

COORDINACIÓN Y ACUERDO

- 11.1. Introducción
- 11.2. Exclusión mutua distribuida
- 11.3. Elecciones
- 11.4. Comunicación por multidifusión
- 11.5. Consenso y sus problemas relacionados
- 11.6. Resumen

En este capítulo, se presentan algunos temas y algoritmos relacionados con la forma en la que los procesos coordinan sus acciones y realizan acuerdos sobre valores compartidos en sistemas distribuidos aun en presencia de fallos. El capítulo empieza presentando algoritmos para conseguir exclusión mutua entre una colección de procesos y para coordinar sus accesos a los recursos compartidos. Continúa examinando cómo puede implementarse una elección en un sistema distribuido. Esto es, describe cómo un grupo de procesos pueden ponerse de acuerdo acerca de un nuevo coordinador de sus actividades tras fallar el coordinador anterior.

La segunda mitad del capítulo examinará los problemas relacionados con la comunicación por multidifusión, el consenso, los acuerdos bizantinos y la consistencia interactiva. En el caso de la comunicación por multidifusión se centra en la consecución de acuerdos en temas tales como el orden en el cual se entregan los mensajes. El consenso y el resto de los problemas se generalizan a partir de la siguiente idea: ¿cómo puede cualquier colección de procesos ponerse de acuerdo en algún valor, con independencia del dominio de los valores en cuestión? Encontramos un resultado fundamental en la teoría de sistemas distribuidos: bajo ciertas condiciones, entre las que se incluyen condiciones de fallo sorprendentemente benignas, es imposible garantizar que los procesos alcanzarán un consenso.

11.1. INTRODUCCIÓN

Este capítulo presenta una colección de algoritmos con objetivos variados, pero que comparten un fin que es fundamental en los sistemas distribuidos: dado un conjunto de procesos, coordinar sus acciones o ponerse de acuerdo en uno o más valores. Por ejemplo, en el caso de un mecanismo complejo como el de una nave espacial, es esencial que los computadores que realizan el control acuerden condiciones tales como si la misión de la nave prosigue o si ésta ha sido abortada. Además, los computadores deben coordinar sus acciones correctamente con respecto a los recursos compartidos (los sensores y actuadores de la nave espacial). Los computadores han de ser capaces de hacerlo incluso cuando no hay una relación establecida maestro-esclavo entre los componentes (que haría la coordinación particularmente simple). La razón para evitar relaciones fijas maestro-esclavo es que, frecuentemente, requerimos de nuestros sistemas que se mantengan trabajando correctamente incluso si ocurre un fallo, por lo que es necesario evitar puntos de fallo individuales tales como maestros prefijados.

Una distinción importante, como se hizo en el Capítulo 10, será si el sistema distribuido en estudio es síncrono o asíncrono. En un sistema asíncrono, no podemos hacer suposiciones con relación a la coordinación temporal. En un sistema síncrono, supondremos que hay límites para el retraso máximo en la transmisión de mensajes, en el tiempo para ejecutar cada proceso y en el promedio de derivas de los relojes. Las suposiciones de sincronía permiten que se usen timeouts para detectar caídas en los procesos.

Otra meta importante del capítulo, cuando se discuta sobre algoritmos, será la de considerar la presencia de fallos y cómo tenerlos en cuenta cuando se diseñan dichos algoritmos. En la Sección 2.3.2 se presentó un modelo de fallo que se usará en este capítulo. El abordar el tema de los fallos es una tarea sutil; por lo tanto, se empezará considerando algoritmos que no toleran fallos, se avanzará introduciendo fallos benignos y se llegará a considerar cómo tolerar fallos arbitrarios. Además, se encuentra un resultado fundamental en la teoría de sistemas distribuidos. Incluso bajo condiciones de fallo sorprendentemente benignas, es imposible garantizar en un sistema asíncrono que una colección de procesos puedan ponerse de acuerdo en un valor compartido; por ejemplo, que todos los procesos involucrados en el control de la nave espacial se pongan de acuerdo en si «la misión sigue» o «la misión aborta».

La Sección 11.2 examina el problema de la exclusión mutua distribuida. Ésta es la extensión a los sistemas distribuidos del conocido problema de evitar condiciones de competición continua en los núcleos y aplicaciones multi-hilo. La exclusión mutua es un problema importante a resolver, dado que lo que ocurre fundamentalmente en un sistema distribuido es que los procesos comparten recursos. A continuación, la Sección 11.3 presenta un asunto relacionado, aunque más general, que es cómo «elegir» dentro de una colección de procesos a uno para que desarrolle un papel especial. Por ejemplo, en el Capítulo 10 se vio cómo los procesos sincronizaban sus relojes con un servidor de tiempos previamente designado. Si el servidor falla y han sobrevivido otros servidores que pueden desempeñar ese papel, entonces es necesario elegir un servidor que lo releve con el fin de mantener la consistencia.

La comunicación por multidifusión es el tema de la Sección 11.4. Tal y como se explicó en la Sección 4.5.1, la técnica de multidifusión es un paradigma de comunicación muy útil, pudiendo aplicarse en temas tan dispares como la localización de recursos o la coordinación de actualizaciones de datos replicados. La Sección 11.4 examina la fiabilidad y la semántica de la ordenación en la técnica de multidifusión y proporciona algoritmos para lograr las distintas variantes. La entrega en la multidifusión es esencialmente un problema de llegar a un acuerdo entre procesos: los receptores acuerdan qué mensajes recibirán y en qué orden. La Sección 11.5 describe el problema de acuerdos de forma más general, principalmente en las formas conocidas como consenso y acuerdos bizantinos.

El tratamiento que se sigue en este libro implica el establecimiento de las hipótesis y los objetivos que deben cumplirse, y dar una explicación informal de por qué los algoritmos que se presentan son correctos. No puede proporcionarse una aproximación más rigurosa dados los problemas de espacio. Debido a esto, remitiremos al lector a los textos que proporcionan informes minuciosos de algoritmos distribuidos, tales como Attiya y Welch [1998] y Lynch [1996].

Antes de introducir los problemas y los algoritmos, discutiremos las hipótesis sobre fallos y la forma práctica de detectar fallos en un sistema distribuido.

11.1.1. SUPOSICIONES SOBRE FALLOS Y DETECTORES DE FALLOS

Para simplificar, este capítulo supone que cada par de procesos están conectados por canales fiables. Esto es, aunque los componentes de la red subyacente puedan sufrir fallos, los procesos utilizan un protocolo de comunicación fiable que enmascara estos fallos, por ejemplo mediante la retransmisión de mensajes perdidos o corruptos. También por simplificar, se supone que ningún fallo en un proceso implica una amenaza para la capacidad de otros procesos de comunicarse. Esto significa que ningún proceso depende de otro para enviar mensajes.

