entonces si podrá reaprovecharse o deberá generarse otro nuevo. Además, también se podrá lograr que las tareas ejecutadas por tales hilos se inicien tras una pausa de una duración determinada o se ejecuten de manera periódica.

Para conocer qué tipos de `Executor` se podrán utilizar, conviene revisar la clase `Executors` que actúa como una factoría o generador de los diferentes ejecutores existentes. Su descripción y lista completa de métodos puede consultarse en [Ora12a]. Entre ellos cabría destacar los siguientes:

- `public static ExecutorService newCachedThreadPool()`. Retorna un nuevo "*thread pool*" que creará nuevos hilos a medida que resulte necesario. Las

llamadas al método `execute()` del `ExecutorService` retornado reutilizarán los hilos existentes en caso de que haya alguno disponible. Si no hubiese ninguno, se generará otro nuevo y se añadirá al "*pool*".

Si alguno de los hilos disponibles en el "*pool*" permanece más de 60 segundos sin ser reutilizado, será eliminado. De esta manera se adapta el número de hilos disponibles a las necesidades de los procesos que utilicen tal reserva de hilos.

- `public static ExecutorService newFixedThreadPool(int nThreads)`. Retorna un nuevo "*thread pool*" que mantendrá un número fijo de hilos, especificado como único argumento. Las llamadas al método `execute()` del `ExecutorService` retornado reutilizarán los hilos existentes en caso de que haya alguno disponible. Si no hubiese ninguno, esa llamada se bloqueará hasta que haya alguno libre.

  Si alguno de los hilos del "*pool*" abortase por cualquier motivo, sería reemplazado por un hilo de nueva creación, para garantizar que siguiese habiendo el número total de hilos solicitado en su creación.

- `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`. Este método realiza una función similar al anterior: gestión de un "*pool*" de tamaño fijo especificado como argumento. La única diferencia es que el `ExecutorService` retornado para gestionar los hilos admitirá la ejecución de tareas periódicas o el inicio retardado de la ejecución de tareas. Véase la descripción de la interfaz `ScheduledExecutorService` en la documentación de Java para obtener mayor información.

5.3.8 Temporización precisa

Antes de la versión 1.5.0, el lenguaje de programación Java únicamente disponía del método `System.currentTimeMillis()` para obtener información sobre el instante actual. Este método devuelve el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970 en el huso horario UTC. Aunque el resultado está expresado en milisegundos, se puede devolver un resultado con esa granularidad (1 ms) u otra mayor (10ms, 50ms,...), dependiendo de la plataforma sobre la que se utilice la máquina virtual Java. Además, el valor retornado en este método suele depender del valor de reloj mantenido por el sistema operativo y éste puede ser modificado por agentes externos (el administrador del sistema, el usuario, un servidor de reloj distribuido, un receptor GPS, ...). Como los procesadores modernos son cada vez más rápidos, los valores retornados por este método no siempre podrán utilizarse para realizar un análisis temporal de las prestaciones de un determinado programa: sería interesante disponer de un reloj con mejor precisión.

Java 1.5.0 introdujo el método `System.nanoTime()` para resolver este problema. Con él ya se obtienen resultados expresados en nanosegundos, pero sigue sin im-

ponerse ninguna restricción concreta respecto a su frecuencia mínima de refresco, de manera que sea idéntica en diferentes plataformas. Lo que sí se asegura es que se proporcionará una lectura que ofrecerá la mejor precisión disponible en cada sistema.

Por otra parte, en versiones anteriores a la 1.5.0, aquellos métodos que bloquean a los hilos durante un determinado intervalo de tiempo solían especificar la duración de tal intervalo en milisegundos. Un ejemplo típico es `Thread.sleep()`, aunque posteriormente se añadió otra variante con dos argumentos, de manera que en el segundo se indicaba cuántos nanosegundos debían complementar tal intervalo de suspensión.

Para resolver estos problemas a la hora de especificar la duración de un intervalo de bloqueo, la biblioteca `java.util.concurrent` ha introducido un *enum* `TimeUnit` con el que resulta posible especificar qué unidad temporal se va a utilizar a la hora de definir un intervalo: `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`, `MICROSECONDS` o `NANOSECONDS`. Además, también incluye métodos para realizar conversiones entre las diferentes unidades posibles. Así, por ejemplo, si se quisiera especificar que la duración de una espera en el método `tryLock()` es de 180 microsegundos, bastaría con escribir un código similar al siguiente:

```
Lock lock = ...;
if (lock.tryLock(180L, TimeUnit.MICROSECONDS)) ...
```

5.4 Resumen

En esta unidad se han revisado las principales interfaces y clases existentes en el package `java.util.concurrent`, que ofrece un buen número de herramientas de sincronización para facilitar el desarrollo de aplicaciones concurrentes en el lenguaje Java. En concreto, se han analizado las siguientes herramientas (mostrando también ejemplos de utilización):

- *locks*, que incluye diferentes clases e interfaces para la gestión de múltiples tipos de locks (como el `ReentrantLock`), así como la definición de diferentes *condiciones* asociadas a los locks (con lo que se evita la limitación de los monitores básicos de Java, en los que solo hay una cola de espera interna implícita a cada monitor).
- *colecciones concurrentes thread-safe*, y en concreto la interfaz `BlockingQueue`, que ofrece una cola de objetos con política FIFO a la hora de extraer sus elementos, mostrando un comportamiento adecuado aunque la cola sea accedida simultáneamente por múltiples hilos.

- *variables atómicas*, como `AtomicInteger`, que garantizan la ejecución no interrumpible (logrando exclusión mutua) de ciertas secuencias de acciones (como incrementar, decrementar, etc.).
- *semáforos*, que ofrecen una gran variedad de utilidades de sincronización, como la exclusión mutua, la limitación del grado de concurrencia, o bien la garantía de cierto orden de ejecución entre dos o más hilos.
- *barreras*, en concreto la clase `CyclicBarrier`, que permite suspender a un grupo de hilos hasta que el último de ellos llegue a la barrera, reactivándose a todos los hilos y quedando la barrera de nuevo cerrada; y la clase `CountDownLatch`, que suspende a un grupo de hilos a la espera de que suceda algún evento generado por un hilo ajeno al grupo.
- *entornos de ejecución de hilos*, como la interfaz `Executor`, que permite gestionar la generación de los hilos.
- temporización precisa, con el *enum* `TimeUnit` que permite especificar qué unidad temporal (días, horas, minutos, segundos, milisegundos, microsegundos, nanosegundos) se debe utilizar al definir un intervalo de tiempo.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar los inconvenientes de las primitivas básicas de Java.
- Describir las herramientas proporcionadas por el package *java.util.concurrent*, que facilitan el desarrollo de aplicaciones concurrentes. En concreto:
  - Ilustrar la utilización de los *locks* (como el `ReentrantLock`) y las *condiciones*. Discutir las limitaciones sobre los monitores básicos de Java que los locks y condiciones permiten romper.
  - Interpretar el uso de colecciones concurrentes *thread-safe*, como la interfaz `BlockingQueue`.
  - Describir el funcionamiento de las clases atómicas. Contrastar la clase `AtomicInteger` con el uso de monitores convencionales.
  - Ilustrar el empleo de semáforos (con la clase `Semaphore`) para distintos tipos de sincronización.
  - Ilustrar el funcionamiento de las barreras, distinguiendo entre `CyclicBarrier` y `CountDownLatch`.
  - Identificar las interfaces para la gestión de la generación de hilos, como la interfaz `Executor`.
  - Identificar el mecanismo de temporización precisa que se ofrece.

Unidad 6

SINCRONIZACIÓN EN SISTEMAS DE TIEMPO REAL

6.1 Introducción

En las aplicaciones de tiempo real, el correcto funcionamiento depende no sólo de los resultados del cómputo sino también de los instantes de tiempo en los que se generan los resultados [Sta88]. Ejemplos de aplicaciones de tiempo real los podemos encontrar en control de tráfico aéreo, sistemas de control y navegación de vehículos (automóviles, aviones o naves espaciales), robótica y control de procesos industriales entre otros.

Una *aplicación de tiempo real* se compone de un conjunto de tareas que cooperan entre sí. Estas tareas deben ejecutar sus acciones dentro de intervalos de tiempo bien definidos. Los requisitos temporales de un *sistema de tiempo real* especifican cuáles son los intervalos de tiempo válidos para la ejecución de cada una de las tareas. El requisito temporal más común es el de tiempo máximo de respuesta o *plazo* ("*deadline*").

Como la funcionalidad requerida en un sistema de tiempo real es muy diversa, existen distintos tipos de actividades o *tareas*[1], que dependen de las características de la aplicación. Diremos que una *tarea* es *crítica* (hard) si el incumplimiento de alguno de sus requisitos temporales, incluso ocasionalmente, supone un fallo intolerable por sus consecuencias en el sistema controlado. Por el contrario, una

[1] Los sistemas de tiempo real son ejemplos de sistemas concurrentes. En este ámbito, las actividades se llaman tareas. En esta unidad utilizaremos ese término para referirnos a las actividades concurrentes que se ejecuten en un sistema de tiempo real.

tarea es *acrítica* (soft) cuando siendo deseable el cumplimiento de sus requisitos temporales, se puede tolerar el incumplimiento ocasional de alguno ellos. Un *plazo* máximo que no puede cumplirse ocasionalmente, pero en el que no hay beneficio por la respuesta retrasada, se denomina *firme*.

Denominaremos *tareas periódicas* a las que se ejecutan repetidamente a intervalos de tiempo regulares iguales a su *periodo*. Normalmente las tareas periódicas deben llevar a cabo sus acciones dentro de plazo en todas las ocasiones en que se ejecuten. Por el contrario, una *tarea aperiódica* se ejecuta de forma irregular, en respuesta a algún evento que ocurre en el sistema controlado. Cuando una tarea aperiódica sea crítica y deba ejecutarse en un plazo de tiempo estricto, la denominaremos *tarea esporádica*. Las tareas esporádicas deben tener asociada también una separación mínima de tiempo entre dos activaciones sucesivas, a la que se denomina período o intervalo mínimo entre llegadas. Sin esta restricción no sería posible garantizar el cumplimiento de los plazos.

La programación de los sistemas de tiempo real críticos es diferente a la de los sistemas de cómputo convencionales. Habitualmente se utilizan como medidas de mérito el *rendimiento* ("*throughput*") o capacidad media de procesado de información y el tiempo de respuesta promedio, mientras que en un sistema de tiempo real estricto las prestaciones medias son un factor secundario, y los realmente importantes son la planificabilidad del conjunto de tareas (habilidad para cumplir todos los plazos) y el tiempo de respuesta en el peor caso.

Uno de los problemas fundamentales en el diseño de sistemas de tiempo real es el de la planificación de la ejecución de sus tareas para que cumplan sus restricciones temporales. Hay que comprobar que los requisitos temporales están garantizados en todos los casos, estudiando para ello el peor caso de carga posible.

Las técnicas utilizadas comúnmente para verificar el cumplimiento de esos requisitos temporales se pueden clasificar en dos tipos diferentes:

- Técnicas *off-line*: antes de que el sistema de tiempo real esté operando se prevé y analiza el conjunto de posibles comportamientos, de forma que estimando cotas superiores de los tiempos de respuesta se verifique el cumplimiento de los plazos de respuesta.
- Técnicas *on-line*: en el instante en que una nueva tarea está lista para ejecutar, se realiza el análisis de planificabilidad considerando la nueva tarea junto con el conjunto de tareas ya existente. Si el nuevo sistema formado siguiera siendo planificable, entonces se acepta la tarea para ejecutar; en caso contrario, se rechaza. Estas técnicas presentan el inconveniente de que pueden rechazar tareas de gran importancia, sin que este hecho se pueda detectar hasta el momento de la ejecución.

El único mecanismo fiable para determinar si la política o políticas de planificación elegidas aseguran el cumplimiento de los plazos consiste en la realización a priori de un test de planificabilidad, ya que la simulación no garantiza que se haya comprobado el comportamiento en cualquier situación. Por ello nos centraremos sólo en el primer tipo de técnicas.

El hecho de tener que garantizar a priori plazos en la respuesta de un sistema hace necesario imponer ciertos requisitos sobre su especificación e implementación. En particular, es preciso que el comportamiento temporal del sistema de tiempo real sea predecible. Esto quiere decir que todos los componentes (tanto hardware como software) del sistema de tiempo real deben ser conocidos y estar perfectamente especificados a priori.

En las aplicaciones de tiempo real, la planificación expulsiva por prioridades fijas es la más popular debido principalmente a que el comportamiento temporal es más fácil de entender y predecir, a que existen técnicas analíticas completas y a que está soportada por estándares de sistemas operativos y lenguajes concurrentes como Ada 2005 RT-annex, Java RTSJ y Real-Time POSIX.

Las secciones siguientes resumen los principales resultados obtenidos en la teoría de planificación expulsiva por prioridades fijas, en lo referente al análisis de planificabilidad básico y compartición de recursos. En [ABD+95] podemos encontrar una evolución cronológica más completa de dicha teoría.

6.2 Análisis básico

El modelo de tareas para el que se desarrolla el análisis de la viabilidad es el siguiente: asumimos un único procesador y política de planificación expulsiva por prioridades fijas; todos los procesos son periódicos (o esporádicos con un tiempo mínimo entre llegadas conocido); tienen un plazo menor o igual a su periodo; son independientes; tienen un tiempo de cómputo variable pero limitado y se conoce el límite para el caso peor; ningún proceso puede suspenderse voluntariamente durante su ejecución; todos los procesos pasan a la cola de procesos preparados para ejecución tan pronto como llegan al sistema; todas las sobrecargas se ignoran.

Dado un conjunto de n tareas periódicas independientes, asignaremos (mediante alguna política) una prioridad fija de base a cada tarea. Utilizaremos la prioridad 1 para representar el nivel más prioritario y n para representar el menos prioritario.

Cada tarea τ_i, tendrá una *prioridad de base* i donde $1 \leq i \leq n$, y una *prioridad activa* que normalmente será igual a la prioridad de base, pero que puede ser alterada en tiempo de ejecución por las operaciones de los protocolos de asignación de recursos.

En tiempo de ejecución una tarea o está preparada para ejecución o está suspendida esperando su activación. En todo momento se seleccionará para ejecución la tarea preparada de mayor prioridad activa. Si estando una tarea en ejecución se activa otra más prioritaria, esta última expulsará del procesador a la que se encontraba en ejecución.

Cada tarea periódica τ_i da lugar a una secuencia infinita de invocaciones separadas por su periodo T_i (en caso de una tarea esporádica T_i representa el intervalo mínimo entre llegadas). En cada invocación necesitará una cantidad de *tiempo de cómputo* limitada por C_i (tiempo de cómputo del caso peor), que debe completarse antes de su plazo D_i (relativo al inicio de cada activación). El *desplazamiento* (o desfase) *inicial* de la primera invocación se denominará O_i.

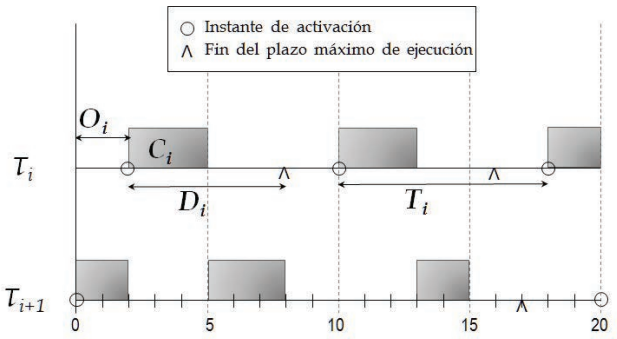

Figura 6.1: Representación gráfica de los atributos de las tareas.

Un conjunto de tareas se dice que es planificable (o viable) si todos sus plazos se cumplen, es decir, si cada tarea periódica acaba su ejecución antes que finalice su plazo.

Existen dos conceptos que ayudan a construir el peor caso de planificación posible con prioridades fijas expulsivas, y tareas periódicas independientes:

- *Instante crítico*. El tiempo de respuesta de peor caso para todas las tareas del conjunto se da cuando todas ellas se activan simultáneamente, es decir, todas las tareas están en fase $O_i = 0$.
- *Comprobar sólo el primer plazo*. Cuando todas las tareas se activan simultáneamente, si una tarea cumple su primer plazo, cumplirá siempre todos sus plazos. Es equivalente a estudiar para cada tarea el intervalo durante el cual el procesador está ocupado ejecutando la tarea i o tareas de prioridad superior a i, que es el intervalo de peor caso.

El principal resultado en que se basan los tests de planificabilidad exactos es el siguiente:

> *"En un sistema de n tareas periódicas independientes con prioridades de base asignadas en algún orden fijo, se cumplen todos los plazos de respuesta para cualquier desfase inicial de las tareas si, cuando se activan todas ellas simultáneamente, cada tarea acaba dentro de plazo en su primera ejecución."*

La comprobación requerida por el anterior teorema puede representarse por un test matemático equivalente, donde se calcula el *tiempo de respuesta* para el caso peor de cada tarea (R_i) y se comprueba que $R_i \leq D_i$. La ecuación siguiente calcula el tiempo de respuesta exacto para cada tarea:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{6.1}$$

siendo:

- $hp(i)$ el conjunto de tareas pertenecientes a niveles estrictamente más prioritarios que i. La sigla "*hp*" hace referencia a "*higher priority*" (prioridad más alta).
- R_i el tiempo de respuesta de caso peor de la tarea i.
- C_i el tiempo de cómputo de caso peor de la tarea i.
- T_i el periodo de la tarea i.
- $\lceil x \rceil$ la función techo sobre x, es decir, redondeo de x al entero superior.

Esta ecuación se resuelve formando una relación de recurrencia que va calculando el tiempo de ejecución necesario para finalizar todo el trabajo hasta el instante de la iteración anterior.

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \tag{6.2}$$

La iteración termina cuando dos pasos consecutivos alcanzan el mismo resultado $R_i^{n+1} = R_i^n$, lo cual indica que no queda más trabajo por realizar a ese nivel de prioridad. La relación o converge o excede el plazo D_i, en cuyo caso el resultado no es válido y la tarea i no es planificable. Joseph y Pandya [JP86] demostraron la convergencia de la ecuación anterior si la utilización del procesador no es mayor del 100 %. Un valor inicial aceptable es $R_i^0 = C_i + \sum_{\forall j \in hp(i)} C_j$.

6.3 Ejemplo de cálculo de los tiempos de respuesta

Vamos a aplicar la ecuación anterior para determinar la planificabilidad del siguiente conjunto de tareas, asumiendo que la prioridad 1 es mayor que la 4:

Tarea	T_i	C_i	D_i	***Prio***
τ_1	12	3	5	1
τ_2	8	2	7	2
τ_3	20	3	16	3
τ_4	25	4	22	4

La figura 6.2 muestra el cronograma de la ejecución del conjunto de tareas, desde el instante crítico $t = 0$ hasta que finaliza la primera activación de la tarea τ_4 en el instante $t = 25$, para poder observar los diferentes intervalos de estudio de cada tarea.

Hay que calcular el tiempo de respuesta de peor caso R_i, y comprobar que $R_i \leq D_i$ para cada tarea i.

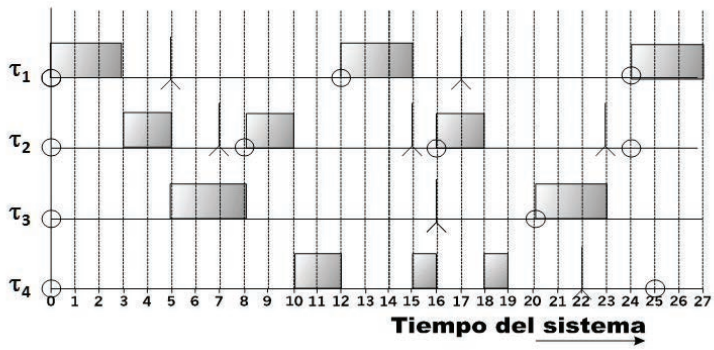

Figura 6.2: Cronograma de ejecución.

Para la tarea τ_1, $R_1^0 = C_1 = 3$ y como es la más prioritaria $R_1^1 = 3$ por lo que el tiempo de respuesta de peor caso es $R_1 = 3$ que es menor que su plazo $D_1 = 5$.

Para la tarea τ_2:

$R_2^0 = C_2 + C_1 = 2 + 3 = 5$

$R_2^1 = C_2 + \left\lceil \frac{R_2^0}{T_1} \right\rceil \cdot C_1 = 2 + \left\lceil \frac{5}{12} \right\rceil \cdot 3 = 5$

el tiempo de respuesta de peor caso es $R_2 = 5$ que es menor que su plazo $D_2 = 7$.

Para la tarea τ_3:

$R_3^0 = C_3 + C_1 + C_2 = 3 + 3 + 2 = 8$
$R_3^1 = C_3 + \left\lceil \frac{R_3^0}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3^0}{T_2} \right\rceil \cdot C_2 = 3 + \left\lceil \frac{8}{12} \right\rceil \cdot 3 + \left\lceil \frac{8}{8} \right\rceil \cdot 2 = 8$
el tiempo de respuesta de peor caso es $R_3 = 8$ que es menor que su plazo $D_3 = 16$.

Para la tarea τ_4:

$R_4^0 = C_4 + C_1 + C_2 + C_3 = 4 + 3 + 2 + 3 = 12$
$R_4^1 = C_4 + \left\lceil \frac{R_4^0}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_4^0}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_4^0}{T_3} \right\rceil \cdot C_3 = 4 + \left\lceil \frac{12}{12} \right\rceil \cdot 3 + \left\lceil \frac{12}{8} \right\rceil \cdot 2 + \left\lceil \frac{12}{20} \right\rceil \cdot 3 = 14$

$R_4^2 = C_4 + \left\lceil \frac{R_4^1}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_4^1}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_4^1}{T_3} \right\rceil \cdot C_3 = 4 + \left\lceil \frac{14}{12} \right\rceil \cdot 3 + \left\lceil \frac{14}{8} \right\rceil \cdot 2 + \left\lceil \frac{14}{20} \right\rceil \cdot 3 = 17$

$R_4^3 = C_4 + \left\lceil \frac{R_4^2}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_4^2}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_4^2}{T_3} \right\rceil \cdot C_3 = 4 + \left\lceil \frac{17}{12} \right\rceil \cdot 3 + \left\lceil \frac{17}{8} \right\rceil \cdot 2 + \left\lceil \frac{17}{20} \right\rceil \cdot 3 = 19$

$R_4^4 = C_4 + \left\lceil \frac{R_4^3}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_4^3}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_4^3}{T_3} \right\rceil \cdot C_3 = 4 + \left\lceil \frac{19}{12} \right\rceil \cdot 3 + \left\lceil \frac{19}{8} \right\rceil \cdot 2 + \left\lceil \frac{19}{20} \right\rceil \cdot 3 = 19$

el tiempo de respuesta de peor caso es $R_4 = 19$ que es menor que su plazo $D_4 = 22$. Por tanto, el conjunto de tareas es planificable y se garantiza que todas las tareas siempre cumplirán sus plazos. En la figura 6.2 puede observarse como el instante $t = 19$ es el instante en que el procesador queda libre, pues el sistema ya ha ejecutado toda la carga que había sido solicitada hasta ese momento.

6.4 Compartición de recursos

En este punto se discute cómo la teoría de planificación puede aplicarse a tareas de tiempo real que deben interaccionar, extendiendo así el análisis hacia situaciones prácticas.

Las primitivas de comunicación más comunes son los semáforos, cerrojos (locks), monitores y la cita de Ada. A pesar de que la utilización de estos métodos u otros equivalentes es necesaria para asegurar la consistencia de datos compartidos o para garantizar la apropiada utilización de recursos en exclusión mutua, su uso puede comprometer la capacidad del sistema para cumplir sus requisitos temporales. De hecho, una aplicación directa de estos mecanismos de sincronización puede conducirnos a un periodo indefinido de *inversión de prioridades*, que ocurre cuando se impide la ejecución a una tarea de prioridad alta por estar esperando a que se libere un recurso que está siendo utilizado por otra tarea de prioridad menor.

La inversión de prioridad (y en especial la no acotada [SRL90]) reduce notablemente la planificabilidad de los sistemas de tiempo real, por lo que es conveniente evitarla o, cuanto menos, reducirla.

El mecanismo normalmente utilizado para garantizar el acceso mutuamente exclusivo es el de semáforos o locks protegiendo las secciones críticas. Este mecanismo, sin embargo, puede ocasionar retrasos muy grandes cuando se aplica en sistemas de tiempo real. La inversión de prioridad no acotada se produce cuando una tarea de prioridad baja bloquea un semáforo que protege a un recurso compartido con otra tarea de prioridad alta y es expulsada por la ejecución de una tarea de prioridad intermedia. Esa expulsión provoca una inversión de prioridad de duración igual a la ejecución de todas las tareas de prioridad intermedia que puedan expulsar a la tarea de baja prioridad, lo cual puede ser un tiempo excesivamente largo.

La figura 6.3 muestra un ejemplo de ejecución de tres tareas en donde ocurre inversión de prioridades. La tarea τ_1 con prioridad alta comparte un semáforo S con la tarea de prioridad baja τ_3. En $t = 5$ τ_3 se activa, comienza su ejecución y en $t = 6$ cierra el semáforo S y entra en su sección crítica. En $t = 8$ es expulsada por τ_1 que comienza su ejecución y en $t = 10$ intenta acceder a su sección crítica pero al estar el semáforo S cerrado debe esperar a que τ_3 salga de su sección crítica y abra el semáforo S. Entonces se produce la inversión de prioridades entre τ_1 y τ_3. Dicha inversión no está acotada al poderse ejecutar tareas de prioridad intermedia como τ_2, que expulsan la ejecución de τ_3. En $t = 19$ se observa que la tarea τ_1 pierde su plazo por la espera sufrida.

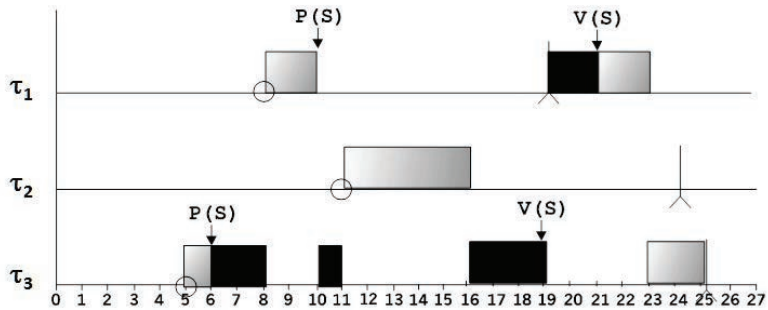

Figura 6.3: Ejemplo de inversión de prioridades.

El uso de semáforos presenta también otro problema, el *interbloqueo* (*deadlock*), en que dos tareas están bloqueadas esperando ambas a que la otra libere un recurso que necesita para ejecutar. Las consecuencias de este efecto son catastróficas,

puesto que ambas tareas se quedan indefinidamente esperando y pierden sus plazos con toda seguridad.

La figura 6.4 ilustra este problema con dos tareas τ_1 y τ_2 que comparten dos estructuras de datos protegidas por los semáforos S_1 y S_2, respectivamente. Supongamos que τ_1 cierra los semáforos en el orden S_1, S_2, mientras que τ_2 lo hace en el orden inverso. Además asumimos que τ_1 tiene mayor prioridad que τ_2. En t_1 τ_2 cierra S_2 y entra en su primera sección crítica; en t_2 τ_1 expulsa a τ_2 comienza su ejecución y en t_3 cierra S_1; estando en su primera sección crítica, en t_4 intenta entrar en su segunda sección crítica guardada por el semáforo S_2 pero como el semáforo ya está cerrado se suspende; siendo τ_2 la única tarea activa, continúa su ejecución hasta que en t_5 se queda también bloqueada al realizar la operación $\mathrm{P}(S_1)$ produciéndose entonces el interbloqueo mutuo. Ambas tareas están esperando que se abra un semáforo, y quien puede abrirlo no podrá hacerlo nunca porque está bloqueada y no puede continuar su ejecución.

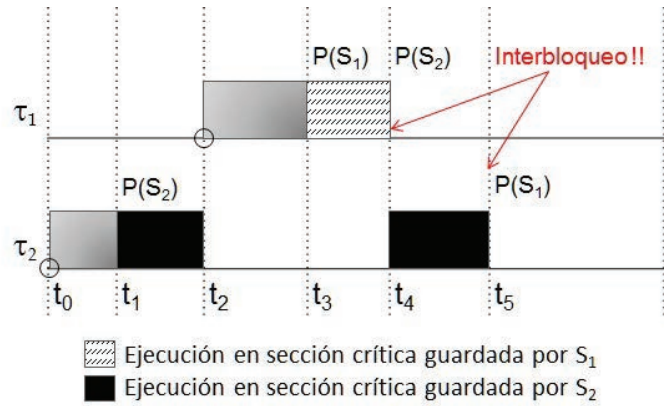

Figura 6.4: Ejemplo de interbloqueo

El objetivo de los protocolos de sincronización de tiempo real es precisamente evitar esas inversiones de prioridad no acotadas, minimizando la duración de los tiempos de bloqueo por las secciones críticas, y evitando los interbloqueos.

Los principales protocolos desarrollados para sistemas basados en prioridades fijas soportados por lenguajes concurrentes y el estándar POSIX son: el *protocolo de herencia básica de prioridades* y el *protocolo de techo de prioridad inmediato* o *protección por prioridad*.

Aunque el primero acota el número de veces que puede bloquearse una tarea de prioridad alta, es posible que se produzcan cadenas de bloqueos dando lugar a tiempos de bloqueo de peor caso muy pesimistas y tampoco evita los interbloqueos.

6.4.1 Protocolo de techo de prioridad inmediato

El protocolo del *techo de prioridad inmediato* se denomina *priority ceiling emulation* en RT-Java, *priority protect protocol* en POSIX y *ceiling locking protocol* en el lenguaje Ada. El protocolo impone las siguientes restricciones en la utilización de los semáforos:

1. Las secciones críticas deben estar anidadas siguiendo una estructura piramidal, es decir:
   P(S_1)...P(S_2)...V(S_2)...V(S_1)
   y no P(S_1)...P(S_2)...V(S_1)...V(S_2).
2. Las secciones críticas deben tener tiempos de cómputo limitados.

Hay dos ideas en el diseño de este protocolo. Primero el concepto de *techo de prioridad de un semáforo*: a cada semáforo le asociaremos un techo, equivalente a la mayor prioridad de las tareas que lo usan. Segundo, una tarea que accede a un semáforo hereda inmediatamente el techo de prioridad del semáforo. Como consecuencia, la sección crítica se ejecuta con la prioridad del techo del semáforo que la guarda.

La figura 6.5 muestra un ejemplo de ejecución de este protocolo para una activación concreta de 5 tareas, con prioridades $1 > 2... > 5$, que usan dos semáforos X e Y, inicialmente abiertos. Las tareas τ_1 y τ_4 utilizan el semáforo Y por tanto el $techo(Y) = 1$; las tareas τ_2, τ_4 y τ_5 utilizan el semáforo X por tanto el $techo(X) = 2$;

- En t_0, τ_5 es la única tarea activa y comienza su ejecución con prioridad 5.
- En t_1, τ_5 cierra el semáforo X y eleva su prioridad a la del techo de X, continuando su ejecución con prioridad 2.
- En t_2, τ_4 se activa, pero no comienza su ejecución pues su prioridad 4 es inferior a la que está en ejecución y no puede expulsarla.
- En t_3, τ_3 se activa, pero no comienza su ejecución pues su prioridad 3 es inferior a la que está en ejecución y no puede expulsarla.
- En t_4, τ_5 abre el semáforo X y recupera su prioridad anterior, continuando su ejecución con prioridad 5. En ese mismo instante τ_2 se activa, y expulsa a τ_5.
- En t_5, τ_2 cierra el semáforo X y eleva su prioridad a la del techo de X, continuando su ejecución con prioridad 2 que ya tenía.

- En t_6, τ_2 abre el semáforo X y recupera su prioridad anterior, continuando su ejecución con prioridad 2. En ese mismo instante τ_1 se activa, y expulsa a τ_2.
- En t_7, τ_1 cierra el semáforo Y y eleva su prioridad a la del techo de Y, continuando su ejecución con prioridad 1 que ya tenía; posteriormente abre el semáforo Y y finaliza su ejecución en t_9; entonces pueden finalizar su ejecución las tareas τ_2 y τ_3 respectivamente.
- En t_{11}, τ_4 comienza su ejecución. Obsérvese que cuando necesita aceder a sus secciones críticas guardadas por los semáforos Y y X respectivamente, puede hacerlo sin ningún bloqueo puesto que ambos semáforos están libres.

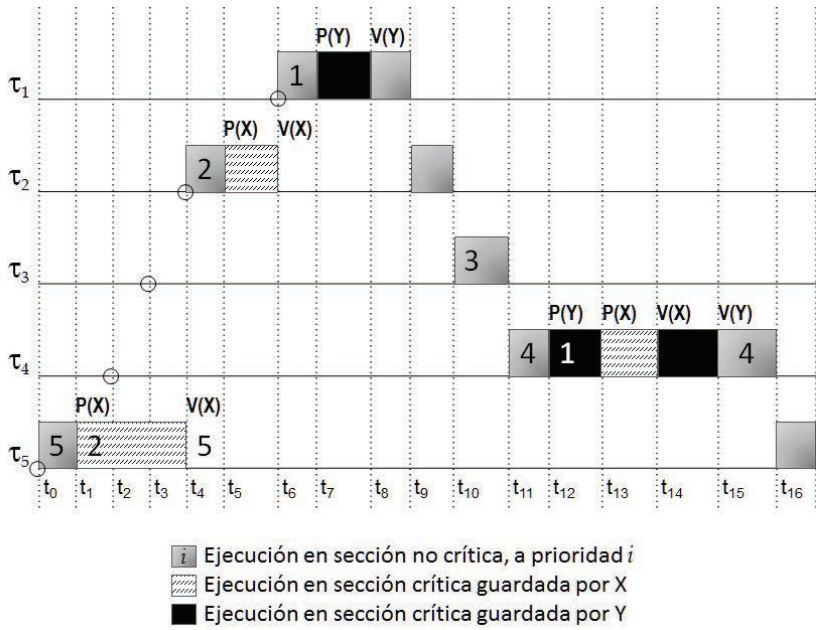

Figura 6.5: Protocolo del techo de prioridad inmediato.

Nótese que aunque la tarea τ_3 no tiene ninguna sección crítica, también sufre un *bloqueo indirecto* entre los instantes t_3 y t_4.

6.4.2 Propiedades

El *protocolo de techo de prioridad inmediato* es un protocolo de sincronización de tiempo real con dos propiedades importantes:

1. Inversión de prioridades limitada. Cualquier tarea τ_i puede ser bloqueada como máximo una sola vez por una tarea de prioridad inferior.
2. Libre de interbloqueos.

El *protocolo de techo de prioridad inmediato* garantiza que una tarea de prioridad alta será bloqueada por a lo sumo una única sección crítica de cualquier tarea de prioridad menor. Además, una tarea sólo podrá ser bloqueada al principio de su ejecución. (Obsérvese esta situación para la tarea τ_3 entre t_3 y t_4, y para la tarea τ_4 entre t_2 y t_4, en la figura 6.5). Una vez que la tarea inicia su ejecución, todos los semáforos que necesite deberían estar libres.

El protocolo también evita el interbloqueo mutuo, como se muestra en el siguiente ejemplo; compárese con el ejemplo de la figura 6.4:

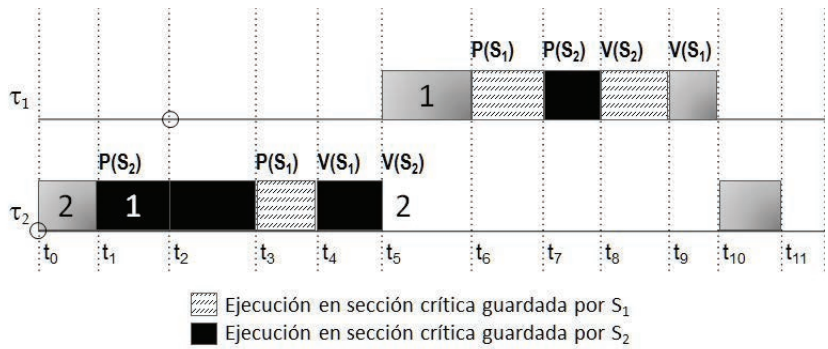

Figura 6.6: Evitación de interbloqueo con el protocolo del techo de prioridad inmediato

Supongamos que tenemos dos tareas τ_1 y τ_2. Además, hay dos estructuras de datos protegidas por los semáforos binarios S_1 y S_2, respectivamente. Supongamos que τ_1 cierra los semáforos en el orden S_1, S_2, mientras que τ_2 lo hace en el orden inverso. Además asumimos que τ_1 tiene mayor prioridad que τ_2.

Como ambas tareas utilizan los dos semáforos, los techos de prioridad de ambos semáforos son iguales a la prioridad de la tarea τ_1. Supongamos que en el instante t_0, τ_2 comienza su ejecución y en t_1 cierra el semáforo S_2. Su prioridad se eleva a la del techo. En t_2, la tarea τ_1 se activa, pero no puede expulsar a τ_2 pues no tiene suficiente prioridad. En el instante t_3, τ_2 cierra el semáforo S_1 y continúa con la misma prioridad pues tiene el mismo valor de techo. En algún momento

τ_2 abrirá los semáforos S_1 y S_2 al finalizar sus secciones críticas recuperando su prioridad original. Entonces τ_1 expulsará a τ_2 y podrá cerrar el semáforo S_1 y posteriormente S_2 sin quedarse bloqueada.

De hecho, si una tarea retiene un recurso (semáforo) mientras espera por otro, entonces ambos recursos tienen el mismo techo de prioridad. Dado que una tarea no puede expulsar a otra de su misma prioridad, se concluye que una vez que una tarea ha accedido al recurso, todos los demás recursos estarán disponibles cuando se necesiten. No hay posibilidad de esperas circulares y se previenen los interbloqueos.

6.4.3 Cálculo de los factores de bloqueo

Este protocolo tiene un comportamiento predecible, en el sentido de que se puede calcular el coste que tiene en el funcionamiento del sistema. Ese coste se puede modelar como un término o factor de bloqueo. El bloqueo sufrido por una tarea τ_i es equivalente a la duración de la sección crítica más larga, con techo de prioridad mayor o igual que la asignada a τ_i, y ejecutada por tareas de prioridad menor que τ_i.

Sean $lp(i)$ el conjunto de tareas con prioridad menor que i, $usa(i)$ el conjunto de semáforos que puede usar la tarea i, $techo(s)$ el techo de prioridad del semáforo s, $pri(i)$ la prioridad de la tarea i, y $C_{i,s}$ la longitud de la sección crítica de la tarea i que envuelve el semáforo s, entonces el *factor de bloqueo* para el caso peor de la tarea i se calcula mediante:

$$B_i = \max_{\{k,s|k\in lp(i)\wedge s\in usa(k)\wedge techo(s)\geq pri(i)\}} C_{k,s} \tag{6.3}$$

es decir, se consideran las tareas k de prioridad inferior a i, y entonces se miran los semáforos s que utilizan; se seleccionan aquellos cuyo techo tenga prioridad mayor o igual a i; se consideran los tiempos de cómputo de las secciones que guardan esos semáforos $C_{k,s}$ y se selecciona la de mayor duración.

Nótese que B_n es siempre cero, ya que la tarea de prioridad más baja no puede, por definición, ser bloqueada por otras tareas de prioridad inferior.

Por ejemplo, para el siguiente conjunto de tareas, donde se indica sus prioridades, la duración de las secciones críticas y el semáforo que las guarda, tenemos que: $techo(S_1) = 1$, $techo(S_2) = 1$, $techo(S_3) = 2$, $techo(S_4) = 3$.

Tarea	***prio***	***Sem***	C_{i,S_j}
τ_1	1	S_1	4
		S_2	3
τ_2	2	S_2	1
		S_3	2
τ_3	3	S_1	3
		S_3	1
		S_4	2
τ_4	4	S_1	3
		S_2	4
		S_4	5

Para calcular el factor de bloqueo de τ_1, B_1, hemos de considerar las tareas de prioridad inferior, en este caso τ_2, τ_3 y τ_4; seleccionar aquellos semáforos que utilizan cuyo techo sea mayor o igual a 1, en este caso S_1 y S_2; mirar las duraciones de las secciones críticas que guardan esos semáforos: 4, 3 y 3, para S_1 y 3, 1 y 4 para S_2; y seleccionar la de mayor duración, que en este caso es 4. Por tanto, $B_1 = 4$.

También podemos construir, a partir de los datos iniciales, la siguiente tabla donde se identifica claramente cada tarea qué semáforo utiliza y durante cuánto tiempo:

	S_1	S_2	S_3	S_4
τ_1	4	3		
τ_2		1	2	
τ_3	3		1	2
τ_4	3	4		5

Para calcular B_2, seleccionamos los semáforos que utilizan las tareas de las filas inferiores cuyo techo sea ≥ 2, en este caso S_1, S_2 y S_3. Mirando las columnas de la tabla anterior identificamos la duración de las secciones críticas que guardan esos semáforos y seleccionamos la de mayor duración. En este caso esa duración es 4.

Siguiendo el mismo procedimiento calcularíamos $B_3 = 5$. La única tarea menos prioritaria que τ_3 es τ_4 y todos los semáforos utilizados por τ_4 tienen un techo de prioridad mayor o igual a la prioridad de τ_3. La duración de la sección crítica más larga protegida por tales semáforos es 5. Por tanto, ese será el valor de B_3.

Finalmente $B_4 = 0$ por definición, pues no hay ninguna tarea con prioridad inferior a la de τ_4.

6.4.4 Cálculo del tiempo de respuesta con factores de bloqueo

Sea B_i la duración más larga de bloqueo que puede experimentar la tarea i. La ecuación 6.1 puede ampliarse para calcular los tiempos de respuesta del caso peor, cuando utilizamos el *protocolo de techo de prioridad inmediato* para sincronizar las tareas periódicas:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{6.4}$$

Obsérvese que lo único que hay que hacer es sumar el factor de bloqueo de una tarea en el cálculo de su propio tiempo de respuesta.

Para ilustrar el uso de la expresión anterior, asumiremos el conjunto de tareas presentado en la figura 6.7, realizando una asignación de prioridades basada en los plazos de manera inversamente proporcional a la extensión de los plazos. Esto es, la tarea con plazo más breve será la de mayor prioridad. Utilizaremos este ejemplo para analizar si ese sistema es planificable.

Tarea	***Periodo (T_i)***	***Plazo (D_i)***	***Cómputo (C_i)***
τ_1	40	30	8
τ_2	12	10	6
τ_3	80	75	20

Tarea	***Semáforo***	***Duración SC***
τ_1	S_2	3
τ_2	S_1	1
τ_3	S_1	4
	S_2	2

Figura 6.7: Sistema formado por tres tareas y dos semáforos.

Empezaremos realizando el cálculo de sus factores de bloqueo. Para ello convendría empezar por la tarea menos prioritaria pues, por definición, su factor de bloqueo será cero. Dada la definición del sistema utilizado en este ejemplo, la tarea menos prioritaria será aquella con un plazo más largo: τ_3. Por tanto, $B_3 = 0$. Esto implica que: $pri(\tau_2) > pri(\tau_1) > pri(\tau_3)$.

Antes de continuar con la siguiente tarea menos prioritaria se procederá a identificar los techos de prioridad de cada semáforo. Así, S_1 es utilizado por τ_2 y τ_3 por lo que su techo es $pri(\tau_2)$ (prioridad máxima). Por su parte, S_2 es utilizado por τ_1 y τ_3. Su techo es $pri(\tau_1)$, con una prioridad intermedia.

La segunda tarea menos prioritaria es τ_1, pues su plazo (30) es inferior al de τ_3 (75) pero mayor que el de τ_2 (10). Para calcular su factor de bloqueo seleccionaremos aquellos semáforos utilizados por tareas con prioridad inferior a la suya (τ_3) que tengan un techo de prioridad igual o superior a la prioridad de τ_1. Como τ_3 utiliza los dos semáforos y ambos tienen una prioridad igual o superior a la de τ_1, el factor de bloqueo B_1 es la duración de la sección crítica más larga que haya en el sistema: 4. Por tanto, $B_1 = 4$.

Finalizaremos el cálculo de los factores de bloqueo, averiguando el valor de B_2. El semáforo por el que habrá que preocuparse es S_1, pues es el único cuyo techo es igual o superior a $pri(\tau_2)$. La duración de la sección crítica más larga relacionada con S_1 es 4. Por ello, $B_2 = 4$.

Ahora que ya se conocen los factores de bloqueo de las tres tareas se procederá a verificar si este sistema es planificable, según el algoritmo explicado en la sección 6.2.

El test de viabilidad se inicia con la tarea más prioritaria, siguiendo un orden decreciente de prioridad. Por tanto, habrá que analizar en primer lugar la planificabilidad de τ_2:

$$R_2^0 = C_2 + B_2 = 6 + 4 = 10$$

Como no hay otras tareas más prioritarias, no es necesario seguir iterando y $R_2 = 10$. Esto cumple la restricción que exige el análisis de viabilidad: $R_i \leq D_i$, ya que $R_2 = D_2 = 10$.

Por tanto, ahora se pasará a analizar la viabilidad de la siguiente tarea en orden de prioridad: τ_1.

$$R_1^0 = C_1 + B_1 + C_2 = 8 + 4 + 6 = 18$$

$$R_1^1 = C_1 + B_1 + \left\lceil \frac{R_1^0}{T_2} \right\rceil \cdot C_2 = 8 + 4 + \left\lceil \frac{18}{12} \right\rceil \cdot 6 = 8 + 4 + 12 = 24$$

$$R_1^2 = C_1 + B_1 + \left\lceil \frac{R_1^1}{T_2} \right\rceil \cdot C_2 = 8 + 4 + \left\lceil \frac{24}{12} \right\rceil \cdot 6 = 8 + 4 + 12 = 24$$

El resultado es $R_1 = 24$ pues las dos últimas iteraciones han proporcionado ese valor. Ese resultado es inferior al plazo de τ_1 (D_1) cuyo valor es 30. Por tanto, τ_1 es viable.

Para terminar, habrá que analizar la viabilidad de la tarea menos prioritaria: τ_3.

$$R_3^0 = C_3 + B_3 + C_2 + C_1 = 20 + 0 + 6 + 8 = 34$$

$$R_3^1 = C_3 + B_3 + \left\lceil \frac{R_3^0}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_3^0}{T_1} \right\rceil \cdot C_1 = 20 + 0 + \left\lceil \frac{34}{12} \right\rceil \cdot 6 + \left\lceil \frac{34}{40} \right\rceil \cdot 8 = 20 + 0 + 18 + 8 = 46$$

$$R_3^2 = C_3 + B_3 + \left\lceil \frac{R_3^1}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_3^1}{T_1} \right\rceil \cdot C_1 = 20 + 0 + \left\lceil \frac{46}{12} \right\rceil \cdot 6 + \left\lceil \frac{46}{40} \right\rceil \cdot 8 = 20 + 0 + 24 + 16 = 60$$

$$R_3^3 = C_3 + B_3 + \left\lceil \frac{R_3^2}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_3^2}{T_1} \right\rceil \cdot C_1 = 20 + 0 + \left\lceil \frac{60}{12} \right\rceil \cdot 6 + \left\lceil \frac{60}{40} \right\rceil \cdot 8 = 20 + 0 + 30 + 16 = 66$$

$$R_3^4 = C_3 + B_3 + \left\lceil \frac{R_3^3}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_3^3}{T_1} \right\rceil \cdot C_1 = 20 + 0 + \left\lceil \frac{66}{12} \right\rceil \cdot 6 + \left\lceil \frac{66}{40} \right\rceil \cdot 8 = 20 + 0 + 36 + 16 = 72$$

$$R_3^5 = C_3 + B_3 + \left\lceil \frac{R_3^4}{T_2} \right\rceil \cdot C_2 + \left\lceil \frac{R_3^4}{T_1} \right\rceil \cdot C_1 = 20 + 0 + \left\lceil \frac{72}{12} \right\rceil \cdot 6 + \left\lceil \frac{72}{40} \right\rceil \cdot 8 = 20 + 0 + 36 + 16 = 72$$

Como estas dos últimas iteraciones generan un mismo valor, el test de viabilidad termina tras esta quinta iteración. Su resultado es $R_3 = 72$, inferior al plazo de τ_3 (D_3) cuyo valor es 75. Por tanto, esta última tarea τ_3 también es viable y, consecuentemente, el sistema es planificable.

6.5 Resumen

Esta unidad introduce los sistemas de tiempo real, describiendo los conceptos más importantes relacionados con este tipo de sistemas. Las actividades que se ejecutan en un sistema de tiempo real reciben el nombre de tareas. Generalmente se requiere que cada tarea finalice su ejecución en un determinado plazo. Existen diversos tipos de tareas según el criterio utilizado para clasificarlas: acríticas o críticas (en función de si se admite o no el incumplimiento de sus plazos); periódicas, aperiódicas o esporádicas (en función de cómo se generen o repitan las tareas). En el caso más general se asumirá un sistema formado por tareas periódicas gestionado por un algoritmo de planificación expulsivo basado en prioridades fijas.

Antes de desarrollar una aplicación de tiempo real conviene realizar un análisis de su planificabilidad. Para ello se estudia el caso peor. Si se asume un sistema formado por tareas periódicas independientes, ese caso peor surge cuando se suponga que todas las tareas se activarán simultáneamente. En esa situación habrá que verificar si cada una de ellas podrá respetar su primer plazo.

Todos los sistemas de tiempo real son ejemplos de sistemas concurrentes, pues el conjunto de tareas que los forman cooperan entre sí. Tal colaboración necesitará que se compartan ciertos recursos. Generalmente esos recursos tendrán que

utilizarse en exclusión mutua, como ya hemos visto en las unidades anteriores. Por ello, tendrán que utilizarse algunos mecanismos de sincronización: locks, monitores, semáforos, ... El uso de estas herramientas puede provocar una inversión de prioridades que podría ocasionar un incumplimiento en los plazos de ejecución de una o más tareas e incluso interbloqueos. Para que se dé una inversión de prioridades, una tarea de baja prioridad estará en una sección crítica, suspendiendo a otra tarea más prioritaria que también quiera ejecutar esa misma sección. Si en esa situación llegase a estar preparada otra tarea con prioridad intermedia, expulsaría a la tarea con menor prioridad y prolongaría el intervalo de suspensión de la tarea más prioritaria. Tal extensión del intervalo de suspensión no podría acotarse y eso conduciría a un incumplimiento de su plazo.

Con el protocolo de techo de prioridad inmediato se evitan esos problemas. Ese protocolo tiene en cuenta la prioridad de las tareas que podrán utilizar cada semáforo, asignando como techo de cada semáforo la prioridad mayor de entre todas sus tareas usuarias. Cuando una tarea ejecute la sección crítica protegida mediante cierto semáforo adoptará la prioridad que éste tenga como techo. Esto evita el problema de inversión de prioridades, pues las tareas con prioridad intermedia no podrán alterar de ninguna manera la ejecución de las más prioritarias.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Caracterizar adecuadamente los sistemas de tiempo real.
- Identificar las dificultades que conlleva la planificación de tareas en sistemas de tiempo real.
- Aplicar el análisis de viabilidad en diferentes supuestos.
- Identificar el problema de inversión de prioridades y las consecuencias que puede tener.
- Implantar el protocolo de techo de prioridad inmediato en aquellos sistemas que lo requieran.

Unidad 7

CONCEPTOS BÁSICOS DE LOS SISTEMAS DISTRIBUIDOS

7.1 Introducción

La mayoría de los ordenadores actuales tienen acceso a algún tipo de red de comunicaciones. Debido a ello, las aplicaciones que se ejecuten en estos sistemas pueden acceder a recursos informáticos ubicados en otros ordenadores y, gracias a esto, múltiples aplicaciones modernas pueden organizarse como un conjunto de componentes que podrán ser instalados en más de un ordenador y que colaborarán entre sí para ofrecer un determinado servicio al usuario.

Para que se pueda hablar de una aplicación o sistema distribuidos, múltiples ordenadores independientes deberán formar parte de tal sistema. Todos ellos deben ser capaces de ofrecer algún servicio común al usuario. Al igual que en cualquier aplicación o sistema concurrente, existirán múltiples actividades que colaborarán entre sí. En este caso dichas actividades se estarán ejecutando en ordenadores diferentes y para colaborar deberán enviarse mensajes entre ellas. Por este motivo todo sistema distribuido es también un sistema concurrente y todo lo que se ha aprendido en las unidades anteriores ofrecerá una buena base para entender cómo debe gestionarse un sistema distribuido.

En esta unidad se ofrecerá una definición de los sistemas distribuidos en la sección 7.2, acompañada por una descripción de sus objetivos más importantes en la sección 7.3. Finalmente, la sección 7.4 discute algunos problemas que deben considerarse a la hora de diseñar cualquier algoritmo distribuido.

7.2 Definición de sistema distribuido

En [Tv08] se define a los *sistemas distribuidos* de la siguiente manera:

> *Un sistema distribuido es una colección de ordenadores independientes que ofrece a sus usuarios la imagen de un sistema único y coherente.*

Por tanto, desde el punto de vista del "hardware" en un sistema de este tipo encontraremos múltiples ordenadores independientes, es decir, cada uno dispondrá de su procesador, memoria, placa base, dispositivos de E/S conectados, etc. Para cualquier observador resultará claro que existen múltiples máquinas en un sistema de este tipo. Por otra parte, cuando un usuario interactúe con este sistema, la imagen percibida será la de un único equipo informático. Es decir, el software se encarga de ocultar la existencia de múltiples máquinas y el usuario cree que está utilizando un solo ordenador con gran capacidad de cómputo y con un amplio conjunto de recursos disponibles.

Para que esta imagen de sistema único pueda proporcionarse, los distintos ordenadores que compongan el sistema tendrán que colaborar entre sí para proporcionar cierto tipo de servicios al usuario. Si tal colaboración no se diera, el usuario seguiría viendo a múltiples ordenadores independientes y cada uno de ellos sería un sistema con sus propios recursos pero desligado de los demás. En ese último caso, el usuario debería disponer de una cuenta en cada uno de los ordenadores que pudiera utilizar y, una vez autenticado, solo podría utilizar los recursos locales de tal ordenador. Por el contrario, en un sistema distribuido bastaría con acceder a uno de los ordenadores del sistema para, a continuación, poder utilizar cualquiera de los recursos presentes en cualquiera de los ordenadores que formen tal sistema.