Nótese que un canal fiable *al final* entrega un mensaje en el búfer de entrada del receptor. En un sistema síncrono, suponemos que existe una redundancia en el hardware allí donde sea necesario de tal forma que un canal fiable no sólo entrega finalmente un mensaje a pesar de los fallos, sino que lo hace dentro de un tiempo límite especificado.

En un intervalo de tiempo cualquiera, la comunicación entre ciertos procesos puede tener éxito mientras que la comunicación entre otros procesos puede verse retrasada. Por ejemplo, el fallo de un encaminador entre dos redes puede significar que un conjunto de cuatro procesos se divide en dos parejas, de tal forma que la comunicación dentro de las parejas es posible sobre sus respectivas redes; pero la comunicación entre las parejas no es posible mientras el encaminador siga fallando. Esto se conoce como una *partición de la red* (véase la Figura 11.1). Sobre una red punto a punto como es Internet, la presencia de topologías complicadas y elecciones de encaminamiento independientes conllevan que la conectividad pueda ser *asimétrica*: la comunicación es posible entre el proceso *p* y el proceso *q*, pero no viceversa. La conectividad puede ser también *intransitiva*: la comunicación es posible desde *p* hasta *q* y desde *q* hasta *r*; pero *p* no puede comunicarse directamente con *r*. Así, nuestra suposición de fiabilidad implica que al final cualquier enlace o encaminador caído será reparado o rodeado. Sin embargo, no todos los procesos son capaces de comunicarse al mismo tiempo.

El capítulo supone, salvo que se diga lo contrario, que los procesos sólo fallan por caídas, una hipótesis que es suficientemente buena para muchos sistemas. En la Sección 11.5 consideraremos cómo tratar los casos en los que los procesos tengan fallos arbitrarios (extraños). Cualquiera que

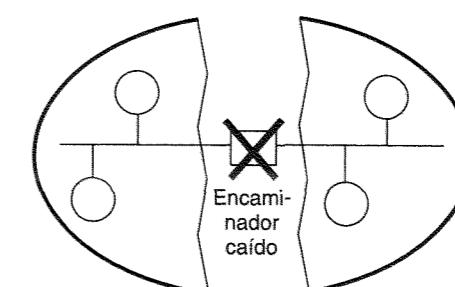


Figura 11.1. Una partición de la red.

sea el fallo, un proceso *correcto* es uno que no muestra fallos en ningún momento de la ejecución que se esté considerando. Nótese que el término corrección se aplica a la ejecución completa, no sólo a una parte de ella. Por lo tanto, un proceso que sufre un fallo por caída se dice que está «sin fallo» antes de ese momento, pero no que sea «correcto» antes de ese momento.

Uno de los problemas en el diseño de algoritmos que puedan superar las caídas de procesos es el de decidir cuándo un proceso ha caído. Un *detector de fallo* [Chandra y Toueg 1996, Stelling y otros 1998] es un servicio al que recurren los procesos para saber si un proceso determinado ha fallado. Frecuentemente, se implementa usando un objeto local en cada proceso (en el mismo computador) que ejecuta un algoritmo de detección de fallos en colaboración con sus equivalentes en los otros procesos. El objeto local a cada proceso se llama un *detector de fallo local*. Se esbozará brevemente cómo implementar detectores de fallo, pero antes hemos de concentrarnos en algunas de las propiedades de los detectores de fallos.

Un «detector» de fallos no es necesariamente exacto. La mayor parte de ellos pueden catalogarse como *detectores de fallo no fiables*. Uno de estos detectores puede generar uno de dos valores cuando se le proporcione la identidad de un proceso: *Sospechoso* o *No sospechoso*. Ambos resultados deben considerarse como indicios que pueden reflejar de forma precisa o no si el proceso ha fallado en realidad. Un resultado de *No sospechoso* significa que el detector ha recibido recientemente evidencias que le sugieren que el proceso no ha fallado; por ejemplo, se recibió recientemente un mensaje enviado por él. Pero, por supuesto, el proceso puede haber fallado desde entonces. Un resultado de *Sospechoso* debe entenderse como que el detector de fallo tiene alguna indicación de que el proceso puede haber fallado. Por ejemplo, puede ser que no haya recibido ningún mensaje por parte del proceso en un período de silencio mayor al permitido (incluso en un sistema asíncrono pueden usarse como indicios algunos límites superiores prácticos). La sospecha puede estar equivocada; por ejemplo, el proceso puede estar funcionando correctamente, pero al otro lado de una partición en la red; o puede estar funcionando más lentamente de lo esperado.

Un *detector de fallo fiable* es aquel que siempre detecta fallos en un proceso de forma exacta. Responde a las preguntas de los procesos bien con el término *No sospechoso*, que, como antes, puede ser sólo un indicio, o con el término *Fallido*. Un resultado de *Fallido* significa que el detector ha determinado que el proceso se ha caído. Recuérdese que un proceso que se ha caído permanece en ese estado, dado que por definición un proceso nunca continúa una vez se ha malogrado.

Es importante darse cuenta de que, aunque se habla de un detector de fallos actuando dentro de una colección de procesos, la respuesta que el detector proporciona al proceso es tan buena como la información de la que dispone el proceso. Además, un detector puede dar algunas veces diferentes respuestas a diferentes procesos, dado que las condiciones de comunicación varían de un proceso a otro.

Podemos implementar un detector de fallos no fiable utilizando el siguiente algoritmo. Cada proceso p manda un mensaje « p está aquí» al resto de procesos, y lo hace cada T segundos. El detector de fallos utiliza una estimación del máximo tiempo de transmisión de un mensaje de D segundos. Si el detector de fallos local en un proceso q no recibe un mensaje « p está aquí» dentro de los $T + D$ segundos desde el último, entonces informa a q que p es *Sospechoso*. Sin embargo, si posteriormente recibe un mensaje « p está aquí», entonces informa a q de que p está correctamente.

En un sistema distribuido real hay límites prácticos en los tiempos de transmisión de mensajes. Incluso los sistemas de correo abandonan tras unos pocos días, dado que es probable que los enlaces de comunicación y los encaminadores hayan sido reparados en ese tiempo. Si elegimos valores pequeños para T y D (que sumen en total 0,1 segundos, por ejemplo) entonces el detector de fallos probablemente sospechará bastantes veces de procesos que no se han caído, y una buena parte del ancho de banda se dedicará a los mensajes « p está aquí». Si elegimos un valor de timeout total grande de (por ejemplo, una semana) entonces los procesos que se han caído serán considerados frecuentemente como *No sospechosos*.

Una solución práctica para este problema es usar valores de timeout que reflejen las condiciones de retraso observadas en la red. Si un detector de fallos local recibe un mensaje « p está aquí» en 20 segundos en lugar del máximo esperado de 10 segundos, entonces tendría que cambiar el valor del tiempo para p de acuerdo con la nueva información. El detector de fallos sigue siendo no fiable y sus respuestas a las preguntas deben seguir considerándose como indicios, pero la probabilidad de que sea exacto aumenta.

En un sistema sincrónico, el sistema detector antes descrito puede convertirse en fiable. Se puede elegir D de tal forma que no sea una estimación sino un límite absoluto en los tiempos de transmisión de mensajes; la ausencia de un mensaje « p está aquí» dentro de los $T + D$ segundos lleva al detector de fallos local a concluir que el proceso p se ha caído.

Los lectores pueden preguntarse si los detectores de fallo tienen alguna utilidad práctica. Por un lado, los detectores de fallos no fiables pueden sospechar de un proceso que no haya fallado (pueden ser *inexactos*); y pueden no sospechar de un proceso que de hecho haya fallado (pueden ser *incompletos*). Por otro lado, los detectores de fallos fiables requieren que el sistema sea sincrónico (y muy pocos sistemas lo son en la práctica).

Se han presentado los detectores de fallos porque ayudan a pensar en la naturaleza de los fallos en un sistema distribuido. Y cualquier sistema práctico que se diseñe para soportar fallos debe detectarlos, aunque no lo haga de una forma perfecta. No obstante, parece que incluso los detectores de fallos no fiables con algunas propiedades bien definidas pueden ayudar a proporcionar soluciones prácticas al problema de la coordinación de procesos en presencia de fallos. Se volverá a este tema en la Sección 11.5.

11.2. EXCLUSIÓN MUTUA DISTRIBUIDA

Los procesos distribuidos necesitan frecuentemente coordinar sus actividades. Si una colección de procesos comparte un recurso o una colección de recursos, entonces se requiere con frecuencia la exclusión mutua para prevenir interferencias y asegurar la consistencia cuando se accede a los recursos. Éste es el problema de la *sección crítica*, conocido en el dominio de los sistemas operativos. Sin embargo, en un sistema distribuido ni las variables compartidas ni las utilidades proporcionadas por un núcleo local pueden usarse para solucionarlo de forma general. Se requiere una solución para la *exclusión mutua distribuida*: una que esté basada exclusivamente en el paso de mensajes.

En algunos casos, los recursos compartidos son gestionados por servidores que, además, proporcionan mecanismos para la exclusión mutua. El Capítulo 12 describe cómo algunos servidores sincronizan los accesos de los clientes a los recursos. Pero en algunos casos prácticos se requiere un mecanismo separado de exclusión mutua.

Considérese el caso de los usuarios que actualizan un archivo de texto. Un método sencillo de asegurar que sus actualizaciones sean consistentes es permitirles que cada vez sólo uno tenga acceso, exigiendo al editor que bloquee el archivo antes de que se hagan las actualizaciones. Los servidores de archivos NFS, descritos en el Capítulo 8, se diseñan para que no tengan estado y, por lo tanto, no admiten el bloqueo de archivos. Por esta razón, los sistemas UNIX proporcionan un servicio de bloqueo de archivos separado, implementado mediante el proceso demonio *lockd*, para gestionar las peticiones de bloqueo por parte de los clientes.

Un ejemplo, particularmente interesante, se da donde no existen servidores de forma que una colección de procesos de igual importancia debe coordinar sus accesos a recursos compartidos por ellos mismos. Esto ocurre de forma habitual en redes tales como las de tipo Ethernet y las redes inalámbricas IEEE 802.11 en modo *ad hoc*, donde las interfaces de red cooperan como procesos iguales, de tal forma que sólo un nodo transmite cada vez en el medio compartido. Otro ejemplo

sería un sistema que realiza el seguimiento del número de vacantes en un aparcamiento de vehículos con un proceso a cada entrada y salida que observe el número de vehículos que entran y salen. Cada proceso mantiene un registro del número total de vehículos dentro del aparcamiento, e informa de si está lleno o no. Los procesos han de actualizar el registro del número de vehículos de forma consistente. Hay distintas formas de conseguirlo, pero sería conveniente para estos procesos que fueran capaces de obtener exclusión mutua únicamente comunicándose entre ellos, eliminando la necesidad de un servidor aparte.

Sería útil tener a nuestra disposición un mecanismo genérico para la exclusión mutua distribuida que fuera independiente del esquema particular de gestión de recursos en cuestión. A continuación se examinarán algunos algoritmos que lo consiguen.

11.2.1. ALGORITMOS PARA LA EXCLUSIÓN MUTUA

Considérese un sistema de N procesos p_i , $i = 1, 2, \dots, N$ que no comparten variables. Los procesos acceden a recursos compartidos comunes, pero lo hacen en una sección crítica. Para simplificar, se supondrá que hay solamente una sección crítica. El procedimiento para ampliar los algoritmos que presentan más de una sección crítica es sencillo.

Se supone que el sistema es asíncrono, que los procesos no fallan y que la entrega de mensajes es fiable, de tal forma que cualquier mensaje enviado finalmente es entregado intacto y exactamente una vez.

El protocolo a nivel de aplicación para ejecutar una sección crítica es como sigue:

```
entrar()           // entrada en la sección crítica – bloquéese si es necesario
acceso_a_Recursos() // acceso a los recursos compartidos en la sección crítica
salir()            // salida de la sección crítica – pueden entrar otros procesos
```

Nuestros requisitos esenciales para la exclusión mutua son los siguientes:

- EM1: (seguridad) A lo sumo un proceso puede estar ejecutándose cada vez en la sección crítica (SC).
- EM2: (pervivencia) Las peticiones para entrar y salir de la sección crítica al final son concedidas.

La condición EM2 implica la ausencia tanto de estancamiento como de inanición. Un estancamiento implicaría que dos o más procesos quedasen atascados indefinidamente intentando entrar o salir de la sección crítica, en virtud de su mutua interdependencia. Pero, incluso sin estancamiento, un algoritmo no muy logrado podría llevar a la *inanición*: el aplazamiento indefinido de la entrada de un proceso que lo ha solicitado.