Los ordenadores de un sistema distribuido deben colaborar entre sí para ofrecer esa imagen de sistema único. Para ejecutar cualquier aplicación distribuida habrá que generar múltiples actividades que podrán ejecutarse simultáneamente para lograr tal colaboración. Queda por ello claro que toda aplicación distribuida será también una aplicación concurrente.

Cualquier *servicio de búsqueda* en Internet (p.ej. Google, Bing o Yahoo!) es un buen ejemplo de sistema distribuido. Quien los utilice no tiene por qué saber que en la parte servidora hay un alto número de ordenadores colaborando entre sí para servir esas consultas. De hecho, se percibe una imagen similar a la que ofrecería un único servidor. Sin embargo, si pensamos que en cada momento habrá muchos usuarios de ese sistema lanzando entre todos un alto número de consultas, entonces se advierte que un solo ordenador (por potente que sea) no podría soportar dicha carga. Así, cada vez que algún usuario realice una consulta utilizando un buscador determinado podría ser atendido por un ordenador diferente. Por tanto, en este ejemplo sí que se observa que esa imagen de sistema único y coherente se ha proporcionado de manera transparente.

Otro ejemplo de sistema distribuido nos lo ofrece el servicio de correo electrónico. Si hemos tenido que configurar alguna vez el programa necesario para utilizar este servicio, se habrá tenido que especificar el nombre de algún servidor (que podría ser distinto para el correo entrante y saliente). Sin embargo, ese servidor no será el único que intervenga para hacer llegar nuestros correos a sus destinatarios. Habrá un buen número de servidores adicionales entre los que se propagarán tales mensajes hasta llegar a sus dominios destinatarios. No obstante, los usuarios de este servicio no tienen por qué saber nada sobre tales servidores. Para ellos, el servicio de correo electrónico es un servicio que funciona y sobre el que poco se tiene que saber. De hecho, aquí se vuelve a obtener la imagen de un sistema en el que, de manera transparente, se proporciona cierta funcionalidad y donde interviene un buen número de ordenadores, pero donde solo el servidor emisor (cuando enviamos correos) y el servidor receptor (cuando los recibimos) llegan a ser visibles al usuario.

Como resultado de esta definición, en todo sistema distribuido se observan las siguiente características :

- *Ocultación*: Debido a esa imagen de sistema único y coherente, se ocultan las diferencias existentes entre todos los ordenadores que componen el sistema. Además, tampoco resultará perceptible la complejidad de los mecanismos de comunicación que lleguen a ser necesarios para que las actividades ejecutadas en cada uno de estos ordenadores cooperen entre sí.

- *Acceso homogéneo*: Independientemente del lugar desde el que se realicen los accesos, tales accesos reciben una misma imagen y no observan una interfaz diferente. Tanto los usuarios como las aplicaciones interactúan con el sistema de una manera uniforme, con independencia del ordenador concreto que haya sido utilizado para atender tal acceso.

- *Escalabilidad*: Como el sistema ya está compuesto por múltiples ordenadores independientes no debería resultar difícil la incorporación de más ordenadores para atender a un mayor número de usuarios.

- *Disponibilidad*: Los servicios ofrecidos por un sistema distribuido deberían estar siempre disponibles. Para ello, las aplicaciones deberán estar compuestas por múltiples módulos y cada uno de esos módulos debería replicarse de manera que cuando algún ordenador falle siempre haya otras copias del módulo en ordenadores que no fallen.

  Obsérvese que si no se respetara esta última característica se perdería la imagen de sistema único que se pretendía garantizar. En ese caso el usuario estaría interactuando con uno de los ordenadores del sistema distribuido y observaría que alguno de los servicios solicitados no funcionaría. Sin embargo, el ordenador que se estaría utilizando no habría fallado y eso sugeriría que el problema estaría causado por el fallo de otro ordenador. Así, el usuario

advertiría que no toda la funcionalidad del sistema estaba siendo proporcionada por un único ordenador, sino por un conjunto de ellos.

Como los sistemas operativos actuales suelen estar centrados en la gestión de sus recursos locales (y no en aquellos que residan en otros ordenadores), esa imagen de sistema único no será proporcionada por un sistema operativo. Se necesita otra capa de software, llamada *middleware* [Ber96] (Figura 7.1), ubicada sobre los sistemas operativos locales, para realizar esa gestión uniforme de los recursos del sistema distribuido. Con ello, aprovechando la interfaz de servicios proporcionada por el middleware ya será más sencillo programar aplicaciones distribuidas. Así, por una parte se dispondrá del sistema operativo, encargado de la gestión de los recursos locales y de proporcionar herramientas de comunicación estándar. Por otra parte, el middleware será el encargado de realizar cierta gestión uniforme de los recursos existentes en todo el sistema distribuido.

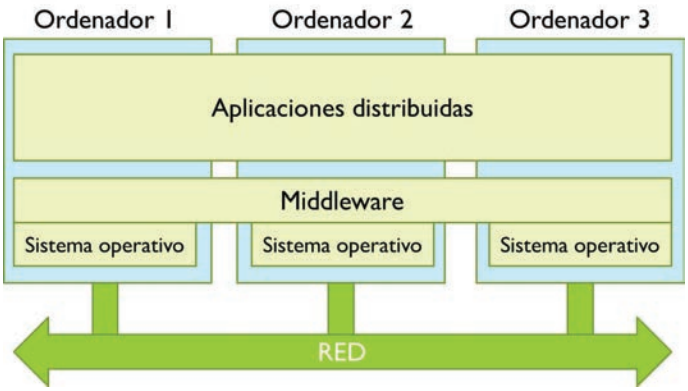

Figura 7.1: Arquitectura de un sistema distribuido.

Para finalizar esta definición de los sistemas distribuidos hay que mencionar que en un buen número de publicaciones relacionadas con esta temática se suele utilizar el término *nodo* para hacer referencia a un componente del sistema distribuido. Dependiendo del contexto, dicho término se podrá referir a un ordenador o a un proceso del sistema. En general, cualquier algoritmo distribuido utilizará múltiples procesos que cooperarán entre sí y que, generalmente, estarán ubicados en ordenadores distintos. A la hora de diseñar algoritmos distribuidos se suele utilizar un *modelo lógico del sistema* [Sch93] en el que gran parte de las características del sistema real no tienen por qué ser consideradas relevantes. En esa fase de diseño, el sistema distribuido puede verse como un grafo en el que los nodos representan a las unidades de cómputo y las aristas representan los enlaces de comunicación. Por ello, no es raro referirse a los componentes del sistema utilizando ese término.

7.3 Objetivos de los Sistemas Distribuidos

Según [Tv08], existen cuatro objetivos importantes que todo sistema distribuido tendrá que cumplir: (a) facilitar el acceso a recursos remotos, (b) proporcionar transparencia de distribución, (c) ser un sistema abierto, y (d) ser un sistema escalable. En las secciones siguientes se describirá cada uno de ellos.

7.3.1 Acceso a recursos remotos

Como un sistema distribuido debe ofrecer una imagen de sistema único y estará compuesto por un conjunto de ordenadores independientes, el usuario tendrá acceso a un elevado número de recursos en un sistema de este tipo. Cada ordenador dispondrá de cierta cantidad de memoria, dispositivos de almacenamiento, otros dispositivos de E/S (impresoras, escáneres,...), al menos un procesador, etc. Aparte, también facilitará una segunda colección de recursos lógicos: ficheros, herramientas de sincronización (semáforos, locks, monitores), herramientas de comunicación (tubos, buzones, *sockets*,...), etc. Todos esos *recursos* deberán ser accesibles para todos los usuarios del sistema distribuido. Por ello, el sistema debe proporcionar mecanismos que permitan el uso de recursos remotos.

El uso de recursos remotos ofrece la ventaja de reducir el coste económico: en lugar de tener una instancia de cada recurso en cada ordenador (o para cada usuario), los recursos más caros (impresoras de elevadas prestaciones, *plotters*, etc.) podrán compartirse entre todos los usuarios. Así, bastará con adquirir un solo ejemplar de tales dispositivos pero todos los usuarios podrán llegar a utilizarlo cuando lo necesiten. Para gestionar el acceso sobre estos recursos compartidos suele existir un proceso servidor que administra las peticiones que vaya recibiendo, secuenciándolas para garantizar un uso en exclusión mutua si así lo requiriese tal dispositivo o recurso.

El inconveniente que puede llegar a plantear un uso de recursos remotos será la gestión de la seguridad en el acceso a tales recursos. Para que los accesos sean seguros se requerirá el uso de algún mecanismo de autenticación y de control de acceso. Para que se mantenga la imagen de sistema único, todos los ordenadores del sistema deberán utilizar el mismo directorio de usuarios y los mismos sistemas de autenticación y control de acceso. Desgraciadamente, no todos los sistemas distribuidos actuales llegan a ese nivel de uniformidad en la gestión de los recursos en caso de apoyarse sobre un conjunto de ordenadores heterogéneo soportado por diferentes sistemas operativos locales. No obstante, cuando se utiliza una misma familia de sistemas operativos en todos los nodos esa gestión uniforme sí que llega a ser posible, como ya demuestran los servicios del Directorio Activo de Windows Server [RKMW08].

En la Unidad 10 se analizarán las gestiones básicas necesarias para que los distintos componentes de una aplicación puedan interactuar entre sí y puedan acceder a diferentes recursos, tanto locales como remotos.

7.3.2 Transparencia de distribución

La *transparencia de distribución* hace referencia a la imagen de sistema único y coherente que se menciona en la definición de los sistemas distribuidos. Es decir, un sistema proporciona esa transparencia si es capaz de ocultar que físicamente está compuesto por múltiples ordenadores independientes.

Transparencia	Descripción
Acceso	Oculta diferencias en la representación de los datos y en cómo se accede a los recursos.
Fallos	Oculta el fallo y recuperación de los recursos.
Migración	Oculta el hecho de que un recurso pueda migrar de un lugar a otro.
Persistencia	Oculta el hecho de que un recurso esté situado en memoria volátil o en memoria persistente.
Replicación	Oculta el hecho de que un recurso pueda tener más de una réplica, impidiendo que un usuario pueda saber con qué réplica está interactuando en cada momento.
Reubicación	Oculta el hecho de que un recurso pueda moverse de un lugar a otro mientras se utilice.
Transacción	Oculta la coordinación entre las actividades que gestionen un conjunto de recursos para mantener su consistencia.
Ubicación	Oculta dónde se ubican los recursos.

Tabla 7.1: Tipos de transparencia.

Según el estándar *ISO/IEC 10746-1:1998(E)* [ISO98] los principales tipos de transparencia de distribución son los que se explican seguidamente y resumen en la Tabla 7.1:

1. ***Acceso***: La *transparencia de acceso* oculta las diferencias en la representación de los datos y en los mecanismos de interacción entre los componentes de una aplicación o sistema distribuido. Por tanto, el objetivo de la transparencia de acceso es facilitar la interacción entre componentes en caso de trabajar sobre un *sistema distribuido heterogéneo* (es decir, aquél en el que sus ordenadores utilicen diferentes arquitecturas hardware).

   En la pila de protocolos de la arquitectura OSI [Zim80], la gestión de la transparencia de acceso recaía en el *nivel de presentación* (nivel 6), ubicado justo por debajo del nivel de aplicación. Por tanto, era una gestión que

debía ser proporcionada por los sistemas de comunicaciones y por la que cualquier desarrollador de aplicaciones a ejecutar sobre una red no tendría por qué preocuparse.

En las arquitecturas de comunicaciones basadas en TCP/IP la responsabilidad a la hora de facilitar esta transparencia de acceso recaería en el nivel de middleware que ya se ha presentado en la figura 7.1. Dicho nivel estaría ubicado sobre el nivel de transporte (nivel 4) y facilitaría una interfaz más cómoda para desarrollar aplicaciones distribuidas dentro del nivel de aplicación.

2. ***Fallos***: La *transparencia de fallos* oculta el hecho de que los componentes de una aplicación fallen. A su vez, tras los fallos que hayan llegado a sufrir, cada componente necesitaría cierto tiempo para llegar a recuperarse y ser capaz de ofrecer una funcionalidad completa. Ninguno de esos hechos llegará a observarse en caso de proporcionar este tipo de transparencia. Con ello, el programador podrá desentenderse de las situaciones de fallo pues jamás resultarán visibles.

   Para obtener este tipo de transparencia habrá que replicar cada uno de los componentes de la aplicación. Además, habrá que facilitar mecanismos que permitan diagnosticar fácilmente los fallos y redirigir las peticiones sobre las réplicas activas de cada componente de la aplicación descartando temporalmente a aquellas réplicas que hayan fallado hasta que se complete su recuperación. Por ello, la transparencia de replicación es un requisito para que se llegue a obtener transparencia de fallos.

   Un concepto relacionado con la transparencia de fallos es la *disponibilidad* de un sistema. La disponibilidad en un instante t se define [Nel90] como la probabilidad de que el sistema o componente esté operativo (es decir, pueda funcionar correctamente) en el instante t. Si se ha podido evaluar ese sistema durante un intervalo temporal prolongado será posible aproximar su disponibilidad utilizando la fórmula:

$$D = \frac{IMEF}{IMEF + IMDR}$$

   siendo IMEF la duración del *intervalo medio entre fallos* e IMDR la duración del *intervalo medio de reparación*. Es decir, se utilizará como numerador la duración media de todos los intervalos entre cada par de fallos consecutivos y como denominador la suma de dicha duración media y la duración media de cada acción de recuperación asociada a dichos eventos de fallo. Obsérvese que el tiempo en que el sistema no funcionó también se incluirá en los intervalos medios de reparación. En la práctica eso es lo mismo que dejar en el numerador el tiempo en que el sistema estuvo funcionando y en el denominador todo el tiempo del estudio realizado (intervalos en funcionamiento más intervalos en que el sistema no llegó a funcionar).

Si un sistema es capaz de proporcionar una transparencia de fallos total, entonces su disponibilidad será 1 (es decir, del 100 %).

3. ***Ubicación***: La *transparencia de ubicación* oculta el uso de información referente al lugar que ocupa cada recurso (es decir, en qué ordenador se ubica realmente) a la hora de identificarlo o usar sus operaciones.

   En cuanto a la identificación, la transparencia de ubicación requiere que el servicio de nombres (a estudiar en la unidad 10) utilice nombres (y espacios de nombrado) que sean independientes de la ubicación física real de las entidades nombradas.

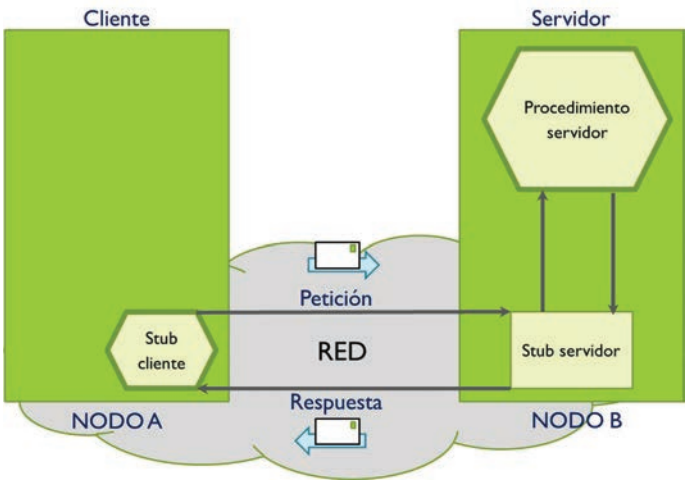

Figura 7.2: Transparencia de ubicación en una RPC.

Respecto al uso de operaciones ofrecidas por los recursos, el mecanismo de llamada a procedimiento remoto (o RPC) que se estudiará en la unidad 8 (sección 8.3) ilustra cómo puede proporcionarse esta transparencia de ubicación. El modelo RPC se basa en la invocación a procedimientos ubicados en máquinas remotas, ocultando al cliente el hecho de que el procedimiento está ubicado en otra máquina. En una RPC (tal como se muestra en la figura 7.2) el proceso que invoque una operación realiza tal invocación sobre un *stub cliente* local que ofrece la misma interfaz que el procedimiento que debía invocarse, ocultando la transferencia de mensajes entre el nodo invocador y el nodo en el que resida el procedimiento invocado. De esta manera se oculta la ubicación real de los procedimientos, ofreciendo siempre la imagen de ser locales.

4. ***Migración***: La *transparencia de migración* oculta a un determinado componente el hecho de que el sistema lo traslade a otra ubicación. El objetivo de la migración es realizar un equilibrado de la carga o reducir el retardo

de las comunicaciones. Así, en caso de que un nodo determinado se llegara a sobrecargar, alguno de los componentes instalados en él serían migrados a nodos cuya carga actual fuera ligera. De manera similar, si la mayoría de los clientes que utilicen los servicios de alguno de los componentes de una aplicación distribuida estuvieran ubicados en una determinada zona que quedara apartada del nodo en el que ahora mismo residiera dicho servicio, convendría migrarlo a un nodo más cercano a ese conjunto de usuarios. De esa manera se reduciría el retardo en la propagación de los mensajes entre dicho componente y sus usuarios.

Obsérvese que si se proporciona transparencia de migración, consiguiendo que quien debe implantar un componente sea incapaz de advertir que dicho componente pase en algún momento de un nodo a otro en un sistema distribuido, también se debería proporcionar transparencia de ubicación para que quien deba utilizar los servicios de dicho componente pueda hacerlo sin ningún problema. Es más, como los componentes de una aplicación distribuida deben colaborar entre sí, en el código que incluya tal módulo también habrá alguna invocación de las operaciones o servicios facilitados por otros módulos. Ello conlleva que dicha transparencia de migración también implique transparencia de ubicación de los procedimientos, pues esas invocaciones siempre ofrecerán la imagen de ser locales, independientemente de dónde residan los componentes que proporcionen tales operaciones.

5. ***Reubicación***: La *transparencia de reubicación* permite que un determinado componente sea migrado a otro nodo mientras otros componentes del sistema todavía estén interactuando con él. La transparencia de migración podía obtenerse de manera general bloqueando la llegada de nuevas invocaciones sobre las operaciones facilitadas por el componente a migrar y esperando a que finalizaran las que ya estuvieran en marcha. En la transparencia de reubicación no es necesario que se bloquee la prestación de servicios. El componente o recurso podrá ser migrado sin ningún problema y se garantizará que tanto sus clientes potenciales como el propio componente no necesiten modificar su propio programa (o su configuración) ni deban sufrir ningún bloqueo para interactuar con otros componentes del sistema mientras dure la reubicación.

   Un ejemplo de transparencia de reubicación se da cuando un usuario esté trasladándose en algún medio de locomoción (por ejemplo, en tren) y esté utilizando su ordenador portátil para realizar alguna actividad que requiera conexión a Internet (facilitada mediante un módem 3G, por ejemplo). Aunque el trayecto recorrido haya sido largo, su conexión a Internet habrá podido mantenerse sin excesivos problemas y cada cierta distancia habrá cambiado de "antena" para obtener cobertura. Todo ello sucede de manera transparente y sin observar ninguna desconexión.

6. ***Replicación***: La *transparencia de replicación* oculta el hecho de que se esté usando un grupo de réplicas de un determinado componente, ofreciendo

la imagen de que tal componente no está replicado. El objetivo de la replicación es mejorar el rendimiento o la disponibilidad de dicho componente (y del sistema en general).

La replicación permite mejorar el rendimiento si el componente replicado ofrece un alto porcentaje de operaciones que solo impliquen una consulta de su estado. Así, cada petición generada por un usuario podría ser redirigida a una réplica distinta, equilibrando la carga generada entre todas ellas. De esta manera se incrementaría el número de peticiones que podrían ser atendidas en cada unidad de tiempo pues múltiples nodos del sistema se estarían repartiendo dicho trabajo.

La replicación también permite mejorar la disponibilidad ya que aunque alguna de las réplicas existentes llegue a fallar, será de esperar que queden otras réplicas activas capaces de atender las nuevas peticiones que vayan generando otros componentes de la aplicación o los usuarios. Por ello, cuando se proporcione transparencia de replicación también se estará proporcionando transparencia de fallos.

Al proporcionar transparencia de replicación, también se proporcionará transparencia de ubicación, pues quien utilice alguna operación del componente replicado no podrá saber qué réplica concreta está atendiendo su petición ni en qué nodo estará ubicada.

7. ***Persistencia***: La *transparencia de persistencia* oculta a un componente la información sobre si otros componentes están activos (es decir, listos para atender de inmediato nuevas peticiones) o inactivos (es decir, si han sido desactivados, persistiendo su estado en almacenamiento secundario hasta su próxima activación). Algunos sistemas operativos y sistemas middleware se encargan de volcar el estado de aquellas aplicaciones que no estén siendo utilizadas durante un tiempo prolongado (es decir, se encargan de desactivarlas), evitando así que acaparen recursos que podrían ser necesarios para la ejecución de otros procesos. La transparencia de persistencia oculta el uso de este tipo de gestión, por lo que los usuarios percibirían la imagen de que todos los componentes de un sistema están siempre activos.

   Generalizando este tipo de gestión, se puede decir que la transparencia de persistencia oculta en qué tipo de memoria (RAM, i.e. memoria volátil que pierde su contenido en caso de que se cortara la alimentación eléctrica; o almacenamiento secundario, que persiste cualquier corte eléctrico) se está almacenando un determinado recurso. Así, se podría estar ofreciendo la imagen de que cierto recurso se mantiene en memoria persistente cuando realmente está almacenado en la RAM de múltiples nodos (manteniendo su contenido ante cualquier problema eléctrico que no afecte a todos esos nodos a la vez) o al contrario: ofrecer la imagen de que un recurso se mantiene en RAM cuando su estado ya ha sido volcado a disco y no queda ninguna copia de él en la memoria principal de ninguno de los nodos. Por ejemplo, algunos

sistemas gestores de bases de datos de elevadas prestaciones garantizan la persistencia de la información pero mantienen siempre los datos en memoria RAM, replicados en múltiples nodos. Dos ejemplos de sistemas de este tipo son *H-store* [KKN+08] y *VoltDB* (http://voltdb.com).

8. ***Transacción***: La *transparencia de transacción* oculta la coordinación entre las actividades que gestionen o utilicen un conjunto de recursos para mantener su consistencia. Obsérvese que cuando múltiples actividades utilicen concurrentemente un conjunto de recursos se corre el riesgo de que alguno de sus accesos deje a alguno de esos recursos en un estado inconsistente. Para evitar tales problemas debe utilizarse algún mecanismo de coordinación o sincronización que garantice que los resultados de las actualizaciones sean consistentes.

   En ediciones anteriores del estándar *ISO/IEC 10746-1:1998(E)* este tipo de transparencia se llamó *transparencia de concurrencia* y con ella se ocultaba el hecho de que un determinado recurso fuera utilizado simultáneamente por más de una actividad o usuario. Sin embargo, su objetivo es el mismo: cuando haya actividades concurrentes todos aquellos mecanismos que resulten necesarios para garantizar la consistencia de los recursos utilizados no deben resultar visibles al programador. El propio sistema debe ofrecer el soporte necesario para que tanto el programador como el usuario no observen ningún problema al programar o utilizar actividades concurrentes.

Aunque cualquier sistema distribuido debería ser capaz de proporcionar estos tipos de transparencia, se debe tener también en cuenta que una transparencia completa puede llegar a ser cara o incluso inalcanzable. Por ejemplo, la transparencia de ubicación debe garantizar que tanto los nombres o identificadores asignados a los recursos como su interfaz de operaciones no debería dar ninguna pista sobre dónde se encuentran los ordenadores que los mantienen. En algunos casos, sí que nos interesará conocer dónde se ubica cada recurso para que así se pueda seleccionar, entre un conjunto de posibles candidatos, aquél que esté más próximo, minimizando así los retardos que implicará la interacción con tal recurso remoto.

Otro tipo de transparencia que será difícil garantizar en todas las situaciones es la transparencia de fallos. Cuando alguna instancia de un recurso que se vaya a utilizar falle, el sistema se encargará de encontrar otras que puedan proporcionar la misma funcionalidad. Sin embargo, no resulta sencillo el diagnóstico de cuándo un recurso o nodo ha llegado a fallar. En ocasiones, un nodo puede que tarde a responder (debido a que soporta mucha carga o a que está ubicado lejos y los mensajes tardan en llegar) pero será difícil fijar un límite para el plazo en que esperaremos sus respuestas. Quizá se considere que ha fallado y esté todavía proporcionando servicio a otros usuarios. Por tanto, distinguir entre un nodo lento y un nodo que ha fallado no resultará fácil.

De manera general *la transparencia suele resultar costosa*. Por ejemplo, cuando se proporciona transparencia de ubicación, los recursos que se estén utilizando parecerán locales pero la mayoría serán remotos y para acceder a ellos tendremos que contactar con sus respectivos servidores y esperar sus respuestas. Esto necesitará que se transmitan mensajes entre diferentes máquinas y que se deba invertir cierto tiempo en dicha transmisión. Eso hace que el rendimiento percibido por el usuario sea inferior al que obtendría en un ordenador local dedicado en exclusiva a su servicio. Como segundo ejemplo se puede citar la transparencia de fallos: los recursos utilizados estarán replicados y cada operación que modifique su estado deberá aplicarse en todas las réplicas antes de proporcionar una respuesta al usuario. De esta manera, si alguna de esas réplicas falla posteriormente, el sistema redirigirá las nuevas peticiones a otras réplicas que permanezcan en funcionamiento y será posible observar que no se ha perdido ninguna de las acciones realizadas previamente. Sin embargo, para mantener la consistencia de todas esas réplicas también se habrá tenido que invertir cierto tiempo y esfuerzo en esa propagación de actualizaciones. Con ello, sería posible observar una ligera pérdida de rendimiento en las operaciones que impliquen una actualización del estado de algún recurso replicado. Afortunadamente, esto se compensa con mejoras en el servicio concurrente de todas aquellas operaciones que únicamente impliquen la consulta del estado del recurso replicado. Por ejemplo, los servidores de bases de datos mejoran su rendimiento en la atención de transacciones de solo lectura cuando la base de datos está replicada en varios nodos: múltiples transacciones pueden ser atendidas simultáneamente en esos nodos.

7.3.3 Sistemas abiertos

Se define como *sistema abierto* aquel que se ajusta a estándares que especifican y describen la sintaxis y la semántica de las operaciones que constituyen el servicio ofrecido por tal sistema.

Al utilizar estándares se potencia la *flexibilidad*:

- Se podrá configurar el sistema combinando múltiples módulos que habrán podido ser desarrollados por organizaciones diferentes. Los sistemas y aplicaciones distribuidos estarán compuestos por múltiples módulos y cada uno de ellos definirá una interfaz clara que los demás asumirán y podrán utilizar. Para que los módulos puedan interactuar entre sí se utilizarán mecanismos y protocolos estandarizados por lo que cada módulo podrá estar desarrollado por un equipo de programadores distinto: el estándar que se esté siguiendo definirá cómo tendrán que interactuar los módulos. Por ello, durante las fases de desarrollo y depuración cada módulo podrá ser comprobado utilizando otros componentes que emulen al resto de módulos que formen parte del sistema en el que finalmente se instalará y ejecutará.

- Resultará sencillo añadir nuevos módulos al sistema cuando sea necesario. Como las funciones a desarrollar por cada componente del sistema estarán bien documentadas y se conocerá en todo momento qué estándares tendrán que respetarse, resultará sencillo integrar nuevos módulos capaces de aportar nuevas funcionalidades, integrándolos de manera sencilla con los que ya formasen parte del sistema hasta ese momento.
- Será sencillo el reemplazo de algún módulo (o conjunto de módulos) por otros que ofrezcan las mismas interfaces, sin que esto afecte al resto del sistema.

Además de estas mejoras en la flexibilidad del sistema, también se obtienen otras características aconsejables:

1. *Interoperabilidad*: Se dice que dos sistemas o componentes presentan interoperabilidad [IEE90] cuando son capaces de intercambiar información y pueden usar posteriormente la información que han llegado a intercambiar. Es decir, gracias al uso de interfaces, mecanismos o protocolos estandarizados diferentes componentes o sistemas son capaces de colaborar entre sí. Esto se logra de manera inmediata si tanto los componentes como cada uno de los sistemas considerados son abiertos.
2. *Portabilidad*: Las aplicaciones que se hayan desarrollado sobre un sistema distribuido abierto serán fácilmente portables a otros sistemas distribuidos abiertos. Esto también es inmediato debido a que tanto el sistema origen como el destino estarán respetando los mismos estándares. Con ello, la aplicación podrá encontrar en ambos sistemas todo aquello que necesite para ser ejecutada. La *portabilidad* se define como [IEE90] la facilidad con que un sistema o componente puede ser transferido desde un entorno *software* o *hardware* a otro.

Como ya se ha comentado al presentar la definición de los sistemas abiertos, su propiedad más importante es la flexibilidad. Para potenciar dicha flexibilidad conviene distinguir entre *políticas* y *mecanismos* a la hora de proporcionar cierta funcionalidad. Por una parte, el *mecanismo* consiste en aquel conjunto de elementos (del hardware y del software) así como el procedimiento que los relacione que deberán utilizarse de cierta manera para lograr alguna funcionalidad. A su vez la *política* establece las reglas o la configuración que deben seguirse a la hora de utilizar los mecanismos para garantizar cierto nivel de prestaciones.

El objetivo en cualquier sistema abierto con un alto grado de flexibilidad será el proporcionar un amplio surtido de mecanismos para desarrollar sus funciones de manera que tanto los administradores como los usuarios puedan decidir qué políticas emplear a la hora de utilizar los mecanismos existentes. De esa manera, cada usuario obtendrá un sistema bien adaptado a sus necesidades de uso.

Así, por ejemplo, si consideramos las transparencias de replicación y fallos, los mecanismos necesarios para proporcionarlas serían los protocolos de replicación que podrían utilizarse. Existe una amplia variedad de ellos, aunque los dos modelos básicos son la *replicación activa* [Sch90] en la que todas las réplicas reciben la misma lista de peticiones y son responsables de atender y ejecutar cada una de ellas; y la *replicación pasiva* [BMST92] en la que una *réplica primaria* atiende de manera directa todas las peticiones generadas por los usuarios para después transmitir al resto de réplicas (o *réplicas secundarias*) las actualizaciones generadas en cada petición. Sin embargo, existen otros modelos de replicación intermedios (semi-pasivo [DSS98], semi-activo [Pow93], ...) que tienen sentido en algunos entornos conservando las mejores características de los dos modelos básicos.

Utilizando tales mecanismos se podrían establecer ciertas políticas, decidiendo entre otros parámetros: el número de réplicas a utilizar, en qué nodo se ubicaría cada réplica, el modelo de consistencia a respetar entre las réplicas, ... Este último parámetro, la *consistencia*, se refiere a las diferencias que llegarán a admitirse entre el estado de cada una de las réplicas. En una consistencia fuerte o estricta no habría diferencias en el estado de las distintas réplicas: todas seguirían la misma secuencia de modificaciones y las aplicarían a la vez. En una consistencia débil o relajada, por el contrario, se admitirían como válidas amplias diferencias tanto en la secuencia como en el instante en que son aplicadas las actualizaciones en cada réplica. Entre un extremo y otro son posibles un buen número de modelos de consistencia, como se describe en [Mos93].

7.3.4 Sistemas escalables

Según [Tv08] la escalabilidad de un sistema distribuido puede analizarse desde tres perspectivas diferentes y complementarias:

- *Escalabilidad de tamaño*: Un sistema distribuido ofrece este tipo de escalabilidad cuando se permite que el sistema crezca tanto en el número de procesos y de ordenadores utilizados para componer el sistema como en el de usuarios atendidos sin afectar a su funcionamiento. Es decir, sin modificar la calidad de servicio obtenida por sus usuarios.
- *Escalabilidad de distancia*: Un sistema distribuido proporciona esta escalabilidad cuando sus componentes pueden ser ubicados en lugares físicamente distantes entre sí. Esta distancia puede ser arbitrariamente grande. Por ejemplo, los sistemas informáticos utilizados por corporaciones internacionales se organizan como sistemas distribuidos y sus ordenadores suelen estar ubicados en diferentes países colaborando en la gestión de la información manejada por tales empresas.
- *Escalabilidad administrativa*: Los sistemas que proporcionen este tipo de escalabilidad estarán formados por múltiples organizaciones y cada una de

ellas tendrá sus propios administradores de los nodos que compongan el sistema informático. Se dará escalabilidad administrativa cuando las tareas administrativas globales resulten sencillas, a pesar de afectar a múltiples organizaciones diferentes. Todos estos subsistemas (entendiendo como tales los conjuntos de ordenadores pertenecientes a cada organización) colaborarán entre sí para proporcionar cierta funcionalidad global. Los *sistemas grid* que se describen en detalle en la sección 11.3.2 son un ejemplo de sistema de este tipo.

En las próximas secciones se explicará con mayor detenimiento cada uno de estos tipos de escalabilidad.

Escalabilidad de tamaño

Como se ha mencionado anteriormente, la escalabilidad de tamaño se da cuando se permite que el sistema crezca tanto en el número de procesos y de ordenadores utilizados para componer el sistema como en el de usuarios atendidos sin afectar a su funcionamiento. Este tipo de escalabilidad se ve amenazada cuando se adoptan estrategias centralizadas para gestionar los servicios, los datos o los algoritmos empleados en un sistema distribuido. Por ejemplo:

- Se tendría un servicio centralizado cuando un solo ordenador fuera el responsable de atender a todos los usuarios de un determinado sistema distribuido. Si el sistema escalara, el número de usuarios iría creciendo progresivamente y llegaría un momento en el que la cantidad de peticiones recibidas por unidad de tiempo desbordaría la capacidad de cómputo de ese ordenador central. Por tanto, un servicio centralizado no puede clasificarse como escalable.

- Se habla de "datos centralizados" cuando el conjunto de información que deba manejar una aplicación distribuida esté contenida en un único almacén. Por ejemplo, si existiera un servicio capaz de mantener y servir las peticiones de visualización o descarga para todas las películas que se hayan llegado a rodar en cualquier estudio cinematográfico, el volumen de información que tendría que mantener sería enorme. Difícilmente podría soportar todas las peticiones que llegasen a efectuar simultáneamente los millones de potenciales usuarios de tal servicio.

- Se dice que un algoritmo es centralizado cuando alguno de los nodos que lo ejecute tenga que recoger toda la información global que maneje tal algoritmo para tomar alguna decisión que condicione su progreso. El resto de nodos que ejecutasen el algoritmo permanecerían esperando a que dicho nodo coordinador tomara su decisión y la propagara a todos los demás. A su vez, el nodo coordinador necesitaría un intervalo prolongado para recolectar toda la información necesaria para tomar su decisión. En algunos casos (por ejemplo,

cuando haya problemas transitorios en alguno de los enlaces de comunicación o cuando alguno de los nodos participantes falle) dicha recolección y la posterior decisión podrían retrasarse indefinidamente.

Por tanto, lo que debe buscarse es precisamente lo contrario: la **descentralización** de servicios, datos y algoritmos. Para ello suelen utilizarse tres técnicas complementarias:

- *División de tareas y datos.* El objetivo es dividir todas las tareas que deba desarrollar una determinada aplicación distribuida de manera que se reparta el esfuerzo que requiera cada una entre diferentes nodos del sistema. Los datos que deban utilizarse para realizar tales tareas también podrían repartirse, de manera que el nodo que realice cada tarea o grupo de tareas únicamente deba mantener y manejar aquellos datos necesarios para ejecutarlas pero no el resto.

  Una primera aproximación hacia este objetivo consiste en proporcionar a cada usuario (e instalar en la máquina que esté utilizando) aquella parte de la aplicación que deba utilizar directamente, dejando el resto en otros servidores centrales. De esta manera el usuario percibiría un tiempo de respuesta breve en caso de que la mayor parte de las tareas que genere puedan ser atendidas por completo por los módulos locales de la aplicación.

  Las aplicaciones distribuidas suelen seguir un modelo de interacción cliente/-servidor (descrito en la unidad 11, sección 11.3.1). Lo que se propone en esta estrategia de división de responsabilidades es la ubicación de tantos módulos de la aplicación como sea posible en los nodos directamente utilizados por los usuarios (adoptando el rol de clientes) de manera que en los nodos que adopten el rol de servidores solo se realice una mínima parte del trabajo generado en cada petición. De este modo, no habrá dificultades para gestionar un número potencialmente alto de usuarios, pues cada uno de ellos aportaría capacidad de cómputo al sistema, a través de las máquinas que los propios usuarios posean. Esta estrategia también ofrecería un buen resultado por lo que respecta a la gestión de los datos, pues las versiones de éstos que se vayan generando durante el procesamiento se mantendrán de manera temporal en los nodos de los usuarios para después dejar sus versiones finales en los ficheros o bases de datos manejados por los nodos servidores, garantizando su persistencia. Obsérvese que con este tipo de procesamiento se consigue también dividir la responsabilidad en la gestión de los datos.

  Para que la gestión de datos sea escalable se requiere una gestión descentralizada. El servicio DNS, encargado de la resolución de nombres (y explicado en la unidad 10, sección 10.4), es otro ejemplo ilustrativo. La misión de los servidores DNS es almacenar el directorio de nombres y direcciones de las máquinas que compongan cierta zona del espacio de nombres. Para ello, el

espacio de nombres de dominio se estructura de manera jerárquica, definiendo un árbol cuyos nodos mantienen las tablas con la correspondencia entre nombres de máquinas y sus direcciones IP asociadas. Así, cada servidor llega a ser responsable de una tabla con un número moderado de entradas y las peticiones relacionadas con dicho conjunto pueden ser atendidas fácilmente por cada nodo servidor. Las peticiones que no pueden ser contestadas con la información local son redirigidas a otros servidores de nombres. Para ello, todos disponen de un algoritmo que determina a qué nodo y de qué manera se debe propagar la petición. Como resultado, el trabajo que comporta este servicio de nombres puede dividirse entre un número alto de servidores y éstos, de manera colaborativa, pueden soportar un alto número de usuarios de sus servicios.

La clave para conseguir esta división de tareas y datos entre múltiples nodos reside en el uso de algoritmos descentralizados. Se habla de ***algoritmo descentralizado*** cuando éste cumpla las cuatro propiedades siguientes:

1. *Ningún nodo mantiene toda la información que necesite el algoritmo.*
   Si alguno de los nodos llegara a tener toda la información sería capaz de centralizar la gestión del algoritmo, imponiendo sus decisiones al resto de los nodos. En ese caso se suele hablar de un nodo *coordinador* que dirige al resto de los nodos y como resultado se obtiene un *algoritmo centralizado*. Por el contrario, cuando la información está repartida entre todos los participantes y cada uno de ellos ejecuta exactamente el mismo algoritmo adoptando todos ellos un mismo rol, se habla de un *algoritmo simétrico*. En ese caso, cada nodo únicamente manejará cierta información parcial, pero no completa.
2. *Los nodos únicamente toman decisiones en base a su información local.*
   Si se cumple la propiedad anterior, todos los nodos que participen en el algoritmo mantendrán información parcial. Por tanto, esta segunda propiedad implica que tal información parcial será suficiente para decidir qué acciones tendrá que aplicar el algoritmo. Esto no implica que los diferentes participantes no intercambien información. De hecho, tendrán que hacerlo pues no son actividades independientes sino que colaboran entre sí para asegurar el buen funcionamiento del algoritmo y del sistema resultante. Lo único que se requiere en esta segunda propiedad es que cada nodo pueda progresar con la información que tenga disponible y que tal progreso no se vea amenazado por la indisponibilidad momentánea de alguno de los participantes.
3. *El fallo de un nodo no impide que el algoritmo progrese.*
   Esta propiedad complementa a la anterior y reafirma lo que allí se ha comentado: aunque alguno de los nodos participantes en el algoritmo falle, el resto de los participantes podrá continuar. Todos ellos seguirán manteniendo cierta información local suficiente para tomar decisiones y garantizar el progreso en la ejecución del algoritmo.

4. *No se asume la existencia de ningún reloj físico global.*

   Si se llegara a asumir un reloj global, algunas acciones del algoritmo podrían tener en cuenta el valor de tal reloj para tomar alguna decisión y eso podría ser peligroso. Como ejemplo, se podría anotar cuándo se recibió por última vez un mensaje desde cada nodo, sabiendo que el algoritmo fuerza a que todos los nodos se intercomuniquen cada cierto tiempo. Con ello, si un nodo A llevara cierto intervalo sin recibir ningún mensaje enviado desde otro nodo B, podría asumir que B ha fallado. Como se explicará en la unidad 9, resulta imposible construir un reloj distribuido cuya lectura esté sincronizada en todos los nodos. Por ello, asumir su existencia puede conducir a tomar decisiones erróneas. Además, las decisiones que se tomen basadas en el transcurso del tiempo suelen estar también condicionadas por la fiabilidad de los canales de comunicación. Jamás se podrá saber con certeza si el hecho de que no sea haya entregado un mensaje está causado por un problema en su nodo emisor o por un problema en el canal de comunicación. Por tanto, no es conveniente que se asuma que existe un reloj global ni que se tome tal reloj como base para tomar decisiones relacionadas con la sincronización de actividades.

Como ejemplo de un algoritmo descentralizado, se podría utilizar el siguiente, destinado a seleccionar un nodo líder en un sistema distribuido. Para ello se asume que los nodos utilizan una red de interconexión con topología de bus en la que resultará sencillo difundir un mensaje. También se asume que cada nodo conoce su propio identificador y que todos los nodos se organizan lógicamente en una topología de anillo. Todos ellos sabrán que el criterio para seleccionar al líder es elegir a aquel nodo con identificador más alto. El algoritmo seguiría estos pasos:

1. El nodo iniciador prepara un *mensaje ELECCIÓN* en el que incluye su propio identificador en un campo "iniciador". Habrá un segundo campo que mantendrá el identificador del proceso más alto visto hasta el momento (que también mantendrá el identificador del iniciador en este primer envío). Es decir, el líder temporal. El iniciador enviará dicho mensaje al siguiente proceso dentro del anillo, esperando su confirmación (es decir, que éste le retorne un mensaje ACK indicando que ha recibido correctamente el mensaje).

   Si la confirmación no llegara en el tiempo previsto (utilizando para ello el reloj local de quien haya enviado el mensaje), se reenvía de nuevo el mensaje al siguiente proceso del anillo. Esto se hará sucesivamente, hasta que alguno de los sucesores en el orden del anillo responda. Obsérvese que esto constituye el mecanismo utilizado por este algoritmo para detectar los fallos de los participantes.

2. Cada proceso que reciba y confirme el mensaje, verificará si el campo iniciador contiene su identificador. De no ser así, leerá el segundo campo

y comparará su valor con su propio identificador. Si el identificador propio fuera superior, reemplazaría al que hubiera hasta ese momento. Una vez hecho esto, el nodo reenvía el mensaje al siguiente participante siguiendo el procedimiento explicado en el paso anterior.

3. Cuando el proceso iniciador reciba de nuevo el mensaje ELECCIÓN, el valor del segundo campo proporcionará el identificador del líder elegido por el algoritmo. El iniciador construirá un *mensaje COORDINADOR* que contendrá el identificador del nuevo líder y difundirá dicho mensaje en el bus físico, para que todos los procesos activos sepan qué proceso ha sido elegido.

Obsérvese que este algoritmo respeta las cuatro propiedades exigidas y, por tanto, puede clasificarse como descentralizado:

1. Ningún nodo conoce toda la información que debería gestionar el algoritmo. Cada cual se conforma con conocer su identificador local, la identidad del líder elegido hasta el momento en que cada uno reciba el mensaje ELECCIÓN y la identidad del siguiente nodo al que tendrá que propagar el mensaje. La información completa sería el conjunto de todos los identificadores de nodo existentes en el sistema y nadie tiene tal información.
2. Cada nodo decide por sí mismo si debe modificar o no la identidad del líder temporal. Para ello compara su propio identificador con el que está contenido en el segundo campo del mensaje ELECCIÓN.
3. El fallo de algún nodo no impide el progreso del algoritmo. Para ello se utiliza un temporizador local y los mensajes ACK que reconocen la entrega del mensaje ELECCIÓN.
4. No se necesita ningún reloj global. En el único paso en que se necesita gestionar el tiempo es en los reenvíos del mensaje ELECCIÓN en caso de que haya habido algún fallo. Para ello basta con consultar el reloj local del emisor. No se necesita que los relojes de todos los participantes estén sincronizados.

- *Replicación.* La segunda técnica que podrá utilizarse para incrementar la escalabilidad de distancia consiste en replicar tanto los datos gestionados por las aplicaciones distribuidas como los procesos responsables de gestionar tales datos. Al tener la información replicada en múltiples nodos cada usuario tendrá alguna réplica en un nodo relativamente cercano. Por tanto, si interactúa con la réplica más cercana conseguirá reducir el retardo en la propagación de los mensajes de petición y respuesta que tendrá que intercambiar con el componente que gestione tales datos.

  Si la mayoría de los accesos que realizan los usuarios únicamente consultan el valor actual de la información mantenida, bastará con acceder a una única réplica. Gracias a ello, las peticiones realizadas por múltiples usuarios

podrán ser atendidas a la vez en varias réplicas, paralelizando su servicio y aumentando el rendimiento de la aplicación distribuida.

No todo serán ventajas al emplear técnicas de replicación. Si los accesos realizados por los usuarios implican una modificación de los datos, tales accesos tendrán que ser propagados tarde o temprano a todas las réplicas para que todas ellas estén actualizadas y mantengan un estado consistente. En tales casos hay que emplear algún protocolo de replicación [BMST92, Sch90, DSS98, Pow93] para realizar la propagación de las actualizaciones, y tales protocolos difícilmente escalan para gestionar un número elevado de réplicas. Afortunadamente, al utilizar estas aplicaciones se suele realizar un mayor número de peticiones de consulta que de actualización. Ante ese escenario, la replicación sí que incrementa el rendimiento, mejorando la escalabilidad de tamaño.

Como ya se ha mencionado anteriormente, los modelos de consistencia [Mos93] establecen qué grado de diferencia se aceptará en el estado de las réplicas de un determinado dato. Si una aplicación puede aceptar un modelo de consistencia relajado, los protocolos de replicación necesarios para propagar las actualizaciones serán sencillos y eficientes, con lo que el rendimiento de la aplicación mejorará apreciablemente al utilizar replicación. Por el contrario, cuando se deba utilizar un modelo de consistencia estricto habrá que emplear algunos mecanismos de sincronización que bloquearán temporalmente los accesos de lectura, esperando a que la última versión de cada dato haya llegado al nodo donde deba realizarse tal acceso. Por ello, en esos casos las prestaciones serán bastante peores que las obtenidas con una consistencia relajada. El modelo de consistencia a emplear dependerá de cada aplicación.

- *"Caching"*. La tercera técnica que podrá emplearse para mejorar la escalabilidad de tamaño es una variante de la segunda. El "*caching*" consiste en recordar las últimas versiones de la información accedida en cada componente de una aplicación distribuida. De esta manera, si se realiza al cabo de poco tiempo otro acceso sobre ese mismo elemento se obtendrá su valor a partir de la copia guardada localmente, sin necesidad de acceder a la instancia "original" de tal elemento. Con ello se mantienen copias relativamente recientes de cada elemento en el mismo nodo en que los usuarios han originado tales accesos.

  Las copias mantenidas en estas cachés no dejan de ser otras réplicas de la información gestionada por las aplicaciones distribuidas. También se necesitará algún criterio para decidir cuándo se "descarta" la versión almacenada en la caché y se renueva su valor realizando una lectura en la ubicación original de tal elemento. Tales criterios dependerán de la aplicación que se esté utilizando y del tipo de dato que se esté accediendo.

Escalabilidad de distancia

La mayor parte de las aplicaciones distribuidas asumen que los nodos que componen el sistema están repartidos en una misma red local. La *escalabilidad de distancia* extenderá el sistema por una amplia zona geográfica y en ese caso no se estará utilizando una única red de área local.

En esas situaciones al utilizarse múltiples redes (quizá con características bien distintas en cada una) y nodos alejados entre sí, algunas propiedades de la comunicación variarán ligeramente:

- Los retardos en la transmisión de los mensajes se incrementarán de manera apreciable. Con ello, la utilización de un mecanismo de comunicación sincrónica (a estudiar en la unidad 8, sección 8.5.1) planteará problemas pues los usuarios difícilmente aceptarían intervalos de bloqueo prolongados. En la comunicación sincrónica se necesita bloquear al emisor hasta que el protocolo de comunicaciones confirme que el mensaje ha podido ser gestionado correctamente. Desafortunadamente, las aplicaciones interactivas suelen utilizar RPC (que necesita comunicación sincrónica) para gestionar la comunicación entre sus módulos.

- La comunicación suele ser mucho menos fiable que en una red local, ya que se utilizan otros nodos para realizar el encaminamiento de los mensajes hacia sus destinos y la probabilidad de que alguno de los nodos participantes llegue a presentar problemas se incrementa de manera directamente proporcional al número de nodos utilizados.

  Esto acarreará de nuevo mayores retardos en la comunicación. En caso de que alguno de esos nodos falle, los mensajes se perderían (y habría que volverlos a enviar) si no se utilizara comunicación orientada a conexión o bien habría que buscar otras rutas de encaminamiento si se utilizaran conexiones.

Por ello, cuando se adapte una aplicación para que pueda gestionar una mayor escalabilidad de distancia deberán considerarse estos inconvenientes. Tanto la pérdida de fiabilidad como el incremento en los tiempos de transmisión de los mensajes deberán considerarse en la fase de diseño. Con ello, deben utilizarse estrategias de comunicación que resulten adecuadas para la funcionalidad que vaya a facilitarse a los usuarios. En la unidad 8 se analizarán los aspectos más relevantes de la comunicación basada en mensajes.

También debe resaltarse que la escalabilidad de distancia está fuertemente relacionada con la escalabilidad de tamaño. Tanto en uno como en otro caso resulta necesario el uso de algoritmos descentralizados. Una gestión centralizada impone fuertes restricciones tanto a la escalabilidad de tamaño como a la escalabilidad de distancia. Por ejemplo, si DNS fuera un sistema centralizado y existiera un único servidor para traducir los nombres de máquina en direcciones, deberíamos

contactar con dicho servidor hasta para averiguar el nombre de cualquiera de los ordenadores de nuestro propio sistema. Eso no funcionaría pues ese servidor se colapsaría fácilmente con la carga que debería soportar.

Existe un buen número de servicios distribuidos que asumen una amplia distribución geográfica. Un ejemplo es el servicio de correo electrónico. En él se utiliza la comunicación asincrónica (descrita en la unidad 8, sección 8.3.3) : cuando enviamos un correo electrónico nuestro programa no espera a que el mensaje correspondiente se entregue a su destinatario final; nos devuelve el control de inmediato. El servicio garantiza que los correos electrónicos lleguen finalmente a su destino, pero no se compromete a hacerlo en un plazo determinado. Generalmente esto sucede en unos pocos segundos, pero podría prolongarse si alguno de los servidores de correo que debiera participar en esta gestión hubiera fallado. A pesar de ello, una vez se recupere, el mensaje sigue con su procesamiento y será entregado tarde o temprano a su usuario destinatario. Los usuarios de este servicio están acostumbrados a este tipo de gestión y la consideran razonable. Gracias a ello, este servicio puede escalar cómodamente tanto en distancia como en tamaño.

Escalabilidad administrativa

Cuando un sistema presenta *escalabilidad administrativa* la gestión de sus componentes debe resultar siempre sencilla incluso cuando el sistema se extienda sobre múltiples organizaciones, cada una con sus propios administradores.

El principal problema que se plantea a la hora de abordar la escalabilidad administrativa tiene que ver con la gestión de la seguridad del sistema: sus mecanismos de autenticación y gestión de identidad, los mecanismos de control de acceso sobre los recursos de cada organización, etc.

La clave para resolver estas dificultades es la apertura. Los sistemas de seguridad deben ser abiertos y basados en mecanismos y protocolos estándar. Cada organización será libre de adoptar sus propias políticas de administración. Los administradores de cada organización serán los responsables de la gestión de los recursos ubicados en su organización y los usuarios, normalmente, solo conocerán a sus propios administradores y confiarán en ellos.

No obstante, si se utilizan protocolos estándar y se fijan ciertas relaciones de confianza entre las diferentes organizaciones que compongan el sistema distribuido, no resultará imposible una gestión global en la que todas las organizaciones colaborarán entre sí a la hora de asegurar sus recursos y controlar que todos los accesos realizados sobre ellos estén autorizados.

7.4 Dificultades en el desarrollo de Sistemas Distribuidos

Tras haber explicado tanto las características como los objetivos de cualquier sistema distribuido, queda claro que el desarrollo de aplicaciones distribuidas no va a ser sencillo. Hay un buen número de aspectos que deben considerarse a la hora de realizar un buen diseño. No basta con aplicar los principios recomendados para el diseño y desarrollo de aplicaciones en un solo ordenador. El hecho de que la aplicación conste de múltiples componentes y éstos puedan y deban ubicarse en diferentes máquinas, necesitando mecanismos de comunicación a través de la red complicará la gestión de estas aplicaciones.

Por regla general, quien empieza a desarrollar su primera aplicación distribuida suele aplicar las reglas utilizadas en el desarrollo de aplicaciones centralizadas (esto es, diseñadas para ser instaladas y ejecutadas en un único ordenador). Por ello, según [Tv08] se suele asumir que:

1. La red es fiable.
2. La red es segura.
3. La red es homogénea.
4. La topología no cambia.
5. El retardo de transmisión de los mensajes es cero.
6. El ancho de banda es infinito.
7. Habrá un administrador.

Obviamente, todas estas simplificaciones no serán realistas. Un entorno distribuido real no cumplirá ninguna de estas características. La red no tiene por qué ser siempre fiable (los mensajes pueden perderse o retrasarse indefinidamente), ni segura (la información transmitida puede ser vista por otras aplicaciones si no hemos tomado las debidas precauciones), ni homogénea (habrá múltiples ordenadores en nuestro sistema y cada uno puede estar utilizando un sistema operativo diferente y tener nodos con distintas características: cantidad de RAM, capacidad del disco duro, procesador,...), la topología podrá cambiar si alguno de los ordenadores del sistema deja de funcionar o algún switch o router se estropea, los mensajes tardan un tiempo variable en ser transmitidos y entregados, el ancho de banda está limitado, el administrador del sistema no siempre estará disponible para resolver los problemas que vayan surgiendo,...

Sin embargo, estas suposiciones pueden tener sentido cuando se esté buscando un primer algoritmo para resolver algún problema dentro del entorno distribuido. Lo más importante es decidir si tal problema tendrá o no solución. Para ello se puede optar por asumir todas las simplificaciones listadas arriba, proporcionando

el entorno más favorable posible. Si así se puede encontrar un primer algoritmo, ya se habrá avanzado un poco. En caso contrario, se podría demostrar que ese problema no tiene solución en un entorno distribuido [FR03].