La ausencia de inanición es una condición de *equidad*. Otro asunto relativo a la equidad es el orden en el cual los procesos entran en la sección crítica. No es posible ordenar la entrada en la sección crítica por el momento en el que el proceso lo solicita, dada la ausencia de relojes globales. Pero un requisito útil que se impone algunas veces para conseguir la equidad utiliza la ordenación *sucedió-antes* (véase la Sección 10.4) entre mensajes que solicitan entrada en la sección crítica:

- EM3: (\rightarrow ordenación) Si una petición para entrar en la SC *ocurrió-antes* que otra, entonces la entrada a la SC se garantiza en ese orden.

Si una solución garantiza la entrada en la sección crítica en una ordenación *sucedió-antes*, y si todas las peticiones están relacionadas mediante el *sucedió-antes*, entonces no es posible para un proceso entrar en la sección crítica más de una vez mientras otro espera para entrar. Esta ordenación, además, permite a los procesos coordinar sus accesos a la sección crítica. Un proceso multihilado puede continuar con otros procesamientos mientras un hilo espera que se le garantice la entra-

da a una sección crítica. Durante este tiempo, podría enviar un mensaje a otro proceso, el cual, a continuación, también intentaría entrar en la sección crítica. EM3 especifica que debe garantizarse el acceso al primer proceso antes que al segundo.

A continuación se evalúa el rendimiento de los algoritmos para la exclusión mutua de acuerdo a los siguientes criterios:

- El *ancho de banda* consumido, que es proporcional al número de mensajes enviados en cada operación *entrar* y *salir*.
- El *retraso del cliente* en el que incurre un proceso en cada operación *entrar* y *salir*.
- El efecto del algoritmo sobre la capacidad de procesamiento del sistema. Ésta es la tasa promedio a la que la colección de procesos, en su totalidad, puede acceder a la sección crítica, teniendo en cuenta que es necesaria alguna comunicación entre procesos sucesivos. Se mide el efecto por medio del *retraso en la sincronización* entre un proceso que sale de la sección crítica y el siguiente proceso que entra en ella; la capacidad de procesamiento es mayor cuando el retraso en la sincronización es menor.

No se tendrá en cuenta en nuestra descripción la implementación de accesos a recursos. Sin embargo, se asumirá que los procesos cliente se comportan bien y emplean un tiempo finito accediendo a los recursos dentro de sus secciones críticas.

◊ **El algoritmo del servidor central.** La forma más simple de conseguir exclusión mutua es emplear un servidor que dé los permisos para entrar en la sección crítica. La Figura 11.2 muestra el uso de este servidor. Para entrar en una sección crítica, un proceso envía un mensaje de petición al servidor y espera una respuesta por su parte. Conceptualmente, la respuesta constituye un testigo que significa permiso para entrar en la sección crítica. Si ningún otro proceso tiene el testigo en el instante de la petición, entonces el servidor responde inmediatamente, dándoselo. Si en ese momento lo tiene otro proceso, entonces el servidor no responde sino que lo pone en una cola la petición. Cuando un proceso sale de la sección crítica envía un mensaje al servidor, devolviéndole el testigo.

Si la cola de procesos en espera no está vacía, entonces el servidor escoge la entrada más antigua en la cola, la elimina y responde al proceso correspondiente. El proceso seleccionado mantiene entonces el testigo. En la figura, se muestra una situación en la cual la petición de p_2 ha sido añadida a la cola, que ya contenía la petición de p_4 . A continuación, p_3 sale de la sección crítica, y el servidor elimina la entrada de p_4 y le responde, concediéndole permiso para entrar. El proceso p_1 no requiere en ese momento entrada en la sección crítica.

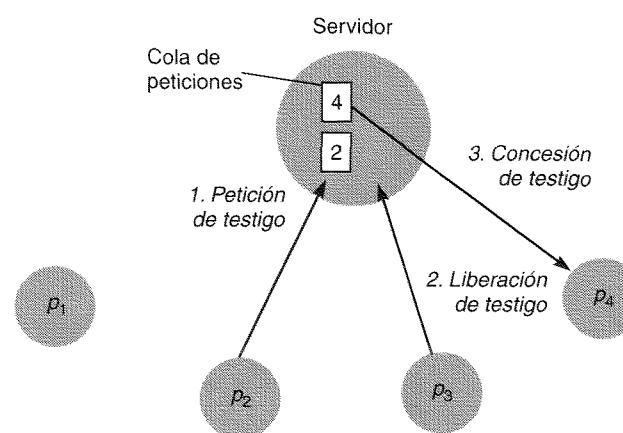


Figura 11.2. Servidor que gestiona un testigo de exclusión mutua para un conjunto de procesos.

Dada nuestra suposición de que no ocurren fallos, es fácil ver que las condiciones de seguridad y pervivencia se consiguen mediante este algoritmo. No obstante, los lectores deben observar que el algoritmo no satisface la propiedad EM3.

A continuación, se evalúa el rendimiento de este algoritmo. La entrada en la sección crítica, incluso cuando no haya un proceso ocupándola, conlleva dos mensajes (una *peticIÓN* seguida por una *concesIÓN*) y retrasa al proceso que realiza la petición debido a la duración de este viaje de ida y vuelta. La salida de la sección crítica implica un mensaje de *liberaciÓN*. Si se supone que el paso de mensajes es asíncrono, esto no retrasa al proceso que sale.

El servidor puede convertirse en un cuello de botella para el rendimiento del sistema considerando en su totalidad. El retraso en la sincronización es el tiempo que se tarda en un viaje de ida y vuelta: un mensaje de *liberaciÓN* hacia el servidor, seguido por un mensaje de *concesIÓN* al siguiente proceso que va a entrar en la sección crítica.

◊ **Un algoritmo basado en un anillo.** Una de las maneras más simples de conseguir la exclusión mutua entre los N procesos sin que se necesite un proceso adicional es organizarlos en un anillo lógico. Esto sólo requiere que cada proceso p_i tenga un canal de comunicación hacia el siguiente proceso en el anillo, $p_{(i+1) \bmod N}$. La idea se basa en conseguir la exclusión obteniendo un testigo mediante un mensaje que se pasa de un proceso a otro en una única dirección alrededor del anillo; por ejemplo, en el sentido de las agujas del reloj. La topología en anillo no tiene por qué estar relacionada con las interconexiones físicas subyacentes entre los computadores.

Si un proceso no requiere entrar en la sección crítica cuando recibe el testigo, entonces, inmediatamente, lo hace avanzar hacia su vecino. Un proceso que requiera el testigo espera hasta recibarlo y, en este caso, lo retiene. Cuando el proceso salga de la sección crítica enviará el testigo hacia el siguiente vecino.

La disposición de los procesos se muestra en la Figura 11.3. Verificar que las condiciones EM1 y EM2 se cumplen con este algoritmo es inmediato; sin embargo, también se cumple que el testigo no se obtiene necesariamente en el orden sucedió-antes (téngase en cuenta que los procesos pueden intercambiar mensajes con independencia de la rotación del testigo).

Este algoritmo consume continuamente ancho de banda de la red (excepto cuando un proceso está dentro de la sección crítica): los procesos envían mensajes alrededor del anillo incluso cuando ningún proceso requiere entrar en la sección crítica. El retraso experimentado por un proceso que está pidiendo entrar en la sección crítica está entre 0 mensajes (cuando acaba de recibir el testigo) y N mensajes (cuando acaba de pasar el testigo). Para salir de la sección crítica se requiere sola-

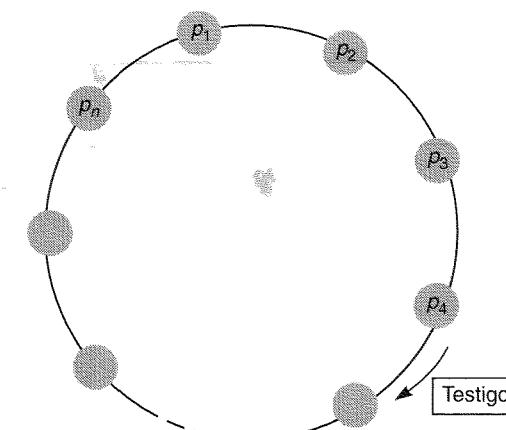


Figura 11.3. Anillo de procesos que transfieren un testigo de exclusión mutua.

```

En la inicialización
estado := LIBERADA;

Para entrar en la sección crítica
estado := BUSCADA;
Multitransmite petición a todos los procesos; } Se aplaza aquí el procesamiento de peticiones
T := marca temporal de la petición;
Espera hasta que (número de respuestas recibidas = (N - 1));
estado := TOMADA;

Al recibir una petición <Ti, pi> en el proceso pi (i ≠ j)
si (estado = TOMADA o (estado = BUSCADA y (T, p) < (Ti, pi)))
entonces
    pon en la cola la petición por parte de pi sin responder;
sino
    responde inmediatamente a pi;
finsi

Para salir de la sección crítica
estado := LIBERADA;
responde a cualquiera de las peticiones en la cola;

```

Figura 11.4. Algoritmo de Ricart y Agrawala.

mente un mensaje. El retraso en la sincronización entre la salida de un proceso de la sección crítica y la entrada del siguiente proceso está en algún punto entre 1 y N transmisiones de mensajes.

◊ **Un algoritmo que usa multidifusión y relojes lógicos.** Ricart y Agrawala [1981] desarrollaron un algoritmo para implementar la exclusión mutua entre N procesos de importancia pareja basado en la técnica de multidifusión. La idea básica es que los procesos que necesitan entrar en una sección crítica envían un mensaje de petición mediante multidifusión y pueden entrar en ella solamente cuando el resto de los procesos haya respondido al mensaje. Las condiciones bajo las cuales un proceso responde a una petición se diseñan para asegurar que se cumplen las condiciones EM1 a EM3.

Los procesos p_1, p_2, \dots, p_N llevan distintos identificadores numéricos. Se supone que disponen de canales de comunicación con el resto y cada proceso p_i mantiene un reloj tipo Lamport, actualizado de acuerdo con las reglas RL1 y RL2 de la Sección 10.4. Aquellos mensajes que soliciten entrar tienen la forma $\langle T, p_i \rangle$ donde T es el marca de tiempo del emisor y p_i es su identificador.

Cada proceso registra en una variable *estado* el estar fuera de la sección crítica (LIBERADA), de querer entrar (BUSCADA) o de estar en la sección crítica (TOMADA). El protocolo se muestra en la Figura 11.4.

Si un proceso solicita entrar y el estado de todos los procesos es LIBERADA, entonces todos los procesos contestarán inmediatamente al que hizo la solicitud y éste obtendrá la entrada. Si algún proceso estuviese en el estado TOMADA no responderá a las peticiones hasta que haya finalizado con la sección crítica; por lo tanto, durante ese tiempo no se concederá la entrada al proceso que realizó la solicitud. Si dos o más procesos solicitan la entrada al mismo tiempo entonces la petición que exhiba una marca temporal más baja será la que recoja las $N - 1$ respuestas, garantizando que será el siguiente en entrar. Si las dos peticiones llevan marcas temporales tipo Lamport iguales, las solicitudes se ordenan de acuerdo a los identificadores correspondientes a cada proceso. Nótese que cuando un proceso solicita la entrada, aplaza el procesamiento de peticiones por parte de otros procesos hasta que su propia solicitud haya sido enviada y se haya grabado la marca temporal T de la misma. Ésta es la razón por la que los procesos realizan decisiones consistentes cuando procesan peticiones.

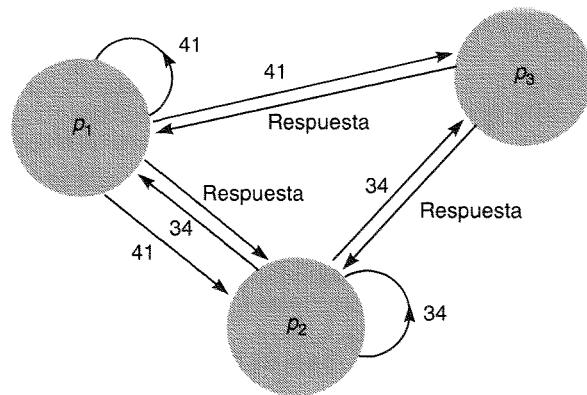


Figura 11.5. Sincronización mediante multitransmisión.

Este algoritmo cumple la propiedad de seguridad EM1. Si fuese posible que dos procesos p_i y p_j ($i \neq j$) entrasen a la sección crítica al mismo tiempo, entonces ambos procesos deberían haberse contestado mutuamente. Pero teniendo en cuenta que los pares $\langle T_i, p_i \rangle$ están totalmente ordenados, esto es imposible. Dejamos a los lectores la verificación de que el algoritmo además cumple los requisitos EM2 y EM3.