En caso de que haya solución, posteriormente se irá eliminando cada una de estas suposiciones y se irán incorporando progresivamente las características del sistema real en el que deba aplicarse la solución. Todo esto debe hacerse todavía en la fase de diseño, antes de iniciar su desarrollo. Con ello se podrá refinar progresivamente esa primera solución hasta adaptarla al entorno físico en el que debía implantarse nuestra aplicación.

7.5 Resumen

Un sistema distribuido consiste en un conjunto de ordenadores independientes que ofrece a sus usuarios la imagen de un sistema único y coherente. Por tanto, se *oculta* a los usuarios las diferencias entre las máquinas, así como la complejidad de los mecanismos de comunicación entre ellas. De este modo los usuarios acceden a los sistemas distribuidos de forma *homogénea*, sea cual sea el lugar desde el que lo hagan. Asimismo, los servicios ofrecidos por un sistema distribuido deberían estar siempre *disponibles*. Finalmente, no debería resultar complicado incrementar su *escalabilidad*, es decir, incorporar más ordenadores al sistema, para así atender a un mayor número de usuarios.

A la hora de diseñar e implementar un sistema distribuido, se deben tener en cuenta cuatro objetivos fundamentales: (i) facilitar el acceso de los usuarios a los recursos remotos (como impresoras, discos duros, máquinas, etc.) para así economizar y facilitar la compartición de la información; (ii) proporcionar transparencia de distribución, ocultando el hecho de que los procesos y recursos están físicamente distribuidos sobre diferentes ordenadores; (iii) concebir el sistema como un sistema abierto, de modo que se ajuste a estándares que describan la sintaxis y semántica de los servicios, permitiendo así configurar el sistema combinando módulos distintos, añadiendo nuevos módulos o reemplazando los existentes, sin afectar al resto del sistema; y (iv) ofrecer escalabilidad, de modo que el aumento de la demanda de servicios se pueda suplir con una aportación de recursos o bien se degrade el tiempo medio de respuesta sin llegar a colapsarse.

En esta unidad se ha presentado el concepto de sistema distribuido, sus características más relevantes y cómo abordar cada uno de los cuatro objetivos importantes que deben cumplirse para que la construcción de un sistema distribuido sea efectiva. En las próximas unidades se describirán algunos conceptos, servicios, problemas y soluciones que tienen sentido en los sistemas distribuidos.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar las ventajas y problemas que comporta el desarrollo de aplicaciones informáticas en un entorno distribuido.
- Caracterizar los sistemas distribuidos y describir los objetivos que deben cumplir.
- Distinguir los diferentes tipos de transparencia que debería facilitar cualquier aplicación distribuida.
- Identificar los diferentes mecanismos necesarios para obtener transparencia en una aplicación distribuida.
- Distinguir las diferentes perspectivas de escalabilidad de un sistema distribuido.
- Caracterizar los algoritmos descentralizados. Identificar sus ventajas y las dificultades existentes para desarrollarlos.

Unidad 8

COMUNICACIONES

8.1 Introducción

Como ya hemos visto en la Unidad 7, todo sistema distribuido estará compuesto por una serie de nodos que colaboren entre sí ofreciendo una imagen de sistema único y coherente. El hecho de que múltiples procesos deban colaborar implica que tendrán que intercomunicarse. Para ello existen dos opciones básicas: utilizar memoria compartida o bien intercambio de mensajes.

Los mecanismos necesarios para sincronizar adecuadamente la ejecución de múltiples tareas en base a memoria compartida han sido tratados en las unidades previas. En esta unidad nos centraremos en algunos aspectos de la comunicación basada en mensajes. Para ello, la sección 8.2 describe las ventajas que aporta una arquitectura de comunicaciones estructurada en múltiples niveles. La sección 8.3 presenta el mecanismo de llamada a procedimiento remoto (RPC), que proporciona transparencia de ubicación a la hora de acceder a los subprogramas de una aplicación distribuida en lenguajes de programación de alto nivel. La invocación a objeto remoto puede considerarse una evolución de RPC para sistemas que adopten un modelo de programación orientado a objetos. El mecanismo de comunicación resultante se describe en la sección 8.4. Por último, la sección 8.5 analiza la comunicación basada en mensajes considerando su grado de persistencia y sincronía.

8.2 Arquitectura en niveles (TCP/IP)

Para gestionar las comunicaciones a través de una red resulta conveniente desarrollar una pila de protocolos, definiendo una arquitectura en niveles. De esta manera, cada nivel utiliza un protocolo determinado y así se preocupa por solucionar alguno de los problemas que la comunicación conlleva. Una primera arquitectura de este tipo fue el modelo OSI ("*Open Systems Interconnection*") [Zim80], definido como un estándar ISO ("*International Standardization Organization*") para estructurar las comunicaciones entre ordenadores.

En una arquitectura estructurada en niveles, cada nivel se responsabiliza de unas tareas determinadas y proporciona una interfaz bien definida a su nivel superior, asumiendo a su vez lo mismo de su nivel inmediatamente inferior. Desafortunadamente, la arquitectura OSI no tuvo un amplio seguimiento. En lugar de los siete niveles presentes en la arquitectura OSI, la mayoría de los sistemas de comunicaciones han optado por la utilización de la arquitectura TCP/IP, centrada en cinco de ellos. Esos niveles son:

1. *Nivel físico*: Determina el tipo de hardware a emplear, su interfaz y cómo llega a transmitir bits a través de un "enlace". Sólo participa el hardware.

2. *Nivel de enlace*: Se preocupa por la transmisión de información entre dos máquinas entre las que físicamente solo existe un "enlace" que permita comunicarlas. Para ello, los mensajes se dividen en tramas y este nivel se preocupa por la adecuada gestión de estos elementos. Mediante CRC pueden detectarse los errores en la transmisión y repetir el envío de una trama hasta que ésta llegue correctamente.

3. *Nivel de red*: Se preocupa del encaminamiento de los mensajes. Es decir, necesita establecer una ruta entre dos nodos que no están directamente interconectados. Se asume que habrá más de dos nodos en el sistema. Ya debe intervenir el "software" (es decir, el sistema operativo). El protocolo comúnmente utilizado para implantar este soporte es *IP* ("*Internet Protocol*") [Pos81a], del que existen dos variantes: IPv4 e IPv6 [NBBB98].

4. *Nivel de transporte*. Gestiona la fiabilidad en la entrega de los mensajes, así como su ordenación. También establece y gestiona las conexiones. Puede proporcionar una interfaz de *comunicación fiable* (es decir, que garantice la entrega de los mensajes), pero donde todavía habrá que resolver algunos detalles que dependerán de la aplicación y que gestionarán los niveles superiores (como pueda ser la sincronización y otros detalles de presentación). Los dos protocolos más utilizados en este nivel son *TCP* ("*Transmission Control Protocol*") [Pos81b, DBEB06] y *UDP* ("*User Datagram Protocol*") [Pos80]. UDP es un protocolo ligero que no garantiza la entrega de los mensajes ni la ausencia de duplicados. Por su parte, TCP es un protocolo bastante más pesado que garantiza la entrega ordenada de los mensajes.

5. *Nivel de aplicación.* Gestión de los protocolos propios de cada aplicación. Ejemplos: FTP para la transmisión de ficheros, HTTP para acceder a páginas web, etc.

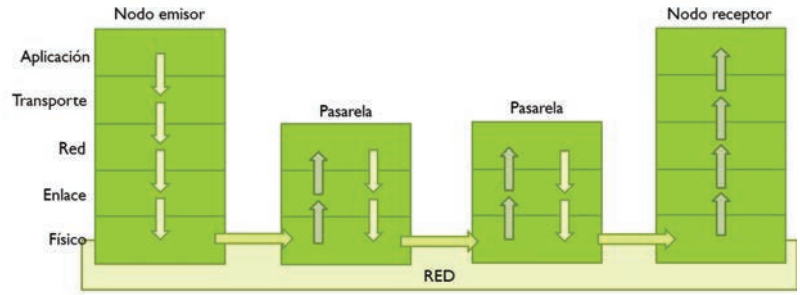

Figura 8.1: Propagación de mensajes (Arquitectura TCP/IP).

La figura 8.1 ilustra cómo intervienen los distintos niveles de la arquitectura TCP/IP a la hora de propagar un mensaje entre dos procesos. En este ejemplo se llegan a utilizar algunos nodos intermedios para las tareas de encaminamiento. Como puede observarse, en dichos nodos solo intervienen los tres niveles inferiores de la arquitectura (físico, enlace y red) pues en el tercero se encuentra la gestión del encaminamiento. Tanto en el nodo emisor como en el receptor, intervendrán todos los niveles.

En la práctica, como ya hemos comentado previamente, cada nivel de la arquitectura TCP/IP ofrece una interfaz bien definida a su nivel inmediatamente superior. La comunicación entre un proceso emisor y otro receptor recorrerá todos los niveles de sus respectivos nodos. Sin embargo, cada nivel utiliza su propio *protocolo*; es decir, su propio conjunto de reglas y procedimientos para gestionar la comunicación. Por ello, en los niveles más altos no resulta visible que hayan intervenido otros nodos intermedios para lograr esta comunicación. Para implantar cada protocolo nos apoyamos en la funcionalidad proporcionada por su nivel inferior, y éstos irán ocultando esos detalles. La figura 8.2 ilustra la ubicación de los distintos protocolos.

Para proporcionar la transparencia necesaria en todo sistema distribuido se suele implantar un middleware [Ber96] entre los niveles de transporte y de aplicación. Este middleware proporciona normalmente una imagen de sistema único a las aplicaciones que se deban diseñar, implantar y utilizar sobre él. Para ello se suele recurrir a protocolos estándar que garanticen la interoperabilidad de los componentes que compongan dicho middleware, aunque los sistemas subyacentes (los que haya en cada uno de los nodos) sean diferentes.

La figura 8.3 refleja qué lugar ocuparía un middleware al implantarlo dentro de una arquitectura de comunicaciones.

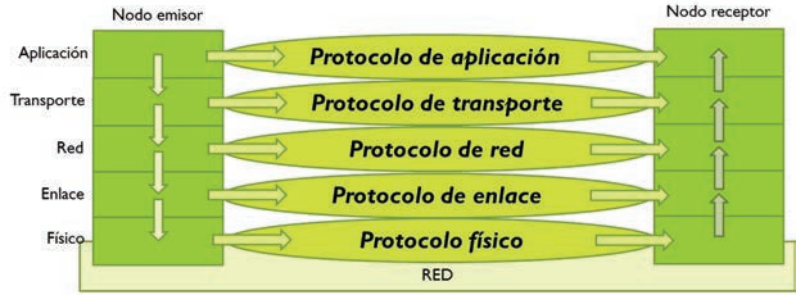

Figura 8.2: Visión basada en protocolos de la arquitectura TCP/IP.

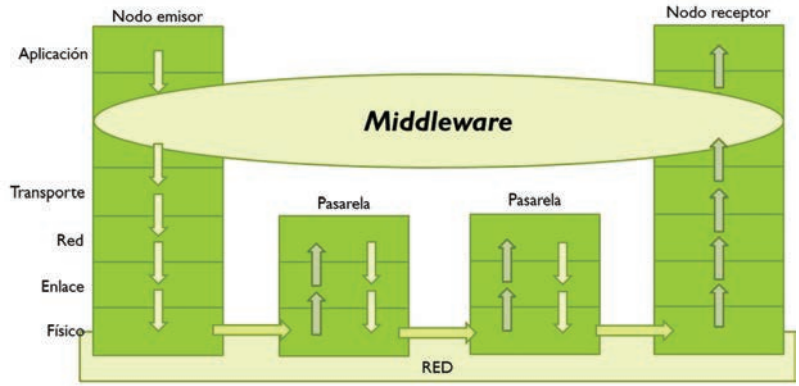

Figura 8.3: Ubicación del middleware en una arquitectura de comunicaciones.

8.3 Llamada a procedimiento remoto (RPC)

Aunque la comunicación entre dos ordenadores distintos deba realizarse siempre a través de una red utilizando mensajes, el modelo proporcionado por una arquitectura estructurada en niveles (como la TCP/IP) ya sugiere que no todos los detalles de la comunicación deben resultar visibles a las aplicaciones. Por tanto, en cierto nivel de nuestra arquitectura distribuida los mensajes "desaparecerán" y utilizaremos ciertos mecanismos de mayor nivel de abstracción. Uno de estos mecanismos es la *llamada a procedimiento remoto* [Nel81, BN84] (o RPC, por sus siglas en inglés: *remote procedure call*), que facilita una imagen local cuando se realiza una invocación a un procedimiento remoto.

Cuando desarrollamos alguna aplicación informática, el diseño estructurado recomienda que ésta se organice en una serie de *módulos* o *procedimientos* (también llamados subprogramas, subrutinas, ...). Cada procedimiento implanta una tarea determinada y es independiente del resto del código de la aplicación. Con un diseño

cuidadoso, los procedimientos serán fácilmente reutilizables en otras aplicaciones donde la misma tarea resulte necesaria.

Los procedimientos tienen una *signatura* determinada, donde se especifica cuáles son sus argumentos de entrada y de salida. Con ello actúan como una caja negra, ocultando cómo están implantados y permitiendo que su código pueda modificarse para eliminar errores u optimizar su rendimiento. Para invocar un procedimiento basta con conocer su identificador y facilitar los argumentos que espera.

El mecanismo concreto que se utiliza para pasar los argumentos desde el procedimiento invocador al procedimiento invocado dependerá de la arquitectura del procesador (se suele utilizar la pila, pero también resulta posible utilizar algunos registros generales cuando el procedimiento invocado requiera pocos argumentos, o bien otros mecanismos alternativos como las ventanas de registros de los antiguos procesadores Sun SPARC). El programador no tiene por qué conocer qué mecanismo concreto se está utilizando. Tomando esta idea como base, Bruce J. Nelson [Nel81] diseñó un mecanismo para invocar de manera transparente a un procedimiento remoto; esto es, a un procedimiento que no resida en el mismo proceso y que no tenga por qué residir siquiera en la misma máquina. Este mecanismo pasó a conocerse como *llamada a procedimiento remoto* (RPC).

Los sistemas operativos actuales utilizan diferentes variantes de la RPC. Por ejemplo, en los sistemas Windows se utiliza RPC para implantar la comunicación entre servidores de su Directorio Activo [RKMW08], así como en su API *DCOM* ("*Distributed Component Object Model*")[RSI09]. A su vez, el propio sistema operativo está organizado con una arquitectura basada en *micronúcleo* y la comunicación entre sus distintos componentes se realiza mediante una versión optimizada de la RPC para entornos locales [RSI09]. En este último caso se ofrece la imagen de que el procedimiento a invocar forma parte del propio proceso cuando realmente está soportado por un proceso diferente de la misma máquina.

Para implantar una llamada a procedimiento remoto aprovecharemos el principio de *ocultación* [Par72] o *encapsulación*: el programador que utilice un procedimiento debe conocer su interfaz pero no necesita conocer cómo está implantado. Por tanto, facilitaremos un procedimiento local (llamado *stub cliente*) para el invocador que proporcionará su misma interfaz pero que no implantará localmente su tarea. En lugar de eso, su misión consiste en recoger todos los argumentos de entrada, empaquetarlos en un formato neutro y hacerlos llegar en un *mensaje de petición* al proceso servidor que implante realmente el procedimiento a invocar. Posteriormente esperará un *mensaje de respuesta* desde el servidor y, una vez recibido, desempaquetará sus argumentos de salida (si los hubiere) y los retornará, junto al control de la ejecución al procedimiento invocador. Como puede verse, hemos sustituido el hipotético procedimiento a invocar por un stub cliente que oculta todos los detalles de la invocación remota, haciendo creer al procedimiento invocador que la llamada se ha realizado sobre un procedimiento local "normal".

Esta secuencia que acabamos de describir en el párrafo anterior ha obviado un buen número de pasos intermedios que explicaremos posteriormente. No obstante, sí que deja claro que el programador del procedimiento invocador no necesita saber nada acerca del mecanismo de una RPC. La ubicación del procedimiento que va a invocar resulta transparente para él. Esto se ha conseguido gracias al stub cliente.

Ocurrirá algo similar en el proceso servidor. Nuestro objetivo sería mantener ese grado de transparencia también para el programador del procedimiento remoto. Para ello, los mensajes de petición y respuesta que emitía y recibía el stub cliente no son gestionados directamente por el procedimiento invocado en el proceso servidor. En su lugar, habrá un *stub servidor* que gestionará dichos mensajes y actuará como el invocador del procedimiento servidor. Así, ese procedimiento servidor podrá mantener su interfaz y funcionalidad iniciales y será el segundo stub (es decir, el stub servidor) quien se encargará de todos los detalles de intercomunicación remota en el proceso servidor. Como resultado, tenemos una transparencia de ubicación completa a la hora de implantar el código de ambos procedimientos: tanto en el invocador como en el invocado.

Además, todo esto resulta sencillo pues existen compiladores de interfaces que son capaces de generar automáticamente los módulos que actúan como stubs cliente y servidor. Para ello basta con especificar las interfaces con un determinado lenguaje (conocido normalmente como *IDL* o "*interface definition language*"; lenguaje de definición de interfaces) y compilarlas con dicha herramienta. En dicha especificación deberá indicarse la signatura del procedimiento así como una indicación acerca del sentido de paso de cada argumento (entrada o salida).

8.3.1 Pasos en una RPC

Tras haber presentado una visión general del mecanismo de llamada remota a procedimiento estudiaremos cómo se desarrolla una invocación. Para ello describiremos la secuencia de pasos que debe seguirse y que se muestra gráficamente en la figura 8.4. Son los siguientes:

1. El proceso cliente invoca al procedimiento. En su propio programa existirá un procedimiento local que ofrecerá una interfaz idéntica, pero que no mantiene el código necesario para realizar esa tarea, sino para realizar la comunicación con el servidor. Es el *stub cliente*.

2. El stub cliente recoge los argumentos de entrada de esa llamada y los empaqueta en un mensaje de petición. Esa tarea de empaquetado se conoce como "*marshalling*". Consiste en incluir una identificación del tipo de datos y el valor correspondiente en una codificación neutra fácilmente interpretable por el receptor (independiente de la arquitectura).

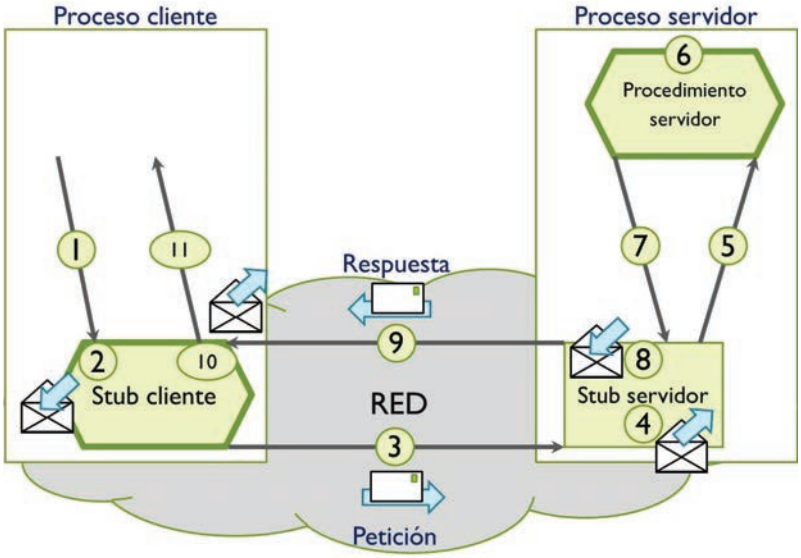

Figura 8.4: Pasos en una llamada a procedimiento remoto.

3. Tras el "marshalling" ya tendremos un mensaje de petición construido. En este tercer paso se envía al proceso servidor. Si dicho proceso no está en la máquina local, habrá que emplear comunicación a través de la red.

   Obsérvese que el proceso cliente (o el hilo que se haya utilizado para realizar el envío) se bloquea en este punto, hasta que llegue el mensaje de respuesta. Eso ocurrirá en el paso 9.

4. El mensaje de petición llega a un *stub servidor*. Éste lo desempaqueta (tarea de "*unmarshalling*") para extraer los argumentos de entrada del procedimiento que debe invocarse.

5. Con esta información se realiza la llamada propiamente dicha.

6. El procedimiento servidor es ejecutado.

7. Se retorna el control y los argumentos de salida al stub servidor.

8. El stub servidor empaqueta los argumentos de salida y el resultado para construir un mensaje de respuesta.

9. Se transmite el *mensaje de respuesta* al stub del proceso cliente. Esto reactiva al (hilo emisor del) proceso cliente.

10. El stub cliente desempaqueta los argumentos de salida y el resultado.

11. Con esa información devuelve el control al procedimiento invocador, terminando así la llamada.

En algunos de los pasos anteriores surgen algunas dificultades que conviene comentar. Por ejemplo, en el paso 3 el stub cliente debe conocer dónde se encuentra el stub servidor para hacerle llegar el mensaje de petición. Para ello, el proceso servidor debería tener un nombre registrado en un *servidor de nombres*. Con ello, en su inicialización o en la primera invocación el stub cliente debería traducir (resolver) ese nombre por su dirección asociada y recordaría esa dirección para futuras invocaciones. La gestión de los nombres de recursos en un sistema distribuido y de las operaciones de resolución se estudiará con detenimiento en la unidad 10.

Por otra parte, cada proceso servidor podría tener más de un procedimiento público que podrían ser invocados por multitud de clientes. El esquema que aparece en la figura 8.4 refleja el caso en que solo hay un procedimiento público. Cuando haya más de uno, este esquema debe complementarse de la siguiente manera:

- En el proceso servidor, además de tener un stub servidor por cada procedimiento público, debería haber una primera rutina gestora que se encargará de recibir los mensajes de petición y redirigirlos al stub servidor adecuado, en función de un identificador de operación.
- En cada stub cliente, además de conocer la dirección del proceso servidor, se tendrá que mantener el identificador del procedimiento que representa y tal identificador se incluirá en los mensajes de petición. De esta manera la rutina gestora del proceso servidor sabrá qué procedimiento tendrá que invocarse.

  Recuérdese que hay un stub cliente por cada uno de los procedimientos presentes en la interfaz del proceso servidor.

8.3.2 Paso de argumentos

En la descripción de la sección anterior se ha asumido que se realizaba un *paso de argumentos por valor*. Es decir, el procedimiento invocado recibe una copia del valor actual de la variable que se suministra como argumento. Cualquier modificación realizada en el procedimiento invocado alterará esa copia, pero no la variable original. Es lo habitual en una llamada a procedimiento remoto.

Un segundo tipo de paso de argumentos en los lenguajes de programación de alto nivel se realiza por referencia. En el *paso por referencia* se facilita al procedimiento invocado la posición de la variable y no se necesita copiar dicha variable. Con ello, cualquier modificación realizada en el procedimiento invocado resultará visible a todos los demás procedimientos en los que pueda accederse a esa misma variable. Obsérvese que eso es factible debido a que todos esos procedimientos comparten un mismo *espacio de direcciones*: el del proceso que los mantiene. Sin embargo,

en una llamada a procedimiento remoto el procedimiento invocador y el invocado residirán en diferentes procesos y éstos no compartirán memoria. Así, pasar la dirección de la variable no tendrá ningún sentido y no resultará factible el paso por referencia.

Para simular el paso por referencia se podría indicar que un determinado argumento debe pasarse en los dos sentidos: tanto en el de entrada como en el de salida. Esto implicará que se realice una copia del valor del argumento para construir el mensaje de petición y que vuelva a realizarse otra tras recibir el mensaje de respuesta. Es decir, no mantendrá la semántica de un paso por referencia y esto podría ocasionar problemas. Por ejemplo, podría haber otras actualizaciones concurrentes realizadas por otros hilos locales que se perderían al terminar la llamada a procedimiento remoto. Además, el paso de argumentos de tipos complejos (por ejemplo, colas, listas o árboles implantados en memoria dinámica) será imposible.

8.3.3 Tipos de RPC

Existen tres variantes de llamada a procedimiento remoto, en función del grado de sincronización que exigen. La que hemos descrito hasta el momento es la variante *convencional*, ilustrada en la figura 8.5. En ella se observa que el proceso cliente queda suspendido tras haber realizado el envío del mensaje de petición. Dicha suspensión concluye al recibir el mensaje de respuesta.

Figura 8.5: RPC convencional.

A su vez, el servidor permanece normalmente suspendido esperando la llegada de algún mensaje de petición. Esa suspensión se representa en la figura 8.5 mediante una línea discontinua. Tras recibir ese mensaje pasará a activarse y a ejecutar el procedimiento solicitado (línea continua). Cuando finaliza la ejecución de ese subprograma, se envía un mensaje de respuesta al cliente y el servidor pasa a esperar la recepción de una nueva petición (representado mediante otra línea discontinua).

En la variante *asincrónica* [SM86] (véase la figura 8.6) el proceso servidor retorna un mensaje de confirmación tan pronto como haya recibido el mensaje de petición.

De esta manera se reduce el intervalo de suspensión del proceso cliente y tanto cliente como servidor pueden avanzar paralelamente. Sin embargo, el uso de esta variante no permite que el procedimiento tenga argumentos de salida ni valores de retorno.

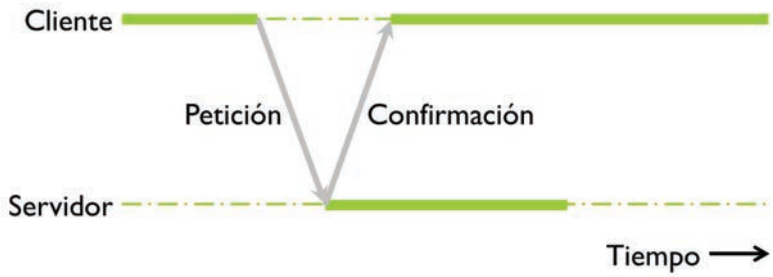

Figura 8.6: RPC asincrónica.

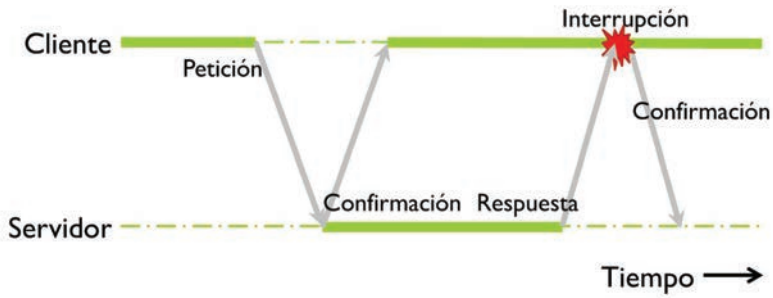

Figura 8.7: RPC asincrónica diferida.

Una tercera variante, conocida como *RPC asincrónica diferida* [LS88](figura 8.7), permite combinar las mejores características de las dos variantes anteriores. Toma como base una RPC asincrónica y así garantiza que el cliente se reactivará lo antes posible. El único inconveniente de la variante asincrónica era la imposibilidad de devolver argumentos de salida en un mensaje de respuesta. Para recuperar ese mensaje de respuesta se realiza una segunda RPC asincrónica, iniciada esta vez por el servidor. El único objetivo de esa segunda "llamada" es retornar los argumentos de salida. Es decir, implanta el mensaje de respuesta que se había eliminado en una RPC asincrónica (segunda variante). El proceso cliente debe disponer de algún mecanismo para recoger esos resultados (creación de un hilo auxiliar, uso de *buffers* de recepción donde depositar los resultados temporalmente, utilización de alguna llamada especial, utilizar un procedimiento de recepción que invoca el sistema operativo, señales, etc.). En la figura 8.7 se utiliza una *interrupción* del hilo de ejecución actual del cliente. Eso puede implantarse como una *señal* en un sistema

UNIX: el hilo en cuestión pasa a ejecutar una rutina asociada a ese tipo de señal para después retomar la instrucción interrumpida. En dicha rutina se obtendrían los resultados de la llamada.

8.4 Invocación a objeto remoto

La mayor parte de las *aplicaciones* actuales son *orientadas a objetos* (OO), por lo que resulta natural diseñar las aplicaciones distribuidas como una colección de objetos distribuidos. Para ello se extiende el modelo estudiado en la sección 8.3: en lugar de invocar procedimientos remotos, se invocan operaciones (métodos) sobre objetos remotos. El mecanismo de invocación resultante recibe el nombre de *invocación a objeto remoto* (o "*ROI*", que es el acrónimo inglés para "*remote object invocation*").

Según el paradigma OO, un *objeto*:

- Encapsula datos y operaciones (los clientes únicamente acceden a su interfaz).
- Implementa las operaciones mediante *métodos*, que resultan accesibles a través de mensajes. En este contexto (cuando se describe el paradigma de programación orientada a objetos) la palabra *mensaje* significa "invocación de un método".
- Responde a las invocaciones efectuadas desde otros módulos clientes.

Dicha definición permite invocar las operaciones de un objeto tanto desde clientes locales como remotos. Definimos un *objeto remoto* como un objeto capaz de recibir invocaciones desde otros espacios de direcciones. Todo objeto remoto se instancia en algún servidor y responde a invocaciones desde clientes locales o remotos.

Las aplicaciones se organizan como una colección dinámica de objetos que residen en nodos distintos e invocan métodos de otros objetos de forma remota. La unidad de distribución es el objeto (un objeto concreto reside totalmente en un nodo)[1]. El modelo de objetos distribuidos presenta múltiples ventajas:

- Se aprovecha la expresividad, capacidad de abstracción y flexibilidad del paradigma OO.
- La encapsulación permite obtener transparencia de ubicación, de forma que la sintaxis de invocación de los métodos de un objeto no depende del espacio de direcciones en el que reside dicho objeto. Esta propiedad permite ubicar

[1] Existen modelos de objetos distribuidos que permiten dividir los objetos en partes localizadas en distintos nodos (objetos fragmentados), pero no se abordan en este texto.

los objetos de acuerdo a distintos criterios (localidad de acceso, restricciones administrativas, seguridad, ...).

- Es posible reutilizar aplicaciones "heredadas" (legacy) encapsulándolas en objetos (utilizando el patrón de diseño "Wrapper").
- Se garantiza escalabilidad, distribuyendo los objetos sobre una red que puede acomodarse al crecimiento de la demanda.

La figura 8.8 ilustra los distintos tipos de invocación que pueden aparecer en un sistema:

Invocación local: Invocador e invocado son dos objetos que residen en el mismo proceso (A y B en la figura).

Invocación remota: Invocador e invocado son dos objetos que residen en procesos distintos, ya sea en el mismo nodo (D, E) o en nodos distintos (C, D).

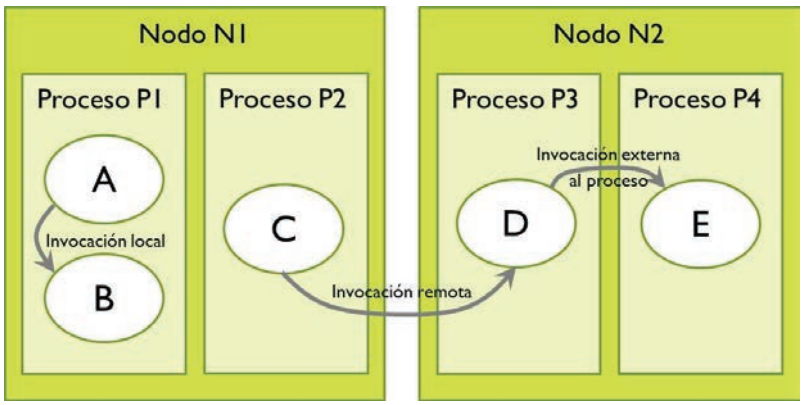

Figura 8.8: Invocación local y remota.

El modelo se basa en servidores que mantienen objetos y permiten que los clientes invoquen sobre ellos las operaciones definidas en su interfaz. Se sigue un protocolo petición/respuesta, pero clientes y servidores pueden desarrollarse en distintos lenguajes de programación, funcionar sobre sistemas operativos diferentes y arquitecturas distintas, y comunicarse mediante distintos protocolos de red.

La invocación local es idéntica a la de un sistema no distribuido, pero la invocación remota requiere un middleware de soporte (*middleware orientado a objetos*). Existen distintas propuestas, cada una con una orientación diferente:

- DCOM (.NET) [EE98]. Es multilenguaje pero no multiplataforma. Utilizado sobre sistemas Microsoft Windows.
- CORBA [Obj11a] es un estándar multilenguaje y multiplataforma (facilita interoperabilidad), pero su complejidad limita su difusión.
- RMI [Ora12f, Ora12d] únicamente soporta Java (no es multilenguaje).
- ICE ("*Internet Communications Engine*") [Hen04] es una alternativa posterior a CORBA. Resulta comparativamente simple y eficiente, pero su difusión es escasa.

El principal componente de un middleware orientado a objetos es el *ORB* ("*Object Request Broker*"), que se encarga de:

- Identificar y localizar objetos. Para ello los objetos deben ser registrados en el ORB. En ese momento se les asigna un determinado identificador con el que se crea su primera referencia. Las referencias son empleadas posteriormente para realizar las invocaciones, permitiendo localizar en qué nodo y proceso reside el objeto asociado a la referencia.
- Realizar invocaciones remotas sobre objetos. Se emplea el mecanismo ROI que es similar a una RPC. La diferencia principal reside en el componente que reemplaza al stub servidor, llamado *esqueleto* pues ahora debe gestionar una interfaz formada por múltiples operaciones, por lo que debe ser capaz de servir las invocaciones recibidas sobre todo ese conjunto de operaciones, en lugar de una sola, como ocurría en las RPC. Por su parte el stub cliente pasa a llamarse *proxy*[Sha86] y ofrece la misma interfaz que el objeto al que representa. El estado de un proxy está formado por una referencia al objeto remoto. En los mensajes de petición se debe incluir como información adicional el identificador del método que se esté invocando. Eso no resultaba necesario en una RPC.
- Gestionar el ciclo de vida de los objetos (creación, registro, activación y eliminación de los objetos).

El esquema básico de funcionamiento para una invocación remota es similar al descrito para RPC, y se muestra en la figura 8.9. En ella se ilustra el camino seguido por los mensajes de petición en aquellas invocaciones que ya aparecieron anteriormente en la figura 8.8 y que deban ser gestionadas por un ORB. No se muestra el camino seguido por los mensajes de respuesta.

En el caso de la interacción entre los objetos C y D, los pasos utilizados son:

- El proceso cliente invoca el método sobre el proxy local del objeto remoto (actúa como representante local del objeto D y posee la misma interfaz que D y una referencia a su esqueleto).

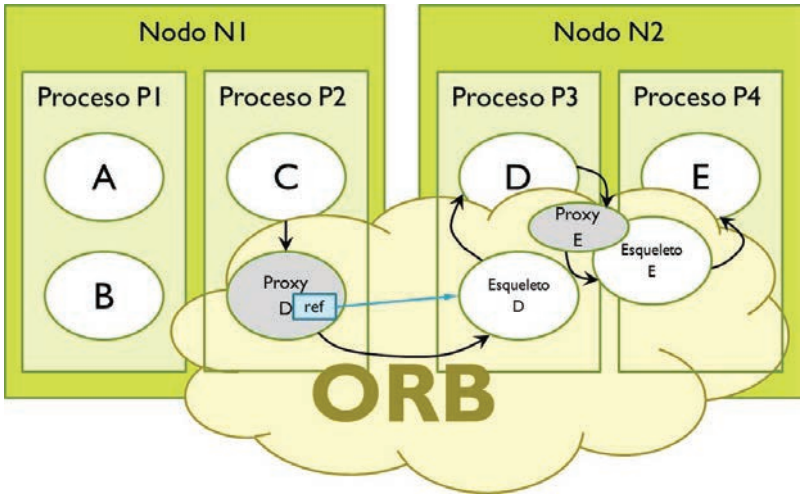

Figura 8.9: Funcionamiento básico de una ROI.

- El proxy empaqueta (marshall) los argumentos y, utilizando la referencia, llama al ORB para que éste gestione la invocación.
- El ORB consigue que el mensaje de petición llegue al esqueleto.
- El esqueleto desempaqueta argumentos y reenvía la solicitud al método invocado. Cuando dicho método termina, el esqueleto empaqueta los resultados y devuelve el control al ORB.
- El ORB hace llegar el mensaje de respuesta al proxy. El proxy desempaqueta los resultados y los devuelve al proceso cliente, completando así la invocación. Con ello retorna el control al código del objeto C.

La invocación entre procesos en el mismo nodo (D,E) podría realizarse como una invocación remota, pero puede optimizarse usando memoria compartida. Se crea un par proxy-esqueleto especializado que actúa como puente entre los espacios de memoria de cliente y servidor, como se observa en la figura 8.9.

8.4.1 Visión de usuario

La infraestructura de objetos remotos proporciona transparencia de acceso y ubicación. Oculta parte de los detalles de distribución, pero no todos (ej. creación de objetos remotos).

La encapsulación garantiza que un objeto (tanto local como remoto) sólo sea accesible a través de su interfaz. La interfaz se puede especificar mediante *IDL* ("*In-*

terface Definition Language"), con una funcionalidad similar al IDL de RPC. Por una parte se utiliza para especificar las interfaces de los objetos que componen una determinada aplicación distribuida que puedan ser invocados remotamente. Así ayuda al programador durante la etapa de diseño, forzándole a que decida qué interfaces debe tener cada componente y qué tipo de información se intercambiará en cada operación, documentando el diseño resultante. Por otra parte genera automáticamente los proxies y esqueletos, bien de forma estática (en tiempo de compilación) o de forma dinámica (en tiempo de ejecución).

El IDL es neutral (no corresponde a ningún lenguaje completo), pero permite generar pares proxy-esqueleto en varios lenguajes.

8.4.2 Creación y registro de objetos

Los objetos se crean en una aplicación distribuida de igual manera que en las aplicaciones no distribuidas. No obstante, si no se realiza ninguna acción más, esos objetos no serán invocables mediante el mecanismo ROI. Para que un objeto llegue a ser invocable de manera remota, se debe proceder a su registro en el ORB. Como consecuencia de ese registro, el ORB devolverá una primera referencia (incluida en un proxy y obtenida por el propio proceso en el que ha sido creado el objeto) y generará el esqueleto correspondiente.

Esta tarea de creación y registro puede organizarse de dos maneras diferentes, según quién la solicite:

- Creación a iniciativa del cliente. Se usa una *factoría* de objetos (servidor que crea objetos de un determinado tipo). Se devuelve al cliente una referencia al objeto creado. La figura 8.10 muestra los pasos que se siguen en esta alternativa:

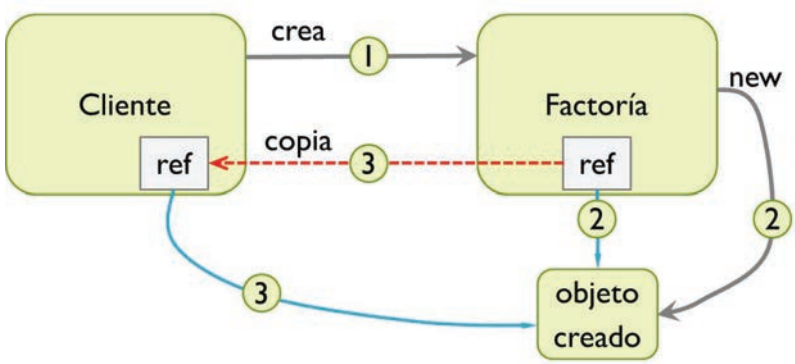

Figura 8.10: Creación de objetos remotos a solicitud del cliente.

1. El cliente solicita a la factoría la generación del objeto. Para ello se asume que el cliente dispone de un proxy para invocar a dicha factoría y que habrá un método que devolverá como resultado o como argumento de salida una referencia (proxy) al objeto que va a crearse seguidamente.
2. La factoría crea el objeto solicitado y lo registra en el ORB. Como resultado de ese registro obtiene una referencia y se instancia un esqueleto a partir del cual se podrán encauzar las futuras invocaciones remotas sobre el objeto que ahora se ha registrado.
3. Al retornar el control al cliente, la factoría le devuelve una copia de la referencia que acaba de obtener. Con ello, el cliente pasará a tener un proxy con el que invocar en un futuro al objeto que se ha generado en el paso 2.

Tanto en la figura 8.10 como en la 8.11 las flechas discontinuas representan operaciones de copia de las referencias. Tales copias se realizan de manera implícita al pasar un objeto como argumento en una invocación. A su vez, las flechas con trazo continuo más estrecho se utilizan para reflejar a qué objeto apunta cada referencia. Por último las flechas de trazo continuo más ancho sirven para representar invocaciones entre los diferentes objetos.

- Creación a iniciativa del servidor. Tal como muestra la figura 8.11, en esta segunda alternativa también se sigue cierta secuencia de pasos. Son los siguientes:

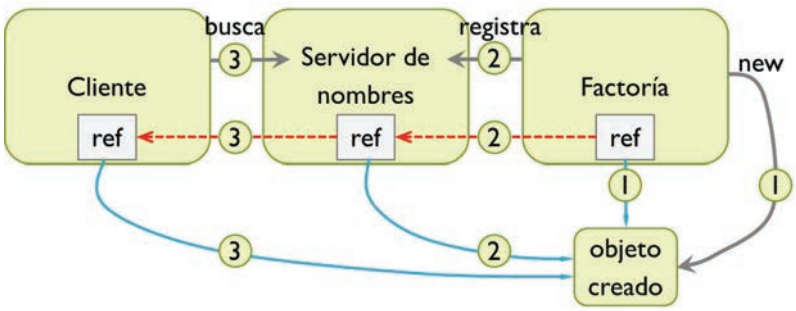

Figura 8.11: Creación de objetos remotos por parte del servidor.

1. Cierto proceso crea el objeto y lo registra en el ORB para que éste genere su primera referencia, proxy y esqueleto. Este proceso pasará a ser, a partir de ese momento, el servidor de las invocaciones recibidas por el objeto que acaba de ser registrado.

   En caso de que el programa ejecutado por tal proceso prevea la generación de múltiples objetos de esa misma clase, este proceso servidor podría considerarse un tipo particular de factoría.

2. Posteriormente el proceso servidor utiliza esa referencia para registrarla en un servidor de nombres, asociándole determinada cadena de texto como nombre.
3. Después, cualquier otro proceso que conozca el nombre con el que se haya registrado el objeto podrá obtener un proxy invocando cierto método del servidor de nombres. Los procesos que hayan obtenido sus proxies de esta manera podrán invocar ese objeto mediante el mecanismo ROI. Por tanto, serán sus clientes.

Estas dos alternativas proporcionan al cliente cierta manera de obtener una referencia a un objeto remoto. Existe una tercera opción. Consiste en recibir dicha referencia como argumento de salida en cualquier ROI, aunque el objeto que proporcione ese método invocado en la ROI no sea una factoría para esa clase de objetos, sino sólo un cliente que ya disponía de tal referencia y se la "pase" a quien le invoque.

8.4.3 Detalles de la invocación remota

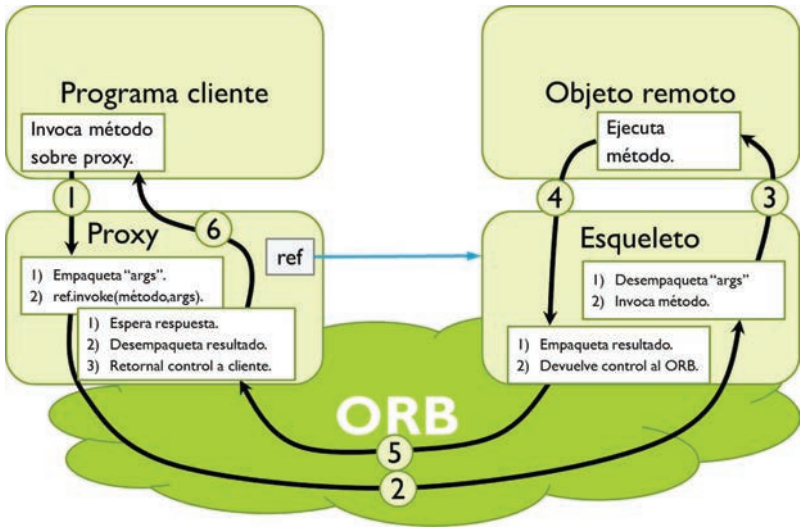

Figura 8.12: Detalles de una invocación remota.

La figura 8.12 describe detalladamente los pasos a realizar en una ROI:

1. Se asume que el proceso cliente habrá obtenido previamente un proxy que permita realizar la ROI. En cierto punto de su ejecución ese proceso cliente necesitará invocar algún método del objeto remoto. Para realizar esa acción invocará al proxy, solicitando la ejecución de ese método.

El proceso cliente no es consciente de que el proxy no es quien ejecuta realmente el método. Como su interfaz es exactamente la misma, obtiene transparencia de ubicación.

Como resultado de esta invocación, el proxy empaqueta los argumentos de entrada recibidos e invoca cierta operación del ORB, proporcionando la referencia al objeto que debe ser invocado (tal referencia es uno de los atributos del proxy), la versión empaquetada de los argumentos y el identificador del método a invocar.

2. El ORB recibe esa petición por parte del proxy (que hemos representado en la figura como "`ref.invoke(método,args)`"). El ORB es capaz de interpretar la información contenida en la referencia. Con tal información sabe en qué nodo reside el objeto, qué protocolo de comunicaciones debe utilizarse para interactuar con él y qué identificador tendrá tal objeto en el nodo servidor. Esto permitirá enviar un mensaje de petición (que, entre otros datos, incluya los argumentos de entrada empaquetados y el identificador del método a invocar) al nodo servidor para que los componentes del ORB ubicados en tal nodo puedan localizar el esqueleto y entregarle a éste dicho mensaje.

3. Cuando el esqueleto reciba este mensaje de petición, lo procesará en función del identificador de método incluido en él. Dicho procesamiento consiste en el desempaquetado de los argumentos de entrada (si los hubiere) y en la invocación del método correspondiente.

   Obsérvese que un mismo objeto puede estar implantando múltiples interfaces. Por ejemplo, ese objeto puede ser de la clase C y ésta puede heredar de la B, que a su vez herede de la A. Cada clase tendrá su propio esqueleto. Por tanto, en los mensajes de petición el proxy habrá tenido que incluir el identificador de interfaz asociado al método que va a invocarse, para que así el ORB del nodo servidor sepa qué esqueleto tendrá que utilizarse para gestionar dicha invocación.

4. El esqueleto queda momentáneamente bloqueado, esperando que finalice la ejecución del método invocado. Cuando eso sucede, retoma el control para empaquetar el resultado y los argumentos de salida y construir con ello un mensaje de respuesta. En caso de que la ejecución del método hubiese generado una excepción, ésta también sería empaquetada de cierta manera, en lugar de los resultados. Hecho esto se devuelve el control al ORB del nodo servidor.

5. El ORB recoge el mensaje de respuesta y lo transmite al nodo cliente, para entregárselo al proxy.

6. El proxy del nodo cliente había quedado esperando tal respuesta desde el final del paso 1. Al recibirla, se reactiva para desempaquetar los argumentos de salida y el resultado (o bien la excepción, si la hubo). Con esta información ya es capaz de contestar la invocación recibida, finalizando así la ROI.

Aunque la visión de usuario es relativamente simple, hay varios aspectos que complican la implementación. Estos aspectos son la información contenida en las referencias a objeto y el paso de argumentos, que son descritos seguidamente.

Referencias a objeto

La *referencia* proporciona acceso a un objeto. Como solo se puede acceder a un objeto a través de su interfaz, la referencia debe proporcionar acceso a dicha interfaz. En caso de un objeto remoto, la referencia debe proporcionar toda la información de acceso necesaria (ubicación de red y protocolo de acceso).

Existen varias alternativas:

- *Referencia directa*. Contiene la ubicación de red (dirección IP y número de puerto, generalmente) y, en caso de que sea posible utilizar diferentes protocolos de comunicación en un mismo ORB, el protocolo de acceso. Su contenido y los componentes necesarios para gestionarla se muestran en la figura 8.13.

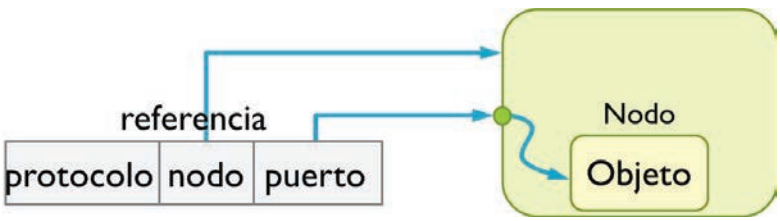

Figura 8.13: Referencia directa.

  Las referencias directas plantean múltiples problemas, por lo que rara vez llegan a utilizarse. Los problemas más importantes son:

  - No permite reubicar el objeto (invalidaría todas sus referencias), por lo que resulta inflexible.
  - No permite mantener más de un objeto en una misma ubicación de red. Esto obligaría a dedicar un puerto distinto por cada objeto existente en ese nodo que pueda ser invocado vía ROI.

- *Referencia indirecta*. Para gestionarlas, cada nodo debe disponer de un *adaptador de objetos*, como se observa en la figura 8.14. El adaptador se encarga de gestionar a múltiples objetos, asignando un identificador distinto a cada uno de ellos. Basta con tener un adaptador por nodo, atendiendo en cierto puerto. El adaptador se encargará de redirigir los mensajes de petición al objeto adecuado. Para ello necesita que las referencias tengan un campo

adicional a los ya descritos para las referencias directas: el identificador de objeto.

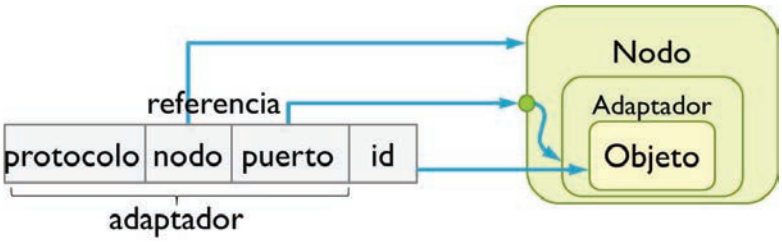

Figura 8.14: Referencia indirecta.

Con el uso de adaptadores se elimina una de las limitaciones observadas en las referencias directas: ya pueden gestionarse múltiples objetos a través de una misma ubicación de red.

- *Referencia a través de localizador*. El contenido de la referencia añade un identificador de gestor al que ya ofrecía el formato anterior, tal como ilustra la figura 8.15. En este caso la tripleta formada por el tipo de protocolo, dirección IP y número de puerto se interpreta en primer lugar y permite acceder a un "*localizador*". El localizador podrá gestionar a múltiples adaptadores, manejando una tabla en la que almacena sus respectivas direcciones y puertos. Por ello, la referencia incluye como siguiente campo el identificador del gestor o adaptador que deberá manejar el campo restante: un identificador de objeto.

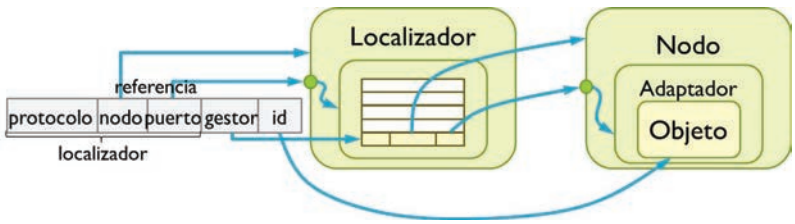

Figura 8.15: Referencia a través de un localizador.

Permite reubicar el adaptador (modificando para ello la información contenida en la tabla gestionada por el localizador) y tener múltiples adaptadores para una misma clase de objetos. Con ello se supera la otra limitación ofrecida por las referencias directas: la imposibilidad de reubicación/migración de objetos.

Para la localización dinámica de los objetos se recurre a un servidor de nombres. Para evitar que la localización del servidor se convierta en un cuello de botella

y punto de fallo único, se puede utilizar un servicio de nombres distribuido. Esos servicios se describirán en la unidad 10.

Paso de argumentos

El paso de argumentos resuelve básicamente dos problemas. El primero se refiere a cómo transmitir valores de tipos básicos a través de la red. Es el paso por valor y se resuelve mediante las tareas de empaquetado y desempaquetado (marshalling/unmarshalling) ya analizadas en el mecanismo RPC. El segundo surge cuando se pasan objetos como argumentos. En caso de que deban pasarse por valor habrá que empaquetar su estado. Como el estado se estructura en múltiples atributos y, además, también debería pasarse el código de cada uno de sus métodos, este proceso de empaquetado es bastante más complejo y recibe el nombre de *serialización*. El receptor de tal argumento tendrá que desempaquetar el objeto transferido, creando una copia del objeto original. Este tipo de pase se ilustra en la figura 8.16. A partir de esa acción cada objeto seguiría su propia evolución, en función de las invocaciones que recibiera. Otra forma de pasar un objeto por valor consiste en facilitar una referencia especial, desde la que se podrá descargar tanto el código como el estado del objeto a transferir.

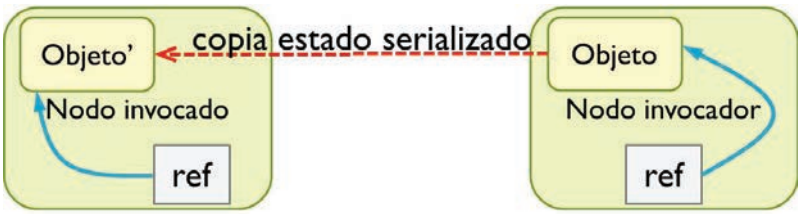

Figura 8.16: Paso de objetos por valor.

Este segundo problema tiene otra variante. Se da cuando los objetos deban pasarse por referencia. Ya se ha descrito qué es una referencia a objeto y cómo se utilizan: mediante los proxies, a la hora de implantar el mecanismo de las ROI. En la práctica, una referencia a objeto en un entorno distribuido ofrece la misma funcionalidad que un puntero en un programa tradicional, diseñado para ser ejecutado en un solo espacio de direcciones: permitir el acceso sobre cierta entidad empleando un nivel de indirección. Así existe una única copia de esa entidad y las modificaciones realizadas por cualquiera de los módulos que tengan acceso a ella serán visibles por parte de todos los demás módulos. Si se copia una referencia y se entrega a otro proceso (en el mismo o en otro nodo), el receptor podrá utilizar tal referencia sin ningún problema. Será eso lo que se utilizará para pasar objetos por referencia: transmitir el contenido de tales referencias; es decir, copiarlas. Así sólo existirá una única copia de ese objeto y todas las invocaciones que realicen sus múltiples clientes afectarán únicamente a esa instancia.

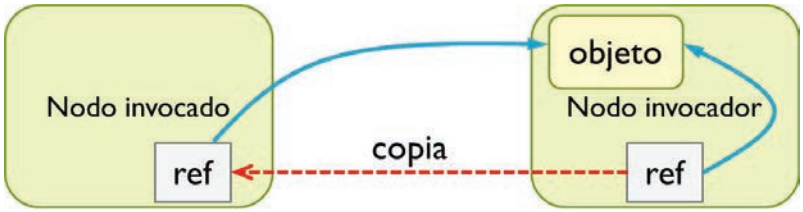

Figura 8.17: Paso por referencia de un objeto local.

Este segundo tipo de paso de objetos como argumentos se ilustra en las figuras 8.17 y 8.18. Obsérvese que en la figura 8.17 el objeto que se está pasando como argumento es un objeto local. El invocador habrá tenido que registrar dicho objeto en el ORB para obtener así una referencia a ese objeto. Tras esto, ya puede facilitar tal referencia como argumento en cualquier método que invoque y necesite un argumento de esa misma clase. Por otra parte, la figura 8.18 muestra como esa misma acción (pasar un objeto por referencia) también puede realizarse sin mayor problema en caso de que la referencia facilitada corresponda a un objeto que no resida en el propio proceso invocador. De hecho, el invocador no observará ninguna diferencia entre un caso y otro.

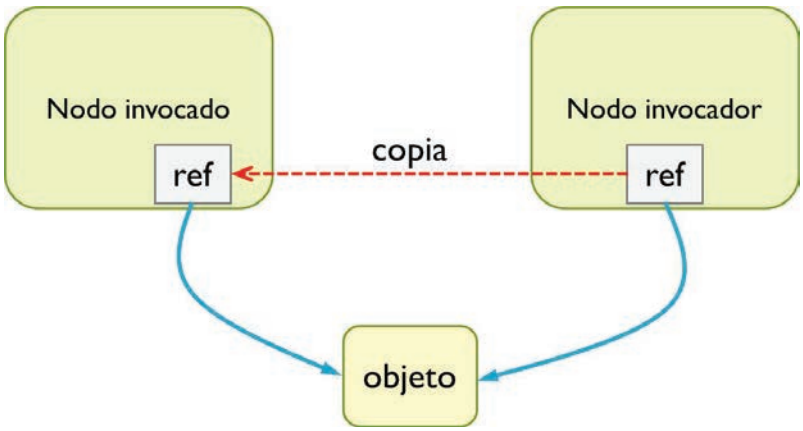

Figura 8.18: Paso por referencia de un objeto remoto.

Para finalizar esta descripción de la gestión de referencias se presenta un ejemplo en el que se realiza una invocación de un método que necesita dos argumentos. En la figura 8.19, el Nodo 1 define el objeto A y mantiene referencias para los objetos B y C creados respectivamente por los nodos 2 y 3. El estado inicial del sistema se muestra en la mitad izquierda de la figura. Si el nodo 1 invoca `B.m(A,C)` y dicho método se ha especificado en IDL indicando que debe realizarse un paso por valor de su primer parámetro y por referencia del segundo, el nodo 2 recibe la copia de

A y la referencia de C. Los efectos de tal invocación quedan reflejados en la mitad derecha de la figura. Las flechas discontinuas expresan un proceso de copia de las entidades relacionadas por tales flechas. Esto corresponde a una serialización del objeto A y a la copia de la referencia al objeto C.

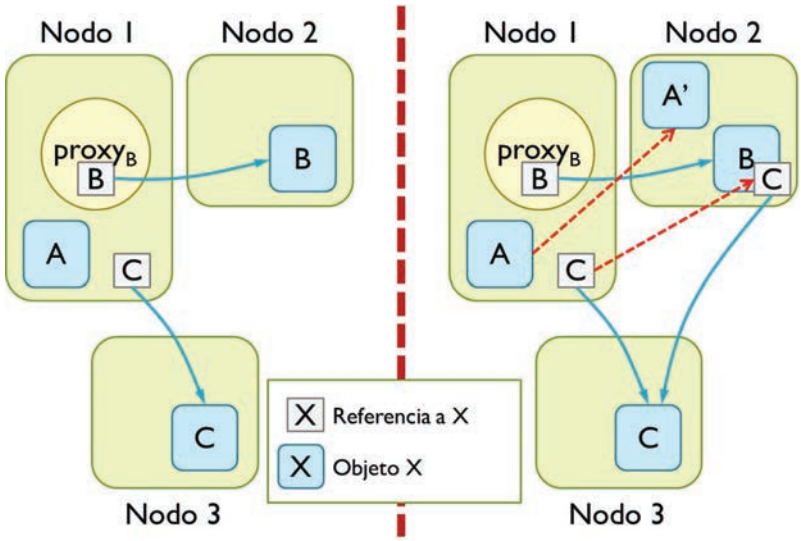

Figura 8.19: Ejemplo de paso de argumentos.

8.4.4 Otros aspectos

Existen otros aspectos a considerar, pero su análisis detallado queda fuera del ámbito del presente texto:

- *Invocaciones concurrentes.* Varios clientes pueden invocar simultáneamente operaciones sobre un mismo objeto. La implementación del objeto servidor debe aplicar las técnicas de programación concurrente descritas en las primeras unidades de este texto. Obsérvese que el código del objeto servidor reside normalmente en un único nodo, por lo que no resulta necesario el uso de técnicas más complejas (siempre y cuando el objeto invocado no esté replicado).
- *Migración.* En sistemas que soporten migración de objetos, debe garantizarse que las referencias apunten al objeto correcto con independencia de su ubicación. Una primera medida para proporcionar este soporte consiste en utilizar referencias con localizador.
- *Replicación.* En sistemas que soporten replicación de objetos debería proporcionarse transparencia de replicación y alguna gestión que garantice la

consistencia entre el estado de las diferentes réplicas. De nuevo, para proporcionar transparencia de ubicación y replicación pueden utilizarse referencias con localizador. En este caso el localizador mantendría un conjunto de direcciones de nodos en los que se ubicarían los adaptadores que gestionarían a cada una de las réplicas. Así, con una sola invocación iniciada por un cliente se podría llegar a todas las réplicas de un determinado objeto. No obstante, no todos los modelos de replicación requieren este tipo de gestión. Además, la gestión de la consistencia entre réplicas es ciertamente delicada y no se resuelve únicamente con la transparencia de replicación y ubicación. La transparencia de fallos es también importante y no depende del mecanismo de invocación utilizado.

8.4.5 RMI (Remote Method Invocation)

Los entornos middleware de comunicación orientada a objetos tradicionales (CORBA, DCOM, ICE, ...) soportan aplicaciones distribuidas cuyos componentes estén implantados en distintos lenguajes de programación. Una alternativa más simple es proporcionar una solución sobre un único lenguaje orientado a objetos que garantice portabilidad. *Java RMI* [Ora12f] implementa el mecanismo de invocación de objetos remotos para el lenguaje Java. Extiende el lenguaje con la posibilidad de invocar métodos de un objeto Java en otra *máquina virtual Java* (*JVM*: "*Java Virtual Machine*"), y pasar objetos Java como argumentos al invocar dichos métodos. Como el lenguaje Java incluye el concepto de interfaz (`interface`), no se necesita un IDL adicional. Los proxy y esqueleto se generan a partir de la especificación de su interfaz remota. En las primeras versiones (antes de la edición J2SE v5.0) se debía utilizar un generador de stubs (`rmic`). Ahora ya no resulta necesario.

Las reglas para programar utilizando objetos remotos son:

- La interfaz del objeto remoto se define como una interfaz cualquiera en Java, pero debe extender la interfaz `java.rmi.Remote` (es un marcador para identificar interfaces remotas). Gracias a ello el compilador de Java generará los proxies y esqueletos para gestionar las invocaciones remotas.
- La invocación de un método sobre un objeto remoto debe indicar que se puede generar la excepción predefinida `java.rmi.RemoteException`.
- Además de implantar la interfaz `Remote` la clase que mantenga el código servidor debe extender la clase `java.rmi.server.UnicastRemoteObject`. Esto resulta necesario para registrar los objetos en el gestor de invocaciones (ORB) de Java. Para ello se utiliza, de manera implícita, el método `exportObject()` de esa clase `java.rmi.server.UnicastRemoteObject`.

Al invocar un método podemos pasar objetos como argumentos. Los objetos locales (los que residen en el nodo del invocador) se pasan por valor y por lo tanto deben ser serializables. Por su parte, los no locales se pasan por referencia, transmitiendo un proxy del objeto.

El modelo de programación RMI no persigue transparencia total: para pasar una aplicación centralizada a distribuida son necesarias ciertas modificaciones:

- Un modelo de objetos remotos se basa en un servidor de nombres. RMI utiliza un servidor de nombres en el que se registran objetos remotos bajo nombres simbólicos (junto a cada nombre se registra la referencia del objeto correspondiente). El servidor de nombres puede residir en cualquier nodo, y resulta accesible tanto a cliente como a servidor bajo una interfaz local denominada `Registry`. La figura 8.20 muestra las operaciones disponibles en esa interfaz.

  En el caso de las distribuciones Java desarrolladas por Oracle, el servidor de nombres se lanza mediante la orden `rmiregistry`.

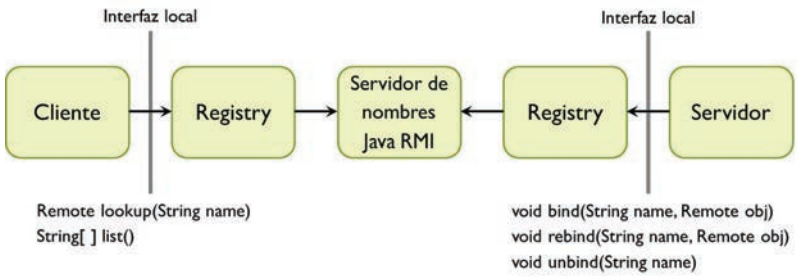

Figura 8.20: Utilización del registro en RMI.

- El servidor puede utilizar los métodos `bind(name, obj)`, `rebind(name, obj)` y `unbind(name)`. Las operaciones `bind()` y `rebind()` permiten registrar objetos, mientras que `unbind()` elimina ese registro.
- El cliente puede utilizar los métodos `lookup(name)` y `list()`. La operación `lookup()` permite obtener la referencia asociada a un nombre, y `list()` retorna un vector de cadenas que contiene los nombres de todos los objetos registrados.

Desarrollo de una aplicación RMI

Para ilustrar cómo funciona RMI se propone seguidamente un ejemplo sencillo. No se muestra todo el código, sino únicamente aquellas partes más importantes. Por ejemplo, no se ha incluido el tratamiento de las posibles excepciones que podrían generarse. En la versión distribuida se asume que se están importando los paquetes `java.rmi` y `java.rmi.registry`. Para tener una visión más completa sobre cómo desarrollar aplicaciones que utilicen RMI se recomienda que el lector siga los tutoriales disponibles. El tutorial de Oracle [Ora12d] es un buen ejemplo.

Este primer listado muestra diversos fragmentos del código que se utilizaría en una versión no distribuida de una aplicación que muestra un mensaje por la salida estándar. El fragmento final se encarga de instanciar un objeto de la clase `ImplHola` y utilizar su método `saluda()` para escribir el mensaje.

Versión centralizada (interfaz, implementación, uso)

```
public interface Hola {
  String saluda();
}



class ImplHola implements Hola {
  ImplHola() {..} // constructor
  public String saluda() {return "Hola a todos";}
}



...;
Hola h=new ImplHola();
System.out.println(h.saluda());
```

En la versión distribuida cliente y servidor pueden estar en JVM distintas e incluso nodos distintos. El código a utilizar sería:

Versión distribuida (interfaz, implementación, servidor, cliente)

```
public interface Hola extends Remote {
  String saluda() throws RemoteException;
}



class ImplHola extends UnicastRemoteObject
implements Hola {
  ImplHola() throws RemoteException {..} // constructor
  public String saluda() throws RemoteException {
    return "Hola a todos";
  }
}



public class Servidor {
  // Asumimos que el servidor de nombres es local.
  public static void main (String[] args) {
    Registry reg = LocateRegistry.getRegistry();
    ...
    reg.rebind("objetoHola", new ImplHola());
    System.out.println("Servidor Hola preparado");
  }
}



// Asumimos que el cliente recibe como argumentos
// el nombre de la máquina y el puerto en que el
// servidor de nombres se esté ejecutando y que ambos
// se han dejado en las variables 'host' y 'port'.
Registry reg = LocateRegistry.getRegistry(host, port);
...;
Hola h= (Hola) reg.lookup("objetoHola");
System.out.println(h.saluda());
```

El cliente debe localizar el objeto remoto, y acceder a él de forma remota. El programa servidor contiene la implementación de la interfaz. El servidor crea el objeto remoto y lo registra bajo un nombre simbólico. Para ello se utiliza la operación

`rebind()` del servicio de nombres. A diferencia de `bind()`, si el nombre ya existe no falla, sino que asocia a éste el nuevo objeto.

El registro también puede ser un servicio distribuido (ejecutándose en una máquina distinta a la del cliente y a la del servidor). Tanto el cliente como el servidor utilizan un proxy del registro (obtenido gracias al método estático `getRegistry()` de la clase `LocateRegistry`) que localiza e invoca al registro actual. Obsérvese que en este ejemplo se ha asumido que el servidor de nombres se encuentra en la misma máquina que el servidor del objeto `ImplHola` y que está atendiendo peticiones en el puerto utilizado por omisión. En el programa cliente se ilustra cómo debe utilizarse el método `getRegistry()` en caso de que el servidor de nombres no esté en el mismo nodo.

La ejecución completa utiliza los siguientes pasos (véase la figura 8.21, donde se muestra el diagrama de secuencia UML correspondiente):

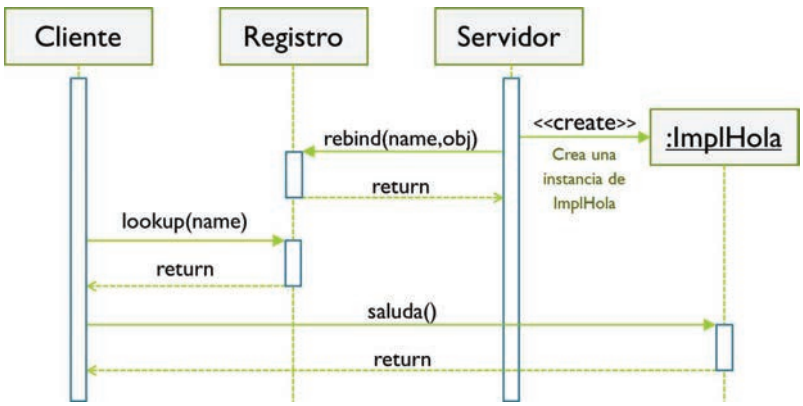

Figura 8.21: Diagrama de ejecución del ejemplo.

1. Se generan las clases proxy y esqueleto gracias a la compilación de la interfaz `Hola`. Esto no llega a mostrarse en la figura, pues ésta recoge los pasos realizados durante la ejecución de los diversos componentes.

2. Se lanza el registro, utilizando la orden `rmiregistry`.

3. Arranca el servidor. El servidor genera la instancia de la clase `ImplHola` y la registra en el servidor de nombres utilizando para ello su método `rebind()`.

4. Arranca el cliente. El programa cliente sabe con qué nombre ha sido registrado el objeto y utiliza tal nombre como argumento del método `lookup()` del servidor de nombres. Como resultado de ello obtiene una referencia (y proxy) para invocar remotamente a `ImplHola`.

5. En el último paso, el cliente invoca el método `saluda()` y con su resultado es capaz de mostrar el mensaje por salida estándar.

8.5 Comunicación basada en mensajes

La llamada a procedimiento remoto es un mecanismo de alto nivel que proporciona transparencia de ubicación. No toda la comunicación en una aplicación distribuida estará basada en ese mecanismo. Se pueden intercambiar mensajes entre los distintos módulos de la aplicación sin necesidad de hacerlo mediante RPC. En ese caso más general, conviene conocer qué características ofrecerá el mecanismo de intercomunicación utilizado. En esta sección describiremos dos de ellas: la sincronización y la persistencia.

Para ello asumiremos que los módulos a intercomunicar pertenecen al nivel de aplicación de la arquitectura de referencia OSI y que existe algún *middleware* que implanta esa comunicación entre módulos. Los protocolos utilizados en este middleware serán los responsables de proporcionar la persistencia y de admitir diferentes grados de sincronización en las comunicaciones.

Obsérvese que en esta sección ya no haremos referencia a los roles de *cliente* y *servidor*, relacionados con una invocación de un servicio, sino a los de *emisor* y *receptor* de un determinado mensaje.

8.5.1 Sincronización

La *sincronización* tiene en cuenta el intervalo de espera que debe soportar el emisor para dar por confirmada la transmisión del mensaje.

Existen diferentes tipos de comunicación en función de su grado de sincronización:

- *Comunicación asincrónica*: El emisor no se bloquea. Envía el mensaje al componente local del middleware de intercomunicación y continúa inmediatamente con su ejecución.

  Este tipo de comunicación se ilustra en la figura 8.22. Al igual que en las figuras anteriores, representamos los intervalos de bloqueo con líneas discontinuas, mientras que los intervalos de actividad se muestran con trazo más grueso y continuo. Como puede observarse, el emisor no llega a suspenderse en este tipo de comunicaciones. Asume de manera optimista que la transmisión del mensaje no acarreará ningún problema.

- *Comunicación sincrónica*: En este caso el emisor llegará a bloquearse esperando algún tipo de confirmación por parte del middleware de comunicaciones. Existen tres tipos de comunicación sincrónica, dependiendo del evento considerado para retornar esa confirmación:

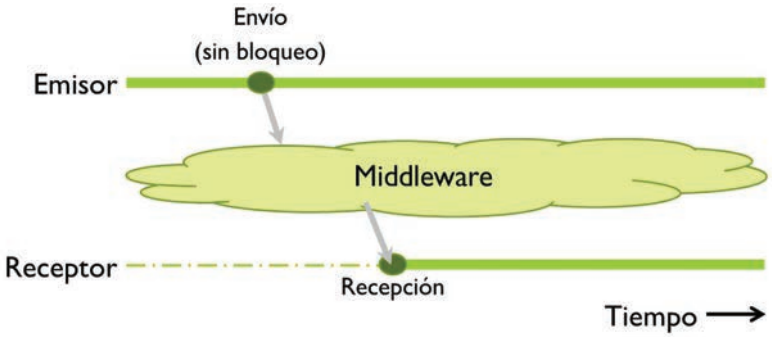

Figura 8.22: Ejemplo de comunicación asincrónica.

1. *Envío*: El middleware responde al emisor confirmando que podrá realizar la transmisión. El mensaje está todavía en los buffers de envío del nodo emisor, pero dichos buffers son gestionados por el middleware.

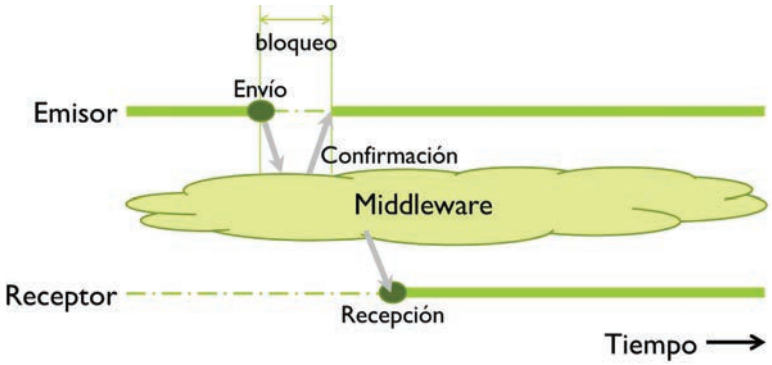

Figura 8.23: Comunicación sincrónica basada en envío.

   Esta gestión se ilustra en la figura 8.23. Como puede observarse, en este caso la confirmación es muy rápida y en ella solo intervienen procesos locales. El mensaje todavía no ha llegado a salir del nodo emisor.

2. *Entrega en el receptor*: El middleware responde al emisor cuando el receptor ha confirmado la correcta entrega del mensaje.

   En este caso (véase la figura 8.24) el mensaje ya ha tenido que llegar al nodo receptor y ha debido entregarse al proceso destinatario. Tras haber realizado dicha entrega, el propio middleware de intercomunicación será el que retorne la confirmación al emisor para reactivarlo.

   Este tipo de gestión coincide con la utilizada en las RPC asincrónicas.

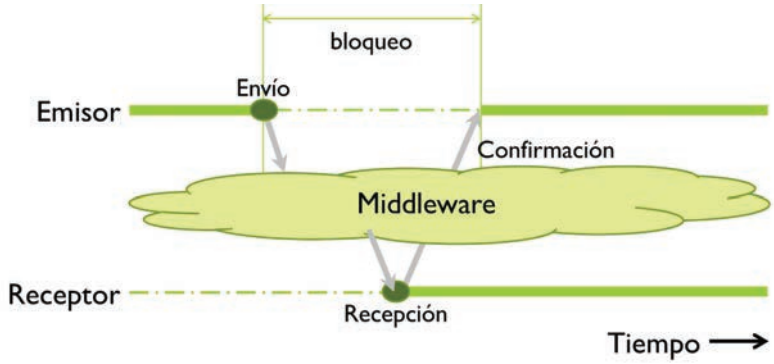

Figura 8.24: Comunicación sincrónica basada en entrega.

3. *Procesamiento por el receptor*: El middleware responde al emisor y lo desbloquea cuando el proceso receptor envía un nuevo mensaje al middleware indicando que ya ha completado el procesamiento del mensaje recibido.

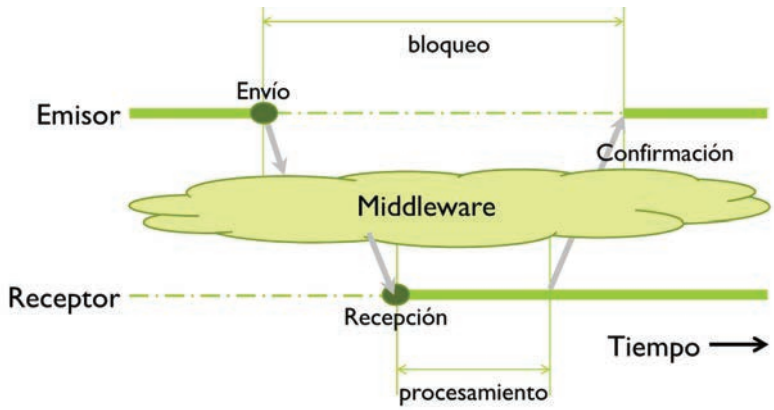

Figura 8.25: Comunicación sincrónica basada en procesamiento.

En este caso (véase la figura 8.25) es el proceso receptor quien debe iniciar la confirmación. Este tipo de gestión coincide con la utilizada en las RPC convencionales.

8.5.2 Persistencia

Por lo que respecta a la persistencia solo podremos encontrar dos tipos de comunicación:

- *Comunicación persistente*: Se da cuando el middleware de intercomunicación es capaz de mantener el mensaje (impidiendo que sea eliminado) hasta que el proceso receptor esté en condiciones de recibirlo. En este caso, los nodos que deban recorrerse para transmitir el mensaje hacia su destino podrán guardar dicho mensaje durante el tiempo que sea necesario (normalmente en almacenamiento secundario). Esto es particularmente importante en caso de fallos en alguno de esos servidores intermedios.

  Debido a este tipo de gestión, se permite que:

  - El receptor no esté activo cuando el emisor inicie la transmisión del mensaje.
  - El emisor no esté activo cuando se le entregue el mensaje al receptor.

  El servicio de correo electrónico es un buen ejemplo de comunicación persistente. Los usuarios utilizan agentes de correo: procesos que permiten redactar nuevos mensajes de correo, así como leer los mensajes que se vayan recibiendo. Para que estos agentes puedan realizar sus tareas dependen de un middleware que gestione el servicio de correo electrónico. Dicho middleware está compuesto por un buen número de servidores, responsables cada uno de cierto conjunto de posibles nodos clientes.

  Cuando nosotros enviamos un correo electrónico a otro usuario, no es necesario que dicho usuario esté en ese momento utilizando su lector de correo. Por tanto, no se exige que el receptor esté activo. Ese destinatario no tiene por qué pertenecer tampoco a nuestra misma organización y/o proveedor del servicio de correo electrónico. Por ello, el servidor de correo de nuestro sistema tendrá que encontrar la ruta más adecuada para llegar al servidor de correo del usuario destinatario y propagar el correo a través de ella. Si alguno de los nodos de esa ruta no estuviera activo, el servidor correspondiente guardará el correo en disco hasta que sea posible interactuar con él, reintentándolo de vez en cuando. Finalmente, el correo llegará a su servidor destinatario. Más pronto o más tarde, el usuario a quien iba destinado ese mensaje de correo abrirá su programa agente y lo leerá. En ese momento, el usuario emisor no tiene por qué estar utilizando todavía su propio proceso lector. De hecho, puede haber pasado un buen rato (incluso días, pues no tenemos por qué estar pendientes del correo) entre un evento (envío) y otro (recepción y lectura).

- *Comunicación no persistente*: En la comunicación no persistente el middleware no es capaz de mantener los mensajes que deban transmitirse. Si el

proceso destinatario del mensaje abortara o no estuviera activo, la comunicación se rompería. Es decir, en este caso tanto el emisor como el receptor deben estar activos para que los mensajes lleguen a transmitirse.

Este tipo de comunicación es el que se lleva a cabo en los protocolos de transporte. Ejemplos válidos serían tanto *UDP* como *TCP*.

8.6 Resumen

En los sistemas distribuidos, debido a la ausencia de memoria compartida, toda la comunicación se basa en el envío y recepción de mensajes (de bajo nivel). Para ello se requieren acuerdos entre los procesos en diversos niveles que van desde detalles de bajo nivel sobre la transmisión de bits, hasta los detalles de alto nivel sobre cómo va a expresarse la información. En esta unidad se han descrito las ventajas que aporta una arquitectura de comunicaciones estructurada en múltiples niveles, tomando como ejemplo la arquitectura TCP/IP.

Aunque la comunicación entre dos ordenadores distintos deba realizarse siempre a través de una red utilizando mensajes, no todos los detalles de la comunicación deben resultar visibles a las aplicaciones. Por tanto, se ofrecerán mecanismos con mayor nivel de abstracción, que permitan ofrecer transparencia de distribución. Uno de estos mecanismos es la *llamada a procedimiento remoto* (RPC), que permite ocultar al cliente el hecho de que el procedimiento que está invocando está ubicado en otra máquina. Para ello el compilador genera cierto código (llamado *stub*), cuya existencia es desconocida por los programadores del programa cliente y del procedimiento remoto, de modo que el *stub cliente* se encarga de empaquetar los argumentos en un mensaje (*marshalling*), enviar el mensaje de invocación al servidor, esperar el mensaje de respuesta, desempaquetar los resultados de la respuesta (*unmarshalling*) y devolver los resultados al código que invocó al *stub*. El *stub servidor* actúa de forma similar en la máquina remota, pero desempaqueta primero, invoca al procedimiento y después empaqueta el resultado. En esta unidad se han estudiado las distintas variantes de este modelo: RPC síncrona, RPC asíncrona (que permite que el cliente no tenga que esperar a que termine el procedimiento, pero sí debe esperar un mensaje de respuesta) y RPC sincrónica retardada (en la que el servidor invocará al cliente cuando termine el procedimiento para proporcionarle los resultados).

Finalmente, de modo más general, en esta unidad se han analizado dos de las características más relevantes que ofrecen los mecanismos de intercomunicación: la sincronización y la persistencia. La *sincronización* tiene en cuenta el intervalo de espera que debe soportar el emisor para dar por confirmada la transmisión del mensaje. Así, en la comunicación sincrónica se bloquea al emisor hasta que el mensaje haya llegado al receptor, mientras que en la comunicación asincrónica el emisor continúa ejecutándose justo después de haber enviado el mensaje a su

buffer local o bien al sistema de comunicaciones. Por su parte, la *persistencia* se refiere a la capacidad de retención o almacenaje de los mensajes que deban transmitirse a través de los distintos nodos de la red. Así, en la comunicación persistente los mensajes enviados al receptor son almacenados por el sistema de comunicación hasta que sean recibidos por el receptor, por lo que el receptor no tiene por qué estar funcionando cuando el emisor envía el mensaje. Por contra, en la comunicación no-persistente, los mensajes solo se almacenan en el sistema de comunicaciones mientras el emisor y el receptor se encuentren en ejecución.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar los diferentes niveles de la arquitectura TCP/IP.
- Caracterizar el mecanismo de llamada a procedimiento remoto (RPC) y conocer los tipos de transparencia que ofrece.
- Identificar la secuencia de pasos que se sigue en una RPC.
- Conocer las particularidad del paso de argumentos en una RPC.
- Identificar las variantes de llamada a procedimiento remoto existentes, en función de su grado de sincronización.
- Conocer cada variante de comunicación mediante mensajes existente, según su grado de persistencia y sincronía.

Unidad 9

SINCRONIZACIÓN DISTRIBUIDA

9.1 Introducción

De manera general, el término *sincronización* hace referencia al conjunto de mecanismos que se utilizan para coordinar múltiples actividades en un determinado sistema o aplicación. En el caso de un sistema distribuido la sincronización plantea algunos problemas adicionales pues no podemos confiar en la existencia de ningún reloj que todos los procesos puedan consultar. Debido a esto, ni siquiera resulta sencillo el determinar en qué orden llegaron a suceder algunos eventos si estos fueron ejecutados por procesos ubicados en distintos ordenadores.

A simple vista, una primera solución a estos problemas parece requerir la utilización de algún algoritmo que consiga sincronizar entre sí los relojes locales utilizados en cada uno de los nodos. Es decir, que aproxime lo máximo posible el reloj de cada máquina a la hora real. Con ello se garantizaría que todos los relojes progresaran a un mismo ritmo y que todo proceso pudiera utilizar su reloj local para etiquetar la ocurrencia de cada evento. Así resultaría factible determinar el orden global en el que se haya llegado a ejecutar cierta secuencia de operaciones en una aplicación distribuida. No obstante, aunque el tiempo de transmisión de los mensajes en las redes actuales sea muy inferior al que se daba hace pocos años, también es cierto que la frecuencia de los procesadores actuales, así como el número de instrucciones que pueden ejecutar en cada unidad de tiempo, han crecido mucho. La mayoría de los algoritmos de sincronización de relojes transmiten mensajes entre los diferentes ordenadores para realizar tal sincronización. Por ello, una variación ligera en los retardos de transmisión de los mensajes implicará que la sincronización no sea

precisa. Eso evitará que los relojes locales puedan tomarse como una referencia temporal globalmente válida en el sistema, por lo que habrá que buscar otras soluciones. La sección 9.2 presenta algunos de los algoritmos clásicos de sincronización de relojes y describe qué soluciones se adoptaron finalmente.

Otro de los problemas a resolver en la sincronización distribuida, relacionado con el anterior, es la determinación del estado global que presenta una aplicación distribuida en un momento determinado. Como no es sencillo ordenar todos los eventos que ocurren en el sistema, tampoco resulta fácil que todos los procesos de una determinada aplicación comuniquen cuál es su estado actual cuando algún usuario o administrador necesite conocerlo. Al igual que en el caso de la sincronización de los relojes, la sección 9.3 presenta una solución válida para este problema. Para llegar a ella se necesita relajar un poco el objetivo inicial: en lugar de determinar un estado global preciso, el algoritmo utilizado reportará un posible estado global del sistema. Quizá no coincida con el estado real del sistema durante la ejecución de nuestras aplicaciones, pero al menos será consistente con los mensajes que han llegado a intercambiar los procesos que las componen. Con una herramienta de este tipo, capaz de reportar imágenes consistentes de la secuencia de estados por los que ha pasado el conjunto de módulos que componen una aplicación distribuida resulta factible la depuración de estos componentes.

Con frecuencia las aplicaciones distribuidas deben tomar algunas decisiones que condicionan el progreso de la propia aplicación. Si los diferentes procesos que componen la aplicación debieran intercambiar múltiples rondas de mensajes para tomar esas decisiones, la ejecución podría interrumpirse en caso de fallos o podría ralentizarse si la entrega de algunos mensajes se retrasara. Por ello, en muchos casos se opta por seleccionar un proceso coordinador y confiar en que este imponga su criterio en esos pasos de la ejecución. Para ello resultan necesarios los algoritmos de elección de líder que se describen en la sección 9.4.

Por último, como ya hemos visto en los temas relacionados con la concurrencia, uno de los mecanismos de sincronización más importantes consiste en asegurar que las secciones críticas de una aplicación se ejecuten en exclusión mutua. Algunos algoritmos distribuidos utilizados para garantizar esa exclusión mutua se presentan en la sección 9.5.

9.2 Relojes

Cuando un sistema informático está compuesto por un solo ordenador resulta trivial la ordenación de sus eventos: basta con acceder al único reloj del sistema y etiquetar ese evento con el instante de tiempo en que suceda. Sin embargo, en un sistema distribuido habrá múltiples ordenadores (cada uno con su propio reloj) pero no existe ningún reloj global que pueda utilizarse para marcar temporalmente

la ocurrencia de los eventos importantes para una aplicación (por ejemplo, para obtener una traza de su ejecución a la hora de depurarla). Desafortunadamente, aunque los relojes actuales son relativamente precisos todavía distan mucho de ser perfectos. Presentan ligeras desviaciones en su progreso respecto a la hora real y eso puede llegar a ofrecer importantes diferencias entre ordenadores distintos, pues los procesadores actuales ejecutan un alto número de instrucciones por cada unidad de tiempo (entre los 3800 *MIPS*[1] de un Intel Atom y los 140000 MIPS de algunos Intel Core i7, por ejemplo). Por ello, aunque la diferencia entre los relojes de dos ordenadores fuera del orden de microsegundos, podría ser demasiado amplia para determinar qué instrucción se ha ejecutado antes en dos ordenadores distintos.

A pesar de estas limitaciones, una primera aproximación para resolver el problema de la ordenación de los eventos de un sistema distribuido consiste en utilizar algún algoritmo distribuido que sincronice los relojes de todos los ordenadores del sistema. La sección 9.2.1 presenta algunos de estos algoritmos.

9.2.1 Algoritmos de sincronización

Antes de describir cómo funciona un algoritmo distribuido de sincronización de relojes es conveniente conocer cómo se implanta el reloj de cada ordenador. Normalmente un ordenador tiene un temporizador (independientemente de cuántos núcleos tenga su procesador) que genera interrupciones con cierta frecuencia y que el sistema operativo programará adecuadamente para llevar una gestión adecuada del instante actual. Así, cuando el ordenador arranque, el sistema operativo averiguará cuál es la hora actual e inicializará de acuerdo con ésta el valor del contador. Por ejemplo, en la mayoría de los sistemas *UNIX* [RT74], el contador recoge el número de "ticks" desde el 1 de enero de 1970 a las 00:00:00 horas y avanza a unos 60 o 100 ticks por segundo.

Si todos los temporizadores utilizados fueran perfectos, en todo instante t, todos los ordenadores de un sistema distribuido tendrían relojes con el mismo valor para su contador local. Sin embargo, los chips de reloj actuales tienen un error relativo de 10^{-5}. Es decir, si el temporizador está programado para generar 60 interrupciones por segundo, debería generar 216000 ticks por hora. Un temporizador convencional generará entre 215998 y 216002 interrupciones.

Los algoritmos de sincronización de relojes tratan de ajustar estos desfases. Estos algoritmos deben afrontar adecuadamente dos problemas:

- Están basados en el intercambio de mensajes entre los nodos que quieran sincronizar sus relojes. La transmisión de un mensaje por la red consumirá tiempo y será imposible medir con exactitud cuánto tiempo se ha invertido en ese

[1] Millones de instrucciones por segundo.

paso, pues los relojes utilizados en el emisor y el receptor todavía no están sincronizados.

- Los relojes utilizados en cada máquina no deberían retroceder nunca. En caso contrario, algunas aplicaciones podrían funcionar mal. Un ejemplo sería la utilidad *make* que examina los instantes de última modificación de una colección de ficheros para emprender ciertas acciones. El uso más habitual de *make* consiste en comparar los instantes de modificación de un fichero fuente y su fichero objeto asociado. Si el fichero objeto es más antiguo, pasa a recompilar el fichero fuente. Si modificásemos el fichero fuente pero el algoritmo de sincronización retrasara el reloj inmediatamente antes de la modificación, podría darse la situación de que su nuevo instante de modificación registrado fuera anterior al que presenta el fichero objeto. Si posteriormente lanzáramos *make*, no lo compilaría. Sin embargo, podría haber otros ficheros de la misma aplicación modificados previamente que sí serían detectados como "no compilados" y que pasarían a compilarse y a enlazarse con los demás. Como resultado tendríamos un fichero ejecutable inconsistente con todos los cambios que hemos efectuado. Quizá pasásemos a ejecutarlo y a observar que falla justo en la rutina que acabábamos de corregir, con lo que invertiríamos un buen rato buscando dónde está el error.

  Para evitar esas situaciones resulta más adecuado no permitir que los relojes se retrasen. Si el reloj local está adelantado N interrupciones, basta con no avanzarlo durante las próximas N (parándolo durante ese intervalo) o bien descartar una de cada M interrupciones durante las próximas M*N (forzando un progreso más lento).

A continuación se describen dos de los algoritmos importantes de sincronización de relojes: el algoritmo de Cristian y el algoritmo de Berkeley.

Algoritmo de Cristian

En el algoritmo de Cristian [Cri89] se asume que existe un ordenador cuyo reloj es preciso[2]. Dicho ordenador actuará como servidor. Los que quieran sincronizarse con él serán sus clientes. Para describir el algoritmo asumiremos que los relojes que mantienen ambos procesos serán C_S en el servidor y C_C en el cliente.

Los pasos que sigue el algoritmo son estos (ilustrados en la figura 9.1, en la que el tiempo avanza hacia la derecha):

1. El cliente pide el valor del reloj al servidor en el instante T0, según el reloj local del cliente (C_C).

[2]Se puede garantizar la precisión del reloj de un determinado ordenador acoplando un receptor especializado. Los más sencillos están basados en el sistema GPS, que para proporcionar servicios de posicionamiento necesita transmitir también la hora, o bien en receptores de la señal horaria transmitida por otros satélites.

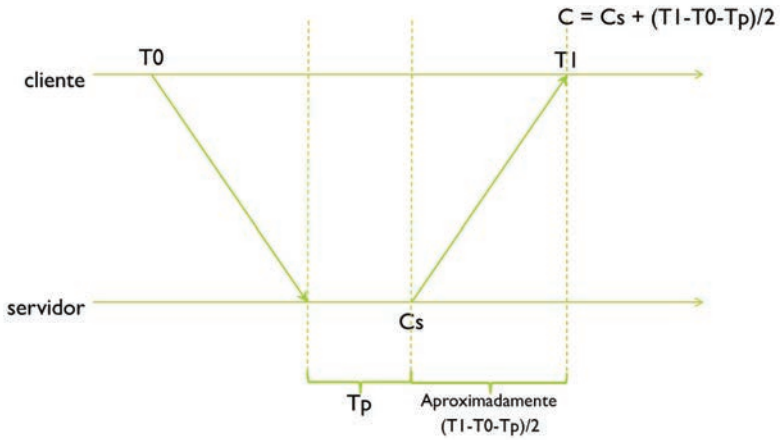

Figura 9.1: Sincronización con el algoritmo de Cristian.

2. El servidor recibe la petición y responde con el valor que tenga su reloj (C_S) justo antes de enviar el mensaje de respuesta.

3. La respuesta llega al cliente en T1 (según el reloj local del cliente).

4. El cliente calcula el valor al que debería asignar su reloj, según la siguiente fórmula:

$$C = C_S + \frac{T1 - T0}{2}$$

   Si además se conociera el tiempo de procesamiento en el servidor (T_p) (tal como se muestra en la figura 9.1), la fórmula pasaría a ser:

$$C = C_S + \frac{T1 - T0 - T_p}{2}$$

5. Si $C > C_C$, entonces el cliente establece su reloj local al valor C. En caso contrario, se descartarán algunos ticks de reloj según el valor de la diferencia $C_C - C$, como ya habíamos descrito previamente.

Obsérvese que este algoritmo asume que el tiempo invertido en la transmisión del mensaje de petición será similar al invertido en la transmisión del mensaje de respuesta. Si la diferencia entre esas dos transmisiones es importante, la sincronización efectuada será imprecisa.

Algoritmo de Berkeley

A diferencia del algoritmo anterior, en el algoritmo de Berkeley [GZ89] no se asume que algún ordenador pueda tener un reloj más preciso que los demás. No obstante, también se escoge a uno de los nodos del sistema para que sea el coordinador del algoritmo y los demás aceptarán sus indicaciones.

El algoritmo asume que se está ejecutando en una red local y que el tiempo de transmisión de los mensajes es suficientemente estable. Utiliza los siguientes pasos:

1. El coordinador difunde periódicamente un mensaje en el que solicita a todos los nodos del sistema el valor de su reloj.

2. Cada uno de los nodos envía su mensaje de respuesta al coordinador.

3. En función del instante de recepción de esa respuesta, el coordinador obtendrá la diferencia entre su reloj local y el de cada uno de los demás nodos, utilizando una fórmula similar a la de Cristian.

   Recuérdese que en esa fórmula se consideran tres instantes ($T0$, C_S y $T1$), que son los que han dado origen a estos tres primeros pasos de este algoritmo y que la precisión en la estimación de la diferencia entre los relojes de los dos procesos depende de los tiempos de transmisión de los dos mensajes intercambiados. En este caso los instantes $T0$ y $T1$ se dan en el nodo coordinador, mientras que el C_S se dará en cada uno de los nodos no coordinadores.

   La descripción original del algoritmo de Berkeley [GZ89] no concreta si en el paso 2 debe devolverse la diferencia entre los dos relojes (tal como se afirma en [Tv08] y en otros textos) o el valor del reloj del nodo no coordinador. Tanto en un caso como en otro el algoritmo funcionaría sin problemas. Esa diferencia no sólo depende de los valores que tuvieran los respectivos relojes en los instantes $T0$ y C_S sino también del tiempo de transmisión de los mensajes, que podría aproximarse conociendo $T1$ (instante en que se recibe la respuesta). El nodo no coordinador no tiene por qué conocer cuánto ha durado la transmisión de la petición del nodo coordinador en el paso 1. Por ejemplo, si asumimos que en el paso 2 se responde con la diferencia entre los valores de los relojes, podríamos encontrar una traza como la siguiente, en la que el nodo A es el coordinador y el nodo B es uno de los que participa en la sincronización:

   a) A difunde su mensaje de petición con el valor $T0$ =13:00:00,000 (asumiendo un formato "hora:minutos:segundos,milésimas").

   b) B recibe ese mensaje cuando su reloj (C_S) marcaba exactamente lo mismo, por lo que retorna que su diferencia es cero (es decir, $C_S - T0 = 0$).

   c) A recibe la respuesta de B en el instante $T1$ =13:00:00,014. Con ello, el valor que debería haber leído B en su reloj habría sido 13:00:00,007,

pero él leyó 13:00:00,000. Por tanto, la diferencia aproximada entre los dos relojes es de -7 milésimas. Es decir, B tiene un reloj retrasado 7 ms. Para que B pueda retornar ya esa diferencia en su respuesta debería haber monitorizado el tiempo de transmisión entre esos dos nodos y tomarlo como referencia para "corregir" su respuesta. Eso también es posible y justifica la variante de este algoritmo presentada en [Tv08].

4. El coordinador calcula la media aritmética de estas diferencias (esto es, las de todas las respuestas) y corrige con ese resultado su propio reloj. Si alguna de las diferencias calculadas en el paso anterior fuera anormalmente alta (en valor absoluto) respecto a las demás, dicho dato sería descartado. Es decir, no se utilizaría para calcular la media.

5. El coordinador calcula las correcciones que tendrá que aplicar cada uno de los demás nodos y envía dicha información a cada nodo para que pueda aplicarla.

   Obsérvese que en este caso no se propaga el nuevo valor del reloj sino cuántos ticks debe adelantarse o retrasarse el reloj de cada receptor. Con ello se evita que el tiempo de propagación de este último mensaje introduzca mayores imprecisiones.

9.2.2 Relojes lógicos

A primera vista, el algoritmo de Berkeley no puede ser tan preciso como el de Cristian. Este último asume que el proceso servidor dispondrá de un reloj preciso y el valor de ese reloj se propagará a todos los demás. En el algoritmo de Berkeley se calcula la media de todos los relojes cuyo valor sea aceptable y todos adoptarán dicho valor medio. Si resultara crítico tener un reloj preciso, el algoritmo de Cristian parece más recomendable. Sin embargo, por lo que respecta a la corrección de los algoritmos y a las tareas que impliquen cierta coordinación entre los procesos de un sistema, poco importará que los valores mantenidos en los relojes de cada nodo coincidan con la hora real o no. Lo que importa es que todos ellos progresen a un mismo ritmo y que sus valores sean lo más cercanos posible entre sí.

Por tanto, el algoritmo de Berkeley ya relaja un poco lo que debemos buscar en un algoritmo de sincronización de relojes. De hecho, bastaría con disponer de algún mecanismo que permitiera ordenar de cierta manera lógica la ocurrencia de todos los eventos importantes en una aplicación distribuida. Eso no tiene por qué ser necesariamente un reloj preciso.

Una primera conclusión que puede extraerse de esta observación es que sólo resultaría necesario sincronizar los relojes de aquellos nodos que interactúen entre sí. Si dos procesos no llegan a intercambiar mensajes, poco preocupará el valor de sus respectivos relojes. Por otra parte, aquellos relojes que debamos sincronizar no tienen por qué coincidir con la hora real. De hecho, bastaría con emplear

contadores, garantizar que dichos contadores nunca retrocedan y que sean capaces de distinguir el instante en que hayan sucedido dos eventos distintos dentro de la ejecución de los procesos de un determinado nodo. Es decir, bastará con utilizar *relojes lógicos* que permitan ordenar los eventos de nuestras aplicaciones distribuidas. Además, ese orden entre dichos eventos solo será relevante cuando exista algún tipo de dependencia entre los eventos.

Leslie Lamport [Lam78] definió la relación "*ocurre antes*" ("*happens before*") para reflejar tales dependencias. Para denotar esta relación se utiliza el operador "$\rightarrow$". Así, la expresión "$a \rightarrow b$" especifica que el evento a ha ocurrido antes que el evento b y significa que todos los procesos están de acuerdo en que el evento a debe aparecer antes que el evento b dentro del orden global.

Esta relación necesita definirse sobre el conjunto de eventos del sistema y cumple las siguientes condiciones:

1. Si a y b son eventos de un mismo proceso, y a se da antes que b, entonces $a \rightarrow b$.
2. Si a es el envío de un mensaje por parte de un proceso y b es la recepción de ese mismo mensaje por otro proceso, entonces $a \rightarrow b$.
3. Transitividad: Si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$.

Dos *eventos* distintos a y b se dice que son *concurrentes* si $a \not\rightarrow b$ y $b \not\rightarrow a$. La expresión $a \| b$ suele utilizarse para denotar que los eventos a y b son concurrentes.

Esta relación establece un *orden parcial* entre los eventos de un sistema. Es decir, no todos los eventos están relacionados entre sí, como demuestra el hecho de que pueda haber eventos concurrentes. También refleja que los eventos importantes para relacionar ejecuciones de distintos procesos son aquellos que intervienen en el intercambio de mensajes.

Todavía no hemos llegado a definir un reloj lógico a partir de esta relación, pero puede hacerse fácilmente. Para ello, Lamport asoció un valor numérico a la ocurrencia de cada evento en el sistema. Así, para un evento a llamó "$C(a)$" al valor del reloj lógico asociado a dicho evento.

Las condiciones que debía cumplir esta función podían extraerse de la propia especificación de la relación "$\rightarrow$". Eran estas:

1. Si a y b son eventos de un mismo proceso, y a se da antes que b, entonces $C(a) < C(b)$.
2. Si a es el envío de un mensaje por parte de un proceso y b es la recepción de ese mismo mensaje por otro proceso, entonces $C(a) < C(b)$.

Obsérvese que si se cumplen estas dos condiciones también se cumplirá la equivalente para la transitividad.

Para satisfacer estas condiciones, se propuso el siguiente algoritmo para asignar valores a los relojes lógicos:

1. Todos los procesos del sistema empiezan con un reloj lógico igual a cero.
2. Cada vez que un proceso p ejecute algún evento, incrementará el valor de su reloj lógico en una unidad.
3. El envío de un mensaje m etiqueta a tal mensaje con el valor de reloj de su emisor (Si p fue su emisor, $C_m = C_p$).
4. Cuando un proceso q reciba un mensaje m en el evento r, actualizará su reloj de la siguiente manera: $C(r) = max(C_q, C_m) + 1$.

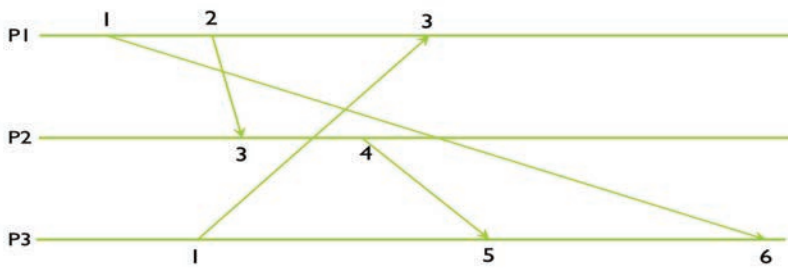

Figura 9.2: Relojes lógicos de Lamport.

La figura 9.2 presenta un ejemplo de ejecución de un sistema con tres procesos en el que se utilizan los relojes lógicos propuestos por Lamport.

Sobre este ejemplo vamos a estudiar qué eventos están relacionados por "$\rightarrow$" y cuáles serían concurrentes. Como no hemos asignado identificadores a los eventos, utilizaremos la notación "*nombre-proceso:*$C(a)$" para referirnos al evento a, pues los relojes lógicos asignados a cada evento sí que se presentan en la figura 9.2.

Con ello, vemos que se respeta lo siguiente (estos serían los "caminos" de dependencias que podríamos observar):

- P1:1 $\rightarrow$ P1:2 $\rightarrow$ P1:3.
- P1:1 $\rightarrow$ P1:2 $\rightarrow$ P2:3 $\rightarrow$ P2:4 $\rightarrow$ P3:5 $\rightarrow$ P3:6.
- P1:1 $\rightarrow$ P3:6.
- P3:1 $\rightarrow$ P3:5 $\rightarrow$ P3:6.

- P3:1 → P1:3.

Además, tendríamos los siguientes eventos concurrentes (en estos casos no existe ningún camino de dependencias que una a estos pares de eventos):

- P1:1 ‖ P3:1
- P1:3 ‖ P2:3
- P1:3 ‖ P3:5
- P2:3 ‖ P3:1
- P1:2 ‖ P3:1
- P1:3 ‖ P2:4
- P1:3 ‖ P3:6
- P2:4 ‖ P3:1

Obsérvese que este ejemplo demuestra de manera práctica que la relación → solo establece un orden parcial, pues se han encontrado varios caminos de dependencias que relacionan a varios eventos pero no se ha podido establecer un único camino que los relacione a todos. Si el orden que espera una aplicación debe ser *total*[3], entonces debemos complementar de alguna manera los valores proporcionados por los relojes lógicos. Lamport [Lam78] también aportó una solución para este problema: basta con añadir a los valores de los relojes, como sufijo (podría entenderse como parte decimal si el valor del reloj fuera un número real o parte de menor peso si fuera un número entero) el identificador del proceso que ha generado cada evento.

El ejemplo presentado en la figura 9.2 quedaría con los valores de reloj que muestra la figura 9.3 al aplicar esta medida. Con ello, ya existe un orden total entre los eventos, fijado por el orden que presentan los valores numéricos de sus relojes. Obsérvese que este orden total no respeta el orden real de ocurrencia de los eventos, pero sí que refleja correctamente las relaciones causa-efecto sugeridas por "→". Su misión se limita a ordenar de alguna manera los eventos concurrentes.

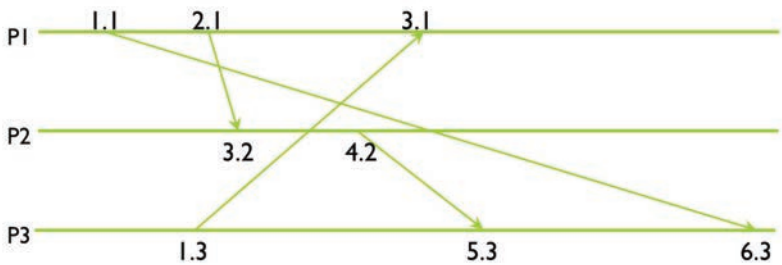

Figura 9.3: Relojes lógicos de Lamport generando un orden total.

[3]Informalmente, un orden total es aquel que es capaz de ordenar a todos los elementos de un determinado conjunto en una misma secuencia. De manera más rigurosa, el orden total se define en la teoría de conjuntos como aquella relación binaria $\leq$ definida sobre un conjunto X tal que $\forall$ a, b, $c \in X$ se cumple: (i) Si $a \leq b \wedge b \leq a$, entonces $a = b$ [Antisimétrica], (ii) Si $a \leq b \wedge b \leq c$, entonces $a \leq c$ [Transitiva], (iii) $a \leq b \vee b \leq a$ [Total].

9.2.3 Relojes vectoriales

Como ya hemos visto, los relojes lógicos son capaces de simplificar enormemente la gestión del tiempo en un sistema distribuido. Muchas aplicaciones podrán utilizarlos para ordenar sus eventos y eso permite diseñar de manera más sencilla otros mecanismos de sincronización.

Desafortunadamente, los relojes lógicos que hemos visto hasta el momento todavía presentan una limitación importante: aunque se conozca el valor del reloj lógico para dos eventos, no siempre se podrá deducir si tales eventos han sido concurrentes o no. De hecho, cuando se da que "$a \rightarrow b$" entonces se puede asegurar que "$C(a) < C(b)$", pero si lo único que sabemos es que "$C(a) < C(b)$" entonces no se puede decidir si "$a \rightarrow b$" o si "$a \| b$". Es decir:

- $a \rightarrow b \Longrightarrow C(a) < C(b)$.
- $C(a) < C(b) \not\Longrightarrow a \rightarrow b$.