Para ilustrar el algoritmo, considérese una situación que involucra a tres procesos, p_1 , p_2 y p_3 que aparecen en la Figura 11.5. Supongamos que p_3 no está interesado en entrar en la sección crítica, y que p_1 y p_2 lo solicitan de forma concurrente. La marca temporal de la petición de p_1 es 41 y la de p_2 es 34. Cuando p_3 recibe sus peticiones, responde inmediatamente. Cuando p_2 recibe la petición de p_1 se da cuenta de que su propia petición tiene una marca temporal más baja y por lo tanto no responde, manteniendo a p_1 fuera. Al mismo tiempo, p_1 encuentra que la petición de p_2 tiene una marca temporal más baja que la de su propia solicitud y responde inmediatamente. Una vez recibida esta segunda respuesta p_2 puede entrar en la sección crítica. Cuando p_2 salga de la sección crítica, responderá a la petición de p_1 , garantizando así su entrada.

Conseguir la entrada en la sección crítica supone $2(N - 1)$ mensajes en este algoritmo: $N - 1$ multidifundir la solicitud, seguido de $N - 1$ respuestas. En el caso en que hubiese un hardware que diese soporte a la multidifusión sólo se requeriría un mensaje para la petición; entonces el número total de mensajes sería N . Esto hace que el algoritmo tenga un mayor coste, en términos de consumo de ancho de banda, que los algoritmos antes descritos. Sin embargo, la espera del cliente durante la solicitud de entrada es otra vez la de un viaje de ida y vuelta (si se ignora cualquier demora que pueda ocasionar el envío de mensajes de petición mediante multidifusión).

La ventaja de este algoritmo es que su retraso de sincronización es de solamente el tiempo de transmisión de un mensaje, mientras que los algoritmos anteriores incurrian en un retraso de sincronización de un viaje de ida y vuelta.

El rendimiento del algoritmo se puede mejorar. En primer lugar, nótense que el proceso que entró en último lugar a la sección crítica, y que no ha recibido peticiones para entrar, todavía sigue el protocolo, aunque pudiese decidir localmente volver a entrar. En segundo lugar, Ricart y Agrawala han refinado el protocolo de tal forma que se requieren N mensajes, en el peor de los casos, que es el más común, para conseguir la entrada sin apoyo de hardware que realice la multidifusión. Esto se describe en Raynal [1998].

◇ **Algoritmo de votación de Maekawa.** Maekawa [1985] observó que para que un proceso entrase en la sección crítica no era necesario que todos los procesos de categoría pareja a la suya le permitiesen el acceso. Los procesos sólo necesitan obtener permiso para entrar por parte de sub-

conjuntos de sus pares, siempre que los subconjuntos utilizados por cualquier par de procesos se solapen. Podemos pensar en los procesos votando para que otro pueda entrar en la sección crítica. Un proceso «candidato» debe recoger suficientes votos para entrar. Los procesos en la intersección de dos conjuntos de votantes aseguran la propiedad de seguridad EM1 (que a lo sumo un proceso pueda entrar en la sección crítica) dando su voto a un único candidato.

Maekawa asoció un *conjunto de votantes* V_i con cada proceso p_i ($i = 1, 2, \dots, N$), donde $V_i \subseteq \{p_1, p_2, \dots, p_N\}$. Los conjuntos V_i se eligen de tal forma que, para todo $i, j = 1, 2, \dots, N$:

- $p_i \in V_i$.
- $V_i \cap V_j \neq \emptyset$ – hay al menos un miembro común a cada par de conjuntos de votantes.
- $|V_i| = K$ – para ser equitativos, cada proceso ha de tener conjuntos de votantes del mismo tamaño.
- Cada proceso p_j está contenido en M de los conjuntos de votantes V_i .

Maekawa demostró que la solución óptima, que minimiza K y permite a los procesos conseguir la exclusión mutua, tiene $K \sim \sqrt{N}$ y $M = K$ (de tal forma que cada proceso está en tantos conjuntos de votantes como elementos hay en cada uno de esos conjuntos). El cálculo de los conjuntos óptimos R_i no es trivial. De forma aproximada, una manera simple de encontrar estos conjuntos R_i tales que $|R_i| = 2\sqrt{N}$ es colocar los procesos en una matriz de \sqrt{N} por \sqrt{N} y hacer que V_i sea la unión de la fila y la columna que contiene a p_i .

El algoritmo de Maekawa se muestra en la Figura 11.6. Para poder entrar a la sección crítica, un proceso p_i envía mensajes de *peticIÓN* a todos los $K - 1$ miembros de V_i . p_i no puede entrar a la sección crítica hasta que no haya recibido $K - 1$ mensajes de *respuESTA*. Cuando un proceso p_j en V_i recibe un mensaje de *peticIÓN* de p_i , envía un mensaje de *respuESTA* inmediatamente, a no ser que su estado sea TOMADA o ya haya contestado («votado») desde que recibió el último mensaje de *liberada*. Si no es éste el caso, guarda en una cola el mensaje de petición (en el orden de llegada), pero no responde todavía. Cuando un proceso recibe un mensaje de *liberada*, elimina la cabeza de su cola de peticiones pendientes (si la cola no está vacía) y envía un mensaje de *respuESTA* (un «voto»). Para dejar la sección crítica, p_i envía mensajes de *liberada* a los $K - 1$ miembros de V_i .

Este algoritmo cumple la propiedad de seguridad EM1. Si fuese posible que dos procesos p_i y p_j entrasen en la sección crítica al mismo tiempo, implicaría que los procesos en $V_i \cap V_j \neq \emptyset$ deberían haber votado por ambos a la vez. Sin embargo, el algoritmo permite a los procesos realizar, a lo sumo, un voto entre recepciones sucesivas de un mensaje *liberada*; por lo tanto, la situación es imposible.

Desgraciadamente, el algoritmo es propenso a las situaciones de estancamiento. Considerense tres procesos p_1 , p_2 y p_3 , con $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ y $V_3 = \{p_3, p_1\}$. Si los tres procesos solicitan entrada a la sección crítica de forma concurrente, entonces es posible que p_1 responda a p_2 pero que haga esperar a p_3 ; que p_2 responda a p_3 pero haga esperar a p_1 ; y que p_3 responda a p_1 pero haga esperar a p_2 . Cada proceso ha recibido una de las dos respuestas por las que espera y ninguno puede seguir adelante.

El algoritmo puede modificarse de tal forma que no se produzcan estancamientos [Saunders 1987]. En la versión modificada del protocolo, los procesos almacenan las peticiones pendientes en orden sucedió-antes, con lo cual, también se satisface el requisito EM3.