En algunas aplicaciones resulta necesario detectar cuándo dos eventos han sido concurrentes. Con los relojes lógicos no podríamos hacerlo y habrá que diseñar otro tipo de relojes que admitan esa detección. Un ejemplo de este tipo de aplicaciones es la detección de operaciones de actualización potencialmente conflictivas en un sistema de ficheros distribuido en el que se repliquen los ficheros. Los problemas que se daban en esa área condujeron al diseño de *relojes vectoriales* [PPR+81, PPR+83] en 1980. Una situación similar se dio en el área de la depuración de aplicaciones paralelas y distribuidas, generando también otras publicaciones [Mat87, Fid88] que propusieron ese mismo mecanismo.

Para poder decidir cuándo dos eventos son concurrentes, los relojes vectoriales reflejan qué eventos generados por otros procesos son conocidos en cada uno de los procesos del sistema. Para ello se necesitará que los relojes sean ahora vectores con tantas componentes como nodos tenga el sistema. Estas componentes se incrementarán utilizando el siguiente algoritmo, en el que asumimos que el sistema tiene N procesos, el algoritmo es ejecutado por un proceso p y su reloj vectorial se llama V_p:

1. El proceso p incrementa $V_p[p]$ cada vez que envía o recibe un mensaje.
2. Un mensaje m enviado por p lleva asociado el valor del reloj de p tras haber aplicado el incremento relacionado con la operación de envío. Es decir, $V_m = V_p$.
3. Al recibir un mensaje m, y tras haber aplicado el incremento citado en el paso 1, el proceso p actualiza su reloj seleccionando el máximo entre el valor local y el valor del reloj contenido en el mensaje para cada una de las componentes. Es decir:

$$\forall i, 1 \leq i \leq N, V_p[i] = max(V_p[i], V_m[i])$$

La clave para trabajar con estos relojes reside en la condición que debe utilizarse para ordenarlos. Es la siguiente:

$$\begin{array}{lll} V(a) \leq V(b) & \Longleftrightarrow & \forall i, 1 \leq i \leq N, V(a)[i] \leq V(b)[i] \\ V(a) < V(b) & \Longleftrightarrow & V(a) \leq V(b) \wedge \exists i, 1 \leq i \leq N, V(a)[i] < V(b)[i] \end{array}$$

Es decir, para que el valor del reloj asociado a un evento a, $V(a)$, se considere menor que el de otro evento b, $V(b)$, cada una de las componentes de $V(a)$ deben ser menor o igual que la respectiva componente de $V(b)$ y además debe haber al menos una de ellas que sea estrictamente menor.

Con estos relojes vectoriales se consigue romper la limitación comentada arriba para los relojes lógicos. Así, con los vectoriales, si $V(a) < V(b)$ entonces se puede asegurar que $a \rightarrow b$.

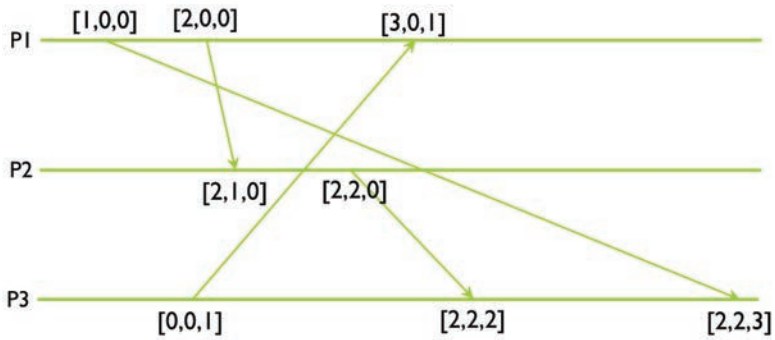

Figura 9.4: Relojes vectoriales.

La figura 9.4 muestra el mismo ejemplo presentado en las anteriores figuras 9.2 y 9.3 utilizando ahora relojes vectoriales. En este caso resulta muy sencillo detectar qué eventos han sido concurrentes entre sí. Las parejas de eventos concurrentes ofrecerían relojes que no serían comparables con las reglas que se acaban de especificar arriba. Serían las siguientes: ([1,0,0], [0,0,1]); ([2,0,0], [0,0,1]); ([3,0,1], [2,1,0]); ([3,0,1], [2,2,0]); ([3,0,1], [2,2,2]); ([3,0,1], [2,2,3]); ([2,1,0], [0,0,1]); ([2,2,0], [0,0,1]).

9.3 Estado global

Existe un buen número de aplicaciones distribuidas que necesitarían conocer el estado global del sistema distribuido, es decir, qué estado presenta cada uno de los procesos que lo forman así como el contenido de los canales de comunicación que estos utilizan. Un ejemplo importante sería una herramienta de depuración, que necesitará consultar ese estado global para comprobar si la aplicación que esté depurándose cumple con sus especificaciones o ha realizado alguno de sus últimos pasos erróneamente. Un segundo ejemplo sería el de los algoritmos que detectasen cuándo el sistema alcanza una *propiedad estable* [CL85], es decir, aquella que pasa a mantenerse una vez se haya alcanzado. Algunos ejemplos de propiedades de este tipo son:

- Que haya un interbloqueo entre algunos de los procesos del sistema. Tras haber detectado una situación así, se debería aplicar un criterio de resolución para eliminar a alguno de los procesos y permitir que la ejecución progrese, como ya vimos en la Unidad 4.
- Que haya finalizado la ejecución de un algoritmo determinado.
- Que desaparezca el token en un determinado algoritmo diseñado para una topología de anillo.

Resulta imposible la captura de una imagen precisa del estado global. Para ello, por ejemplo, se podría confiar en un reloj global y forzar a que los distintos procesos anoten su estado y el de sus canales cuando se llegue a un instante determinado. Como ya hemos visto en la sección 9.2.1, ni ese reloj global existe ni resulta posible sincronizar los relojes de cada nodo con una precisión suficiente. Por tanto, será necesario diseñar algún algoritmo distribuido que tome esa imagen del estado global. Dicho algoritmo sólo podrá recolectar una imagen aproximada, pues además de las restricciones comentadas acerca de la precisión de los relojes, también se utilizarán mensajes para ejecutar dicho algoritmo y los mensajes tendrán un tiempo de transmisión difícilmente predecible o acotable.

Antes de describir las soluciones existentes, habrá que definir de manera más precisa qué es el *estado global* de un sistema. Ese estado global está compuesto por:

- El estado (aquellas variables de interés) de cada uno de los *procesos* que se ejecuten en el sistema.
- El estado de cada uno de los *canales* de comunicación que puedan utilizar tales procesos. Es decir, la secuencia de mensajes enviados por cada canal y que todavía no se hayan podido entregar.

9.3.1 Corte o imagen global

El algoritmo de detección del estado global tendrá que capturar una imagen consistente de cada uno de esos estados (procesos y canales). Esa imagen define un *corte* en una ejecución. Este corte recoge un prefijo en la ejecución local de cada proceso para componer así cierta imagen global de la ejecución del sistema. El objetivo ideal sería que el corte fuera *preciso*, es decir, que finalizaran todas la ejecuciones locales que queremos recoger en el mismo instante real. Para ello, como se ha dicho anteriormente, se necesitaría un reloj global y eso no se puede asumir en un sistema distribuido. La figura 9.5 ofrece un ejemplo de un corte de este tipo en una ejecución distribuida.

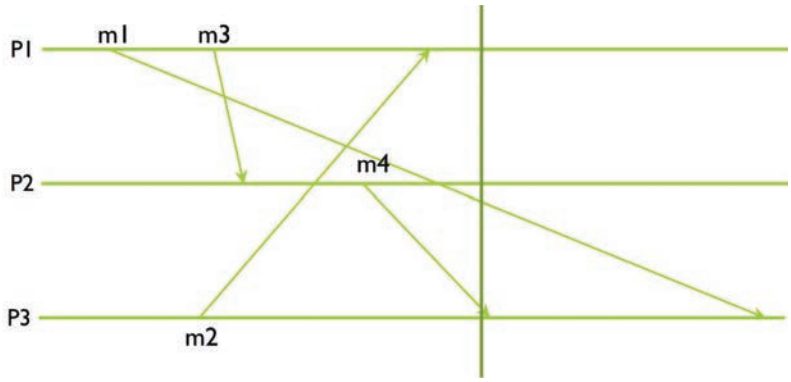

Figura 9.5: Corte preciso de una ejecución.

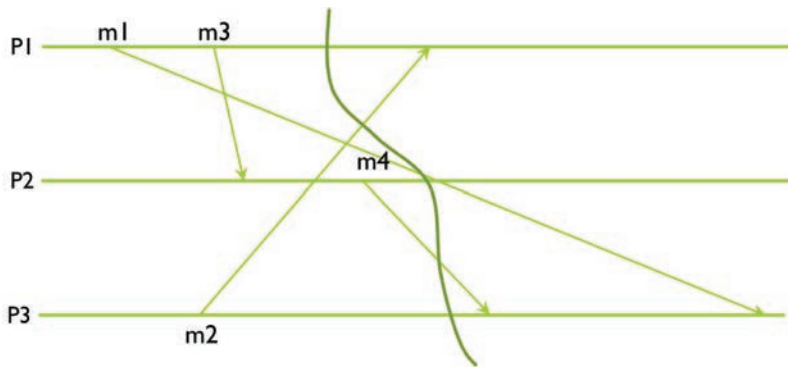

Figura 9.6: Corte consistente de una ejecución.

En lugar de ese grado de precisión, habrá que conformarse con la obtención de una imagen consistente. Es decir, cada proceso podrá finalizar su prefijo para la composición del estado global allí donde pueda. Por tanto, el instante real en que finalizará el prefijo de cada proceso podrá ser distinto. Sin embargo, lo que

sí se garantizará es que en esos estados no se llegue a reflejar la entrega de un mensaje cuyo envío no haya sido también recogido en el estado de su proceso emisor. Por otra parte, sí que podremos recoger también mensajes en tránsito. Es decir, aquellos que hayan sido enviados (y que así refleja el estado de su proceso emisor) pero todavía no entregados. La figura 9.6 muestra un ejemplo de *corte consistente*. En el estado global que se ha obtenido no se recoge la entrega de ningún mensaje cuyo envío no aparezca en la ejecución de su proceso emisor. De hecho, la única recepción que se ha reflejado es la del mensaje $m3$ en el proceso $P2$. Sin embargo, sí hay mensajes en tránsito. Tanto $m1$, $m2$ como $m4$ están en esta situación. Además, el instante en que se cortó la ejecución de $P1$ es anterior al de $P2$ y éste al de $P3$. Esto podría deberse a que fuera $P1$ quien iniciara el algoritmo para recoger el estado global.

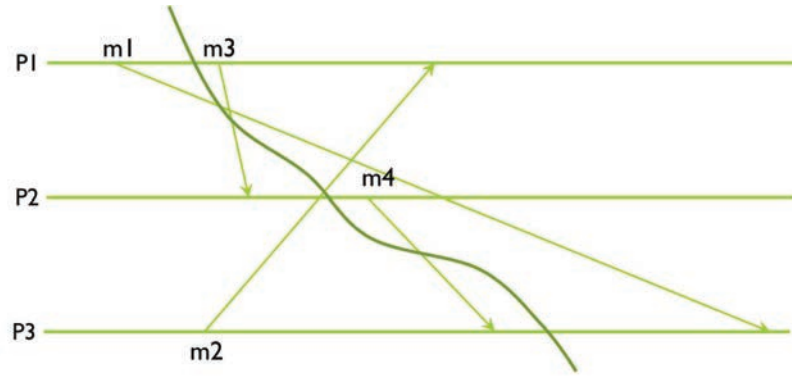

Figura 9.7: Corte inconsistente de una ejecución.

Por último, la figura 9.7 facilita un ejemplo de *corte inconsistente*. En este caso, el algoritmo utilizado no se ha preocupado por comprobar la consistencia en el estado de los canales y ha obtenido una imagen que recoge la entrega de $m3$ y $m4$ pero que no recoge sus eventos de envío en los procesos $P1$ y $P2$, respectivamente.

9.3.2 Algoritmo de Chandy y Lamport

Una solución válida para obtener una imagen consistente del estado global fue propuesta en [CL85]. En esa solución se asume una red con topología completa (esto es, existen canales entre todo par de procesos) cuyos canales son fiables (es decir, no pierden los mensajes que propagan) y transmiten sus mensajes en *orden FIFO*, es decir, que los mensajes que se transmiten por un determinado canal se entregan en el orden en que fueron enviados. Además, se asume que los canales son unidireccionales. Es decir, entre cada par de procesos p y q se asumirá que existe un primer canal de p a q, denominado (p, q), y un segundo de q a p, denominado (q, p).

Cada proceso guardará la imagen de su estado local cuando reciba algún mensaje que le ordene hacerlo. Ese paso no resulta complicado. Resulta más delicado el decidir cuándo debe recogerse el estado de un determinado canal. Para ello debe respetarse lo siguiente: su estado debe estar compuesto por la secuencia de mensajes enviados a través del canal antes de que el estado de su proceso emisor haya sido registrado, pero eliminando aquellos mensajes de dicha secuencia que ya hubieran sido recibidos por su proceso destinatario cuando el estado de este último se registre.

Por ejemplo, consideremos un canal $(p1, p2)$. Asumamos que el proceso $p1$ había enviado los mensajes $m1$, $m2$, $m3$ y $m4$ a través de ese canal antes de registrar su estado. Posteriormente, $p2$ es capaz de recibir $m1$ y $m2$. Tras esto, $p2$ registra su estado. El estado que debemos reflejar en la imagen global para el canal $(p1, p2)$ contendrá, en ese orden, los mensajes $m3$ y $m4$.

El algoritmo propuesto por Chandy y Lamport sigue estos pasos, donde asumimos que existe un proceso p cualquiera del sistema que actuará como el iniciador del algoritmo:

1. El proceso iniciador p guarda su estado local y luego envía un mensaje MARCA a todos los demás procesos por cada uno de sus canales de salida.

2. Cuando un proceso reciba un mensaje MARCA:

   a) Si todavía no ha guardado su estado local, lo guarda y envía MARCA a todos los demás procesos (incluyendo al proceso que le acaba de enviar el mensaje de MARCA).

   Debe anotarse la identidad del proceso iniciador, pues la necesitará en el último paso del algoritmo.

   A partir de este momento almacenará lo que vaya llegando por los canales entrantes.

   b) Si ya había guardado su estado local, pasa a registrar todos los mensajes recibidos por ese canal de entrada y "cierra" ese canal. Es decir, con ello recoge el estado de dicho canal y tal estado contendrá la secuencia de mensajes recibidos desde que guardó su estado local hasta la llegada del mensaje de MARCA por este canal.

3. Un proceso termina cuando ha recibido MARCA por todos sus canales de entrada.

   Cuando termina, envía su estado y el de sus canales al iniciador.

Para ilustrar cómo funciona este algoritmo, realizaremos una traza asumiendo que en la ejecución mostrada en la figura 9.5, el proceso P2 inicia el algoritmo antes de enviar el mensaje $m4$.

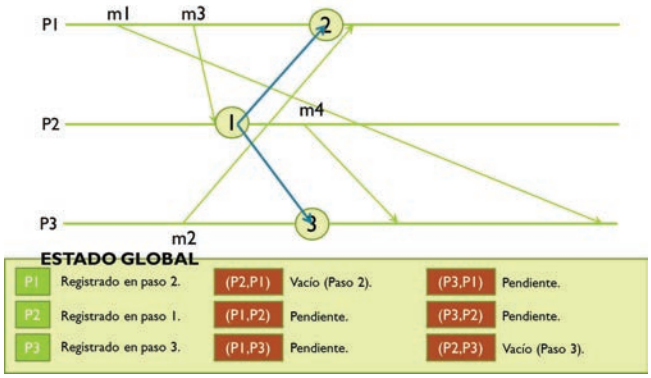

Figura 9.8: Traza del algoritmo de Chandy y Lamport (Pasos 1 a 3).

En la figura 9.8 se observa cómo este proceso difunde los mensajes de MARCA a P1 y P3. La transmisión de esos mensajes se ha representado en color azul y con líneas de mayor grosor para distinguirlos de los mensajes convencionales. El proceso P2 habrá registrado su estado local en el paso 1, en el que realiza la difusión. Por su parte, los procesos P1 y P3 registran su estado local al recibir tales mensajes de marca. A su vez, también registran el estado de los canales de llegada que tengan como emisor al iniciador como vacíos.

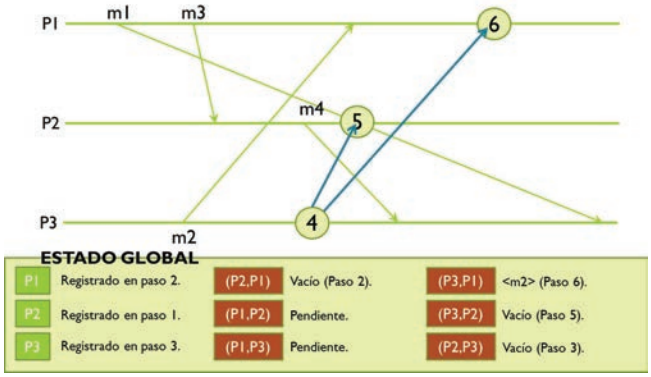

Figura 9.9: Traza del algoritmo de Chandy y Lamport (Pasos 4 a 6).

La reacción del proceso P3 a ese mensaje de MARCA se observa en la figura 9.9. En ella, el proceso P3 envía mensajes de MARCA tanto a P1 como a P2. Entre tanto, P1 habrá recibido el mensaje $m2$ y lo habrá registrado en el estado del canal $(P3, P1)$. Con ello, al recibir el mensaje de MARCA en el paso 6, podrá cerrar dicho canal, donde ya hemos registrado a ese mensaje en tránsito. A su vez, en el paso 5 el proceso P2 ha podido marcar el canal $(P3, P2)$ como vacío.

Por otra parte, la reacción del proceso P1 al mensaje de MARCA del iniciador se observa en la figura 9.10. En ella, el proceso P1 envía mensajes de MARCA tanto a P2 como a P3. Entre tanto, P3 habrá recibido el mensaje $m1$ y lo habrá registrado en el estado del canal $(P1, P3)$. Con ello, al recibir el mensaje de MARCA en el paso 9, podrá cerrar dicho canal, donde ya hemos registrado a ese mensaje en tránsito. Por su parte, en el paso 8 el proceso P2 ha podido marcar el canal $(P1, P2)$ como vacío, pues el mensaje $m3$ ya se había entregado cuando P2 inició el algoritmo.

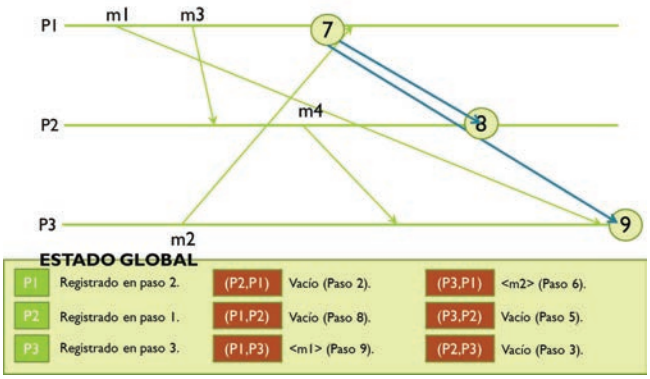

Figura 9.10: Traza del algoritmo de Chandy y Lamport (Pasos 7 a 9).

Obsérvese que los pasos 4, 5 y 6 vistos en la figura 9.9 y los pasos 7, 8 y 9 de la figura 9.10 se han presentado en esa secuencia para facilitar su división en esas dos figuras. El orden real en que pudieron suceder puede ser cualquier otro que respete las siguientes dependencias: $4 \rightarrow 5$, $4 \rightarrow 6$, $7 \rightarrow 8$, $7 \rightarrow 9$. Pero no hay dependencias entre el orden de ocurrencia del evento 7 y del evento 4. Por ejemplo, esta secuencia habría sido admisible: 7, 4, 6, 9, 5, 8. A su vez, también los eventos 2 y 3 son concurrentes entre sí, por lo que esa primera parte de la secuencia que hemos presentado también habría podido ocurrir en el orden 1, 3, 2.

Por último, la figura 9.11 refleja la imagen global que ha llegado a tomar esta ejecución del algoritmo, remarcando los instantes en que se han tomado los estados locales de cada proceso y el hecho de que había dos mensajes en tránsito.

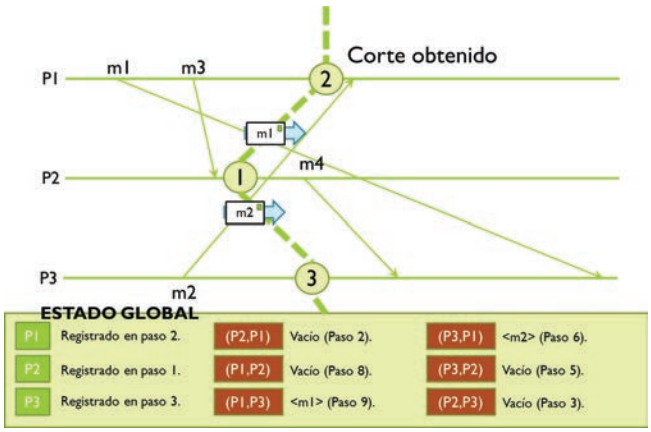

Figura 9.11: Traza del algoritmo de Chandy y Lamport (Estado global).

9.4 Elección de líder

En algunos algoritmos distribuidos resulta necesario seleccionar a uno de los procesos para que coordine las acciones de los demás. De esta manera, tras haber elegido a ese proceso *líder* o *coordinador* se puede simplificar la gestión de otras acciones posteriores en ese mismo algoritmo o aplicación. Si existe un proceso coordinador éste puede dirigir a los demás en caso de que existan múltiples alternativas para superar una determinada acción. Para ello, bastará con una ronda de mensajes que difunda la alternativa a seguir. Si no existiera un proceso coordinador, todos los procesos deberían llegar a un acuerdo y eso exigiría múltiples rondas de mensajes para que algunos procesos realizaran alguna propuesta y entre todos se votara y decidiera cuál es la mejor.

Para que la elección de líder sea posible, los procesos participantes deben tener identificadores distintos y cada uno de ellos debe conocer la identidad del resto de los participantes. Si no fuera así no habría manera de efectuar ninguna elección, pues generalmente se utiliza el mismo algoritmo en todos los procesos y no se podría distinguir a un proceso de los restantes.

Si cada proceso puede conocer la identidad de todos los demás procesos del sistema resulta muy sencillo seleccionar a uno de ellos. Basta con que todos apliquen un mismo criterio (p. ej., que sea líder el que tenga el identificador más alto) para efectuar tal selección. Sin embargo, podrá aparecer un segundo problema: habría que comprobar con cierta frecuencia si los procesos siguen activos o han llegado a fallar. Estas primeras soluciones integraban esa comprobación de fallos en el propio algoritmo de elección. Las secciones 9.4.1 y 9.4.2 presentan dos ejemplos de este tipo.

Actualmente existen algunos componentes que proporcionan *servicios de comunicación a grupo* [CKV01] y que pueden utilizarse en cualquier aplicación distribuida. Aunque la misión principal de estos servicios es proporcionar una interfaz para difundir mensajes a todos los procesos que formen un grupo, también ofrecen una interfaz para consultar en cada momento qué procesos integran el grupo en cada momento. Es decir, se preocupan por monitorizar el estado de todos los participantes. Con ello, proporcionan ya la base suficiente para que los procesos de una determinada aplicación puedan seleccionar un proceso coordinador sin necesidad de utilizar un algoritmo especializado.

Sin embargo, para que un servicio de pertenencia pueda proporcionar una información idéntica a todos los procesos de la aplicación, internamente tendrá que ejecutar algún algoritmo de *consenso*. Es decir, todos los procesos deben llegar a un acuerdo sobre quién integra al grupo. En su momento se demostró que en un sistema distribuido asincrónico (aquel en el que cada proceso pueda progresar a un ritmo distinto y en el que no se pueda acotar el tiempo de transmisión de los mensajes), donde los procesos puedan fallar no se podrá resolver el problema de consenso [FLP85].

A primera vista el problema de consenso implícito en un servicio de pertenencia sería irresoluble, pues normalmente los sistemas suelen ser asincrónicos y los procesos llegan a fallar. Precisamente, esto último era lo que motivaba el diseño de los servicios de pertenencia: se quería conocer qué procesos seguían funcionando. Para romper dicha imposibilidad se propuso una herramienta complementaria: los *detectores de fallos* [CT96], que introducían el grado de sincronía [LFA04] suficiente para resolver el problema de consenso.

Estas soluciones actuales son más precisas y generales que los primeros algoritmos. Sin embargo, son bastante más complejas y su estudio requeriría más detalle del que se pretende abordar con esta unidad didáctica. Por ello, en esta sección únicamente se describen algunos de los algoritmos clásicos para elegir un proceso coordinador.

9.4.1 Algoritmo "Bully"

En el algoritmo "Bully" [GM82] se asume que la comunicación es fiable y que se conoce el tiempo de transmisión de los mensajes. Este algoritmo es iniciado por un proceso que no reciba respuesta por parte del coordinador en alguno de los pasos de otros algoritmos, o bien por un proceso que finalice su recuperación tras un fallo previo. Consta de los siguientes pasos:

1. El proceso iniciador envía un *mensaje ELECCIÓN* a todos los nodos *con identificador superior al suyo.*

2. Si no responde nadie, él será el nuevo líder y difundirá un *mensaje COORDINADOR* para comunicar este hecho al resto.

3. Si alguien respondiera, el proceso no hace nada más. No será el coordinador.

4. Cuando un proceso reciba ELECCIÓN, envía un *mensaje OK* como contestación a quien se lo envió. De esta forma, le avisa de que él también participa y le ganará en su intento de ser líder.

   Además, este nuevo proceso participante inicia a su vez otra ejecución de este algoritmo.

9.4.2 Algoritmo para anillos

Este algoritmo asume que todos los procesos están dispuestos en un anillo lógico y enviarán sus mensajes a través de los canales que definen tal anillo.

Para seleccionar un proceso coordinador mediante este algoritmo se hace lo siguiente:

1. El proceso iniciador prepara un *mensaje ELECCIÓN* en el que incluye su propio identificador dentro de un campo que recogerá el conjunto de nodos participantes y en un segundo campo que mantendrá el identificador del proceso iniciador. El iniciador enviará dicho mensaje al siguiente proceso dentro del anillo, esperando su confirmación (es decir, que éste le retorne un mensaje ACK indicando que ha recibido correctamente el mensaje).

   Si la confirmación no llegara en el tiempo previsto, se reenvía de nuevo el mensaje al siguiente proceso del anillo. Esto se hará sucesivamente, hasta que alguno de los sucesores en el orden del anillo responda. Obsérvese que esto constituye el mecanismo utilizado por este algoritmo para detectar los fallos de los participantes.

2. Cada proceso que reciba y confirme el mensaje, verificará si el campo iniciador contiene su identificador. De no ser así, incluirá su propio identificador en el conjunto de participantes y lo reenvía al siguiente siguiendo el procedimiento explicado en el paso anterior.

3. Cuando el proceso iniciador reciba de nuevo el mensaje ELECCIÓN, el conjunto que ahora mantiene contendrá a todos los procesos participantes. El iniciador seleccionará al proceso de ese conjunto que tenga el identificador mayor, construirá un *mensaje COORDINADOR* que contendrá el identificador del nuevo líder y propagará dicho mensaje a través del anillo, para que todos los procesos activos sepan qué proceso ha sido elegido.

Aunque el algoritmo haya sido iniciado por múltiples procesos simultáneamente, no habrá ningún problema. Todos ellos elegirían a un mismo proceso como coordi-

nador, ya que todos detectarían el mismo conjunto de nodos participantes y, por tanto, seleccionarían el mismo valor máximo como identificador del líder.

9.5 Exclusión mutua

Como ya se ha visto en las primeras unidades de este libro, uno de los problemas a resolver cuando se implantan programas concurrentes es garantizar que las *secciones críticas* existentes en diferentes componentes del programa se ejecuten en *exclusión mutua*. Es decir, que únicamente las ejecute un solo proceso o hilo en cada momento.

Existen algunas soluciones clásicas a este problema. Están descritas en las próximas secciones.

9.5.1 Algoritmo centralizado

En la primera solución se utiliza un proceso coordinador (que ha podido elegirse utilizando alguno de los algoritmos presentados en la sección 9.4). El algoritmo resultante sigue estos pasos:

1. Cuando un proceso deba acceder a su sección crítica enviará un *mensaje SOLICITAR* al proceso coordinador indicando que pretende acceder a la sección crítica.

2. Si el coordinador observa que la sección crítica está libre actualmente, responderá con un *mensaje CONCEDER*. De esa manera el proceso solicitante reanudará su ejecución y ejecutará su sección crítica.

   Por el contrario, si el coordinador sabe que la sección crítica está ocupada por otro proceso, se anotará la identidad del solicitante y no responderá. Como el proceso solicitante espera esa respuesta, de momento permanecerá suspendido.

3. Cuando un proceso finalice la ejecución de su sección crítica, enviará un *mensaje LIBERAR* al coordinador, reportando tal finalización. El coordinador comprobará si existe algún proceso suspendido esperando permiso para entrar en la sección. Si hubiera alguno, seleccionará alguno de ellos (por ejemplo, el que hubiese realizado la solicitud en primer lugar) y responderá a su solicitud con un mensaje CONCEDER.

   Si no hay ningún solicitante bloqueado, el coordinador se anotará que la sección crítica queda libre. De esta manera, el primer nuevo solicitante podrá acceder sin necesidad de suspenderse.

Este algoritmo centralizado presenta la ventaja de que requiere pocos mensajes para completar la gestión del acceso a una sección crítica. Basta con tres mensajes por cada proceso (SOLICITAR, CONCEDER y LIBERAR).

También se puede justificar de manera sencilla su buen funcionamiento. El coordinador sabe si la sección crítica está libre u ocupada y en función de ello sólo permite que un único proceso la ejecute en cada momento.

Desafortunadamente, también tiene algunos puntos débiles. La comunicación debe ser fiable, pues la pérdida de algún mensaje puede tener efectos graves. Por ejemplo, la pérdida de un mensaje SOLICITAR implicaría que el solicitante correspondiente permaneciera suspendido indefinidamente. A su vez, si se pierde un mensaje CONCEDER tendremos un efecto semejante para el proceso solicitante, pero además el proceso coordinador creería que la sección libre estaría ocupada y no permitiría que ningún otro proceso accediera a ella. Por tanto, esa sección quedaría bloqueada de manera indefinida. Si se perdiera el mensaje LIBERAR también mantendríamos el bloqueo sobre la sección, aunque en ese caso el proceso liberador continuaría su ejecución sin problemas.

Otro problema surge con la caída del proceso coordinador pues se perdería toda la información relacionada con el estado actual de la sección y la identidad de su proceso propietario, en caso de que hubiese alguno y los mensajes que deben esperarse para proseguir con una gestión adecuada. Por tanto, el proceso coordinador constituye un *punto único de fallo* y debería replicarse para garantizar la continuidad de este servicio.

9.5.2 Algoritmo distribuido

Aunque una primera versión de algoritmo distribuido de exclusión mutua ya se sugirió en [Lam78], posteriormente fue optimizada en [RA81]. En ambos algoritmos se utilizaban los relojes lógicos de Lamport, complementados con los identificadores de los nodos (tal como se explicó en la sección 9.2.2, con un ejemplo en la figura 9.3) para implantar un orden total en el etiquetado temporal de los eventos.

El algoritmo utilizado en [RA81] seguía estos pasos:

1. Cuando un proceso quiera acceder a la sección crítica difundirá un *mensaje TRY* a todos los procesos del sistema.
2. Cuando un proceso reciba un mensaje TRY, actuará de la siguiente forma:
   - Si no está en su sección crítica ni intentaba entrar, responderá con un *mensaje OK*.
   - Si está en su sección crítica, no contesta, pero encola el mensaje.

- Si no está en su sección crítica, pero quiere entrar, compara el número de evento del mensaje entrante con el que él mismo envió al resto. Vence el número más bajo. Así, si el mensaje entrante es más bajo, el receptor responde OK. Si fuera más alto, no responde y encola el mensaje.

3. Un proceso entra en la sección crítica cuando recibe OK de todos.

   Cuando abandone la sección, enviará OK a todos los procesos que enviaron los mensajes que retuvo en su cola.

Es fácil justificar que el algoritmo funciona correctamente. Si sólo hubiese un proceso interesado en acceder a la sección crítica, todos los demás le responderían con el correspondiente OK y accedería sin problemas. En caso de que haya más de un peticionario, aquellos procesos que no hayan solicitado la entrada responderán a todos los solicitantes. El conflicto entre los solicitantes (un posible empate no resuelto bloquearía a todos ellos) se resuelve gracias a los relojes lógicos utilizados. Como estos están complementados con los identificadores de los procesos emisores, jamás podrá haber un empate. Aquel proceso que haya difundido un TRY con el valor de su reloj lógico más bajo será el único que podrá acceder a la sección.

Por motivos similares a los discutidos en el algoritmo centralizado, es básico que la comunicación sea fiable. En caso contrario podría bloquearse alguno de los procesos solicitantes y, a su vez, evitar que otros solicitantes posteriores accedieran alguna vez a la sección crítica.

9.5.3 Algoritmo para anillos

En el algoritmo para anillos, todos los procesos participantes se estructuran en un anillo lógico y cada uno de ellos conoce la identidad y dirección del siguiente proceso del anillo.

El algoritmo está basado en la existencia de un token que va circulando por el anillo de la siguiente manera:

1. Un proceso no puede iniciar la ejecución de su sección crítica mientras no reciba el token.

2. Al recibir el token, si se quería acceder a la sección ya se podrá ejecutar ésta. El token se mantendrá mientras no finalice la ejecución de la sección crítica. Cuanto termine, se pasará al siguiente proceso.

   Si el proceso no quería acceder a su sección cuando recibió el token, lo pasará inmediatamente al siguiente proceso en el anillo.

Este algoritmo funciona correctamente mientras no fallen los procesos que participan en él y no se duplique involuntariamente el token. Es obvio que garantizará ex-

clusión mutua, pues solo debería existir un token y únicamente aquel proceso que lo mantenga podrá ejecutar su sección crítica.

Cuando un proceso falle habría que comprobar lo antes posible si tenía el token o no. No basta con esperar algún tiempo para que "aparezca" el token, pues resulta impredecible cuánto le costará a cada proceso ejecutar su correspondiente sección. Por ello, sería conveniente utilizar un algoritmo para obtener una imagen consistente del estado global, como el descrito en la sección 9.3.2.

9.5.4 Comparativa

Las características principales de los tres algoritmos que acabamos de describir están resumidas en la tabla 9.1, asumiendo un sistema en el que existen n procesos.

Algoritmo	Mensajes por sección	Rondas para acceder	Problemas
Centralizado	3	2	Caída del coordinador.
Distribuido	2(n-1)	2(n-1)	Caída de cualquier proceso.
Anillo	1 a ∞	0 a n-1	Pérdida del token

Tabla 9.1: Comparativa de algoritmos de exclusión mutua.

La primera columna analiza cuántos mensajes debe intercambiar el proceso solicitante con los demás para llegar a solicitar, ejecutar y liberar su sección crítica. El algoritmo centralizado es el más sencillo en este apartado, pues únicamente requiere tres mensajes (un mensaje hacia el coordinador y su respuesta para conseguir la entrada y después un mensaje de liberación al coordinador para terminar la sección). Por su parte, el algoritmo distribuido necesita que el solicitante envíe su mensaje de petición a cada uno de los demás procesos (n-1 mensajes TRY emitidos por el solicitante) y que los demás aprueben su petición (n-1 mensajes OK emitidos por los demás procesos). Cuando finalice la ejecución de la sección, el proceso también llega a enviar algún mensaje OK para desbloquear a otros solicitantes, pero este número sería variable (oscila entre cero y n-1) y depende de la contención que llegue a haber. Por último, en el algoritmo para anillos, bastará con un solo mensaje si todos los procesos están siempre interesados en acceder a la sección cuando llega su correspondiente turno. Sin embargo, si las secciones críticas fueran muy pequeñas y los procesos rara vez solicitaran su ejecución el token podría estar horas dando vueltas por el anillo. Por tanto, no existe cota superior para el número máximo de mensajes que podrían necesitarse.

La segunda columna analiza la pausa que debe superarse (en rondas de mensajes que llegan a intercambiarse) desde que un proceso solicita su sección crítica hasta que finalmente se aprueba su acceso. En el algoritmo centralizado basta con dos rondas: la petición al coordinador y la respuesta de éste. Si asumiéramos que no

transita más de un mensaje a la vez por la red del sistema, el algoritmo distribuido requeriría 2(n-1) rondas, pues deben propagarse n-1 mensajes TRY y deben recibirse n-1 mensajes OK para que un proceso acceda a su sección. Finalmente, el algoritmo para anillos necesitaría un mínimo de cero rondas (si tenemos la suerte de recibir el token justo cuando se intentaba acceder a la sección) y un máximo de n-1 (si acabamos de enviar el token justo antes de solicitar la entrada).

Por último, la columna final resume en qué condiciones habrá problemas en el progreso del algoritmo. En todos los casos estamos asumiendo comunicación fiable, pues resulta necesaria en los tres algoritmos. En el algoritmo centralizado sólo tendríamos dificultades ante el fallo del proceso coordinador, pues nos obligaría a elegir otro y reconstruir su estado. Si fallara el proceso que ahora estaba ejecutando la sección crítica, el coordinador conoce su identidad y podría otorgar la entrada al siguiente solicitante (si hubiera alguno) o considerarla libre (si no hubiera ningún solicitante más). En el algoritmo distribuido lo tenemos bastante más difícil pues el fallo de cualquiera de los procesos del sistema impide que el algoritmo progrese (pues tal proceso caído será incapaz de conceder su permiso a las solicitudes venideras, bloqueando a esos solicitantes). Por último, el algoritmo para anillos planteará problemas cuando caiga el proceso que ahora mismo mantenía el token. Es una situación difícil de detectar, como ya hemos explicado en la sección 9.5.3.

Por tanto, el algoritmo más robusto de entre los tres presentados es el centralizado. Esto parece contradecir el principio de descentralización que se había explicado en unidades anteriores. No obstante, es lógico, pues los algoritmos de exclusión mutua tratan de imponer una coordinación entre los procesos interesados en la ejecución de una sección crítica y tanto las necesidades de comunicación como la complejidad de los algoritmos se reducen en estos casos si se elige un proceso coordinador, como ya explicamos en la sección 9.4.

9.6 Resumen

La sincronización en los sistemas distribuidos resulta mucho más compleja que en los sistemas centralizados, especialmente en el acceso a los recursos compartidos y la ordenación de eventos. Así, mientras que en los sistemas centralizados se emplea un reloj único, en los sistemas distribuidos cada ordenador tiene su propio reloj y pueden haber divergencias en las velocidades de cada uno de ellos. Por tanto, se requiere sincronizar los relojes de diferentes nodos para tener un reloj común. En esta unidad se han revisado los algoritmos de sincronización de relojes más relevantes (en concreto, los algoritmos de Cristian y de Berkeley), en los cuales un ordenador actúa de servidor y los demás se sincronizan con él.

Para muchas aplicaciones es suficiente con que exista acuerdo con el orden global en que ocurren los eventos y no el tiempo real en que suceden. En estos casos, se emplean relojes lógicos, que marcan el instante en que ocurren los eventos, asociando un valor a cada evento. En esta unidad hemos visto el algoritmo de Lamport para relojes lógicos, que permite ordenar totalmente los eventos de un sistema distribuido. Una mejora de dicho algoritmo consiste en el empleo de relojes vectoriales, construidos de forma que cada nodo mantiene un vector de marcas temporales con los eventos ocurridos en los otros nodos. Los relojes vectoriales permiten determinar el orden causal entre dos eventos, si se conocen los valores de sus registros de tiempo lógicos.

Conocer el estado global de un sistema distribuido, es decir, el estado local de cada proceso y el estado de cada canal (mensajes enviados y todavía no entregados) permite detectar la terminación de los procesos y realizar depuraciones distribuidas. Un algoritmo de detección del estado global deberá capturar una imagen consistente de cada uno de los estados de los procesos y canales. Esa imagen define un corte de la ejecución de los procesos. Si se realiza un corte consistente, se reflejarán estados consistentes donde todas las entregas de mensaje tienen antes sus envíos correspondientes. El algoritmo de Chandy-Lamport, estudiado en esta unidad, permite obtener una imagen consistente del estado global.

En algunos algoritmos distribuidos resulta necesario seleccionar a uno de los procesos para que coordine las acciones de los demás. Así, la elección de un proceso líder o coordinador permitirá simplificar la gestión de otras acciones posteriores que sean necesarias realizar, como por ejemplo la selección entre múltiples alternativas para realizar una determinada acción. En esta unidad se han descrito algunos de los algoritmos clásicos para elegir un proceso coordinador. En concreto, hemos visto cómo funciona el algoritmo Bully (o algoritmo del abusón) y el algoritmo para anillos. En ambos casos, se escogerá como líder al que tenga el identificador más alto, de entre los nodos o procesos participantes.

Finalmente, uno de los problemas importantes a resolver cuando se implantan programas concurrentes consiste en garantizar que las secciones críticas existentes en diferentes componentes del programa se ejecuten en exclusión mutua. En esta unidad se han revisado algunas soluciones clásicas de algoritmos de exclusión mutua distribuida. Estos algoritmos garantizan que, en una colección de procesos distribuidos, al menos un proceso tenga en cierto momento acceso a un recurso compartido. En concreto, hemos comparado tres soluciones: un algoritmo centralizado (en el que un proceso coordinador es quien gestiona las solicitudes de acceso a secciones críticas), un algoritmo distribuido (donde se hace uso de los relojes lógicos de Lamport para resolver el conflicto entre las solicitudes de acceso a una misma sección crítica), y un algoritmo para anillos (donde un token recorre el anillo y un proceso debe recibir el token para acceder a la sección crítica).

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Describir los problemas que comporta la gestión del tiempo en un entorno distribuido.
- Identificar las ventajas introducidas por una ordenación lógica de los eventos en una aplicación distribuida.
- Ilustrar las soluciones clásicas para algunos problemas de sincronización en un entorno distribuido: imagen del estado global, elección de líder, exclusión mutua.

Unidad 10

GESTIÓN DE RECURSOS

10.1 Introducción

Como ya se comentó en la Unidad 7, uno de los objetivos importantes que todo sistema distribuido tendrá que cumplir es facilitar el acceso a los recursos del sistema. En cualquier aplicación distribuida habrá un buen número de recursos que sus componentes tendrán que utilizar: ficheros, dispositivos, componentes remotos, nodos del sistema, etc. Una de las gestiones básicas a realizar en estas aplicaciones será la asignación de nombres a estos recursos cuando la aplicación los genere, así como la obtención de sus puntos de entrada para utilizarlos posteriormente. En esta unidad analizaremos esas gestiones básicas, necesarias para que los distintos componentes de una aplicación puedan interactuar entre sí y puedan acceder a diferentes recursos tanto locales como remotos.

No todos los componentes de una determinada aplicación distribuida necesitarán un nombre para que otros componentes puedan utilizar sus operaciones. Por ejemplo, en una aplicación *Java RMI*[Ora12f] sus módulos serán objetos y algunos de ellos tendrán que estar dados de alta en el registro; es decir, tendrán un nombre. Esos objetos registrados pueden comportarse como "factorías" de otros objetos y devolverlos como argumentos de salida en algunas de las operaciones de sus interfaces. Esa será otra manera de proporcionar acceso a objetos remotos: que sus "referencias" se retornen como argumentos de salida.

Aunque exista esa segunda manera de acceder a componentes remotos, el mecanismo básico para ello es el uso de un servicio de nombres en el que se debe registrar el nombre y dirección de cada componente. En este tema se describirán estos servicios de nombres. Para ello, esta unidad describe algunos conceptos básicos sobre los servicios de nombres en la sección 10.2. Posteriormente, la sección 10.3

describe el subservicio de localización, que gestiona direcciones e identificadores, permitiendo una gestión más fácil de aquellos componentes que puedan cambiar frecuentemente su nombre. A su vez, la sección 10.4 describe el servicio DNS como un ejemplo de servicio de nombres con organización jerárquica. Para terminar, la sección 10.5 presenta LDAP, un servicio de nombres basado en atributos en el que, además del nombre, podremos mantener y estructurar información adicional sobre cada entidad gestionada.

10.2 Nombrado

Una gestión básica en cualquier sistema operativo, tanto centralizado como distribuido, es la asignación de nombres a los recursos que gestione. Esto permite que los usuarios (y las propias aplicaciones) puedan acceder a tales recursos cuando conozcan sus nombres. Veamos seguidamente algunos conceptos básicos relacionados con la gestión de los nombres.

10.2.1 Conceptos básicos

Llamaremos *entidad* a todo elemento o recurso de un sistema distribuido que ofrezca una interfaz de operaciones que pueda ser utilizada por otros elementos del sistema. Habrá múltiples clases de entidades: procesos, ficheros, dispositivos, herramientas de sincronización, buzones, semáforos, procedimientos públicos accesibles mediante RPC, objetos accesibles mediante RMI,...

Para que una entidad pueda ser accedida habrá que publicar su nombre en algún servidor. Este *nombre* será una cadena de caracteres. Su valor debería ser fácilmente legible y memorizable por el usuario, que lo utilizará para referirse a la entidad.

Como ya hemos comentado, las entidades deben proporcionar cierta interfaz de operaciones. Esas operaciones son accesibles a través de algún punto de entrada. Ese punto de entrada puede verse también como una entidad más cuyo nombre (es decir, aquello necesario para referirse al punto de entrada) será una *dirección*. En la práctica, a la hora de gestionar los accesos sobre las entidades no se entra en detalle y se obvia la existencia del punto de entrada. Eso implica que en lugar de hablar sobre direcciones de los puntos de entrada de una entidad, hablamos simplemente de las direcciones de la entidad.

Acabamos de decir que la dirección es el nombre del punto de entrada. También que el punto de entrada pertenece a la entidad. Por tanto, una dirección ya es un nombre para una entidad. ¿Por qué no las utilizamos como tales? Sencillamente, porque no son fácilmente utilizables para que un usuario haga referencia a la entidad. Suelen ser una secuencia de dígitos cuyo valor no proporciona ninguna

sugerencia sobre la identidad de la entidad referenciada, y tampoco son fáciles de recordar.

Por otra parte, los puntos de entrada de una determinada entidad pueden cambiar con facilidad. Normalmente los puntos de entrada están ligados a la máquina en la que resida la entidad. Por ello, si se migra la entidad, los puntos de entrada cambian y sus direcciones asociadas también. Por el contrario, los nombres que hayamos asignado a las entidades no deberían cambiar aunque éstas se trasladaran.

Por último, obsérvese que una misma entidad puede tener múltiples puntos de entrada y, por tanto, múltiples direcciones. Para presentar un ejemplo no informático, tomemos a una persona como ejemplo de entidad. Un posible punto de entrada (algo que permita comunicarse con ella) sería el teléfono. Una misma persona puede tener múltiples números de teléfono (móvil, fijo en el hogar, fijo en el trabajo,...) y con el transcurso del tiempo esos teléfonos podrían cambiar (al cambiar de trabajo, por ejemplo). Sin embargo, siempre debería mantener un mismo nombre, independiente de los puntos de entrada y direcciones necesarios para acceder a esa "entidad".

Por desgracia, habrá casos en que los nombres resultarán ambiguos, pues solemos utilizar cadenas cortas como nombres y será sencillo que alguna de esas cadenas haya sido registrada por distintos usuarios para referirse a entidades diferentes (¿Cuántas personas se llaman *John Smith* en los países de lengua inglesa?). Debido a esto, conviene que el sistema utilice internamente identificadores para referirse a las entidades.

Un *identificador* es un nombre con las siguientes propiedades:

- Referencia como máximo a una entidad.
- Cada entidad está relacionada como máximo con un identificador.
- No es reutilizable. No podrá referirse a otra entidad cuando la entidad inicialmente relacionada con él desaparezca.

De esta manera, el sistema distribuido usará internamente algún tipo de identificador para referirse a las entidades, aunque dichos identificadores rara vez son públicos. Esto ya sucedía en los sistemas operativos utilizados como base. Por ejemplo, en los sistemas UNIX se utilizan los números de nodos-i para identificar a los ficheros creados en un determinado dispositivo de almacenamiento. Sin embargo, los usuarios no utilizan directamente estos números (pues serían excesivamente difíciles de recordar), sino los nombres. Esos nombres se asocian a los ficheros en las entradas de directorio, donde basta con mantener la correspondencia entre los nombres y los números de nodo-i.

10.2.2 Espacios de nombres

Cuando un usuario (o el sistema operativo) cree una nueva entidad, tendrá que asignarle un nombre. Para que dicha asignación tenga éxito, el nombre que vaya a utilizarse debería ser nuevo; esto es, no debería estar ya asignado a otras entidades en ese mismo ámbito de nombrado.

Un *espacio de nombres* define el contexto o ámbito en el que se crearán y utilizarán los nombres. Como generalmente las aplicaciones distribuidas admiten que sus componentes sean utilizados por procesos remotos, no tiene sentido la utilización de un espacio de nombres "plano", en el que todos los nombres asignados a todas las entidades compartieran un único contexto de nombrado. Obsérvese que en ese caso, para evitar colisiones a la hora de generar nuevos nombres, estaríamos obligados a utilizar cadenas de caracteres extremadamente largas y la probabilidad de error a la hora de escribir tales nombres sería bastante alta.

Por ello, la utilización de un espacio de nombres jerárquico es la opción más sensata y escalable. Cada nivel de la jerarquía establece subespacios de nombres donde resultará posible asignar nombres a un número moderado de entidades. Así, la búsqueda y resolución de nombres podrá:

- Distribuirse entre múltiples servidores, cada uno responsable de uno de esos subespacios.
- Ser más eficiente, pues cada servidor no tendrá por qué reservar grandes cantidades de memoria ni realizar costosas operaciones de indexación.
- Ser más robusta, pues la caída de uno de estos servidores no implicará dejar a todo el sistema sin servicio.

Para modelar un espacio de nombres jerárquico se representará a éste como un árbol. Véase un ejemplo en la figura 10.1. Cada nodo hoja de esa jerarquía representaría a una entidad y mantendría algunos atributos de ésta (como mínimo su dirección). Los nodos internos (los que no sean hojas) representarán a los *directorios* (que son un tipo especial de entidad dedicado a asignar nombres al resto de entidades) y las aristas dirigidas que los conecten a otros nodos hojas o a otros directorios de nivel inferior dentro de la jerarquía estarán etiquetados con los nombres asignados a tales entidades o directorios. Estas aristas parten del nodo que actúa como directorio y llegan al nodo que represente a la entidad a nombrar (que, a su vez, podría ser otro directorio).

Para garantizar que no haya ambigüedad, todo nodo de este árbol tendrá asignado un identificador, conocido por el servicio de nombres. En la práctica, los nodos que modelan a los directorios pasan a implantarse como tablas cuyas entradas contienen dos campos:

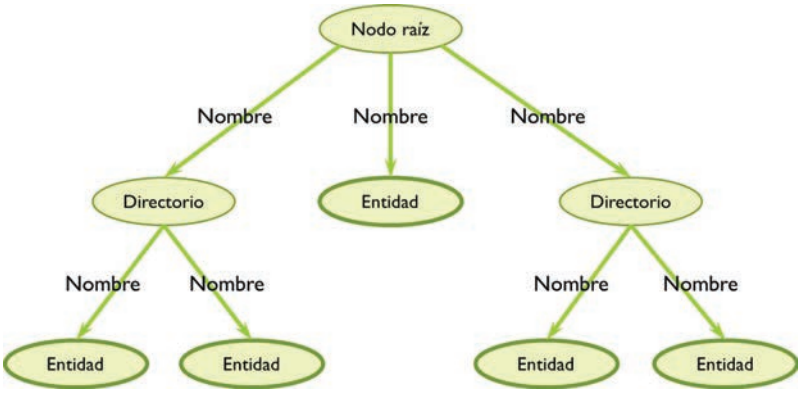

Figura 10.1: Directorios y entidades en un espacio de nombres.

- El nombre asignado a la entidad. Es decir, la cadena que hemos asociado a la arista que relacionaba a ese nodo interno (el directorio) con el nodo que modelaba a la entidad.
- El identificador de la entidad nombrada.

El único nodo del árbol que solo tenga aristas salientes y ninguna arista entrante será el nodo raíz o *directorio raíz*.

En un mismo directorio no podrá haber dos entidades distintas con un mismo nombre. Sin embargo, resulta difícil controlar (ni llega a ser práctico y, por ello, no se realiza tal control) que un determinado nombre no se haya llegado a usar en distintos directorios. Debido a esto, para referirnos a una entidad no basta con dar el nombre con el que haya sido registrada en un determinado directorio sino su nombre de ruta. El *nombre de ruta* de una entidad es la lista de nombres que haya recibido dentro de cierta rama del árbol de nombrado.

Existen dos tipos de nombres de ruta:

- *Absoluto*: Es el que define el camino completo que debe seguirse desde el directorio raíz del árbol de nombrado hasta el nodo que represente a la entidad que se quería nombrar.
- *Relativo*: Es aquel que describe un camino parcial, debido a que el origen de la ruta no coincide con el directorio raíz sino con otro directorio. Por ejemplo, aquél que determine el contexto de nombrado actual para el proceso que utilice el nombre de ruta.

Obsérvese que el significado de ambos conceptos coincide con el de los espacios de nombres utilizados en los sistemas de ficheros tradicionales.

Los espacios de nombres constituyen la información gestionada por un *servicio de nombres*. Los procesos que gestionan estos servicios se conocen como *servidores de nombres*. Cada uno de estos servidores se hace cargo de la gestión de un directorio. Por ello, los servidores de nombres también presentan una organización jerárquica, al igual que los espacios de nombres que ellos administran.

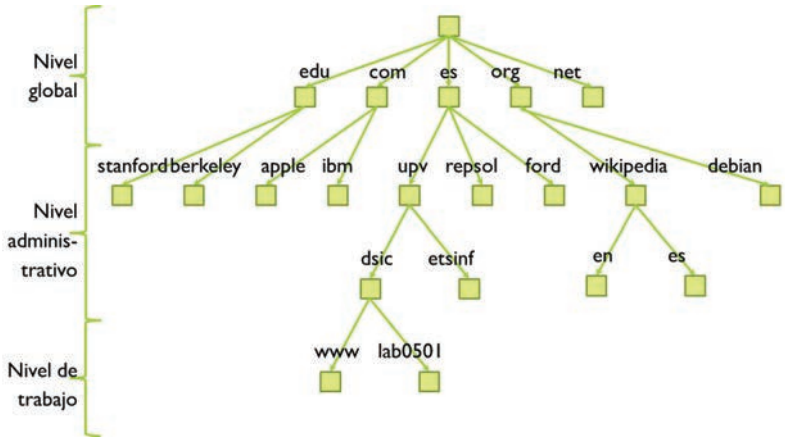

Figura 10.2: Niveles en un espacio de nombres jerárquico.