La utilización del ancho de banda por parte del algoritmo es de $2\sqrt{N}$ mensajes por entrada en la sección crítica y de \sqrt{N} mensajes por salida (asumiendo que no existan utilidades hardware para realizar la multidifusión). El número total de $3\sqrt{N}$ es superior al de $2(N - 1)$ mensajes que requiere el algoritmo de Ricart y Agrawala, siempre que $N > 4$. La espera para el cliente es la misma que la del algoritmo de Ricart y Agrawala, pero la espera en la sincronización es peor: un viaje de ida y vuelta en lugar de un tiempo de transmisión de un mensaje individual.

```

En la inicialización
estado := LIBERADA;
votado := FALSO;

Cuando  $p_i$  quiere entrar en la sección crítica
estado := BUSCADA;
Multitransmite petición a todos los procesos en  $V_i - \{p_i\}$ ;
Esperar hasta que (número de respuestas recibidas =  $(K - 1)$ );
estado := TOMADA;

Al recibir  $p_j$  una petición de  $p_i$  ( $i \neq j$ )
si (estado = TOMADA o votado = CIERTO)
entonces
    pon en la cola la petición por parte de  $p_i$  sin responder;
sino
    envía respuesta a  $p_j$ ;
    votado := CIERTO;
fin si

Para salir  $p_i$  de la sección crítica
estado := LIBERADA;
Multitransmite liberar a todos los procesos en  $V_i - \{p_i\}$ ;

Al recibir en  $p_j$  una liberación por parte de  $p_i$  ( $i \neq j$ )
si (la cola de peticiones no está vacía)
entonces
    quita la cabeza de la cola – por ejemplo,  $p_k$ ;
    envía respuesta a  $p_k$ ;
    votado := CIERTO;
sino
    votado := FALSO;
fin si

```

Figura 11.6. Algoritmo de Maekawa.

◇ **Tolerancia a fallos.** Al evaluar los algoritmos anteriores con respecto a su tolerancia a fallos, las principales consideraciones a tener en cuenta serán:

- ¿Qué ocurre cuando se pierden mensajes?
- ¿Qué ocurre cuando un proceso se cae?

Ninguno de los algoritmos descritos anteriormente toleraría la pérdida de mensajes, en el caso de que los canales no fuesen fiables. El algoritmo basado en anillo no puede tolerar un fallo por caída de ningún proceso individual. Como ya se vio, el algoritmo de Maekawa puede tolerar que algunos procesos se caigan: si un proceso que se ha caído no está en un conjunto de votantes que están siendo requeridos, entonces ese fallo no afectará al resto de procesos. El algoritmo del servidor central puede soportar la caída de un proceso cliente que ni tenga ni haya pedido el testigo. El algoritmo de Ricart y Agrawala, tal y como ha quedado descrito, puede adaptarse para tolerar la rotura de un proceso haciendo que, de forma implícita, conceda todas las peticiones.

Se invita al lector a considerar cómo adaptar los algoritmos para tolerar fallos suponiendo que esté disponible un detector de fallos fiable. Incluso con un detector de fallos fiable se ha de tener cuidado para tomar en consideración un fallo en cualquier instante de tiempo (incluso durante el proceso de recuperación) y para reconstruir el estado de los procesos tras haberse detectado un fallo. Por ejemplo, en el algoritmo del servidor central, si el servidor falla entonces se debe establecer si él tiene el testigo o si lo tiene uno de los procesos cliente.

Se examinará en la Sección 11.5 el problema general de cómo los procesos deben coordinar sus acciones en presencia de fallos.

11.3. ELECCIONES

Un algoritmo para escoger un proceso único que juegue un papel específico se llama *algoritmo de elección*. Por ejemplo, en una variante de nuestro algoritmo de «servidor central» para exclusión mutua, el «servidor» se escoge entre los procesos p_i , $i = 1, 2 \dots, N$ que necesitan usar la sección crítica. Se necesita un algoritmo de selección para escoger cuál de los procesos jugará el papel de servidor. Es esencial que todos los procesos estén de acuerdo en la elección. Tras ésta, si el proceso que juega el papel de servidor desea retirarse, entonces se requiere otro proceso de selección para escoger un sustituto.

Se dirá que un proceso *pide una elección* si lleva a cabo una acción que inicie una ejecución específica del algoritmo de elección. Un proceso individual no pide más de una elección cada vez, pero, en principio, los N procesos podrían pedir N elecciones concurrentes. Un proceso p_i , en cualquier instante de tiempo, es o bien un *participante* (lo que significa que está comprometido en alguna ejecución del algoritmo de elección) o bien, un *no participante* (lo que significa que no está comprometido actualmente en ninguna elección).

Un requisito importante es que la selección del proceso elegido sea única, incluso si distintos procesos piden elecciones de forma concurrente. Por ejemplo, dos procesos podrían decidir de forma independiente que un proceso coordinador ha fallado y pedir ambos elecciones.

Sin pérdida de generalidad, se requiere que el proceso elegido sea escogido como aquél con el mayor identificador. El »identificador» puede ser cualquier valor útil siempre que los identificadores sean únicos y estén totalmente ordenados. Por ejemplo, podríamos elegir el proceso con la menor carga computacional, teniendo que usar cada proceso $<1/carga, i>$ como su identificador, donde la $carga > 0$ y el índice del proceso i se usa para ordenar los identificadores con la misma carga.

Cada proceso p_i , ($i = 1, 2 \dots, N$) posee una variable $elegido_i$, que contendrá el identificador del proceso elegido. Cuando el proceso se convierta en participante en una elección por primera vez, fijará la variable al valor especial « \perp » para señalar que no ha sido definido todavía.

Nuestros requisitos serán que durante una ejecución en particular del algoritmo:

- | | |
|-------------------|--|
| E1: (seguridad) | Un proceso participante p_i tiene $elegido_i = \perp$ o $elegido_i = P$, donde P se elige como el proceso con el identificador mayor que no se ha caído al final de la ejecución. |
| E2: (pervivencia) | Todos los procesos p_i participan y, al final, fijan $elegido_i \neq \perp$; o bien se han caído. |

Observe que puede haber procesos p_j que no son aún participantes y que almacenan en $elegido_j$ el identificador del proceso elegido anteriormente.

Se mide el rendimiento de un algoritmo de elección por su utilización del ancho de banda total de la red (que es proporcional al número total de mensajes enviados) y por el *tiempo de vuelta* (*turnaround*) para el algoritmo, número de veces de transmisión de mensajes serializados entre el comienzo y la finalización de una ejecución individual.