En esta organización jerárquica podemos distinguir tres niveles, tal como se ilustra en la figura 10.2:

1. *Nivel global*: Formado por directorios que nunca cambian. Representan el nivel más alto de la jerarquía. En el caso del servicio DNS (que estudiaremos en la sección 10.4) en este nivel se encuentran los dominios correspondientes a cada uno de los países y a los apartados más relevantes ("com", "edu", "org", "net", etc.).

2. *Nivel administrativo*: Representan a organizaciones y departamentos dentro de determinadas organizaciones. Estos directorios reciben pocos cambios, pero sí más que los de nivel global.

3. *Nivel de trabajo*: Formado por directorios y entidades que cambian con relativa frecuencia. Suelen estar administrados por los propios usuarios finales.

Los servidores de nombres suelen estar replicados para garantizar su disponibilidad. Así, se aconseja que la replicación se realice al menos en los dos niveles superiores (global y administrativo). Sería crítico que fallara un servidor de nivel global y su información dejara de estar disponible, pues de él dependería la gestión de los nombres contenidos en esa rama del espacio de nombres.

Existe una diferencia en los requisitos exigibles al nivel global y administrativo. En el global no resulta crítico el tiempo de respuesta. Como su información se modifica muy pocas veces, gran parte de ella se mantiene en las cachés de los niveles inferiores, que habrán actuado como clientes en peticiones previas. Por ello, no pasa nada si se tarda algunos segundos en responder a una petición en este nivel. Aparte, suele haber también pocas inserciones y, para que su información llegue a los clientes se toleran retardos prolongados.

No ocurre lo mismo con la información del nivel administrativo (donde se mantendrían los nombres y direcciones de los departamentos/secciones que forman una determinada organización/empresa). En algunos casos, el rendimiento de algunos componentes de una aplicación puede depender de las direcciones mantenidas en estos directorios. Es crítico que la respuesta sea rápida y que las actualizaciones en esta información resulten visibles lo antes posible.

Por último, la disponibilidad de la información de directorio no es crítica en el "nivel de trabajo". Por otra parte, también sería costoso mantenerla con un alto grado de consistencia si hubiera replicación, pues es información que puede llegar a modificarse frecuentemente.

El servicio de nombres proporciona tres operaciones básicas relacionadas con el nombrado. Son las siguientes: asignación, eliminación y resolución de nombres.

- La *asignación de un nombre* a una entidad determinada no presenta dificultades. Para realizarla basta con seleccionar el directorio y solicitar dicha asignación. Habrá que proporcionar la dirección de la entidad a nombrar y la cadena a utilizar como nombre. En algunos casos, también será necesario proporcionar un identificador. El servicio de nombres pasará entonces a comprobar que no haya otras entidades en ese mismo directorio cuyo nombre coincida con el que pretendíamos insertar. Si no hay ninguna, la asignación tendrá éxito y el directorio pasará a mantener la información necesaria. En otro caso, la asignación fallará y tendríamos que repetirla utilizando un nombre distinto para que tuviera éxito.

- La *eliminación de un nombre* únicamente necesita que se elimine la correspondiente entrada de directorio en la que se asignaba un determinado nombre a una entidad. Para ello basta con comprobar que dicha entrada exista y que el proceso solicitante tenga los permisos adecuados para realizar tal borrado.

- La *resolución de nombres* consiste en obtener la dirección de una entidad a partir de su nombre de ruta. Como los espacios de nombres suelen presentar una organización jerárquica, esta operación requiere una serie de pasos. Para empezar, debe localizarse al servidor que gestione el directorio que haya sido tomado como origen para iniciar el nombre de ruta. Será el directorio raíz para los nombres de ruta absolutos.

Una vez localizado ese directorio origen, habrá que contactar con cada uno de los servidores que gestionen el resto de directorios que formen parte del nombre de ruta, hasta llegar al último de ellos. Ese servidor de nombres será el que podrá retornar la dirección de la entidad por la que se esté preguntando. Existen dos tipos de resolución, según cómo se realicen estos pasos:

- *Resolución iterativa*: En este tipo de resolución cada servidor retornará al cliente la dirección del servidor responsable del siguiente directorio dentro del nombre de ruta, así como la parte del nombre de ruta que todavía quedará pendiente de resolución. Con ello, el cliente avanza un paso en la resolución del nombre de ruta. Una vez realizado tal paso, repetirá el proceso con el próximo servidor, y así sucesivamente. De esta manera, la resolución podrá verse como una secuencia de iteraciones donde cada una de ellas es capaz de resolver la dirección del próximo servidor dentro del nombre de ruta. Esto terminará cuando se llegue al último directorio, cuyo servidor será ya capaz de devolver la dirección de la entidad asociada a ese nombre.

  Obsérvese que en esta técnica siempre debe ser el cliente el que inicie cada una de las iteraciones.

- *Resolución recursiva*: En la resolución recursiva, a diferencia de la anterior, el cliente solo interactúa una vez con los servidores de nombres. Dicha interacción se realiza con el servidor de nombres que gestione el directorio original del nombre de ruta.

  En este caso los servidores analizan el nombre y, en lugar de devolver al cliente la dirección del próximo servidor dentro de la ruta, contactan con tal servidor y le pasan el resto del nombre de ruta que todavía quede por resolver. El servidor que reciba una petición de este tipo, volverá a hacer lo mismo: analizar la primera componente de la ruta, localizar en su tabla local la dirección del servidor responsable de ese directorio y pedirle que resuelva el fragmento restante. Esto terminará cuando se llegue al servidor del último directorio, quien ya será capaz de retornar la dirección de la entidad a su peticionario.

  Estas respuestas se van retornando siguiendo el camino inverso al de las peticiones, hasta llegar finalmente al servidor del directorio origen, quien podrá retornar esa respuesta al cliente.

  Utilizando esta técnica, como puede observarse, toda la gestión de los sucesivos pasos la realizan los servidores sin involucrar al cliente.

10.3 Servicios de localización

Como hemos visto en la sección anterior, un servicio de nombres debe mantener la correspondencia entre nombres y entidades, retornando una dirección o conjunto de direcciones como respuesta a una resolución de nombre. Para ello, normalmente los directorios se implantan como tablas con múltiples entradas y cada una de esas entradas mantiene la dirección o direcciones asociadas a un determinado nombre. Esto resulta suficiente cuando una determinada entidad siempre mantiene un mismo nombre. Sin embargo, en ocasiones esto no es así. Puede ocurrir que una determinada entidad modifique con cierta frecuencia su ubicación y que en cada uno de esos cambios deba también modificarse su nombre. Esto llega a ocurrir, por ejemplo, cuando las entidades son los propios ordenadores que forman parte del sistema distribuido y trasladamos algunos de ellos (por ejemplo, los portátiles) a un dominio de nombrado diferente.

En situaciones como esas, conviene dividir un servicio de nombres en dos servicios complementarios:

- *Nombrado.* Asocia un nombre a un identificador de entidad. De esta manera, cuando la entidad cambie de nombre bastará con modificar únicamente la información gestionada por este servicio.
- *Localización.* Asocia un identificador a una dirección. Como los identificadores son invariables, este segundo servicio no debe sufrir ningún cambio cuando la entidad modifique su nombre.

Las ventajas de esta división se aprecian principalmente cuando una misma entidad pueda tener más de un nombre y múltiples puntos de entrada. En ese caso, si la entidad decide modificar alguno de sus nombres, con un servicio de nombres convencional se vería obligada a modificar un buen número de entradas de directorio, pues debería actualizar el nombre asociado a sus múltiples direcciones. Por ejemplo, si una determinada entidad tuviera 3 nombres y 4 direcciones, necesitaría 12 entradas de directorio. La modificación de alguno de sus tres nombres implicaría el tener que modificar 4 entradas de directorio, una por cada dirección asociada al nombre antiguo.

Sin embargo, con un servicio de nombres dividido en un subservicio de nombrado y otro de localización, solo se necesitarían 7 entradas para guardar esa información. En el subservicio de nombrado tendríamos tres, en las que mantendríamos los tres nombres que se podrían utilizar, asociados todos ellos a un mismo identificador (el que referencie a esa entidad). Por su parte, en el subservicio de localización tendríamos cuatro entradas más, una por cada una de sus direcciones. Todas ellas estarían también asociadas al identificador de la entidad. Si quisiéramos modificar uno de los nombres, solo habría que modificar una de las entradas de subservicio de

nombrado (a diferencia de las cuatro entradas que debimos cambiar en un servicio de nombres convencional).

Por tanto, lo que estamos haciendo es añadir un nivel de indirección. Como hemos visto, esto resulta ventajoso a la hora de gestionar la multiplicidad de nombres o direcciones para una determinada entidad. Sin embargo, el nivel de indirección que se añade implicará que en las operaciones de resolución de nombres se tenga que duplicar el número de pasos necesarios para responder a cada petición. Obsérvese que en lugar de obtener directamente una dirección al proporcionar un nombre, ahora obtendremos un identificador y ese identificador debe ser transformado posteriormente en la dirección por la que se preguntaba.

La implantación de un servicio de nombres mediante directorios ya se había descrito en la sección 10.2.2. Pasemos a ver seguidamente cómo se puede implantar un servicio de localización, estudiando diferentes variantes en cada una de las próximas secciones. En concreto, veremos la localización por difusiones y los punteros adelante.

10.3.1 Localización por difusiones

Un primer esquema de gestión de localización es el uso de difusiones. En un esquema de este tipo se asume una red de área local con topología de bus.

Con una topología de bus bastará con dejar un mensaje de petición en el bus para propagar la petición a todos los nodos conectados a dicha red. Esta petición solo incluye el identificador de la entidad cuya dirección nos interese. El nodo que tenga dicho identificador responderá al solicitante, retornando su propia dirección.

De esta manera solo se utilizan dos mensajes para localizar a una determinada entidad. Es un protocolo ligero por lo que respecta al número de mensajes que deben utilizarse, aunque necesita que todos los nodos accesibles participen y examinen cada una de las peticiones efectuadas.

Este tipo de servicio se utiliza, entre otros, en el protocolo *ARP* ("*Address Resolution Protocol*")[Plu82]. En este protocolo los identificadores corresponden a las direcciones del nivel de red (*direcciones IP* o nivel 3 de la arquitectura TCP/IP) y los valores retornados son las direcciones de nivel de enlace (nivel 2 de la arquitectura TCP/IP, lo que solemos llamar *direcciones MAC*).

10.3.2 Punteros adelante

Los "*punteros adelante*"[SDP92] fueron un mecanismo diseñado para gestionar la migración de objetos en un sistema distribuido. Estos "punteros adelante" se empleaban en el mecanismo de ROI del sistema SOR[1]. En este caso, los identificadores serían las referencias a objeto mantenidas en los proxies y las direcciones a obtener corresponderían a la ubicación actual del objeto o procedimiento que quería invocarse mediante dicho proxy.

Cuando el objeto que va a invocarse no ha sufrido ninguna migración, el proxy utilizado ya mantiene la dirección correcta. Con ello, no se necesita ninguna "traducción" y el servicio de localización no requiere ninguna interacción adicional.

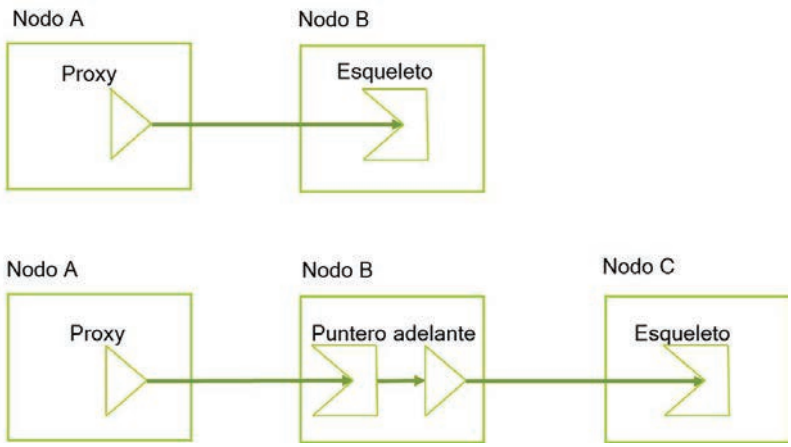

Figura 10.3: Inserción de un puntero adelante.

No obstante, cuando la entidad a invocar cambie su ubicación de un nodo B a otro C se seguirá manteniendo su esqueleto en el nodo B. En lugar de eliminarlo, se "conecta" a un proxy que mantiene la dirección de la entidad en C y permite redirigir las invocaciones recibidas en B hacia la nueva ubicación de esa entidad en C. Esto se ilustra en la figura 10.3, en la que se ha asumido que el primer proxy que tuvo esa entidad se dejó en el nodo A. Si la entidad volviera a cambiar de nodo, añadiríamos otro par <proxy, esqueleto> a la cadena ya existente.

Las entidades "móviles" generarán una larga cadena de punteros. Esa cadena debería recorrerse cada vez que alguien utilizase un proxy que apunte hacia la ubicación inicial de la entidad. Cuando estas cadenas sean largas, sería fácil que alguno

[1]SOR fue un sistema distribuido académico desarrollado por el INRIA francés. Fue uno de los primeros sistemas en los que se extendió el mecanismo de RPC tradicional para que fuera capaz de gestionar inferfaces de objetos.

de los nodos que las componen fallase. En ese caso, la entidad dejaría de ser accesible.

Para evitar que las cadenas crezcan indefinidamente, se debería diseñar algún mecanismo que actualizara las referencias mantenidas en los proxies para que pasen a apuntar a la ubicación actual, en lugar de hacerlo hacia la inicial. Para ello se debe modificar ligeramente el mecanismo ROI, permitiendo que los proxies que inicien las invocaciones (y solo ellos, no así los que formen parte de la cadena y reenvíen las peticiones) dejen en el mensaje de petición alguna referencia propia (la dirección del propio proceso cliente). Cuando el esqueleto responda, no enviará su mensaje al último proxy de la cadena (como ocurriría utilizando una ROI convencional), sino que utilizará esa dirección del cliente original para pasarle directamente a él la respuesta. Ese mensaje de respuesta debe incluir también la referencia actualizada de la entidad invocada. Con esa referencia se actualiza el proxy para que en lugar de seguir la cadena de punteros pase ahora a referirse a la ubicación actual de la entidad. La figura 10.4 ilustra un ejemplo de este tipo de gestión.

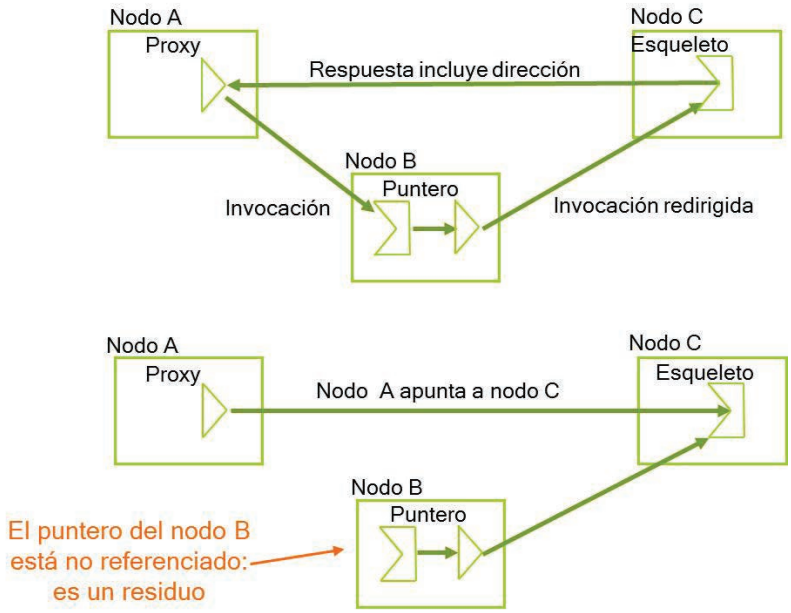

Figura 10.4: Recorte de una cadena de punteros adelante.

Esto conseguirá que las cadenas "desaparezcan". No obstante, esa eliminación no es completa. Puede que el sistema mantenga durante un largo intervalo un buen número de "punteros adelante" que ya nadie llegará a utilizar. Es decir, habrá ciertos esqueletos que dejarán de estar referenciados por proxies, por lo que

no tiene sentido que sigan en el sistema. A ese tipo de componentes se les considera "*residuos*", pero no resulta fácil detectarlos y eliminarlos.

Recolección de residuos

Los servicios de nombres permiten que los procesos de un sistema distribuido puedan obtener las direcciones de otros componentes de su misma aplicación o las direcciones de otros servidores. Para mantener la transparencia de ubicación, dichas direcciones pueden utilizarse a través de stubs clientes, en el caso de RPC, o de proxies en el caso de ROI. Si no importa la transparencia, pueden utilizarse directamente.

En ocasiones, una entidad puede dejar de ser interesante para sus posibles clientes. Éstos irán reemplazando las referencias que permitían utilizar sus operaciones por otras que apunten a otros servidores. Así, llegará un momento en el que cierta entidad dejará de tener referencias que apunten a ella. Si eso ocurre, la entidad ha pasado a ser un residuo y debería eliminarse pues tendrá algunos recursos asignados que nadie podrá aprovechar.

Las entidades registradas en un servidor de nombres nunca llegarán a ser residuos, pues al menos el servidor de nombres mantiene una referencia válida hacia ellas. No obstante, otras entidades pueden generarse como resultado de invocaciones a las operaciones disponibles en la interfaz de una entidad con nombre. Ese segundo tipo de entidad es el que suele correr el riesgo de transformarse en residuo.

Resulta difícil decidir cuándo una entidad ha pasado a ser un residuo, pues debería contactarse con todos sus posibles clientes y comprobar que efectivamente ninguno de ellos mantiene una referencia hacia esa entidad. Existen varios tipos de algoritmos distribuidos [PS95] para realizar dicha detección:

- Mantener una *cuenta de referencias* en el stub servidor. Cada vez que se genere una nueva referencia tendrá que incrementarse el contador. Cuando se elimine una referencia, el contador tendrá que decrementarse. Resulta complejo manejar estos algoritmos pues los incrementos y los decrementos los generan los procesos clientes y estos deben propagar sus mensajes de incremento y decremento hacia el servidor correspondiente. Si llegaran antes los decrementos que los incrementos (en caso de actividades paralelas), podría eliminarse indebidamente alguna entidad, al ser identificada como residuo cuando realmente no lo era.

- Mantener una *lista de referencias* en el stub servidor. En este caso, el servidor mantiene una lista con la identidad de todos los clientes que mantienen alguna referencia hacia esa entidad. No se necesitan operaciones de incremento o decremento de contadores sino de inserción o eliminación de referencias

inversas. Así no se corre el riesgo de eliminar indebidamente a una entidad, pero se necesita mucho más espacio para manejar la lista de referencias inversas.

- Realizar una *trazado* del sistema. En estos algoritmos se realiza una primera fase de marcado, en la que se van asignando marcas a las entidades que resulten alcanzables a partir de las referencias gestionadas por el sistema. En una segunda fase (fase de barrido o limpieza) las entidades que no hayan sido marcadas son eliminadas puesto que se considerará que son residuos.

10.4 Servicios de nombres jerárquicos: DNS

Los servicios de localización que acabamos de describir en las secciones 10.3.1 y 10.3.2 no están diseñados para escalar fácilmente. Un servicio distribuido se considera *escalable* [Bon00] cuando sea capaz de atender a un número extremadamente alto de clientes sin reducir su calidad de servicio. Es decir, manteniendo un tiempo de respuesta predecible y aceptable. Para ello debe utilizarse algún algoritmo que permita distribuir las tareas entre un número variable de servidores, repartiendo las peticiones entre todos ellos. Así, cuando tanto el número de clientes como la tasa de llegada de peticiones crezcan, el algoritmo permitirá que el número de servidores también crezca. Además, aunque el número de servidores sea alto, el algoritmo debe minimizar (o incluso eliminar) la necesidad de interacción entre ellos. Así, esa intercomunicación no supondrá ningún "cuello de botella".

En el caso de la localización por difusión estamos obligados a propagar los mensajes en una sola red local con topología de bus. Un sistema así no admite un alto número de nodos. Por su parte, en la localización por punteros adelante, a medida que las entidades incrementen el número de migraciones efectuadas, la cadena de punteros crece e incrementa tanto el tiempo de respuesta como su fragilidad ante la ocurrencia de algún fallo.

La solución para mejorar la escalabilidad reside en una organización jerárquica de los servidores de nombres. En una organización de este tipo, ya descrita al presentar los espacios de nombres en la sección 10.2.2, cada servidor será responsable de uno o más directorios. Todo nombre que esté almacenado en un mismo grupo de directorios puede ser gestionado directamente por su servidor responsable. Cuando el nombre de ruta proporcionado por el cliente en una resolución no se limite a la información del directorio local, el servidor también sabe (analizando la ruta) qué información tendrá que retornar (en caso de resolución iterativa) o a qué otros servidores tendrá que preguntar para completar la resolución (en caso de resolución recursiva). En cualquier caso, el número de mensajes que tendrán que utilizarse en cada servidor será muy bajo y el tiempo que el cliente invertirá esperando una respuesta suele ser razonable.

Un ejemplo de servicio de nombres jerárquico es *DNS* (sigla para "*Domain Name System*"). La primera especificación de este servicio aparece en [Moc83a, Moc83b], pero ha habido otras versiones posteriores.

En DNS, cada uno de los nodos del espacio jerárquico es un "*dominio*". Cada dominio podrá ser el "nodo padre" de otros dominios en el nivel inmediatamente inferior. Un determinado conjunto de dominios relacionados entre sí podrán constituir una "*zona*". Cada zona tiene un servidor de nombres que se responsabiliza de su gestión.

El nombre asignado a un ordenador define la ruta desde el nodo raíz de la jerarquía hasta el lugar que ocupa tal ordenador en el espacio de nombres. Cada "dominio" se comporta como un directorio y para formar el nombre de ruta ("*nombres de dominio*" en DNS) se utiliza el punto (".") como separador de los nombres de directorio. Otra característica importante de esta estructura de nombrado es que las rutas se dan en orden inverso: se empieza por el nombre del ordenador que queríamos nombrar y se termina con el directorio raíz. Este nodo raíz es una excepción en el esquema de nombrado y solo existe de manera lógica, siendo su nombre la cadena vacía.

Por ejemplo, el nombre que recibiría el servidor de la web de nuestra escuela sería: "www.etsinf.upv.es.". El punto final (tras "es") también forma parte del nombre de dominio y vincula al dominio "es" con el nodo raíz. No obstante, tales puntos finales se consideran opcionales y rara vez se utilizan. Debido a esto, podríamos decir que la estructura del espacio de nombres en DNS es un "bosque" en lugar de un árbol. Así, existirá una serie de dominios en el nivel más alto de la jerarquía, también llamados "*dominios de nivel superior*" (*TLD*: "top-level domain"). Ejemplos de TLD serían: "com", "edu", "org", "net", "mil", "gov",... junto a los dominios que representan a cada país: "es" (España), "fr" (Francia), "de" (Alemania), "it" (Italia), "uk" (Reino Unido), "pt" (Portugal), "ch" (Suiza), "gr" (Grecia), etc.

El formato que deben seguir los nombres se explica seguidamente. Un nombre está formado por una serie de etiquetas separadas por puntos. Cada etiqueta representa a un nodo del espacio de nombres y tiene una longitud entre 1 y 63 caracteres. Se admitirían hasta 127 etiquetas en un nombre de dominio (127 niveles en el árbol de nombrado), pero también debe respetarse que el nombre completo de dominio no supere los 255 caracteres.

La estructura jerárquica resultante permite que cada servidor de nombres solo deba responsabilizarse de un conjunto de entradas relativamente pequeño. Si dicho conjunto creciera y el servidor fuera responsable de una zona compuesta por varios dominios, se podría dividir dicha zona en varias zonas más pequeñas (con al menos un dominio) y asignar un servidor distinto a cada una de ellas. Así, el servicio de nombres DNS puede escalar de manera sencilla.

Para que un servidor pueda administrar una zona determinada tendrá que utilizar cierto fichero en el que mantendrá tanto la correspondencia entre nombres y direcciones para las máquinas de su zona como otra información de configuración. Este fichero se organiza como una lista de registros. Cada registro se estructura como una serie de campos (cuyo número es variable y depende del tipo de registro), entre los que siempre podremos encontrar un tipo de registro, un nombre y algún valor. Este último deberá interpretarse en función de su tipo. Los tipos más importantes son:

- A: Su valor es la dirección IP (IPv4) de la máquina especificada en el campo "nombre".
- AAAA: Su valor es la dirección IP (IPv6) de la máquina especificada en el campo "nombre".
- CNAME: Permite asignar un "alias" para la máquina especificada en el campo "nombre".
- MX: Su valor es el nombre del servidor de correo electrónico para el dominio especificado por el campo "nombre".
- NS: Su valor es el nombre del servidor de nombres para una de las zonas del dominio especificado por el campo "nombre".
- SOA ("Start of authority"): Proporciona información sobre una zona DNS, tal como su servidor de nombres primario, la dirección de correo electrónico del administrador (aunque sustituyendo el carácter arroba por un punto), el número de serie de la zona y algunos temporizadores para gestionar la información contenida en el directorio de la zona.
- SRV: Se utiliza de manera general para guardar los nombres de los nodos que gestionan otros servicios que complementen a DNS. Un ejemplo de uso de estos registros lo podemos encontrar en los "*domain controllers*" del Directorio Activo de Windows Server [RKMW08]. Estos nodos guardan mediante registros de este tipo el nombre de los servidores LDAP, entre otros.

  Para ello utilizarían registros similares a los siguientes:

  ```
  _ldap._tcp.ejemplo.com. 600 IN SRV 0 100 389 serv1.ejemplo.com
  ```

  Cada uno de los campos empleados en este ejemplo de registro indicaría esto:

  - _ldap: Indica que el servidor gestionará el servicio LDAP.
  - _tcp: Indica que se utilizará el protocolo TCP.
  - ejemplo.com: Nombre del dominio en el que se utilizará dicho servicio.
  - 600: TTL (Time to live) de este registro, en segundos.
  - IN: Clase del registro. IN indica que es un registro para algún servicio convencional (clase "Internet").

- SRV: Tipo de registro.
- 0: Especifica la prioridad de este registro para el cliente. En caso de que hubiera múltiples entradas con información que afectase a un mismo dominio y para un mismo servicio, el cliente debería utilizar en primer lugar aquellos servidores con número de prioridad más bajo. Si estos hubieran fallado, entonces pasaría a comprobar y utilizar aquellas entradas con números más altos, siguiendo siempre un orden de menor a mayor.
- 100: Especifica el peso de esta entrada. En caso de que existan múltiples entradas con información para un mismo servicio y dominio y todas ellas tuvieran una misma prioridad, deberían utilizarse con mayor frecuencia aquellas que tuvieran un mayor peso.
- 389: Puerto por el que "escuchará" el servidor. En este caso, 389 es el número de puerto a utilizar por omisión en un servicio LDAP sobre TCP (cuando no se utilice SSL).
- serv1.ejemplo.com: Nombre del servidor que proporciona el servicio especificado en este registro.

La figura 10.5 muestra un ejemplo de fichero con la información de una zona DNS que incluye un único dominio ("mycomp.com").

Para soportar las situaciones de *fallo* [Nel90], los servidores de nombres están replicados en DNS. Para cada zona existirá un servidor de nombres primario, que es quien se encarga de atender todas las peticiones de inserción, borrado o resolución de nombres. Para su respaldo existirá algún servidor secundario. El servidor primario propaga la información completa de su fichero cada cierto tiempo al servidor o servidores secundarios en caso de que tal fichero haya sufrido alguna modificación desde la última transferencia. De esta manera, si el servidor primario dejara de estar disponible, el servidor secundario contaría con una copia de su información y podría retomar el servicio sin problemas.

10.5 Servicios basados en atributos: LDAP

Los servicios de nombres vistos en las secciones anteriores solo se preocupaban por retornar la dirección de la entidad por la que se preguntara. Para ello bastaba con relacionar a esa entidad con un nombre o identificador que permitiera encontrarla. Esto puede no ser suficiente para algunas aplicaciones, que necesitarían obtener mayor información sobre la entidad.

Para solucionar este problema se diseñaron otros servicios de gestión de entidades. Son los servicios de nombres basados en atributos, entre los que *LDAP* (sigla del inglés "*Lightweight Directory Access Protocol*") es un buen ejemplo. En él, el

```
$TTL 3D
@ IN   SOA mycomp.com. root.mycomp.com. (
                200109206 ; serial, todays date+todays serial #
                8H ; slave refresh (8 hours)
                2H ; slave retry (2 hours)
                4W ; slave expiration time (4 weeks)
                1D ) ; maximum caching time if failed lookups
        NS mycomp.com.
        NS ns2.anyorg.net.
        MX 10 mycomp.com. ; Primary Mail Exchanger
        TXT "MyComp Corporation"
localhost       A 127.0.0.1
router          A 208.6.177.1
mycomp.com.     A 208.6.177.2
ns              A 208.6.177.3
www             A 207.159.141.192
ftp             CNAME mycomp.com.
mail            CNAME mycomp.com.
news            CNAME mycomp.com.
another         A 208.6.177.2

;; Workstations ;
ws-177200       A 208.6.177.200
ws-177201       A 208.6.177.201
ws-177202       A 208.6.177.202
ws-177203       A 208.6.177.203
ws-177204       A 208.6.177.204
```

Figura 10.5: Ejemplo de fichero de zona DNS.

servicio de consulta proporcionado a los clientes no será una mera resolución de nombres, retornando direcciones, sino consultas en base al valor de alguno de los atributos que caractericen a las entidades, retornando a su vez cierto conjunto de atributos para aquellas entidades que satisfagan las condiciones de la consulta. En LDAP el concepto de entidad es muy general. Por ejemplo, puede abarcar también a los usuarios y no es raro que uno de los atributos gestionados sean las contraseñas. Así, entre otras, una de las funciones de LDAP suele ser la gestión de contraseñas para tareas de autenticación en sistemas distribuidos. Pero también gestiona cualquier otro tipo de recurso dentro del sistema distribuido. De hecho, puede trabajar como un servicio de nombres tradicional, implantando así el *nivel de trabajo* del espacio de nombres jerárquico integrándose con DNS, por ejemplo. LDAP se utiliza actualmente en la mayoría de los sistemas operativos modernos. El *Directorio Activo* de los sistemas Microsoft Windows Server implanta LDAP, y la mayoría de los sistemas Linux pueden usar OpenLDAP.

Cuando un servicio de nombres es extendido para manejar múltiples atributos de cada entidad pasa a conocerse como un "*servicio de directorio*". LDAP se diseñó como una versión más sencilla del protocolo *DAP* (acrónimo del inglés "*Directory Access Protocol*"). Este último se utilizaba en el nivel de aplicación de la arquitectura OSI, ofreciendo servicios de directorio. El problema principal de DAP era que estaba diseñado específicamente para una arquitectura OSI y que en sus versiones iniciales no podía trabajar directamente sobre TCP/IP. El objetivo de LDAP fue la adaptación de DAP a los protocolos TCP/IP. La primera especificación de LDAP, correspondiente a LDAP v1, se publicó en 1993 [YHK93]. La versión actual de los protocolos LDAP es la v3, especificada inicialmente en 1997 y reorganizada en 2006.

De manera similar a DNS, los servidores LDAP también pueden organizar sus directorios jerárquicamente. La información gestionada directamente por cada servidor se mantiene como un conjunto de registros o entradas. A su vez, cada entrada está formada por un conjunto de atributos que caracterizan a la entidad que representan. Estos atributos pueden ser:

- *Simples*: Cuando admiten un solo valor. Un ejemplo sería el DNI de una persona.
- *Múltiples*: Admiten más de un valor simultáneamente y gestionan a todos ellos como un conjunto sin ningún orden establecido. Ejemplo: las direcciones IP en un ordenador que tuviera múltiples tarjetas de red.

Se puede especificar qué atributos se mantendrán para cada entidad y de qué tipo será cada uno, mediante un *esquema*.

Aunque el administrador puede dar su propio esquema para la información mantenida en un determinado servidor, existen también atributos estándar para los registros LDAP. Si asumimos que las entidades que vamos a gestionar son personas, algunos de sus atributos estándar son (entre paréntesis se especifica su abreviatura, que puede utilizarse para identificarlos):

- commonName (cn): Nombre completo.
- surname (sn): Apellidos.
- givenName: Nombre de pila. Para formar el nombre completo, la especificación LDAP asume que se concatenarían los atributos givenName, initials y surname, en ese orden.
- countryName (c): País, como abreviatura de dos caracteres, utilizando abreviaturas similares a las empleadas en DNS.
- Locality Name (l): Ciudad.

- state (st): Nombre de un estado o provincia.
- Street Address (street): Calle y número en una dirección postal.
- Organisation Name (o): Nombre de la organización o empresa en la que trabaje la persona representada en la entrada o registro.
- Organisational Unit Name (ou): Nombre del departamento dentro de la organización o empresa.
- friendlyCountryName (co): Nombre completo del país.

Aunque LDAP sea un servicio de directorio basado en atributos, también puede trabajar como un servicio de nombres. Para ello, debe agrupar algunos de los atributos de la entidad para formar un nombre global único. En la terminología de LDAP este nombre global único se conoce como "*distinguished name*" y puede abreviarse como "dn". Cada uno de los atributos que lo constituyan será un "*relative distinguished name*" o "*RDN*". Para formar el nombre global único, los RDN se concatenan usando el carácter "/" o bien el "," como separador. Por ejemplo, el nombre global único para nuestra Escuela Técnica Superior de Ingeniería Informática sería: "c=ES,o=UPV,ou=ETSInf".

A la hora de realizar una consulta LDAP habrá que especificar cierta información en una serie de campos. La aplicación utilizada para realizar tal consulta ya indicará cómo puede hacerse. Como mínimo, sería necesario lo siguiente:

- La entidad sobre la que se realizará la consulta. Para ello el cliente debería proporcionar el "dn" (nombre global único) de la entidad a buscar. En caso de que la consulta se realice sobre un conjunto de entidades, habría que especificar los valores de los RDN que compartan todas ellas.
- El filtro o consulta a realizar.
- Los nombres de los atributos cuyos valores deberá retornar el sistema como respuesta a esa consulta.

La especificación de los filtros a utilizar en las consultas se ha reformado recientemente y aparece descrita en [SH06]. Como principios generales se debe respetar lo siguiente:

- El filtro estará constituido por uno o más términos. Cada uno de los términos tendrá que estar encerrado entre paréntesis.
- Habrá dos tipos de operadores. Los primeros se utilizan en el interior de un término y sirven para relacionar el nombre del atributo con su valor. Los segundos se aplican sobre uno o más términos y utilizan notación prefija; es

decir, se darán justo antes del término o secuencia de términos sobre los que operen.

Algunos ejemplos de operadores internos son:

- = Separa el nombre de un atributo (que debe estar a su izquierda) de su posible valor (a su derecha). Indica que el atributo especificado debe tener un valor igual al indicado. Ejemplo: (co=España) indicaría que el país de la entidad que se está buscando debe ser España.
- >= Indica que el valor del atributo debe ser lexicográficamente mayor o igual al indicado. Puede utilizarse el asterisco como carácter comodín para indicar que en su posición podrá aparecer cualquier subcadena. Por ejemplo: (sn>=F*) indicaría que las personas buscadas tienen un apellido que empiece por cualquier letra lexicográficamente mayor o igual a 'F'.
- <= Indica que el valor del atributo debe ser lexicográficamente menor o igual al indicado. Puede utilizarse el asterisco como carácter comodín para indicar que en su posición podrá aparecer cualquier subcadena. Por ejemplo: (sn<=C*) indicaría que las personas buscadas tienen un apellido que empieza con 'A', 'B' o 'C'.

A su vez, algunos operadores a utilizar entre términos serían:

- ! El carácter de admiración sirve para expresar que la expresión utilizada en el siguiente término debe entenderse en sentido contrario. Es decir, es una negación lógica. Ejemplo: (!(sn=Herrero)) indicaría que se están buscando personas cuyo apellido no sea Herrero.
- & El "ampersand" expresa un AND lógico de las expresiones que aparezcan en la lista de términos que sigan a este operador. Ejemplo: (&(givenName=David) (l=Valencia)) buscaría los registros correspondientes a personas cuyo nombre sea David y que vivan en Valencia.
- | La barra vertical expresa un OR lógico de las expresiones que aparezcan en la lista de términos que sigan a este operador. Ejemplo: (|(sn=B*)(sn=F*)) buscaría los registros correspondientes a personas cuyo apellido empiece con B o F.

10.6 Resumen

Uno de los objetivos importantes que todo sistema distribuido tendrá que cumplir es facilitar el acceso a recursos remotos. Para ello, se debe proporcionar una gestión de la asignación de nombres a los recursos, así como la obtención de sus puntos de entrada para utilizarlos posteriormente.

Un nombre es una cadena de caracteres que se utiliza para referirse a una entidad, que en un sistema distribuido puede ser cualquier recurso que sea deseable referenciar de forma remota (máquinas, impresoras, discos, ficheros, buzones, procesos, objetos, etc.). Las entidades se identifican entre sí de forma unívoca por medio de los identificadores. Un *identificador* siempre referencia a la misma entidad y nunca se reutiliza. Para operar sobre una entidad es necesario conocer su punto de entrada, cuyo nombre es una *dirección*. Las entidades pueden ofrecer más de un punto de entrada, y éstos pueden cambiar durante su vida.

Los nombres se organizan en *espacios de nombres* y al mecanismo que dado un nombre proporciona una entidad, se le conoce como resolución de nombres. El servicio de localización de recursos introduce un nivel de indirección, en base a nombre-identificador-dirección, de modo que el servicio de nombres retorna un identificador, mientras que el servicio de localización, dado el identificador busca la dirección actual de la entidad, para poder acceder a ella. En esta unidad se han revisado diferentes variantes para implantar un servicio de localización: la localización por difusiones (en las que se envía un mensaje de difusión al bus conteniendo el identificador de la entidad y solo contesta el nodo que contenga el punto de entrada), la localización por punteros adelante (en donde cuando una entidad se traslada de A a B, se deja un puntero en A hacia su nueva ubicación; y en este tipo de localización se debe considerar la recolección de residuos). Para mejorar la escalabilidad del sistema, se suele utilizar una organización jerárquica de los servidores de nombres. Un ejemplo de servicio de nombres jerárquico es DNS, que asocia nombres de dominio a direcciones IP. Por su parte, en los servicios basados en atributos, entre los que destaca LDAP, se asignan atributos a las entidades para permitir búsquedas en función del valor de algunos de sus atributos.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Distinguir los conceptos de entidad, nombre de entidad, punto de entrada, dirección, identificador.
- Caracterizar un espacio de nombres: identificarlo, describir sus niveles y explicar cómo se lleva a cabo el proceso de resolución de nombres.
- Identificar el servicio de localización y describir diferentes implementaciones: la localización por difusiones y la localización por punteros adelante.
- Identificar servicios de nombres jerárquicos.
- Identificar servicios basados en atributos.
- Utilizar adecuadamente los servicios de nominación/localización de recursos en un entorno distribuido.

Unidad 11

ARQUITECTURAS DISTRIBUIDAS

11.1 Introducción

Cualquier aplicación distribuida estará formada por múltiples componentes que podrán ser instalados en más de un ordenador. La *arquitectura* de cualquier aplicación informática debe definir cómo se estructuran estos componentes y qué tipo de consecuencias tendrá tal organización. En el caso de las aplicaciones distribuidas, se distinguirán dos tipos de arquitectura: lógica y física. La arquitectura lógica (o "arquitectura *software*") determina la organización que se ha utilizado para estructurar los componentes de la aplicación durante su diseño, así como las interacciones que podrán establecerse entre estos componentes. Por su parte, una vez la arquitectura lógica esté definida y la aplicación sea desarrollada, llegará un momento en que pasará a ser instalada y empleada por sus usuarios. En este último paso habrá que decidir en qué máquinas será instalado cada componente y qué tipo de ordenadores y redes serán necesarios. Esa instanciación final de la aplicación definirá su arquitectura física (o "arquitectura de sistema") y condicionará la escalabilidad y rendimiento de la aplicación.

En esta unidad se describirán las variantes más importantes en cada tipo de arquitectura. Así, la sección 11.2 describe las *arquitecturas software*, mientras que la sección 11.3 enumera y explica las *arquitecturas de sistema* más relevantes, proporcionando algunos ejemplos de modelos de aplicación que siguen tales arquitecturas.

11.2 Arquitecturas *software*

Una *arquitectura software* (o arquitectura lógica) es la que expresa cómo se organizan los componentes de una aplicación y cómo deberían interactuar entre sí. Se han llegado a definir múltiples estilos arquitectónicos, que indican cuáles son los componentes de una aplicación o sistema distribuido y cómo están configurados.

Los modelos o estilos de arquitectura software que se siguen con mayor frecuencia son:

- *Arquitectura de capas o niveles.* Los componentes se estructuran de manera lógica en capas, de manera que cada capa presente una interfaz de operaciones hacia la capa superior. Dentro de una misma capa los componentes podrán interactuar entre sí sin restricciones. La interacción entre componentes de niveles distintos sí que está limitada. Así, un componente ubicado en el nivel N solo podrá invocar aquellas operaciones ofrecidas por la capa N-1 (inferior) y, a su vez, podrá responder a aquellas invocaciones que haya recibido desde el nivel N+1 (superior). Obsérvese que en las interacciones entre niveles resultará imposible "saltarse" un nivel. Es decir, un componente de nivel N jamás podrá invocar directamente una operación presente en la interfaz del nivel N-2 o en otros niveles inferiores al N-2. La interacción entre niveles se ilustra en la figura 11.1.

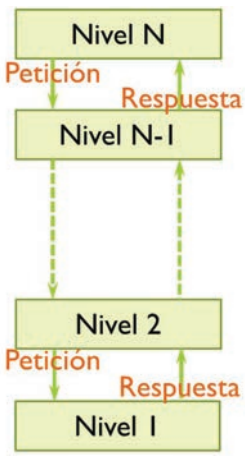

Figura 11.1: Arquitectura software en niveles.

  Un ejemplo de arquitectura lógica basada en niveles es la pila de protocolos de comunicación del estándar ISO-OSI [Zim80].

- *Arquitectura basada en objetos.* En una arquitectura de este tipo, cada uno de los componentes será un objeto que presentará una determinada interfaz.

Los componentes interactuarán entre sí por medio de la invocación de los métodos disponibles en tales interfaces. Para ello existirá algún mecanismo de invocación (remota) de métodos.

Para diseñar y representar una arquitectura de este tipo se pueden utilizar algunas herramientas de modelado. El estándar UML ("*Unified Modeling Language*") [Obj11b] especifica múltiples tipos de diagramas con esta finalidad. La mayor parte de estos diagramas no dependen para nada de cuántos ordenadores tendrán que utilizarse para instalar la aplicación (de hecho, son también válidos para el diseño e implantación de aplicaciones orientadas a objeto que vayan a utilizarse en un solo ordenador). Algunos de ellos son:

- Los diagramas de clases especifican los métodos y atributos de cada una de las clases o interfaces utilizadas en la aplicación distribuida, así como las relaciones existentes entre tales elementos (herencia, asociación, instanciación, ...).
- Los diagramas de objetos pueden utilizarse para modelar el conjunto de objetos que deberíamos instanciar de cada una de las clases a la hora de implantar la aplicación.
- Los diagramas de secuencia permiten reflejar las interacciones entre diferentes objetos generadas por la invocación de algún método.
- Los diagramas de comunicación son capaces de reflejar tanto las interacciones entre los objetos (cosa que ya hacen los diagramas de secuencia) como algunas de las relaciones existentes entre las clases implantadas por dichos objetos (aspecto que se refina en los diagramas de clases). Al igual que en el caso de los diagramas de secuencia, se necesitará un diagrama de comunicación distinto para representar las interacciones generadas por la invocación de cada método de cierto objeto, reflejando así con qué otros objetos se llega a interactuar para completar esa invocación.

Independientemente de la herramienta de modelado y de los tipos de diagrama que podamos utilizar para diseñar una aplicación distribuida bajo este estilo arquitectónico, lo que define a una arquitectura de este tipo es el hecho de utilizar los objetos como componentes de la aplicación y que la colaboración entre las múltiples actividades que formen la aplicación distribuida se llevará a cabo utilizando invocaciones de métodos.

Toda arquitectura basada en objetos asume que la aplicación modelada sigue un modelo de interacción cliente/servidor. Así, cada invocación a método sigue una secuencia en la que se distinguen tres pasos: petición, servicio y respuesta. Esa secuencia se explicará con detenimiento en la sección 11.3.1.

- *Arquitectura centrada en datos.* Tienen sentido en aquellas aplicaciones en las que la interacción entre sus actividades se realice a través de cierto almacén común de información. Un ejemplo de almacén de este tipo sería un conjunto

de ficheros gestionados por un sistema de ficheros distribuido. Los procesos que componen esta aplicación intercambiarían información mediante estos ficheros compartidos.

- *Arquitectura basada en eventos.* En estas arquitecturas, los componentes de una aplicación distribuida se comunican a través de un bus de notificación de eventos. Para ello, los componentes deben registrarse en el sistema y especificar qué tipo de eventos podrán generar (o publicar) y en qué tipos de eventos están interesados, a los que se suscribirán (especificando para ello cierta área temática). El sistema resultante también se conoce como *sistema de publicación-suscripción* [EFGK03]. Así, cada proceso va publicando los eventos que considere relevantes y el sistema se encargará de hacerlos llegar (con la información que lleven asociada) a aquellos procesos que se hayan suscrito a determinadas áreas relacionadas con esos eventos. Una de las ventajas de estos sistemas es que sus procesos están débilmente acoplados: no tienen por qué conocerse entre sí. Basta con que tengan interés en una misma temática para que lo que cada cual publique llegue a ser consultado por los demás. Para que los procesos suscriptores lleguen a consultar tal información deberán ser notificados por el sistema (la notificación es un segundo tipo de evento, al igual que la publicación es su evento complementario).

  En la figura 11.2 se muestra un ejemplo en el que existen dos áreas temáticas representadas como dos buses de eventos, uno más general (representado con un color más claro y con un bus más amplio) y otro específico (representado con color oscuro y emplazado dentro del anterior). En el segundo bus únicamente se aceptan los eventos publicados explícitamente en el bus específico. Por el contrario, en el bus general se aceptan tanto los eventos publicados en el bus específico como los únicamente destinados al bus general. Los componentes A y C están suscritos al bus general, mientras que el componente B se ha suscrito al bus específico. El componente D publica una información en el bus general dentro de un tema no incluido en el bus específico. Por ello, esa información generada por D jamás llegará al componente B (pues no está registrado en ese bus), pero sí a los componentes A y C.

  Obsérvese que un modelo de interacción basado en publicación-suscripción permite desacoplar a los procesos generadores y receptores de eventos considerando tres aspectos:

  1. Identidad. El publicador no tiene por qué conocer al suscriptor. La información llegará pero para ello no tienen por qué conocerse entre sí (el publicador no necesita saber ni la dirección ni el identificador del suscriptor). El sistema se encargará de notificar a los suscriptores interesados.
  2. Tiempo. Cuando el publicador genera el evento, los suscriptores no tienen por qué estar activos. De igual manera, cuando cada suscriptor

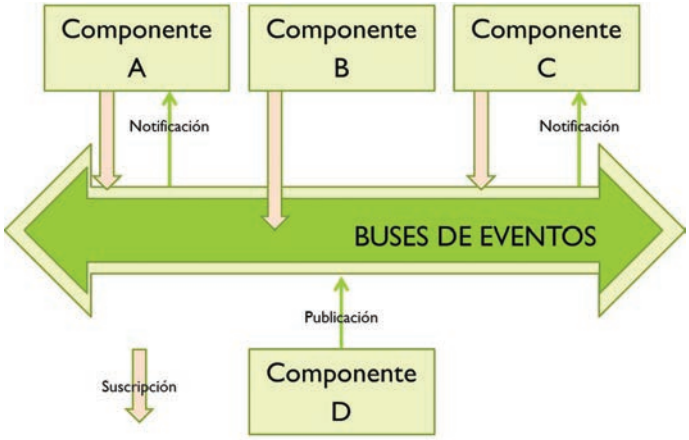

Figura 11.2: Arquitectura software basada en eventos.

obtenga la información publicada, su publicador puede estar realizando otras tareas no relacionadas con el sistema de gestión de los eventos.

3. Sincronización. Los publicadores no necesitan bloquearse mientras generen sus eventos y los suscriptores pueden ser notificados asíncronamente (más tarde) si durante el intervalo de publicación no permanecieron activos.

11.3 Arquitecturas de sistema

Una *arquitectura de sistema* (o *arquitectura física*) expresa cuál ha sido la instanciación final de una arquitectura software. Es decir, indica cómo y dónde se ha ubicado cada componente de la aplicación distribuida en un conjunto de ordenadores reales.

Existen tres alternativas a la hora de definir una arquitectura de sistema, en función de las características que pretendamos optimizar (rendimiento, escalabilidad, seguridad, mantenibilidad, disponibilidad, etc.):

- *Arquitectura centralizada*: En una arquitectura centralizada se asume un modelo de interacción cliente/servidor en el que se utilizará un componente servidor por cada recurso de la aplicación que deba ser compartido por varias actividades. De esta manera se centraliza la gestión del recurso, pues un solo proceso será el responsable de atender y gestionar las diferentes peticiones de acceso sobre tal recurso que lleguen desde otros componentes. Esto permite que diferentes actividades utilicen concurrentemente el recurso y que

tales accesos se puedan gestionar con mecanismos de control de concurrencia locales (locks, semáforos, monitores, etc.).

En la mayoría de los casos una arquitectura centralizada utiliza una *distribución vertical*. En una distribución vertical se asume una arquitectura software en niveles (donde cada nivel es responsable de un único tipo de servicio o recurso) y se selecciona en qué ordenador se instalará cada componente/nivel.

Una arquitectura centralizada minimiza las necesidades de coordinación entre los componentes de una aplicación, pues cada uno de ellos tendrá una misión bien definida y solo interactuarán cuando se requiera el uso de recursos gestionados por otros componentes. De esta manera se optimiza la *mantenibilidad* (hay pocos componentes, con funcionalidad e interacciones claras, pudiendo identificarse rápidamente en qué componente puede existir un defecto en caso de un mal comportamiento del sistema, tomando rápidamente las acciones necesarias para eliminar tal defecto) y la *seguridad* ("*safety*": garantía de que ante cualquier situación imprevista, el comportamiento de la aplicación será fiable y no provocará desastres) de la aplicación resultante.

- *Arquitectura descentralizada*: En una arquitectura descentralizada se utiliza *distribución horizontal*. Para ello se divide cada componente físicamente en múltiples partes que son equivalentes lógicamente (es decir, cada una de ellas desarrolla la misma función) y se ubica cada una de esas partes en un nodo distinto del sistema. Es decir, se utilizan los principios de distribución (reparto de la carga y de los datos) y de replicación ya vistos en la Unidad 7 como base para mejorar la escalabilidad de tamaño.

  Una arquitectura descentralizada optimiza la escalabilidad, rendimiento y disponibilidad de una aplicación distribuida. Obsérvese que al realizar un reparto de la carga y datos se mejorará la escalabilidad y el rendimiento, pues cada proceso servidor será responsable de un menor número de recursos y la carga global generada por las peticiones de los clientes se puede dividir entre ese conjunto de servidores. A su vez, se debe tener más de una réplica de cada recurso, con lo que se mejorará la disponibilidad (en caso de que falle una de ellas, las restantes podrán atender las peticiones que generen otras actividades de la aplicación) y también el rendimiento (si las peticiones recibidas solo consultan la información mantenida, basta con que una sola réplica las gestione; con ello, cuantas más réplicas haya, mayor número de peticiones concurrentes se podrán atender).

- *Arquitectura híbrida*: Como ya se ha visto en los dos casos anteriores, con arquitecturas estrictamente centralizadas o estrictamente descentralizadas no se pueden optimizar todas las características que los usuarios esperan de una aplicación distribuida. Por ello se suele optar por la combinación de un modelo de interacción cliente/servidor con la descentralización, generando

así arquitecturas híbridas que mantienen las mejores características de cada una de las alternativas anteriores. La mayoría de las aplicaciones distribuidas colaborativas actuales adoptan una arquitectura híbrida.

En las siguientes secciones se explicará con mayor detalle cada uno de estos tipos de arquitecturas de sistema.

11.3.1 Arquitecturas centralizadas

Una *arquitectura centralizada* adopta un modelo de interacción *cliente-servidor*. Algunos componentes se comportarán como servidores, gestionando un determinado recurso del sistema, centralizando así la atención de las peticiones generadas por otros componentes (los clientes) para utilizar dicho recurso. Por tanto, en una gestión centralizada se distinguen dos tipos de roles:

- *Servidor*: Es el responsable de la gestión de un recurso determinado que podrá ser utilizado concurrentemente por múltiples componentes de una aplicación distribuida. Proporciona una interfaz de operaciones que define una serie de servicios. Su misión consiste en esperar la llegada de peticiones de servicio (es decir, invocaciones de las operaciones presentes en su interfaz), procesándolas seguidamente para generar una respuesta para cada una de ellas.
- *Cliente*: Es cualquier componente que utilice los servicios de otro. Para ello realizará una petición y esperará la respuesta del servidor, en la que se indicará el resultado del servicio solicitado.

Los conceptos "cliente" y "servidor" se refieren a roles (es decir, papeles interpretados por cada componente) y dependen de cada interacción concreta. Por ello, un mismo proceso o componente del sistema puede adoptar roles diferentes en diferentes interacciones. Así, por ejemplo, la figura 11.3 muestra mediante un diagrama de comunicación UML cómo se realiza una secuencia de interacciones entre componentes en un sistema distribuido. La figura describe la arquitectura software de dicho sistema. En este ejemplo, un usuario inicia cierta operación sobre el componente "Objeto A" que a su vez necesita la invocación de un método `op1()` del "Objeto B". En esta primera interacción, "Objeto A" se comporta como cliente y "Objeto B" como servidor. Durante la ejecución del método `op1()` se genera otra invocación del método `op2()` del "Objeto C". Por ello, en esa segunda interacción, el componente "Objeto B" pasa a ser un cliente del servidor "Objeto C". Puede observarse que el "Objeto B" ha adoptado ambos roles (cliente y servidor) en este ejemplo, actuando de manera distinta en cada una de esas invocaciones. Debe tenerse en cuenta que se está utilizando todavía un diagrama que representa la arquitectura software del sistema y que, posteriormente, se tendrá que decidir en qué ordenador se instalará cada componente. Cada componente podría ser ubicado en un ordenador diferente y así cada componente sería ejecutado por un proceso

distinto. Cada uno de esos procesos sería quien realmente adoptaría el rol cliente o servidor.

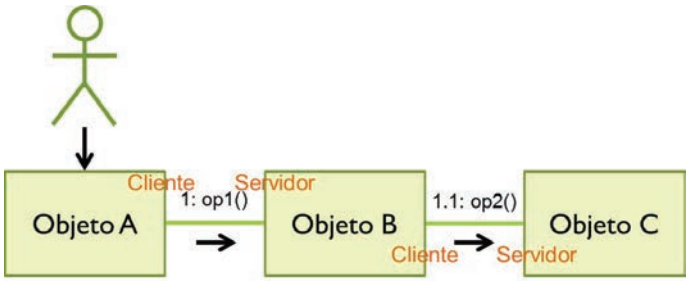

Figura 11.3: Roles cliente y servidor en diferentes invocaciones.

Los componentes clientes y servidores que participen en una determinada interacción podrán residir en máquinas diferentes. Tanto si es así como si la comunicación es local, se establece un mecanismo general de comunicación conocido como *secuencia petición-respuesta*. La figura 11.4 ilustra esta secuencia. En ella, los intervalos en que cada proceso está activo aparecen con trazo grueso, mientras que los intervalos en que han permanecido suspendidos se muestran con trazo fino y discontinuo. La interacción es iniciada por el cliente que genera una *petición* al servidor. En caso de asumirse una arquitectura basada en objetos, en esa petición se especifica qué método de la interfaz del servidor se desea invocar y qué argumentos se facilitan. Cuando el servidor recibe la petición, se activa e inicia un intervalo de servicio en el que se procesa esa petición, realizando todas las tareas asociadas a ella. Dicho intervalo de servicio finaliza con la generación de una *respuesta* y su envío al cliente. El cliente habrá permanecido suspendido desde que emitió su petición. Dicho intervalo de bloqueo finaliza al recibir la respuesta. Con ella, la interacción cliente-servidor terminará y el cliente reanudará su ejecución.

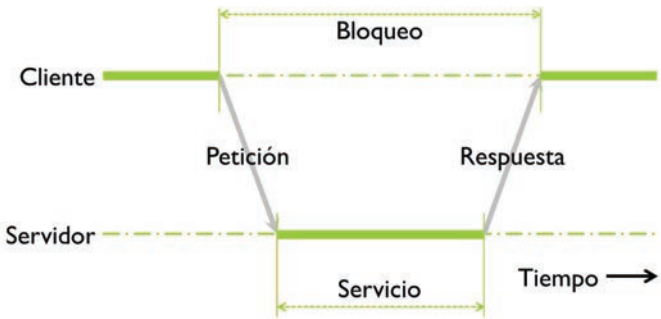

Figura 11.4: Secuencia petición-respuesta en una interacción cliente-servidor.

La mayor parte de las aplicaciones que siguen el modelo cliente-servidor presentan una arquitectura software en niveles. Se suelen distinguir tres niveles (o capas) en esta arquitectura software:

- *Capa de interfaz*. Contiene aquellos componentes que ofrecen la interfaz de usuario de este tipo de aplicaciones. Es el nivel superior de la arquitectura.
- *Capa de procesamiento*. También se la conoce como "*capa de lógica de negocio*". Es la que contiene todas las reglas de funcionamiento de la aplicación. Suele organizarse sin integrar datos concretos, que se tomarán a partir de las entradas proporcionadas por los usuarios y de la capa de datos persistentes. Los resultados se retornarán al usuario y puede que provoquen también la actualización de la capa de datos inferior. La capa de procesamiento define el nivel intermedio de esta arquitectura.
- *Capa de datos*. Mantiene de manera persistente los datos que los usuarios manipularán mediante el resto de componentes de la aplicación. Esta capa suele estar soportada por un *sistema gestor de bases de datos* (SGBD). Es el nivel inferior de la arquitectura.

A la hora de implantar esta arquitectura software y definir su arquitectura de sistema se suele seguir una *distribución vertical*. Esto es, cada uno de los niveles puede implantarse en un ordenador distinto.

Dentro de cada uno de los niveles podrá haber múltiples componentes que colaborarán entre sí para ofrecer la funcionalidad asumida en esa capa. Si en un sistema concreto únicamente dispusiéramos de dos máquinas para instalar estos componentes (una cliente, en la que el usuario utilizará el sistema, y otra servidora, en la que tendremos aquellos componentes no ubicados en la máquina cliente), se podría generar una de las cinco arquitecturas de sistema que se muestran en la figura 11.5.

En esta figura se observa en su mitad izquierda la arquitectura software utilizada, que sigue la estructura en tres capas que hemos descrito. En cada una de las capas se muestran algunos componentes. En la mitad derecha aparecen los dos ordenadores utilizados para "desplegar" los componentes. Mediante las líneas horizontales discontinuas se ha representado cada una de las cinco alternativas que se podrían utilizar a la hora de repartir los componentes. Así, en la alternativa (a) solo algunos de los componentes de la capa de interfaz residirían en la máquina cliente y todos los demás estarían en la máquina servidora. Por el contrario, en la alternativa (e) la mayor parte de la aplicación residiría en la máquina cliente y solo algunos componentes de la capa de datos se mantendrían en la máquina servidora. Ante la cuestión de qué variante convendría utilizar, la respuesta depende del parámetro que se pretenda optimizar. Si el objetivo fuera mejorar la escalabilidad y el rendimiento percibido por los usuarios, en ese caso interesaría

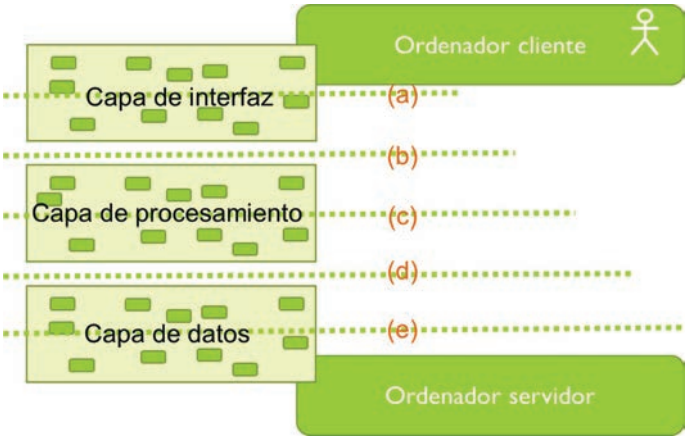

Figura 11.5: Cinco ejemplos de despliegue de una arquitectura en capas.

adoptar la alternativa (e) o (d) pues así el ordenador servidor solo sería responsable de la gestión de datos y cada usuario debería preocuparse por descargarse los componentes necesarios para ejecutar la aplicación (todo lo relacionado con las capas de interfaz y procesamiento) en su propia máquina y ejecutarlos allí. Así, aunque el número de usuarios creciera, cada uno tendría que aportar su ordenador para cubrir la mayor parte del trabajo que generase y el sistema podría escalar fácilmente. Por el contrario, si el objetivo fuera asegurar tanto la seguridad como la consistencia de los datos gestionados y de la propia aplicación, la opción más recomendable sería la (a) o la (b) pues cada una de ellas expone una parte mínima de la aplicación bajo el dominio de cada usuario, con lo que se evita que éstos puedan "trastear" con aquellos componentes que deban descargarse en sus propias máquinas.

De manera general, el número de ordenadores que podrán utilizarse para definir la arquitectura del sistema no tiene por qué estar fuertemente condicionado. A la hora de desarrollar cada componente se utilizarán los mecanismos de comunicación necesarios para que éstos puedan interactuar entre sí con independencia de su ubicación. Dichos mecanismos se describen en detalle en la Unidad 8. Por tanto, se deben tener en cuenta otros factores para definir la arquitectura de sistema, tales como:

- Interesará instalar en una misma máquina aquellos componentes que necesiten intercambiar mucha información entre sí, pues de otra manera se necesitaría un alto número de mensajes y se incrementaría excesivamente el uso de la red de interconexión. En la Unidad 8 se muestra que el uso de la comunicación por red sincroniza la ejecución de las actividades. Por ejemplo, quien deba recibir un mensaje permanecerá suspendido hasta que tal men-

saje llegue. Por ello, los componentes se bloquearían con excesiva frecuencia si el número de interacciones entre ellos fuera alto. Además, si hubiera algún problema en la red, estos componentes verían comprometido el progreso de sus actividades.

- Por otra parte, debe evaluarse con cuidado la capacidad de cada máquina así como la carga de peticiones que podrá recibir cada componente que se pretenda instalar en ellas. Si instalásemos demasiados componentes en un mismo ordenador, se minimizaría el tráfico que habría en la red pero el rendimiento obtenido sería muy bajo, pues cada componente no dispondría de suficiente tiempo de procesador para servir en el tiempo previsto cada una de las peticiones recibidas. Esto debe evitarse.

Por tanto, durante la fase de desarrollo y depuración de una aplicación distribuida se tendrá que evaluar el rendimiento que obtenga cada componente y modificar el despliegue inicialmente previsto en caso de que alguno de los ordenadores se sature (repartiendo los componentes que inicialmente se instalaron en él, pasando parte de ellos a otras máquinas), o bien se incremente excesivamente el retardo en las comunicaciones (redistribuyendo los componentes de manera que aquellos que generasen un alto tráfico de mensajes por estar separados pasen a residir ahora en una misma máquina).

11.3.2 Arquitecturas descentralizadas

Como ya se ha comentado previamente, una *arquitectura descentralizada* es aquella que utiliza distribución horizontal. Cuando se emplea distribución horizontal cada componente de la arquitectura software se divide en varias partes lógicamente equivalentes, y cada parte se ubica en un nodo diferente. Al aplicar una distribución horizontal, el sistema resultante se verá obligado a utilizar algoritmos descentralizados (explicados en la Unidad 7), potenciando así la escalabilidad de tamaño.

Si la distribución horizontal se aplica sobre los servidores, se generará un *sistema replicado*. Si se dividen horizontalmente los clientes y se eliminan los servidores, se genera un *sistema "peer-to-peer"*.

Un ejemplo típico de sistema replicado lo podemos encontrar en los servidores web (Figura 11.6). Normalmente el servidor se replica sobre múltiples máquinas (S1, S2, ..., SN) conectadas por una misma red de área local. Otra máquina (el *propagador* en la figura 11.6) da conexión hacia el exterior y recibe todas las peticiones que llegan al sistema. Su misión es redirigir estas peticiones a cada uno de los servidores. Para hacer esto, la solución más sencilla consiste en repartir las peticiones siguiendo un turno rotatorio. Así, la primera petición iría al servidor S1, la segunda al S2, la tercera al S3, y así sucesivamente hasta que todos los servidores tuvieran una. Después, en el siguiente turno de reparto se volvería a empezar del mismo

modo. Con esto se repartiría la carga equitativamente entre todos los servidores. Cada réplica mantendría en un disco local una copia completa de todas las páginas web atendidas por el servidor replicado. Si en algún momento se modificara alguna página (cosa que haría el administrador del sistema), se copiaría la nueva versión sobre todas las réplicas servidoras.

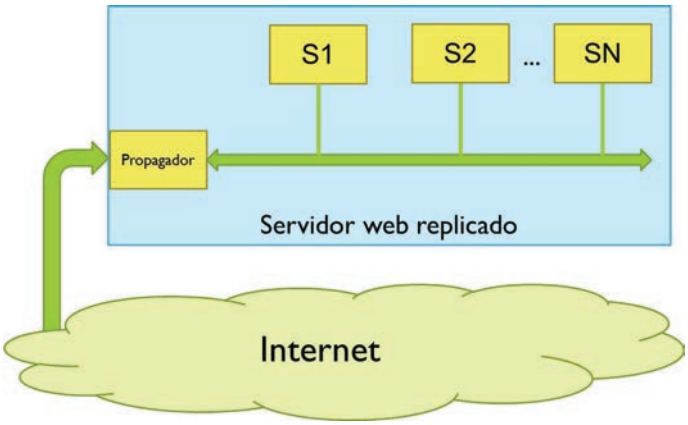

Figura 11.6: Distribución horizontal en un servidor web replicado.

Obsérvese que el nodo propagador no "centraliza" el algoritmo de servicio. Su funcionalidad no guarda relación directa con los algoritmos de servicio de páginas web ni con el protocolo HTTP. Él simplemente reenvía los mensajes a la réplica servidora que cree que estará libre, pero no coordina a esas réplicas (desconoce en qué estado se encuentra cada una en cuanto al servicio de las peticiones que ha redirigido) ni toma ninguna decisión que pueda afectar o condicionar sus algoritmos de servicio. De hecho, en cada réplica se puede seguir utilizando un servidor web diseñado para una única máquina. La gestión de la replicación se limitará a que el propagador observe si hay algún nodo servidor que no funcione, para dejar de enviarle nuevas peticiones y avisar al administrador para que lo sustituya.

Para un usuario de este servidor replicado resultaría imposible advertir que hay múltiples máquinas atendiéndole. Desde el exterior solo se observa una única dirección IP (la del propagador) y el resto del sistema permanece invisible. Con esta distribución horizontal se estaría consiguiendo transparencia de replicación y transparencia de fallos. A su vez, el propagador también tendría que ser un nodo fiable y esto implicaría que tendría que disponer de hardware replicado (múltiples tarjetas de red, múltiples buses, múltiples procesadores,...) para evitar que el fallo de cualquiera de sus componentes dejara al sistema inservible.

Sistemas "Peer-to-peer"

Según [AS04] un *sistema peer-to-peer* (o sistema P2P) puede definirse como "un sistema distribuido compuesto por nodos interconectados capaces de autoorganizarse en determinadas topologías con el propósito de compartir recursos tales como contenido, procesador, almacenamiento o ancho de banda; con capacidad para adaptarse a los fallos y a distribuciones transitorias de los nodos manteniendo a la vez una conectividad y rendimiento aceptables, y sin necesitar la mediación o soporte de un servidor o autoridad centralizada global". Al no existir ningún servidor o autoridad centralizada global, todos los nodos de estos sistemas deberían adoptar un mismo rol a la hora de interactuar con otros nodos. Ese sería el comportamiento ideal en un sistema de este tipo. Sin embargo, pocos sistemas reales llegan a cumplir estrictamente esta definición.

En líneas generales, cualquier sistema o aplicación peer-to-peer debería presentar estas características:

- Es un sistema distribuido de gran escala con amplia distribución geográfica. Gran parte de estas aplicaciones se utilizan para distribuir ficheros y las propias aplicaciones están disponibles en la red y pueden ser descargadas e instaladas por cualquier usuario. Por ello, no hay apenas restricciones para su distribución geográfica.

- En estos sistemas habrá un gran número de nodos poco fiables. El término "fiabilidad" referido a una aplicación distribuida expresa [Nel90] la probabilidad condicionada de que tal aplicación pueda realizar la función para la que fue diseñada en el instante t sabiendo que estaba operativa en el instante $t = 0$. Los nodos de un sistema P2P son poco fiables pues sus usuarios pueden apagarlos o desconectarlos de la red a voluntad, incluso mientras se estuviera realizando alguna transacción con otros nodos del sistema.

  A pesar de la baja fiabilidad de cada uno de los nodos que formen el sistema, los ficheros que se suelen distribuir suelen permanecer disponibles durante largo tiempo. Cuando un usuario está bajando un fichero determinado, aunque alguno de los nodos que proporcionen los fragmentos de tal fichero se desconecte, siempre habrá otros nodos que guardarán otras copias del mismo fichero. El fragmento en cuestión se obtendrá de otro nodo que mantenga alguna de esas copias. Para ello, los algoritmos que gestionan las descargas se adaptan adecuadamente a estas situaciones.

- Los nodos que participan en estas aplicaciones realizan tareas similares y ofrecen una funcionalidad equivalente. Respetan así la distribución horizontal que se ha comentado anteriormente.

Una de las características más delicadas de un sistema P2P descentralizado es su servicio de localización de recursos, entendiendo como recurso aquel elemento que

se pretende compartir en una aplicación de este tipo: espacio de almacenamiento, ciclos de cómputo, ficheros, ancho de banda, etc. Los usuarios de estas aplicaciones necesitan utilizar dicho servicio de localización para encontrar ese recurso que se pretenda obtener (por ejemplo, qué máquinas tienen el fichero que pretendemos bajar en el caso de una aplicación de distribución de ficheros, qué ordenadores son más fiables y están usando menos su ancho de banda en el caso de una aplicación como Skype para transmisión de voz o vídeo, etc.). La organización de este servicio de localización condiciona la topología de la red lógica ("*overlay network*") que los nodos definen para interactuar entre sí. Esta red lógica puede clasificarse en función de su grado de centralización y de su estructura [AS04].

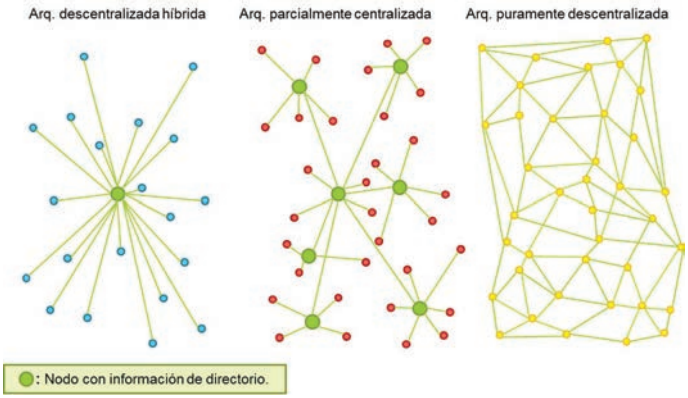

Figura 11.7: Grados de centralización en un sistema P2P.

Considerando el **grado de centralización** se pueden distinguir las siguientes clases de red lógica (véase la figura 11.7):

- *Arquitecturas puramente descentralizadas*: En este caso todos los nodos realizan exactamente las mismas funciones y no hay ningún nodo que actúe como servidor de localización (es decir, que mantenga un directorio en el que se guarde una colección de entradas compuestas cada una por un identificador de fichero y la lista de las direcciones de los nodos que mantengan copias de éste). Los nodos de un sistema que siga esta arquitectura se suelen llamar "*SERVENTs*" (nombre generado como contracción de "server" y "client").

  Para encontrar un fichero en una arquitectura como ésta, el nodo que lo busque propaga entre todos sus vecinos un mensaje que contendrá el identificador del fichero buscado o algún patrón o característica o palabra clave que deban cumplir todos los ficheros a obtener. Aquellos nodos que posean algún fichero que satisfaga dicha búsqueda, retornarán su propia dirección y la lista de ficheros que la cumplan. Con ello, el nodo iniciador podrá establecer conexiones con ellos y bajar tales ficheros. Por el contrario, los nodos

que no dispongan de ningún fichero para responder, repropagarán el mensaje recibido entre sus propios vecinos y con ello el mensaje de búsqueda se irá difundiendo progresivamente por la red. Para evitar que esos mensajes se propaguen indefinidamente, los algoritmos empleados suelen incluir en el mensaje un número máximo de saltos de propagación que se irá decrementando en cada reenvío. Cuando dicho contador llegue a cero, los nodos que lo reciban ya no lo reenviarán a otros.

Otro problema que conviene resolver es la obtención de alguna lista inicial de nodos con los que contactar para incorporarse al sistema. Es decir, los demás nodos deben darse cuenta de que un nuevo nodo se está incorporando y el propio nodo nuevo debe poder obtener las direcciones de otros que ya formaran parte del sistema. Para conseguir todo esto la mayor parte de las aplicaciones P2P puramente descentralizadas incorporan en su ejecutable la dirección de algunos nodos "estables". Cuando el programa es iniciado por un usuario, se pasa a contactar con estos nodos estables y alguno de ellos devolverá direcciones de nodos con los que poder interactuar para iniciar nuevas búsquedas de ficheros. Los propios nodos estables no pueden utilizarse para realizar búsquedas de recursos ya que de esa manera se sobrecargarían en exceso y estarían centralizando el servicio de localización.

Un ejemplo de sistema de este tipo es *Gnutella* [Bab04], utilizando la versión 0.4 de su protocolo. En esa versión, el número medio de "vecinos" para cada nodo en la red lógica era 5 y el límite de pasos de propagación para los mensajes de búsqueda se estableció en 7.

- *Arquitecturas parcialmente centralizadas.* Esta arquitectura se inspira en la anterior, pero los nodos con mayor capacidad de cómputo o de ancho de banda se responsabilizan de "recordar" qué recursos mantienen sus vecinos directos. De esta manera, esos nodos adoptan un rol especial (pasan a llamarse *supernodos*) y los mensajes de búsqueda son dirigidos inicialmente a ellos. A su vez, cada supernodo conoce a otros supernodos y repropaga las búsquedas en ellos, reduciendo la participación de los nodos "normales".

  Un ejemplo de sistema de este tipo es el protocolo *FastTrack* empleado por algunas aplicaciones P2P como Kazaa y Mammoth. Este protocolo también sirvió como base para desarrollar *Skype* [GDJ06], aunque en este caso los supernodos se utilizan para trazar las rutas sobre las que transmitir voz y/o vídeo entre sus usuarios.

- *Arquitecturas descentralizadas híbridas.* En esta arquitectura existe un servidor central de directorio, al que cada nodo podrá preguntar por los recursos que quiera obtener o utilizar. Por ello, la gestión del servicio de localización está centralizada. Sin embargo, una vez se haya localizado el recurso y se haya obtenido la dirección de la máquina en la que resida, el servidor central ya no será responsable de nada más. La interacción entre el usuario y la máquina que mantenga el recurso se podrá realizar sin tener que utilizar

ningún otro intermediario. Si el sistema P2P estuviera destinado a distribuir ficheros, este servidor central no mantendría réplicas de ninguno de los ficheros existentes en el sistema. Si tal servidor fallara en algún momento, no podrían realizarse nuevas búsquedas, pero aquellos nodos que ya conocieran quién mantiene los ficheros de su interés podrían interactuar con tales nodos sin ninguna dificultad.

Por tanto, en esta arquitectura se centraliza el servicio de localización pero se descentraliza el servicio de transferencia de datos. Eso explica por qué se califica a estos sistemas como "híbridos".

Napster fue uno de los primeros sistemas que ofreció una arquitectura de este tipo.

Por otra parte, si se tiene en cuenta la **estructura de la red lógica** es posible distinguir estas arquitecturas:

- *Arquitectura no estructurada.* En un sistema no estructurado, los recursos pueden ubicarse en cualquier nodo del sistema y su ubicación no depende de la topología lógica que tenga el sistema. Para localizar un recurso tendrá que utilizarse algún servicio que no dependerá de la topología.

- *Arquitectura estructurada.* En una arquitectura estructurada los recursos se mantienen en posiciones concretas de la red lógica. La propia red lógica define cierta estructura que facilita la función de localización de los recursos. Con esta ayuda resulta muy sencillo localizar un recurso.

  Ha habido multitud de propuestas para definir arquitecturas estructuradas con servicios de localización muy eficientes. Algunos ejemplos son: Chord [SMK+01], CAN [RFH+01], Pastry [RD01], ...

  Para ilustrar su funcionamiento, la figura 11.8 muestra un ejemplo de red lógica en el sistema Chord. En Chord, cada nodo del sistema tiene un identificador cuyo valor es un número natural. Los nodos se estructuran de manera lógica en un anillo, en función de su identificador. El espacio de identificación será grande (números positivos de 48 bits, por ejemplo), pero para no complicar el ejemplo asumiremos que trabajamos con naturales de 3 bits (valores entre 0 y 7, ambos inclusive). Obsérvese que se sigue un orden circular. Por ello, el valor cero se considera también superior al 7. Los recursos también reciben un identificador en ese mismo rango, aplicando una función de *hashing* a su contenido. Como el dominio para los identificadores es muy amplio, por regla general en el anillo habrá pocos lugares ocupados por sus respectivos nodos. En la figura 11.8 solo los nodos 1, 4 y 7 están funcionando en ese sistema. Eso se ha representado resaltando tales posiciones con puntos más grandes de color verde. Cada nodo será responsable de mantener aquellos recursos que tengan un identificador igual o inferior al suyo, pero superior al del anterior nodo dentro del anillo. Así, el nodo 1 se encarga de mantener

aquellos recursos cuyo identificador sea también 0 o 1. Por su parte, el nodo 4 mantendrá los recursos con identificador 2, 3 o 4 y, por último, el nodo 7 mantendrá aquellos recursos cuyos identificadores pertenezcan al conjunto {5, 6, 7}.

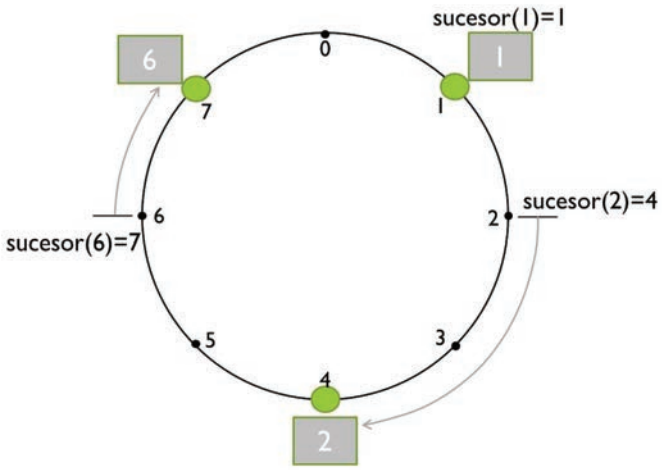

Figura 11.8: Ubicación de recursos en el sistema Chord.

En la figura se observa que, hasta el momento, solo se han registrado tres recursos en el sistema, cuyos identificadores son 1, 2 y 6 (representados mediante cuadrados) y que residen en los nodos 1, 4 y 7, respectivamente. Este ejemplo demuestra que la propia topología de la red lógica impone las reglas a seguir para localizar un recurso dentro del sistema. Si conocemos los identificadores de los nodos existentes y el criterio utilizado para ubicar los recursos, sabremos en qué nodo se guardará el recurso que busquemos.

Sin embargo, estos sistemas son dinámicos (poco fiables): los nodos van entrando y saliendo del sistema a voluntad. Por ello se deben mantener múltiples copias de cada recurso (no solo en el nodo que dicte la estructura sino en algunos más que estén próximos en la red lógica, por ejemplo). Además, cada nodo del sistema debe tener alguna tabla que le indique cuáles son los siguientes nodos en el anillo y a qué distancia de él se encuentran. Es decir, para mantener esta estructura también se necesitan algunas tablas similares a las utilizadas en el encaminamiento de la red y estas tablas deben actualizarse en caso de que lleguen nuevos nodos o se desconecten algunos de los que formaban parte del sistema. En [SMK+01] se describen estos mecanismos en profundidad.

	Grado de centralización		
	Híbrido	Parcial	Descentr.
No estructurado	Napster	Mammoth Kazaa Gnutella 0.6	Gnutella 0.4
Estructurado	–	–	CAN Chord Pastry

Tabla 11.1: Clases de sistemas P2P.

No todas las combinaciones de estructura y grado de centralización de la red lógica llegan a ser posibles. La tabla 11.1 refleja qué alternativas tienen sentido, presentando ejemplos de sistemas P2P que ofrecen esas características.

Sistemas "Grid"

Un segundo tipo de arquitectura de sistema descentralizada y escalable es la de los *sistemas grid* [FK98]. Un sistema de este tipo consiste en una federación de recursos computacionales ubicados en múltiples dominios administrativos que persiguen un objetivo común. Por tanto, en un sistema de este tipo hay que potenciar los tres tipos de escalabilidad: tamaño (para utilizar tantos nodos como sea posible, aunque el número de usuarios será normalmente bajo), de distancia (pues las organizaciones que colaboran en estos sistemas pueden abarcar una amplia extensión geográfica) y administrativa (pues habrá múltiples organizaciones en un sistema de este tipo).

Las aplicaciones que suelen ejecutarse en estos sistemas no son interactivas y necesitan un elevado volumen de información que suele ubicarse en múltiples ficheros. Son aplicaciones de computación numérica que buscan un alto grado de paralelismo en su ejecución. Por ello, cuantos más ordenadores puedan utilizarse simultáneamente, antes se obtendrán los resultados esperados.

Los sistemas grid se están empleando para ejecutar todas aquellas aplicaciones que requieran un alto rendimiento computacional. Algunos ejemplos serían: secuenciación del genoma humano, cálculo de estructuras en proyectos de arquitectura, diagnóstico médico basado en imagen, etc.

Para construir un sistema de este tipo suele utilizarse un *middleware* [Ber96] que proporciona la imagen de sistema único. *Globus Toolkit* [FK98] es uno de los más antiguos, pero también el más utilizado actualmente.

La arquitectura de sistema está basada en distribución horizontal para utilizar múltiples ordenadores dentro de cada nivel de la arquitectura lógica. Los sistemas grid son abiertos pues utilizan protocolos estandarizados para garantizar una correcta interacción entre sus componentes.

Sistemas "Cloud"

Un *sistema cloud* se define [VRMCL09] como "un gran almacén de recursos virtualizados fácilmente utilizables y accesibles (tales como infraestructura, plataformas de desarrollo y/o servicios). Estos recursos pueden ser reconfigurados dinámicamente para adaptarse a una carga variable (escalabilidad), facilitando una utilización óptima de cada recurso. Un conjunto de recursos se utiliza típicamente bajo un modelo de pago por uso en el que las garantías son ofrecidas por el proveedor del sistema mediante un SLA ("*Service level agreement*", o acuerdo de calidad de servicio)."

Como el conjunto de recursos que resultará necesario en una determinada aplicación distribuida se obtendrá de un proveedor cloud y éste los gestionará en sus propias instalaciones, los usuarios perciben a estos sistemas como algo "externo" cuya ubicación, administración y uso resulta "transparente". Por ello se habla de una "nube" al referirnos a ellos: una entidad externa que ofrecerá los recursos solicitados y por la que no habrá que preocuparse.

Para los usuarios, la principal ventaja de estos sistemas es que solo utilizaremos aquellos recursos que realmente se necesiten y que solo habrá que pagar por lo que se haya usado. Muchas veces resulta difícil prever qué inversión en equipamiento informático necesita una empresa que deba gestionar alguna aplicación web, pues no siempre será sencillo prever cuántos usuarios utilizarán sus servicios y qué carga impondrá cada uno de ellos. Con el uso de un sistema cloud eliminaríamos tal incertidumbre, pues resulta sencillo adaptarse a cualquier número de usuarios.

Una segunda ventaja es que se puede acceder a cualquiera de los recursos ubicados en la nube en cualquier momento y desde cualquier tipo de ordenador. Lo único que debe cumplir tal ordenador es que disponga de conexión a Internet.

Los sistemas cloud dan soporte al concepto de "*utility computing*":

- Empresas con exceso de capacidad de cómputo pueden de forma rentable dejar usar sus sistemas a múltiples clientes.
- A su vez, empresas con demanda de capacidad de cómputo pueden alquilar la infraestructura de quien ofrezca el mejor servicio o precio. Gracias a ello, no se necesitará realizar ninguna inversión en ninguno de estos tres conceptos:

- Adquisición de equipos y de locales donde poder mantenerlos. En lugar de construir un centro de datos propio, se alquilará a un proveedor cloud cierto número de equipos virtuales.
- Administración del sistema. Si se ha llegado a comprar un conjunto de máquinas y se ha instalado la red necesaria para interconectarlos, también se tendrá que contar con personal que pueda administrar el sistema resultante, gestionando su configuración y adaptándola a las necesidades de las aplicaciones que se lleguen a instalar. Son labores complejas que precisan de personal especializado.

  Al utilizar los servicios cloud, la responsabilidad de la administración de los recursos utilizados recae en el propio proveedor cloud. Por ello, sus usuarios no tienen por qué preocuparse por tales tareas.
- Consumo eléctrico. La gestión de la instalación eléctrica y el pago de los costes relacionados con el suministro eléctrico necesario para que los equipos funcionen corre a cargo del proveedor del sistema cloud.

Así, el término "utility computing" resalta que la gestión de los recursos informáticos pasará a ser una herramienta más sobre la que se podrá comerciar. Habrá proveedores de tal tipo de servicio (aquellos que faciliten sus sistemas cloud) y usuarios para éstos (aquellos que instalen sus aplicaciones informáticas en estos entornos o utilicen dichas aplicaciones).

Dependiendo del tipo de recurso informático que se esté facilitando a los usuarios, se pueden distinguir tres tipos de sistemas cloud [VRMCL09]:

- *Infraestructura como servicio* (IaaS, "*Infrastructure as a Service*"): En este caso el recurso que se está facilitando es el propio hardware, sin ningún complemento relacionado con el software: los ordenadores completos, así como la infraestructura de red necesaria para intercomunicarlos. Gracias a los mecanismos de virtualización que incorporan los procesadores modernos no será necesario asignar máquinas reales a cada usuario. Basta con alquilar máquinas virtuales con ciertas prestaciones y después gestionar cómo se soportan tales máquinas virtuales sobre los equipos del proveedor.

  En un sistema como éste los usuarios deben generar "imágenes" de máquina virtual en las que decidirán qué sistema operativo utilizar y qué otros componentes habrá que instalar sobre éste: los que formen a la aplicación que se quiera ejecutar en estos sistemas cloud. Una vez creadas tales imágenes de máquina virtual, se podrán desplegar sobre la infraestructura y así arrancar la aplicación.

  Ejemplos de sistemas IaaS son: Amazon EC2, Sun Grid e IBM Blue Cloud.

  Dentro de este mismo tipo de sistema también se llega a ofrecer la capacidad de almacenamiento persistente como un recurso específico (como un componente especial de tal infraestructura que se podrá administrar y alquilar

de forma independiente). En ese caso se puede hablar de "Almacenamiento como servicio" y también existen diferentes propuestas de este tipo: Amazon EBS, Amazon S3, Amazon RDS, Nirvanix SDN, ...

- *Plataforma como servicio* (PaaS, "*Platform as a Service*"): En un sistema de este tipo se facilita una plataforma programable en lugar de una infraestructura. La interfaz facilitada dependerá de cada proveedor. Así, por ejemplo, en el sistema Google AppEngine se ofrece el soporte necesario para desarrollar servicios web, mientras que en Microsoft Azure se ofrece la interfaz necesaria para desarrollar aplicaciones bajo su modelo ".NET". En cualquiera de estos casos, el propio sistema se encarga de decidir cuántos ordenadores serán necesarios para soportar tales aplicaciones, cuántas réplicas habrá de cada dato manejado por la aplicación, etc. y esos detalles no le importarán al usuario de estas plataformas (no solo al ejecutar la aplicación, sino también a la hora de programarla).

  Por todo esto un sistema PaaS proporciona un soporte cómodo y eficiente para desarrollar aplicaciones distribuidas escalables. Todo lo relacionado con la gestión de tal escalabilidad es responsabilidad del proveedor cloud.

- *Software como servicio* (SaaS, "*Software as a Service*"): Existe un buen número de aplicaciones típicas que gran número de usuarios informáticos llegan a utilizar. Un primer ejemplo son los procesadores de textos y hojas de cálculo. En algunos sistemas cloud son estas mismas aplicaciones lo que se ofrece al usuario. De esta manera, no es necesario instalar localmente una aplicación de este tipo: basta con utilizar un navegador web, acceder al sistema cloud cargando la página inicial de tal aplicación (tras habernos identificado) y utilizar la aplicación.

  En la mayoría de los casos, el usuario no solo obtiene acceso a una aplicación remota sino que también dispone de un almacenamiento persistente ligado a tal aplicación. Así, por ejemplo, no solo utilizamos un procesador de textos sino que también mantenemos en el sistema cloud todos los documentos que hemos generado con él (que también podremos descargar cuando resulte necesario). Además, podremos otorgar permisos de edición a otros usuarios del sistema y así colaborar fácilmente en su redacción.

  Ejemplos de sistemas de este tipo son: Google Docs (que ha pasado a llamarse Google Drive en abril de 2012), Movistar Aplicateca y Google Apps for Business (dirigidos a empresas), Microsoft Windows Live, Apple iCloud (aunque sus prestaciones como aplicación son limitadas y podría considerarse un "almacenamiento como servicio"), etc.

  Las principales ventajas que observa el usuario al utilizar un sistema SaaS son:

  - No resulta necesario instalar las aplicaciones en el ordenador local.

- Podemos reanudar el uso de la aplicación desde cualquier otro ordenador conectado a Internet.
- Se admite el trabajo colaborativo por parte de múltiples usuarios en la gestión de un mismo proyecto (por ejemplo, edición de un documento con una aplicación compartida de proceso de textos).
- El software puede actualizarse de manera totalmente transparente: no implica ninguna interrupción en las sesiones de trabajo de los usuarios ni la necesidad de actualizar nada en los ordenadores que éstos utilicen.

11.4 Resumen

La arquitectura de cualquier aplicación informática define cómo se estructuran los componentes que la forman y qué tipo de consecuencias tendrá tal organización. En las aplicaciones distribuidas se distinguen dos tipos de arquitectura: lógica y física. La *arquitectura lógica* (o arquitectura software) determina cómo se estructuran los componentes de la aplicación durante su diseño y qué interacciones se podrán establecer entre dichos componentes. La *arquitectura física* (o arquitectura de sistema) define en qué máquinas se instalan los componentes y qué redes se necesitan para comunicarlos.

En esta unidad se han descrito las variantes más relevantes de cada tipo de arquitectura. Así, para las arquitecturas software, se han descrito los modelos o estilos arquitectónicos que se siguen con mayor frecuencia: arquitectura de capas o niveles, arquitectura basada en objetos, arquitectura centrada en datos y arquitectura basada en eventos.

Por su parte, para las arquitecturas de sistema se han analizado las tres alternativas existentes para definirla: arquitectura centralizada, arquitectura descentralizada y arquitectura híbrida. La *arquitectura centralizada* adopta una distribución vertical y un modelo de interacción cliente-servidor. La *arquitectura descentralizada* adopta una distribución horizontal: sobre los servidores (sistemas replicados), o sobre los clientes (sistema peer-to-peer). Además, hace uso de algoritmos descentralizados. Como ejemplos de arquitecturas descentralizadas, se han analizado los distintos tipos de sistemas peer-to-peer (en función de su grado de centralización y de su estructura), así como los sistemas grid y los sistemas cloud.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Distinguir entre arquitectura software y arquitectura de sistema.
- Clasificar diferentes modelos de arquitectura software.
- Clasificar las alternativas de arquitecturas de sistema.

- Describir el modelo de interacción cliente-servidor y el mecanismo de comunicación de secuencia petición-respuesta.
- Distinguir entre distribución vertical y distribución horizontal.
- Identificar las características de los sistemas peer-to-peer y clasificarlos en función de su grado de centralización y de la estructura de la red lógica.
- Distinguir entre sistemas grid y sistemas cloud. Clasificar los diferentes tipos de sistemas cloud.

Unidad 12

DIRECTORIO ACTIVO DE WINDOWS SERVER

12.1 Introducción

Los ordenadores existentes en cualquier organización se encuentran formando parte de redes de ordenadores, de forma que puedan intercambiar información. Desde el punto de vista de la administración de sistemas, la mejor manera de aprovechar esta característica es la creación de un dominio de sistemas, en donde la información administrativa y de seguridad se encuentra centralizada en uno o varios servidores, facilitando así la labor del administrador.

Active Directory [RKMW08] es la denominación que da Microsoft a su solución para organizar y administrar una red de ordenadores empresarial. Esta unidad describe diferentes componentes y servicios integrados en *Active Directory*, incidiendo especialmente en los conceptos básicos de esta solución.

12.2 Servicios de dominio

Un *dominio* en *Active Directory* es un conjunto de ordenadores y usuarios, donde usuarios y ordenadores son autenticados e identificados por los servicios del propio dominio. El conjunto de ordenadores que forma un determinado dominio recibirá sus nombres dentro de un mismo dominio DNS cuyo nombre se utiliza también para nombrar el dominio *Active Directory*. La figura 12.1 representa gráficamente el dominio *eovic.csd* en el que es posible identificar algunos ordenadores y usuarios.

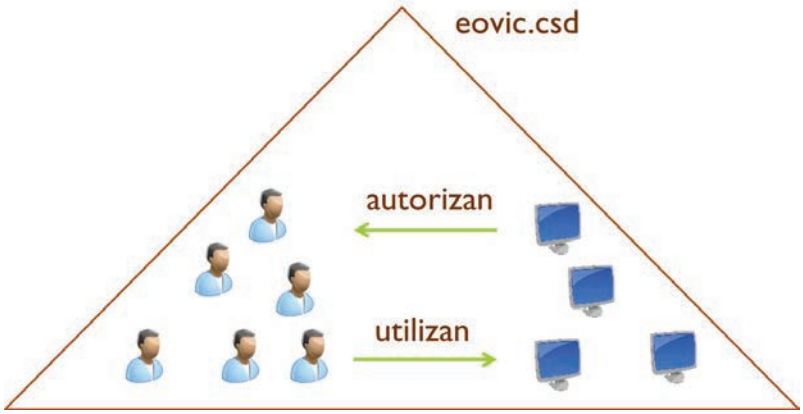

Figura 12.1: Elementos en un dominio.

Los *Servicios de Dominio* en *Active Directory* (que abreviaremos como AD DS, por ser las siglas inglesas para identificar estos servicios: *Active Directory Domain Services*) almacenan toda la información relevante de los elementos que constituyen el dominio. Tales elementos son: usuarios, ordenadores, grupos de usuarios, ... Esta información se utiliza en los procesos de autenticación y autorización que suceden en el dominio. La *autenticación* consiste en comprobar la identidad de un usuario. Para ello pueden utilizarse diversos mecanismos, aunque lo más habitual es solicitar una contraseña y compararla con la que en su momento el usuario habrá registrado en el sistema. Por su parte, la *autorización* es el proceso mediante el que se comprueba que el usuario tiene permiso para realizar aquello que haya solicitado.

12.2.1 Controlador de dominio

Los servicios AD DS deben instalarse en un ordenador cuyo sistema operativo sea alguna edición de *Windows Server*; por ejemplo, Windows Server 2008 R2. A este ordenador se le denomina *controlador de dominio* (o DC, por ser la sigla para *Domain Controller*). Este controlador es el responsable de las tareas de autenticación de los usuarios del dominio.

Para asegurar la disponibilidad de estos servicios, resulta aconsejable que en un mismo dominio exista más de un DC. De esta forma, si alguno de ellos falla y deja de ser accesible, los demás seguirán ofreciendo servicio a los usuarios y máquinas del dominio. Cada uno de estos DC dispone de una copia de la información de los servicios AD DS. Tal información se replica sobre todos los DC y todos los DC adoptan un mismo rol: todos son capaces de aceptar modificaciones solicitadas por los usuarios para después propagarlas sobre el resto de DC.

La técnica de replicación utilizada en este servicio está basada en el principio de propagación de actualizaciones. Esto implica que:

- No se propaga todo el estado mantenido por un servidor cada vez que hay alguna modificación, sino solo aquellos elementos que hayan sido modificados. Esto reduce la cantidad de información a propagar.
- Cuando haya alguna modificación no se puede garantizar que todos los DC recibirán esa modificación inmediatamente. Con ello se pierde momentáneamente la consistencia entre todos los controladores, pero eso no es excesivamente grave. Tal consistencia se recuperará pronto: cuando las modificaciones hayan sido propagadas y aplicadas en todos los demás controladores. El intervalo necesario para tal propagación dependerá de la cantidad de información modificada, de la distancia existente entre los controladores (y el tipo de red que los interconecte), así como de la frecuencia de propagación que se haya configurado (por omisión [RKMW08, Capítulo 4], cada 15 segundos entre controladores de una misma subred y cada 3 horas entre controladores que deban utilizar una WAN para comunicarse).

Como ya se ha comentado previamente, todos los controladores de dominio adoptan un mismo rol por lo que respecta a su replicación. Cuando un usuario deba autenticarse podrá dirigir su petición a cualquiera de los controladores existentes y éste podrá responderle sin tener que interactuar con otras réplicas. Le basta con su información local.

12.2.2 Árboles y bosques

En ocasiones convendrá crear dominios dentro de otros dominios ya existentes. Cuando eso suceda, los dos dominios presentarán una organización jerárquica (padre-hijo) en caso de representarlos mediante un grafo. Dicha relación puede extenderse sobre varios niveles: se podría crear un tercer dominio dentro del segundo, y así sucesivamente. En esos casos, los nombres de los dominios que se vayan creando extienden los nombres de sus antecesores. Por ejemplo, si el primer dominio se llamó *eovic.csd*, se habrá podido crear un segundo dominio dentro de él llamado *spain.eovic.csd* y después un tercero llamado *valencia.spain.eovic.csd*. A su vez, se pueden crear otras "ramas" dentro de esa misma organización jerárquica de dominios. Por ejemplo, un cuarto dominio llamado *france.eovic.csd*. Un conjunto de dominios organizados jerárquicamente mediante relaciones padre-hijo constituye un *árbol de dominios*. Los árboles pueden ser apropiados para estructurar los dominios de una empresa que presente una organización jerárquica.

En otros casos, una misma organización empresarial querrá gestionar múltiples árboles de dominios, sin ninguna relación padre-hijo entre el nombrado de tales árboles. Entonces se habla de un *bosque de dominios*. Esto puede tener sentido en

grandes corporaciones, en las que cada una de las empresas que las forman mantendrá su propio árbol, pero también es aplicable en otros contextos; por ejemplo, en empresas que prefieran un árbol diferente para cada uno de sus departamentos.

Todos los árboles de un mismo bosque, a pesar de no estar relacionados por los nombres DNS que utilicen en sus respectivos dominios, comparten una misma configuración (todos sus DC se conocen entre sí y saben que forman parte de un mismo bosque) y un mismo conjunto de usuarios administradores. Como resultado de esto, los directorios de todos los dominios contenidos en el bosque forman un directorio común. A su vez, todos los usuarios del bosque pueden ser autorizados a utilizar recursos de cualquier ordenador del bosque.

El primer dominio creado en un determinado bosque adopta el rol de *dominio raíz del bosque*. Sus DC serán los responsables de gestionar la creación o eliminación de nuevos dominios (y árboles) en ese bosque.

La figura 12.2 muestra una representación gráfica de un bosque en el que se distinguen tres árboles diferentes y en el que el primer dominio creado fue *eovic.csd*. Por tanto, el dominio raíz de este bosque es *eovic.csd*.

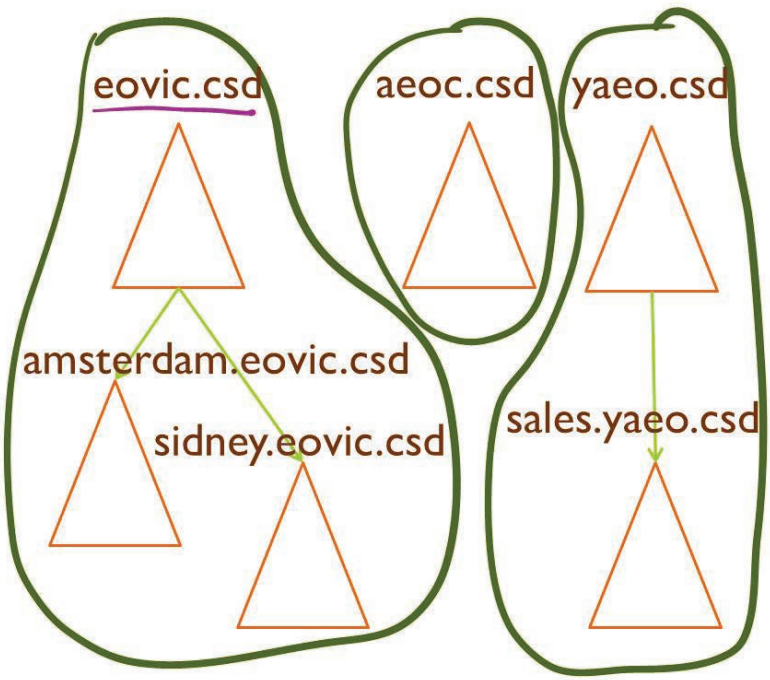

Figura 12.2: Ejemplo de bosque de dominios.

12.2.3 Principales protocolos empleados

Active Directory recurre a protocolos estándar para gestionar algunas de sus tareas administrativas. De esa manera, al utilizar protocolos que ya han demostrado su funcionalidad y eficiencia, se puede confiar en su correcto funcionamiento y además se facilita la interoperabilidad con aquellos sistemas que también los utilicen. Esta es la base para desarrollar un *sistema distribuido abierto*, como ya se explicó en la sección 7.3.3. Tres de estos protocolos son:

- *DNS* (ya descrito en la sección 10.4). Es el sistema de nombrado estándar para los ordenadores que formen parte de cualquier sistema distribuido (es decir, de aquellos que tengan acceso a la red de comunicaciones). Se utiliza para localizar los DC.
- *LDAP* (ya descrito en la sección 10.5). Proporciona un servicio de directorio distribuido basado en los atributos de las entidades registradas en tal directorio. Es la base de AD DS. Así se puede mantener de forma sencilla la información sobre los usuarios, grupos y ordenadores que formen parte de cada dominio.
- *Kerberos* [NYHR05]. Permite que dos entidades ubicadas en ordenadores que se comuniquen utilizando una red insegura puedan demostrar su identidad de manera segura. Se utiliza para autenticar usuarios y ordenadores.

12.3 Servicios de directorio

Uno de los módulos más importantes dentro de los servicios de dominio es el que ofrece los *servicios de directorio*: *Directory Data Store*. Su misión es la gestión del directorio, atendiendo las operaciones de creación y eliminación de elementos que constituyen la información del directorio, así como las operaciones que pueden desarrollar tales elementos.

Los elementos que podremos encontrar en este directorio son: usuarios, ordenadores, grupos de usuarios, contenedores y unidades organizativas. Cada dominio tiene su propio directorio y reside en los controladores de dominio. La información de un directorio de un dominio puede ser usada en cualquier otro dominio de ese mismo bosque.

Para acceder a las herramientas de administración del directorio, basta con seguir esta secuencia en un ordenador que sea DC:

1. Pulsar el botón “Inicio” del escritorio de Windows (normalmente ubicado en la barra inferior, en su esquina izquierda).

2. Seleccionar la entrada "Herramientas administrativas" que aparece en la parte derecha del menú emergente.
3. Seleccionar la entrada "Usuarios y equipos de Active Directory".

En el panel izquierdo de la ventana mostrada por esta aplicación se listarán los tipos de entrada que pueden mantenerse en un directorio: *Builtin*, *Computers*, *Domain Controllers*, *ForeignSecurityPrincipals*, *Managed Service Accounts* y *Users*.

La sección 12.3.1 explica la arquitectura de estos servicios, describiendo cada uno de sus componentes. Posteriormente, la sección 12.3.2 describirá cada uno de los objetos que aparecen en el directorio así como la gestión asociada a ellos.

12.3.1 Arquitectura y componentes

En este módulo de *Active Directory* (que está instalado en los DC) se pueden distinguir otros componentes que se encargan de implantar partes de estos servicios (véase la figura 12.3). Son los siguientes:

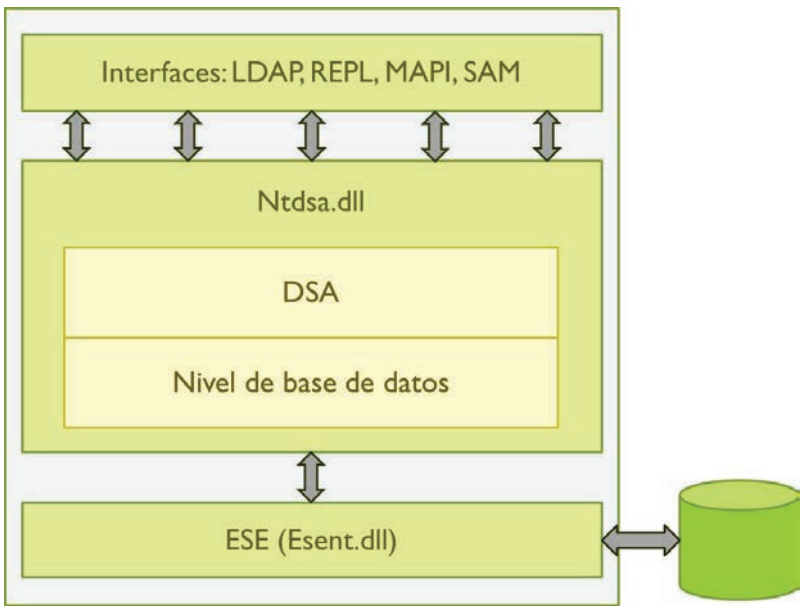

Figura 12.3: Arquitectura de los servicios de directorio en un DC.

- Para acceder a los servicios de directorio (tanto desde un ordenador cliente como desde otros DC) se ofrecen cuatro interfaces complementarias:
  - *LDAP*: Es el protocolo que define cómo los ordenadores clientes deben acceder a la información de directorio.

- *REPL*: Define la interfaz y protocolo utilizados para gestionar la replicación de la información contenida en los DC. Utiliza *RPC* como mecanismo de intercomunicación entre controladores de dominio para gestionar la propagación de los datos modificados.
- *MAPI* (*Messaging API*): MAPI es utilizado por aquellas aplicaciones clientes basadas en la gestión de mensajes para acceder a la información mantenida en el directorio. Un ejemplo de cliente de este tipo es *Outlook*, que utiliza esta interfaz para acceder a la información mantenida por *Microsoft Exchange Server* en el propio directorio. Al igual que en el caso de la interfaz REPL, MAPI utiliza RPC como mecanismo de intercomunicación.
- *SAM* (*Security Accounts Manager*): Esta última interfaz es una interfaz propietaria para permitir que los procesos clientes que se ejecuten sobre sistemas Windows NT 4.0 (o anteriores) puedan interactuar con el resto de los componentes de estos servicios de directorio. Utiliza RPC como mecanismo de intercomunicación, al igual que las dos interfaces anteriores.

- *DSA* (Agente de servicios de directorio: *Directory Service Agent*): Es el componente encargado de toda la gestión relacionada con los servicios de directorio. Este componente está presente en todos los controladores de dominio. Internamente contiene un "nivel de base de datos" que se encarga de proporcionar una visión basada en objetos de toda la información contenida en el directorio.
- *ESE* (*Extensible Storage Engine*): Proporciona la interfaz de acceso a la información mantenida en los ficheros del directorio. Es el componente responsable de indexar la información de directorio y de proporcionar las operaciones necesarias para leer y escribir tal información. También se encarga de proporcionar un soporte transaccional para tales accesos.

12.3.2 Objetos del directorio

Como ya se ha comentado arriba, el directorio de estos sistemas es capaz de manejar múltiples tipos de objeto: usuarios, equipos, grupos, contenedores y unidades organizativas. Veamos qué información se almacena para cada uno de estos objetos y cómo se gestiona ésta.

Objeto usuario

Los objetos *usuario* representan a los usuarios de la red empresarial. Proporcionan a cada usuario una identidad única en el bosque al que pertenezca el dominio. Para ello se construye un identificador cuya primera componente es el nombre de dominio y cuya segunda componente es el "login" del usuario, separados por una contrabarra ("\"). Así, por ejemplo, el identificador de un usuario `juan` en el dominio `eovic` sería "`eovic\juan`".

Objeto equipo

Los objetos *equipo* representan a los equipos (ordenadores) de la red empresarial. También proporcionan una identidad única a cada equipo en el bosque. Para ello les basta con el nombre DNS de cada equipo.

Objeto grupo

Un *grupo* es una colección formada habitualmente por usuarios. No obstante, también se contempla la formación de grupos de equipos e incluso de grupos mixtos con usuarios y equipos.

Un grupo puede formar parte de otro grupo, lo que equivale a incluir los usuarios del primer grupo en el segundo. Un usuario puede pertenecer a tantos grupos como se considere conveniente.

A la hora de crear un grupo se debe especificar cuál va a ser su nombre y también de qué tipo y ámbito será.

Existen dos tipos de grupos:

- *Seguridad*: Sirven para autorizar de manera conjunta a varios usuarios la realización de una acción determinada.

  En cualquier sistema operativo actual se asocian *listas de control de acceso* [SS75] a cada uno de los recursos que deba ser protegido. En tales listas debe especificarse a qué usuarios se les permitirá realizar qué operaciones. Si no se gestionaran grupos de seguridad, estas listas podrían ser extremadamente largas en sistemas con muchos usuarios y recursos que admitieran el acceso por parte de un alto porcentaje de ellos. Para remediar esta situación se manejan grupos y con ellos es posible reducir el tamaño de tales listas: en lugar de incluir a todos los usuarios autorizados, basta con incluir qué grupos tienen permiso.

- *Distribución*: Los grupos de distribución no pueden ser utilizados como entidades a la hora de gestionar la protección del sistema. Para ello ya se utilizan

los grupos de seguridad. Los grupos de distribución suelen utilizarse para definir listas de correo electrónico, de manera que cuando se envíe un correo electrónico al grupo, sea recibido por todos sus usuarios. Para que esto sea posible, se debe haber instalado *Microsoft Exchange Server* en los DC del dominio en que se defina el grupo [RKMW08].

A su vez, existen tres ámbitos en los que definir un grupo:

- *Dominio local*: Resulta visible únicamente en su propio dominio. Es decir, sólo puede ser utilizado en dicho dominio. Sin embargo, puede contener usuarios de cualesquiera de los dominios del bosque.
- *Global*: Resulta visible en todo el bosque, pero sólo puede contener usuarios del dominio en el que haya sido definido.
- *Universal*: Resulta visible en todo el bosque y puede contener usuarios de cualesquiera de los dominios del bosque.

La decisión sobre cuál debe ser el ámbito a aplicar a un nuevo grupo no resulta sencilla. A la hora de tomarla deben seguirse las siguientes pautas:

- A la hora de asignar permisos en ficheros y carpetas, sólo deberían utilizarse grupos de dominio local.
- Los grupos globales deben utilizarse para agrupar usuarios que desempeñen el mismo rol en su propio dominio.
- Los grupos universales deben utilizarse para agrupar usuarios que desempeñen el mismo rol en todo el bosque.
- Se podrán incluir grupos globales y universales en los grupos de dominio local con el fin de asignar permisos a roles.

Contenedores y unidades organizativas

El *Active Directory* se estructura como una jerarquía de contenedores. Un *contenedor* es un objeto que puede mantener dentro de él a otros objetos. Las *unidades organizativas* son un tipo particular de contenedor. Es el único tipo de contenedor que puede crearse desde la herramienta "Usuarios y equipos de Active Directory". El panel izquierdo de esta aplicación nos muestra gráficamente la jerarquía de contenedores existente en un determinado dominio. Para ello, la entrada con el nombre de dominio mantendría al contenedor raíz. Dicha entrada se puede desplegar, mostrando una lista de al menos seis carpetas más, siendo cada una de ellas un contenedor distinto. Entre ellas cabe resaltar las siguientes:

- *Computers*: Contiene todos los equipos del dominio que no sean DC.
- *Domain Controllers*: Contiene todos los equipos del dominio que sean controladores.
- *Users*: Contiene los usuarios y grupos del dominio.

La decisión sobre qué unidades organizativas conviene crear en un determinado dominio no es sencilla. A la hora de definir nuevas unidades organizativas debe considerarse que:

- Un objeto del directorio (usuario, grupo, etc.) sólo puede pertenecer a una unidad organizativa.
- Las unidades organizativas son la base utilizada para delegar tareas administrativas.
- Los administradores pueden vincular objetos de política de grupo ("*Group Policy Objects*", GPO) a estas unidades.

Aunque se definan múltiples unidades organizativas estructurándolas de manera jerárquica y en ellas incluyamos equipos, esto no tendrá ningún efecto sobre el esquema de nombrado DNS. Tales ordenadores seguirán manteniendo su nombre DNS y sus nombres no se verán afectados por el lugar en el que se ubique cada ordenador en la jerarquía de unidades organizativas que hayamos definido.

Para concluir esta sección debe recordarse que las unidades organizativas están destinadas a dotar al directorio de cierta estructura jerárquica. Las tareas administrativas asociadas a una unidad organizativa pueden especificarse mediante GPO. Por su parte, los objetos grupo solo tienen una funcionalidad relacionada con la protección y el control de acceso. También resulta posible generar una jerarquía de grupos (incluyendo a algunos de ellos como miembros de otros), pero su objetivo es distinto al de las unidades organizativas, pues los grupos únicamente intervienen en las tareas de autorización relacionadas con el acceso a recursos. Es decir, como un mecanismo que permite reducir el tamaño de las listas de control de acceso y que únicamente importa a la hora de comprobar si cierto acceso se permite o no.

12.4 Gestión de permisos

Los sistemas operativos actuales utilizan algún mecanismo de protección para controlar el acceso sobre los recursos existentes. Uno de los mecanismos más extendidos es el uso de *listas de control de acceso* (o ACL: *Access Control List*). Esas listas están asociadas a los recursos que deben proteger. En cada lista se especifica qué permisos se ha concedido a cada usuario o grupo existente en ese sistema. Los

sistemas Windows utilizan este mecanismo y *Active Directory* permite su extensión a un entorno distribuido. Para conocer su funcionamiento las próximas secciones describirán los permisos que podrán manejarse en una ACL, la estructura de estas listas y el algoritmo utilizado para decidir si una petición de acceso va a admitirse o no, en función de lo que especifique la lista correspondiente.

Windows Server 2008 distingue entre los permisos estándar de carpetas y los de archivos. Estos permisos estándar son combinaciones predefinidas de permisos individuales (o especiales), que son los que controlan cada una de las acciones que se pueden realizar sobre carpetas y archivos. La existencia de estas combinaciones predefinidas es el resultado de una agrupación "lógica" de los permisos individuales para facilitar la labor del administrador (y de cada usuario cuando administra los permisos de sus archivos). Por este motivo, los permisos estándar se conocen también como "plantillas de permisos". En esta asignatura se llega a nivel de permisos estándar.

12.4.1 Permisos para archivos

Un *permiso* es la especificación que se da en una lista de control de acceso sobre la posibilidad de que un determinado usuario o grupo de usuarios realice una determinada operación sobre un recurso. En algunos sistemas se asume que un usuario o grupo no podrá realizar cierta operación a menos que en la lista de control de acceso se le conceda explícitamente el permiso correspondiente. En otros sistemas, las especificaciones que aparecen en la lista de control de acceso pueden ser de dos tipos: concesión de un permiso o denegación de éste. El comportamiento por omisión en este segundo tipo de sistema dependerá del algoritmo utilizado para gestionar las ACL. Para el caso particular de *Active Directory*, esta gestión se describirá en la sección 12.4.4.

Existen cinco clases de permiso aplicables a un fichero o archivo, dependiendo de las operaciones que autoricen. Son las siguientes:

- *Lectura*: Permite leer el contenido del fichero.
- *Escritura*: Se puede cambiar el contenido del fichero.
- *Lectura y ejecución*: Proporciona el permiso de "*Lectura*", al que añade la posibilidad de ejecutar el fichero en caso de que éste mantenga un programa ejecutable.
- *Modificar*: Añade el permiso de eliminación (borrado) del fichero sobre los permisos de "*Lectura y ejecución*" y "*Escritura*".
- *Control total*: Toma como base el permiso "*Modificar*" sobre el que añade la toma de posesión (es decir, el usuario o grupo que utilice este permiso

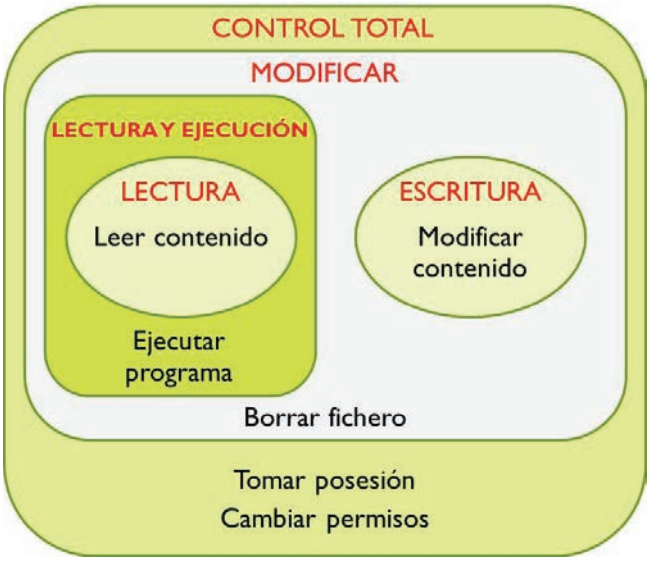

Figura 12.4: Relaciones entre los diferentes tipos de permisos (para archivos).

se convierte en el propietario del fichero) y la posibilidad de cambiar los permisos asociados a dicho fichero (modificando su ACL).

La figura 12.4 muestra gráficamente las relaciones entre estos permisos. En ella se presenta en texto de color negro el conjunto de operaciones admitidas en cada uno de los permisos y en texto de color rojo el nombre de cada uno de ellos.

12.4.2 Permisos para carpetas

El conjunto de permisos disponibles para gestionar el acceso a las carpetas modifica y extiende ligeramente el que se ha presentado para los archivos. En el caso de las carpetas se utilizarán los siguientes:

- *Mostrar*: Permite ver la lista de elementos contenidos en la carpeta, así como en las subcarpetas contenidas en ésta. Sin embargo, no implica poder acceder al contenido de tales elementos. Para ello se necesitaría alguno de los permisos que se comentan seguidamente.
- *Lectura*: Concede el permiso "*Mostrar*" al que añade la posibilidad de leer el contenido de los archivos contenidos en la carpeta y sus subcarpetas.
- *Lectura y ejecución*: Concede el permiso "*Lectura*" al que añade la posibilidad de ejecutar aquellos archivos que contengan un programa ejecutable (tanto en la carpeta como en sus subcarpetas).

- *Escritura*: Permite crear archivos y subcarpetas en la carpeta (y sus subcarpetas). A esto se añade la posibilidad de cambiar el contenido de cualquier archivo.
- *Modificar*: Combina los permisos "*Lectura y ejecución*" y "*Escritura*" a los que añade el permiso de borrado sobre la propia carpeta, sus subcarpetas y sus archivos.
- *Control total*: Toma como base el permiso "*Modificar*" sobre el que añade la toma de posesión (de la carpeta, subcarpetas y archivos) y la posibilidad de cambiar los permisos (de la carpeta, subcarpetas y archivos) y eliminar subcarpetas y archivos (aun sin tener permiso para ello).

La figura 12.5 muestra gráficamente las relaciones existentes entre estos tipos de permisos aplicables a carpetas.

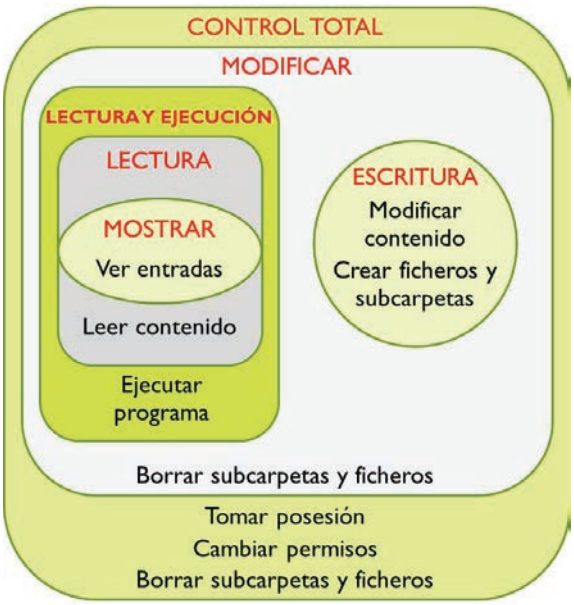

Figura 12.5: Relaciones entre los diferentes tipos de permisos (para carpetas).

12.4.3 Listas de control de acceso

Como ya se ha comentado previamente, los sistemas operativos necesitan algún mecanismo de protección para controlar qué tipos de operaciones podrá realizar cada usuario sobre cada recurso del sistema. La forma habitual de implantar tales mecanismos está basada en el uso de listas de control de acceso. El sistema asocia una lista de este tipo a cada recurso sobre el que se deba controlar los accesos que puedan realizar los distintos usuarios. En dicha lista se guardará una determinada colección de entradas, permitiendo o denegando el uso de una operación o conjunto de operaciones. Existirá cierto algoritmo que especificará cómo tiene que consultarse la lista de entradas y qué efecto tendrá cada una de ellas.

En las secciones 12.4.1 y 12.4.2 ya se ha indicado qué operaciones están incluidas cuando se especifica cada tipo de permiso (tanto para el caso de archivos como de carpetas, pues estos son los dos tipos de recursos gestionados por el sistema de protección en Windows). Habrá que ver ahora qué otra información aparece en cada *entrada de control de acceso* (o *ACE*: "*Access Control Entry*"). Cada entrada mantiene tres campos:

- El primero especifica si el permiso utilizado en tal entrada se está concediendo ("*Permitir*") o no ("*Denegar*").
- El segundo indica a qué usuario o grupo corresponde esta entrada.
- El tercero indica qué permiso se está concediendo o denegando.

Debe considerarse que las ACL tienen sus entradas ordenadas. Eso es importante a la hora de comprobar si un determinado acceso se va a permitir o no. Como ejemplo, veamos una posible ACL para un archivo `Prueba.txt`, en la que encontraríamos las siguientes cuatro entradas:

1. {Permitir, Laura, Modificar}.
2. {Permitir, Juan, Lectura}.
3. {Denegar, Ejecutivos, Escritura}.
4. {Permitir, Directores, Modificar}.

En este ejemplo, la usuaria `Laura` tiene permiso para leer y modificar su contenido, así como borrar el fichero. Por su parte, el usuario `Juan` puede leer el contenido del fichero, mientras que los usuarios del grupo `Ejecutivos` no podrán modificar su contenido. Por último, los usuarios del grupo `Directores` podrán leer y modificar su contenido, así como borrar el fichero.

Una ACL tiene dos tipos de entradas: las heredadas y las explícitas. Una *ACE explícita* es aquella que se ha declarado explícitamente para una carpeta o archivo

determinado. Una *ACE heredada* será cualquiera de las ACE explícitas de alguna de las carpetas antecesoras.

Si asumimos que el fichero `Prueba.txt` mencionado previamente se encontraba en la carpeta `C:\TEMP`, las entradas de su ACL podrían haberse declarado de la forma presentada en la tabla 12.1.

`Carpeta C:`	
{Permitir, Directores, Modificar}	Explícita
`Carpeta C:\TEMP`	
{Denegar, Ejecutivos, Escritura}	Explícita
{Permitir, Directores, Modificar}	Heredada de `C:`
`Archivo C:\TEMP\PRUEBA.TXT`	
{Permitir, Laura, Modificar}	Explícita
{Permitir, Juan, Lectura}	Explícita
{Denegar, Ejecutivos, Escritura}	Heredada de `C:\TEMP`
{Permitir, Directores, Modificar}	Heredada de `C:`

Tabla 12.1: Ejemplo de ACL con entradas explícitas y heredadas.

Una carpeta puede desactivar la herencia. En ese caso su ACL no contendrá ninguna de las entradas presentes en las ACL de sus carpetas antecesoras. Así, si se hubiese desactivado la herencia en la carpeta `C:\TEMP`, el ejemplo presentado en la tabla 12.1 ofrecería el resultado que muestra la tabla 12.2.

`Carpeta C:`	
{Permitir, Directores, Modificar}	Explícita
`Carpeta C:\TEMP` (Herencia desactivada)	
{Denegar, Ejecutivos, Escritura}	Explícita
`Archivo C:\TEMP\PRUEBA.TXT`	
{Permitir, Laura, Modificar}	Explícita
{Permitir, Juan, Lectura}	Explícita
{Denegar, Ejecutivos, Escritura}	Heredada de `C:\TEMP`

Tabla 12.2: Ejemplo de ACL con desactivación de herencia.

Para obtener información sobre la ACL de un determinado archivo o carpeta puede seleccionarse tal objeto desde el explorador de Windows y pulsar el botón derecho del ratón una vez lo tengamos seleccionado. Aparecerá un menú contextual en el que habrá que seleccionar la opción "*Propiedades*". En la ventana que aparecerá a continuación se seleccionará la ficha "*Seguridad*". En la información mostrada en esta ficha, se podrá seleccionar un grupo o usuario dentro de la lista presentada como "*Nombres de grupos o usuarios*" y con ello se mostrarán sus permisos en la

lista de la mitad inferior de la ventana. Si queremos modificar tales permisos se tendrá que pulsar el botón "*Editar...*".

En caso de que el objeto sea una carpeta, podremos pulsar el botón "*Opciones avanzadas*" y en la ventana que se mostrará a continuación se indicará si los permisos correspondientes son heredados o explícitos. En tal ventana también encontraremos una casilla seleccionable "*Incluir todos los permisos heredables del objeto primario de este objeto*", que en caso de estar inactiva inhabilitará la herencia de ACEs. El "*objeto primario*" al que se refiere en dicho texto es la carpeta antecesora.

12.4.4 Uso de las ACL

Las ACL resultan necesarias para la toma de decisiones a la hora de autorizar las peticiones de acceso realizadas por los diferentes procesos de un sistema Windows. Este mecanismo de autorización se inicia cuando un usuario solicita una acción respecto a un archivo o carpeta. Ejemplos: editar un archivo, listar el contenido de una carpeta, crear una subcarpeta, borrar un archivo, etc.

La secuencia completa de pasos llevada a cabo por el sistema para autorizar o denegar la acción solicitada es la siguiente:

1. Determina el conjunto de permisos necesario para llevar a cabo la acción.

   Por ejemplo, si el proceso hubiera solicitado editar un fichero de texto, se necesitaría tanto leer como modificar el contenido de ese fichero. Esto requiere el permiso "*Lectura*" y el permiso "*Escritura*" o cualquier otro permiso de mayor nivel que incluya a ambos (véase la figura 12.4).

2. Crea un conjunto de identidades formado por el usuario (es decir, el usuario que haya lanzado el proceso que ahora solicita acceder al fichero o carpeta) y los grupos a los que éste pertenece.

   Por ejemplo, si el usuario es `David` y pertenece a los grupos `Directores` y `Todos`, las identidades que incluiríamos en dicho conjunto serían esas tres: `David`, `Directores` y `Todos`.

3. Se consulta la ACL en el orden siguiente:
   a) ACEs explícitas que denieguen.
   b) ACEs explícitas que permitan.
   c) ACEs heredadas de la carpeta antecesora inmediata (es decir, aquella en la que se ubique el recurso que se ha solicitado acceder) que denieguen.
   d) ACEs heredadas de la carpeta antecesora inmediata (es decir, aquella en la que se ubique el recurso que se ha solicitado acceder) que permitan.
   e) ...

f) ACEs heredadas de la carpeta antecesora más lejana que denieguen.

g) ACEs heredadas de la carpeta antecesora más lejana que permitan.

Se analizarán en esta consulta únicamente aquellas entradas que hagan referencia a alguna de las identidades del conjunto generado en el paso 2.

Para cada entrada analizada:

a) Si la entrada deniega **alguno** de los permisos de la solicitud: **se deniega el acceso y se deja de recorrer la ACL**.

b) Si la entrada concede alguno de los permisos de la solicitud, se acumula a otros permisos previamente acumulados. Si **todos** los permisos solicitados forman parte del conjunto de permisos acumulados **se concede el acceso y se deja de recorrer la ACL**.

Si finalmente se hubiese recorrido toda la lista sin haber llegado a tomar ninguna decisión, **se deniega el acceso**.

Para ilustrar cómo se consulta una ACL, utilizaremos como ejemplo la lista mostrada en la tabla 12.3.

`Archivo C:\CARP1\CARP2\DOCUMENT.TXT`			
Tipo	**Usuario/Grupo**	**Permiso**	**Heredado de...**
Permitir	Laura	Modificar	Explícito
Denegar	Ismael	Lectura	`C:\CARP1\CARP2`
Permitir	Project3	Escritura	`C:\CARP1\CARP2`
Permitir	Project1	Lectura y Ejecución	`C:\CARP1`
Permitir	Project2	Lectura y Ejecución	`C:\CARP1`
Permitir	Administradores	Control total	`C:`

Tabla 12.3: Ejemplo de ACL.

En cierto instante, el usuario `Alberto` perteneciente a los grupos `Project2` y `Project3` inicia un proceso para editar el fichero `C:\CARP1\CARP2\DOCUMENT.TXT`. El sistema pasaría a utilizar el algoritmo que se acaba de enunciar, con las siguientes consecuencias:

1. La edición requiere los permisos "*Lectura*" y "*Escritura*" sobre dicho fichero.
2. El conjunto de identidades a utilizar será: `Alberto`, `Project2` y `Project3`.
3. La única ACE explícita existente no es aplicable a ese conjunto de identidades, por lo que se pasa a buscar entre las heredadas.

   La ACE denegatoria que se ha heredado de `C:\CARP1\CARP2` tampoco afecta a este usuario. Sin embargo, la otra ACE heredada de esa misma carpeta

sí que es aplicable pues el grupo `Project3` está dentro del conjunto de identidades generado en el paso 2 de este algoritmo. Por tanto, el permiso "*Escritura*" está concedido y pasamos a guardarlo en el conjunto de permisos otorgados hasta el momento. Todavía no están todos los permisos necesarios: falta "*Lectura*" o bien otro que lo englobe.

Pasamos a buscar ahora entre las ACE heredadas de `C:\CARP1`. La primera afecta a `Project1` y no es aplicable a esta petición. La segunda afecta a `Project2` (uno de los grupos de `Alberto`). Por tanto, ese permiso sí que se puede acumular. Como "*Lectura y Ejecución*" implica el permiso de "*Lectura*" que todavía quedaba pendiente, esto finaliza el análisis de la lista, proporcionando como resultado que **el acceso se autoriza**.

Supongamos que poco después el usuario `Ismael` perteneciente a esos dos mismos grupos (`Project2` y `Project3`) intenta también editar ese mismo fichero. En este segundo caso, el algoritmo haría lo siguiente:

1. La edición requiere los permisos "*Lectura*" y "*Escritura*" sobre dicho fichero.
2. El conjunto de identidades a utilizar será: `Ismael`, `Project2` y `Project3`.
3. La única ACE explícita existente no es aplicable a ese conjunto de identidades, por lo que se pasa a buscar entre las heredadas.

   La ACE denegatoria que se ha heredado de `C:\CARP1\CARP2` afecta directamente a `Ismael` denegándole el permiso de lectura. Como ese permiso resultaba necesario para realizar la acción solicitada, el sistema **deniega la petición** y el análisis de la lista termina aquí.

Poco después `Laura`, que pertenece al grupo `Project2` también solicita editar el fichero. En este caso:

1. La edición requiere los permisos "*Lectura*" y "*Escritura*" sobre dicho fichero.
2. El conjunto de identidades a utilizar será: `Laura`, `Project2`.
3. La única ACE explícita existente sí que es aplicable a ese conjunto de identidades e indica que se concede a esta usuaria el permiso "*Modificar*". Según se ha comentado en las secciones anteriores, ese permiso implica tanto "*Lectura*" como "*Escritura*" por lo que ambos entrarían a formar parte del conjunto de permisos acumulados y con ello ya se podría aceptar la petición. La exploración de la lista ya no debe continuar y **la petición se autoriza**.

Finalmente, el usuario `Juan` de los grupos `Project1` y `Administradores` solicita borrar ese fichero. Veamos qué ocurre:

1. El borrado requiere al menos el permiso "*Modificar*" sobre dicho fichero. Recuérdese que ese permiso también forma parte de "*Control total*" con lo que cualquiera de esos dos será suficiente para autorizar la acción solicitada.

2. El conjunto de identidades a utilizar es: `Juan`, `Project1` y `Administradores`.

3. La única ACE explícita existente no es aplicable a ese conjunto de identidades, por lo que se pasa a buscar entre las heredadas.

   La ACE denegatoria que se ha heredado de `C:\CARP1\CARP2` no afecta a ese conjunto de identidades, por lo que no es tenida en cuenta. La otra ACE heredada desde esa misma carpeta tampoco afecta.

   Pasamos a buscar ahora entre las ACE heredadas de `C:\CARP1`. La primera afecta a `Project1` por lo que sí que es aplicable a esta petición. Sin embargo, no incluye ningún permiso relacionado con la operación de borrado, por lo que tampoco afecta a la gestión de la petición realizada. La segunda ACE es aplicable a `Project2` y no tiene ninguna influencia en la petición analizada. De momento, la lista de permisos acumulados sigue vacía.

   Finalmente llegamos a las ACE heredadas de `C:`. Sólo hay una ACE de este tipo. En ella se nos dice que el grupo `Administradores` tendrá "*Control total*" sobre este objeto. Es aplicable, pues ese grupo está dentro del conjunto de identidades generado en el paso 2. El permiso concedido admite la acción solicitada. Por tanto, el análisis termina aquí y **la petición se autoriza**.

Modifiquemos este último ejemplo y asumamos que `Juan` sólo pertenece al grupo `Project1` (y no a `Administradores`). En ese caso, la exploración de la lista concluye sin encontrar ninguna entrada que autorice o deniegue la acción solicitada. Con ello, la petición sería denegada.

12.4.5 Diseño de ACL

El diseño de listas de control de acceso adecuadas no es una tarea sencilla. Para llevarlo a cabo conviene respetar las siguientes recomendaciones:

- Si los permisos que se heredarían de una carpeta determinada no son los adecuados, conviene desactivar la herencia en lugar de usar la denegación de permisos.
- Hay que utilizar únicamente *Grupos de Dominio Local* en las ACL.
- Hay que definir un grupo distinto por cada permiso distinto que se pretenda conceder.
- Usar un convenio de nombrado que facilite la identificación del propósito de cada *Grupo de Dominio Local*. Un esquema recomendable sería:

ACL_*nombre-del-recurso_permiso*

Con este criterio, la lista de control de acceso para una carpeta `BASEDIR` tendría el siguiente contenido. Así, el administrador iría incorporando posteriormente a cada grupo los usuarios que necesitaran tales permisos:

Archivo C:\BASEDIR		
Tipo	**Usuario/Grupo**	**Permiso**
Permitir	ACL_basedir_escritura	Escritura
Permitir	ACL_basedir_lectura	Lectura
Permitir	ACL_basedir_modificar	Modificar
Permitir	ACL_basedir_controlTotal	Control total

12.5 Recursos compartidos

Un *recurso compartido* permite acceder de manera remota a una carpeta ubicada en otro ordenador del sistema. La figura 12.6 muestra un ejemplo que se describe seguidamente.

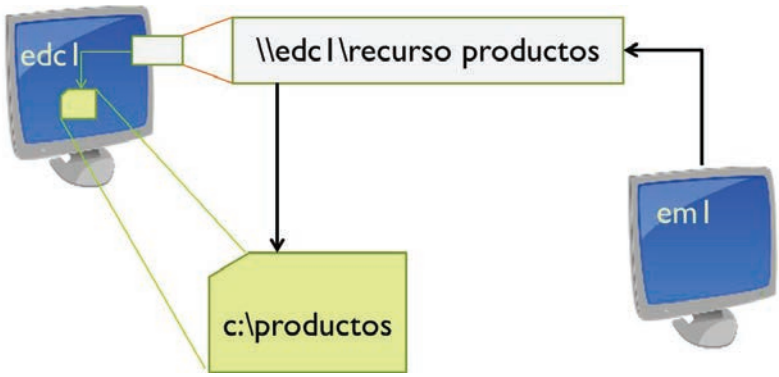

Figura 12.6: Recurso compartido.

A partir de la carpeta "`C:\productos`" del ordenador `edc1` se ha creado un recurso compartido vinculado a ella. El nombre asignado a tal recurso es "`recurso productos`". Como puede observarse, no tiene por qué ser igual al de la carpeta que va a compartirse e irá precedido por una doble contrabarra inicial (esto es: "\\") seguida por el nombre del ordenador en el que se encuentre la carpeta ("`edc1`" en este caso) y una última contrabarra que separa ambos componentes.

En la figura se muestra cómo un usuario hace referencia desde el ordenador `em1` a la carpeta compartida utilizando el nombre apropiado: "`\\edc1\recurso productos`". Con ello el sistema sabe que debe contactar con el servidor "`edc1`" y realizar el acceso sobre la carpeta asociada a tal nombre.

12.5.1 Creación del recurso compartido

Para crear el recurso compartido mostrado en la figura 12.6 se habría seguido este procedimiento:

1. En el ordenador donde resida la carpeta que desee compartirse, se utilizará el explorador de Windows para acceder al lugar en que ésta esté ubicada.
2. Una vez allí, se seleccionará la carpeta a compartir y se pulsará el botón derecho del ratón para desplegar el menú contextual. Se seleccionará la opción "*Propiedades*" en tal menú.
3. Seleccionamos la ficha "*Compartir*". Una vez en ella, se pulsa el botón "*Uso compartido avanzado...*".
4. En la ventana emergente, activaremos la casilla "*Compartir esta carpeta*", dándole un nombre en el campo "*Nombre del recurso compartido:*". Como ya se ha dicho previamente, este nombre no tiene por qué coincidir con el que tenga la carpeta. Puede ser cualquiera.
5. Una vez hecho esto, se pulsa el botón "*Aceptar*" y esto completa la creación del recurso.

Para comprobar que la creación ha funcionado correctamente, se puede utilizar la aplicación "*Administrador de almacenamiento y recursos compartidos*" disponible entre las "*Herramientas administrativas*" (accesible desde el menú de Inicio) del ordenador donde hayamos creado el recurso compartido. En su ficha "*Recursos compartidos*" aparecerá, entre otras, la carpeta compartida que acabamos de crear.

Otra forma de comprobarlo consiste en introducir el nombre completo del recurso ("`\\edc1\recurso productos`" en este ejemplo) en el explorador de Windows de cualquier otro ordenador de ese mismo dominio. Si todo funcionara correctamente, no se mostraría ningún mensaje de error y la ventana del explorador presentaría el contenido de la carpeta compartida.

12.5.2 Asignación de un identificador de unidad

Una vez ya se ha definido el recurso compartido, desde otros ordenadores se podrá acceder a él, como se ha descrito al final de la sección anterior. Para facilitar el uso de tales recursos, es posible asignar a tales carpetas compartidas un identificador de unidad (es decir, un nombre de unidad en Windows, tal como "`Z:`").

Para realizar tal asignación se tendrá que utilizar el explorador de Windows. El procedimiento concreto a utilizar en cada versión de los sistemas operativos Windows podrá variar ligeramente, pero seguirá una secuencia similar a ésta (que es la utilizada en Windows 7):

1. Iniciar el explorador de Windows. Por ejemplo, mediante un doble *click* sobre el icono "*Equipo*" del escritorio, o bien seleccionando la opción "*Equipo*" desde el menú desplegado al pulsar el botón "*Inicio*" del escritorio.

2. Pulsar el botón "*Conectar a unidad de red*" que se muestra en la barra superior de la ventana principal del explorador.

3. En el campo de entrada "*Carpeta:*", habrá que introducir el nombre del recurso compartido. Siguiendo con el ejemplo dado anteriormente, la cadena a introducir habría sido: "\\`edc1`\`recurso productos`".

   En el primer campo de entrada aparece ya un nombre válido de unidad (y todavía libre) para asignar a ese recurso compartido. Puede seleccionarse otro si se considera necesario.

   Por omisión, la casilla "*Conectar de nuevo al iniciar sesión*" aparece activa. Con ello la asignación del nombre de unidad a ese recurso compartido se mantendrá en futuras sesiones y resultará más sencillo acceder a él.

4. Pulsar el botón "*Finalizar*".

12.5.3 Autenticación y autorización

Tanto el ordenador que "exporta" el recurso compartido como el utilizado por el usuario para acceder a él deben colaborar de alguna manera para autenticar a ese usuario y comprobar que las operaciones que solicite estén autorizadas. Para ello se procede de la siguiente manera. Asumamos que el ordenador A exporta un recurso compartido y que un usuario B intenta acceder a él desde un ordenador C. Cuando B solicite alguna operación sobre ese recurso compartido, las credenciales de B se facilitan al ordenador A para que lo autentique. Si tal operación se realiza sobre un dominio, A tendrá que contactar a su vez con el DC de dicho dominio para realizar la autenticación. En caso contrario, sería una operación de autenticación local en A.

Una vez autenticado el usuario, el ordenador A comprobará si B está autorizado para realizar la operación que solicita. Para ello se tendrá que analizar por una parte la ACL del recurso compartido y por otra la de la carpeta o fichero que se utilizó para definir tal recurso compartido. En el ejemplo que hemos utilizado anteriormente (véase la figura 12.6) el recurso compartido se llamó "`recurso productos`" mientras que la carpeta utilizada para definirlo fue "`productos`". Cada uno de esos dos elementos tendrá su propia ACL. Por omisión, estos permisos habrían sido "*Modificar*" para el grupo "`Todos`" en la carpeta "`productos`" y "*Lectura*" para el grupo "`Todos`" en el recurso compartido "`recurso productos`". De ser así, cualquier usuario remoto (entre ellos B) únicamente podrá leer el contenido de esa carpeta o mostrar un listado de ésta, pero no estará autorizado para realizar otras operaciones.

Si queremos modificar la ACL del recurso compartido habrá que seguir estos pasos:

1. Abrir el explorador de Windows y acceder a la carpeta donde se encuentre el fichero o carpeta que generó el recurso compartido.

2. Una vez allí, se seleccionará ese fichero o carpeta y se pulsará el botón derecho del ratón para desplegar el menú contextual. Se seleccionará la opción "*Propiedades*" en tal menú.

3. Seleccionamos la ficha "*Compartir*". Una vez en ella, se pulsa el botón "*Uso compartido avanzado...*".

4. En la ventana emergente, bastará con pulsar el botón "*Permisos*" y aparecerá una segunda ventana emergente donde será posible visualizar cada una de las ACEs de la ACL, seleccionando para ello a cada uno de los usuarios o grupos que aparezcan en la mitad superior de esa ventana.

   En esta ventana es posible modificar los permisos asignados en cada ACE.

5. Una vez hecho esto, se pulsa el botón "*Aceptar*" y esto completaría la consulta (y modificación, si la hubiere) de la ACL.

Ya que por omisión sólo se concede el permiso "*Lectura*" para el grupo "`Todos`" sobre los recursos compartidos y éste puede ser excesivamente restrictivo, se considera una buena recomendación dentro de un dominio el modificar ese permiso y transformarlo en "*Control total*" para el grupo "`Todos`". De esta manera, el recurso compartido no impone ninguna restricción. En ese caso, los permisos que realmente restringirán el acceso serán aquellos que tenía la carpeta o archivo tomados como base para definir el recurso compartido. Tales permisos no deben modificarse y serán los que controlarán de manera efectiva qué operaciones podrán realizar los usuarios sobre dicho elemento.

12.6 Resumen

El Directorio Activo de Windows (*Active Directory*) es la solución que proporciona Microsoft para organizar y administrar una red de ordenadores empresarial. Dicho directorio activo almacena información acerca de los recursos de la red y permite el acceso de los usuarios y las aplicaciones a dichos recursos, convirtiéndose en un medio de organizar, controlar y administrar centralizadamente el acceso a los recursos de la red. Al instalar el Directorio Activo en uno o varios sistemas Windows Server de nuestra red, convertimos a dichos ordenadores en los Controladores de Dominio, mientras que el resto de los equipos de la red actúan de clientes de estos servicios de directorio, recibiendo de ellos la información de cuentas de usuario, grupo, equipo, etc., así como los perfiles de usuario y equipo, directivas de

seguridad, servicios de red, etc. Por tanto, el Directorio Activo es la herramienta fundamental de administración de toda la organización.

En esta unidad se han descrito los componentes y servicios integrados en *Active Directory* que permiten gestionar los dominios de un sistema distribuido, así como la gestión de los permisos para el control del acceso sobre los recursos existentes. Un *dominio* permite agrupar a un conjunto de ordenadores y usuarios, que serán autenticados e identificados por los *servicios del dominio*. Estos servicios se instalan en controladores de dominio, que se responsabilizan de la autenticación de los usuarios del dominio, y que podrán ser replicados para asegurar su disponibilidad.

Los dominios se pueden organizar jerárquicamente, estableciéndose un *árbol de dominios*. Los árboles resultan apropiados para estructurar los dominios de una empresa con organización jerárquica. En otros casos, se emplean *bosques de dominios*, en los que se gestionan múltiples árboles de dominio que no guardan ninguna relación jerárquica entre ellos.

La gestión de los elementos del directorio (tales como usuarios, ordenadores, grupos de usuarios, contenedores, unidades organizativas), se lleva a cabo a través de los *servicios de directorio*. Cada dominio tiene su propio directorio, que reside en los controladores de dominio. Los objetos *usuario* representan a los usuarios de la red empresarial; los objetos *equipo* representan a los equipos de dicha red; un *grupo* es una colección de usuarios, de equipos o ambos, y pueden ser definidos en ámbito de dominio local, global o bien en ámbito universal; un *contenedor* es un objeto que puede mantener dentro de él a otros objetos; mientras que las *unidades organizativas* son un tipo particular de contenedor y dotan al directorio de cierta estructura jerárquica.

Respecto a la gestión de permisos, en esta unidad se ha explicado uno de los mecanismos de protección más extendidos para controlar el acceso sobre los recursos existentes: las *listas de control de acceso*. En estas listas se especifica qué permisos se concede a cada usuario o grupo existente en el sistema. Existen cinco clases de permisos aplicables a archivos o carpetas, dependiendo de las operaciones que autoricen. Las listas de control de acceso guardan una determinada colección de entradas de control de acceso (heredadas o explícitas), que permiten o deniegan el uso de una operación o conjunto de operaciones. En esta unidad se ha detallado el protocolo que se sigue para consultar dichas listas y determinar si se permite o no el acceso a un archivo o carpeta solicitado por un usuario. También se ha revisado cómo se comprueban los permisos si se trata de un recurso compartido.

Resultados de aprendizaje. Al finalizar esta unidad, el lector deberá ser capaz de:

- Identificar las principales abstracciones relacionadas con los servicios de dominio (dominio, bosque, unidad organizativa, usuario, grupo, etc.) y explicar su utilidad para la administración de sistemas dentro de una organización.
- Identificar el uso de las principales abstracciones dentro de un dominio y relacionar este uso con sus respectivos procedimientos y herramientas administrativas.
- Identificar las principales abstracciones relacionadas con la gestión de permisos (listas de control de acceso, clases de permiso, entradas de control de acceso) y describir la gestión de permisos que Active Directory lleva a cabo.
- Describir la gestión de los recursos compartidos.
- Administrar aspectos concretos de un dominio mediante las herramientas adecuadas para cumplir con un conjunto de requisitos de funcionamiento.

Bibliografía

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade y Vera Watson. System R: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[ABD+95] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken Tindell y Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[Apa12] Apache Software Foundation. About the Apache HTTP server project. Disponible en http://httpd.apache.org/ABOUT_APACHE.html, marzo 2012.

[AS04] Stephanos Androutsellis-Theotokis y Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[Bab04] Arne Babenhauserheide. Gnutella for users. Disponible en http://draketo.de/inhalt/krude-ideen/gnufu-en.pdf, junio 2004.

[Ber96] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):87–99, febrero 1996.

[BFC95] Peter A. Buhr, Michel Fortier y Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, 1995.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider y Sam Toueg. Optimal primary-backup protocols. En *6th Intnl. Wshop. Distrib. Alg. (WDAG)*, volumen 647 de *Lect. Notes Comput. Sci.*, págs. 362–378, Haifa, Israel, noviembre 1992. Springer.

[BN84] Andrew D. Birrel y Bruce J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, febrero 1984.

[Bon00] André B. Bondi. Characteristics of scalability and their impact on performance. En *2nd Intnl. Wshop. on Software and Performance (WOSP)*, págs. 195–203, Ottawa, Canadá, septiembre 2000. ACM Press.

[Bri73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall Inc., 1973.

[Bri75] Per Brinch Hansen. The programming language concurrent Pascal. *IEEE Trans. Software Eng.*, 1(2):199–207, 1975.

[CES71] Edward G. Coffman Jr., M. J. Elphick y Arie Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[CHP71] Pierre-Jacques Courtois, F. Heymans y David Lorge Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, 1971.

[CKV01] Gregory Chockler, Idit Keidar y Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[CL85] K. Mani Chandy y Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[Cri89] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CT96] Tushar Deepak Chandra y Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DBEB06] Martin H. Duke, Robert Braden, Wesley M. Eddy y Ethan Blanton. RFC4614: A roadmap for transmission control protocol (TCP) specification documents. Internet Engineering Task Force Request for Comments, Network Working Group, septiembre 2006.

[Dij64] Edsger Wybe Dijkstra. Over seinpalen (EWD-74), 1964. URL: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF.

[Dij65] Edsger Wybe Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Dij68] Edsger Wybe Dijkstra. Cooperating sequential processes. En F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, págs. 43–112. Academic Press, 1968. Disponible en: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF.

[Dij71] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[DSS98] Xavier Défago, André Schiper y Nicole Sergent. Semi-passive replication. En *17th Symp. Reliab. Distrib. Sys. (SRDS)*, págs. 43–50, West Lafayette, Indiana, EE.UU., octubre 1998. IEEE-CS Press.

[EE98] Guy Eddon y Henry Eddon. *Inside Distributed COM.* Microsoft Press, abril 1998. ISBN 978-1572318496.

[EFGK03] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui y Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[FGM+99] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach y Tim Berners-Lee. RFC2616: Hypertext transfer protocol – HTTP/1.1. Internet Engineering Task Force Request for Comments, Network Working Group, junio 1999.

[Fid88] Colin J. Fidge. Partial orders for parallel debugging. En *Workshop on Parallel and Distributed Debugging*, págs. 183–194, Madison, Wisconsin, EE.UU., mayo 1988. ACM Press.

[FK98] Ian T. Foster y Carl Kesselman. The Globus project: A status report. En *Heterogeneous Computing Workshop*, págs. 4–18. IEEE-CS Press, 1998.

[FLP85] Michael J. Fischer, Nancy A. Lynch y Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FR03] Faith Ellen Fich y Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

[GDJ06] Saikat Guha, Neil Daswani y Ravi Jain. An experimental study of the Skype peer-to-peer VoIP system. En *5th Intnl. Wshop. on Peer-To-Peer Systems (IPTPS)*, Santa Barbara, CA, EE.UU., febrero 2006.

[GM82] Héctor García-Molina. Elections in a distributed computing system. *IEEE Trans. Comp.*, 31(1):48–59, enero 1982.

[GZ89] Riccardo Gusella y Stefano Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Trans. Software Eng.*, 15(7):847–853, 1989.

[Hab69] Arie Nicolaas Habermann. Prevention of system deadlocks. *Commun. ACM*, 12(7):373–377, 1969.

[Hen04] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[Hoa74] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10), octubre 1974.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5.

[Hol72] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[IEE90] IEEE. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries. The Institute of Electrical and Electronics Engineering, 345 East 47th Street, New York, NY 10017, EEUU, 1990. ISBN 1-55937-078-3.

[IEE08] IEEE. IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4694974, diciembre 2008.

[IK82] Toshihide Ibaraki y Tsunehiko Kameda. Deadlock-free systems for a bounded number of processes. *IEEE Trans. Computers*, 31(3):188–193, 1982.

[ISO98] ISO. International standard ISO/IEC 10746-1:1998(E): Information technology - open distributed processing - reference model: Overview. International Standard Organization, Case Postale 56, CH-1211 Genève 20, Suiza, diciembre 1998.

[JP86] Mathai Joseph y Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg y Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[LFA04] Mikel Larrea, Antonio Fernández y Sergio Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Computers*, 53(7):815–828, 2004.

[LR80] Butler W. Lampson y David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[LS88] Barbara Liskov y Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. En *ACM SIGPLAN Conf. Programm. Lang. Design and Impl. (PLDI)*, págs. 260–267, Atlanta, Georgia, EE.UU., junio 1988.

[LT93] Nancy G. Leveson y Clark Savage Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[LYBB11] Tim Lindholm, Frank Yellin, Gilad Bracha y Alex Buckley. *The Java^{TM} Virtual Machine Specification: Java SE 7 Edition*. Oracle America, Inc., 7^{a} edición, julio 2011. Disponible en: http://docs.oracle.com/javase/specs/.

[Mat87] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[McC08] Scott McCloud. Google on Google Chrome - comic book. Disponible en: http://blogoscoped.com/google-chrome/, septiembre 2008.

[Mic10] Microsoft Corp. Detecting and ending deadlocks. MSDN Library, SQL Server 2008 R2 Documentation, Database Engine Book, abril 2010. URL: http://msdn.microsoft.com/en-us/library/ms178104(v=sql.105).

[Min82] Toshimi Minoura. Deadlock avoidance revisited. *J. ACM*, 29(4):1023–1048, 1982.

[Moc83a] Paul V. Mockapetris. RFC882: Domain names - concepts and facilities. Internet Engineering Task Force Request for Comments, Network Working Group, noviembre 1983.

[Moc83b] Paul V. Mockapetris. RFC883: Domain names - implementation and specification. Internet Engineering Task Force Request for Comments, Network Working Group, noviembre 1983.

[Mos93] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[NBBB98] Kathleen Nichols, Steven Blake, Fred Baker y David L. Black. RFC2474: Internet protocol. Internet Engineering Task Force Request for Comments, Network Working Group, diciembre 1998.

[Nel81] Bruce J. Nelson. *Remote Procedure Call*. Tesis doctoral, Carnegie Mellon Univ., 1981.

[Nel90] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, 1990.

[NYHR05] Clifford Neuman, Tom Yu, Sam Hartman y Kenneth Raeburn. RFC4120: The Kerberos network authentication service (v5). Internet Engineering Task Force Request for Comments, Network Working Group, julio 2005.

[Obj11a] Object Management Group. Common Object Request Broker Architecture (CORBA), version 3.2. Disponible en: http://www.omg.org/spec/CORBA/, noviembre 2011.

[Obj11b] Object Management Group. OMG Unified Modeling Language, superstructure specification, version 2.4.1. Disponible en: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/, agosto 2011.

[OL82] Susan S. Owicki y Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[Ora11] Oracle Corp. Oracle database concepts 11g release 2 (11.2), capítulo 9, Data concurrency and consistency. Disponible en: http://docs.oracle.com/cd/E11882_01/server.112/e25789/consist.htm#autoId19, septiembre 2011.

[Ora12a] Oracle Corp. Class `executors` (Java Platform SE 7). Disponible en: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html, febrero 2012.

[Ora12b] Oracle Corp. Java SE documentation at a glance. Disponible en: http://www.oracle.com/technetwork/java/javase/documentation/index.html, abril 2012.

[Ora12c] Oracle Corp. The Java^{TM} tutorials - essential classes - lesson: Concurrency. Disponible en: http://docs.oracle.com/javase/tutorial/essential/concurrency/, febrero 2012. También disponible como libro electrónico en formato ePUB (http://download.oracle.com/otn-pub/java/tutorial/2012-02-28/essentialtrail.epub) y MOBI (http://download.oracle.com/otn-pub/java/tutorial/2012-02-28/essentialtrail.mobi).

[Ora12d] Oracle Corp. The Java^{TM} tutorials: Rmi. Disponible en: http://docs.oracle.com/javase/tutorial/rmi/index.html, noviembre 2012.

[Ora12e] Oracle Corp. Oracle technology network for Java developers. Disponible en: http://www.oracle.com/technetwork/java/index.html, abril 2012.

[Ora12f] Oracle Corp. Remote Method Invocation home. Disponible en: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, noviembre 2012.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, diciembre 1972.

[Plu82] David C. Plummer. RFC826: An Ethernet address resolution protocol. Internet Engineering Task Force Request for Comments, Network Working Group, noviembre 1982.

[Pos80] Jon Postel. RFC768: User datagram protocol. Internet Engineering Task Force Request for Comments, Network Working Group, agosto 1980.

[Pos81a] Jon Postel. RFC791: Internet protocol. Internet Engineering Task Force Request for Comments, Network Working Group, septiembre 1981.

[Pos81b] Jon Postel. RFC793: Transmission control protocol. Internet Engineering Task Force Request for Comments, Network Working Group, septiembre 1981.

[Pos12] The PostgreSQL Global Development Group. PostgreSQL 9.2.0 documentation, capítulo 13: Concurrency control, sección 13.3.3: Deadlocks. Disponible en: http://www.postgresql.org/docs/9.2/static/explicit-locking.html, septiembre 2012.

[Pow93] David Powell. Distributed fault tolerance - lessons learned from Delta-4. En *Hardware and Software Architectures for Fault Tolerance*, volumen 774 de *Lect. Notes Comput. Sc.*, págs. 199–217. Springer, 1993.

[PPR+81] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser y Charles S. Kline. Detection of mutual inconsitency in distributed systems. En *Berkeley Workshop*, págs. 172–184, 1981.

[PPR+83] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser y Charles S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

[PS95] David Plainfossé y Marc Shapiro. A survey of distributed garbage collection techniques. En *Intnl. Wshop. Memory Mgmnt. (IWMM)*, volumen 986 de *Lect. Notes Comput. Sc.*, págs. 211–249, Kinross, Reino Unido, septiembre 1995. Springer.

[RA81] Glenn Ricart y Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, enero 1981.

[RD01] Antony I. T. Rowstron y Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. En *Intnl. Conf. on Distr. Syst. Platf. (Middleware)*, volumen 2218 de *Lect. Notes Comput. Sc.*, págs. 329–350, Heidelberg, Alemania, noviembre 2001. Springer.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp y Scott Shenker. A scalable content-addressable network. En *ACM SIGCOMM Conf. on Appl., Techn., Arch. and Prot. for Comput. Commun. (SIGCOMM)*, págs. 161–172, San Diego, CA, EE.UU., agosto 2001.

[RKMW08] Stan Reiner, Conan Kezema, Mike Mulcare y Byron Wright. *Windows Server 2008 Active Directory Resource Kit*. Microsoft Press, Redmond, Washington, EE.UU., abril 2008.

[RSI09] Mark E. Russinovich, David A. Solomon y Alex Ionescu. *Windows Internals*. Microsoft Press, Redmond, Washington, EE.UU., junio 2009. ISBN 0-7356-2530-1.

[RT74] Dennis Ritchie y Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch93] Fred B. Schneider. What good are models and what models are good? En Sape Mullender, editor, *Distributed Systems*, capítulo 2, págs. 17–25. Addison Wesley, 2^a edición, 1993.

[SDP92] Marc Shapiro, Peter Dickman y David Plainfossé. Robust, distributed references and acyclic garbage collection. En *Symp. Principles Distrib. Comput. (PODC)*, págs. 135–146, Vancouver, Canadá, agosto 1992. ACM Press.

[SH06] Mark Smith y Tim Howes. RFC4515: LDAP: String representation of search filters. Internet Engineering Task Force Request for Comments, Network Working Group, junio 2006.

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. En *6th Intnl. Conf. on Distrib. Comput. Sys. (ICDCS)*, págs. 198–204, Cambridge, Massachusetts, EE.UU., mayo 1986.

[SM86] Robert J. Souza y Steven P. Miller. UNIX and remote procedure calls: A peaceful coexistence? En *6th Intnl. Conf. Distr. Comput. Syst. (ICDCS)*, págs. 268–277, Cambridge, Massachusetts, EE.UU., mayo 1986. IEEE-CS Press.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek y Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. En *ACM SIGCOMM Conf. on Appl., Techn., Arch. and Prot. for Comput. Commun. (SIGCOMM)*, págs. 149–160, San Diego, CA, EE.UU., agosto 2001.

[SRL90] Lui Sha, Ragunathan Rajkumar y John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[SS75] Jerome H. Saltzer y Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SS94] Mukesh Singhal y Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. McGraw-Hill, Inc., New York, EE.UU., 1994. ISBN 0-07-113668-1.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, octubre 1988.

[Tv08] Andrew S. Tanenbaum y Maarten van Steen. *Sistemas Distribuidos: Principios y Paradigmas*. Pearson, 2^a edición, 2008. ISBN 9789702612803.

[VRMCL09] Luis Miguel Vaquero, Luis Rodero-Merino, Juan Cáceres y Maik A. Lindner. A break in the clouds: towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[WK00] Steve Wilson y Jeff Keselman. *Java Platform Performance: Strategies and Tactics*. Prentice-Hall, 1^a edición, junio 2000. ISBN 978-0201709698. Disponible en http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html.

[YHK93] Wengyik Yeong, Tim Howes y Steve Kille. RFC1487: X.500 lightweight directory access protocol. Internet Engineering Task Force Request for Comments, Network Working Group, julio 1993.

[Zim80] Hubert Zimmermann. OSI reference model: The ISO model of architecture for open systems interconnection. *IEEE Trans. on Commun.*, 28(4):425–432, abril 1980.

Índice alfabético