◇ **Un algoritmo de elección basado en anillo.** A continuación se presenta el algoritmo de Chang y Roberts [1979], que es apropiado para una colección de procesos dispuestos en un anillo lógico. En éste cada proceso p_i tiene un canal de comunicación con el siguiente proceso del anillo, $p_{(i+1)modN}$, y todos los mensajes se mandan en el sentido de las agujas del reloj alrededor del anillo. Se supone que no ocurren fallos y que el sistema es asíncrono. El objetivo del algoritmo es elegir un proceso individual llamado el *coordinador*, que es el proceso con el identificador más grande.

Inicialmente, cada proceso se etiqueta como *no participante* en una elección. Cualquier proceso puede comenzar una elección. Para ello se marca a sí mismo como un *participante*, colocando

su identificador en un mensaje de *elección* y enviándolo hacia el próximo vecino en el sentido de las agujas del reloj.

Cuando un proceso recibe un mensaje de *elección*, compara el identificador que viene en el mensaje con el suyo. Si el identificador que le llega es mayor, entonces hace avanzar el mensaje hacia su vecino. Si el identificador que llega es más pequeño y el receptor es un *no participante*, entonces cambia el identificador del mensaje por el suyo y lo vuelve a enviar; si en este mismo caso, el receptor ya fuera un *participante*, no lo enviaría nuevamente. En cualquier caso, en el momento en el que se envía un mensaje de *elección*, el proceso se etiqueta a sí mismo como *participante*.

Sin embargo, si el identificador que se recibe es el del propio receptor, entonces ese identificador ha de ser el mayor y se convierte en el coordinador. El coordinador se señala a sí mismo como *no participante* una vez más y envía un mensaje de *elegido* a su vecino, anunciando su elección e incluyendo en el mensaje su identidad.

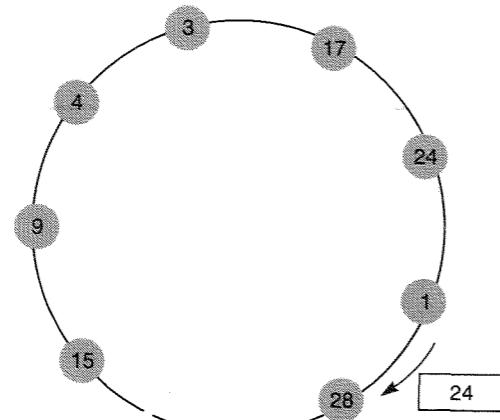
Cuando un proceso p_i recibe un mensaje de *elegido*, se marca a sí mismo como *no participante*, fija su variable $elegido_i$ al valor del identificador que hay en el mensaje y, a no ser que sea el nuevo coordinador, envía el mensaje a su vecino.

Se observa fácilmente que se cumple la condición E1. Se comparan todos los identificadores, ya que un proceso ha de recibir su propio identificador antes de enviar un mensaje de *elegido*. Para cualquier par de procesos, aquél con el identificador más grande no pasará el identificador del más pequeño. Por lo tanto, es imposible que ambos reciban sus propios identificadores de vuelta.

La condición E2 se obtiene de forma inmediata ya que se garantiza que se atraviesa el anillo (no hay fallos). Nótese cómo los estados *no participante* y *participante* se utilizan de tal manera que, si surgen mensajes cuando otro proceso comienza una elección al mismo tiempo, desaparecen tan pronto como sea posible y siempre antes de que el resultado de «ganador» de la elección haya sido anunciado.

Si un único proceso comienza una elección, entonces el caso de peor rendimiento ocurre cuando su vecino de al lado, en sentido anti-horario, tiene el identificador mayor. En ese caso, se necesitan $N - 1$ mensajes para llegar a ese vecino, que no anunciará su elección hasta que haya completado otra vuelta al anillo, dando lugar a N nuevos mensajes. Después, el mensaje *elegido* se envía N veces, totalizando $3N - 1$ mensajes. El tiempo necesario para completar la tarea es también $3N - 1$, ya que todos los mensajes se envían secuencialmente.

En la Figura 11.7 se muestra el funcionamiento de una elección basada en anillo. El mensaje de *elección* contiene 24 en ese momento, pero cuando llegue al proceso 28, éste lo reemplazará con su identificador.



Nota: La elección fue iniciada por el proceso 17. El identificador de proceso más elevado encontrado a continuación es el 24. Los procesos participantes están indicados con un tono más oscuro.

Figura 11.7. Progreso de una elección basada en anillo.

Aunque el algoritmo basado en anillo es útil para comprender las características de los algoritmos de elección en general, el hecho de que no tolere fallos hace que tenga escaso valor práctico. No obstante, si se dispone de un detector de fallos fiable es posible, en principio, rehacer el anillo cuando un proceso se cae.

◊ **El algoritmo abusón (bully).** El algoritmo del abusón [García-Molina 1982] permite la caída de procesos durante una elección, aunque supone que la entrega de mensajes entre procesos es fiable. A diferencia del algoritmo basado en anillo, este algoritmo supone que el sistema es síncrono, esto es, que utiliza timeouts para detectar un fallo en un proceso. Otra diferencia es que el algoritmo basado en anillo supone que los procesos tienen un conocimiento mínimo *a priori* de cada uno de los otros procesos: cada uno sabe cómo comunicarse únicamente con su vecino y ninguno conoce los identificadores de los otros procesos. Por otro lado, el algoritmo del abusón supone que cada proceso conoce qué procesos tienen identificadores mayores y que puede comunicarse con todos esos procesos.

Hay tres tipos de mensajes en este algoritmo. Un mensaje de *elección* se envía para anunciar un proceso de elección; un mensaje de *respuesta* se envía para responder a un mensaje de elección; y un mensaje *coordinador* se envía para anunciar la identidad del proceso elegido, el nuevo «coordinador». Un proceso comienza una elección cuando se da cuenta, a través de los timeouts, que el coordinador ha fallado. Además, varios procesos pueden descubrirlo de forma concurrente.

Dado que el sistema es síncrono, se puede construir un detector de fallos fiable. Existe un retraso máximo en la transmisión de un mensaje T_{trans} y un retraso máximo para el procesado de un mensaje $T_{procesado}$. Por lo tanto, podemos calcular un tiempo $T = 2T_{trans} + T_{procesado}$ que es el límite superior sobre el tiempo total transcurrido desde que se envió un mensaje a otro proceso hasta que se recibió la respuesta. Si no llega una respuesta dentro de ese tiempo T , entonces el detector de fallos local puede informar que el supuesto receptor de la petición ha fallado.

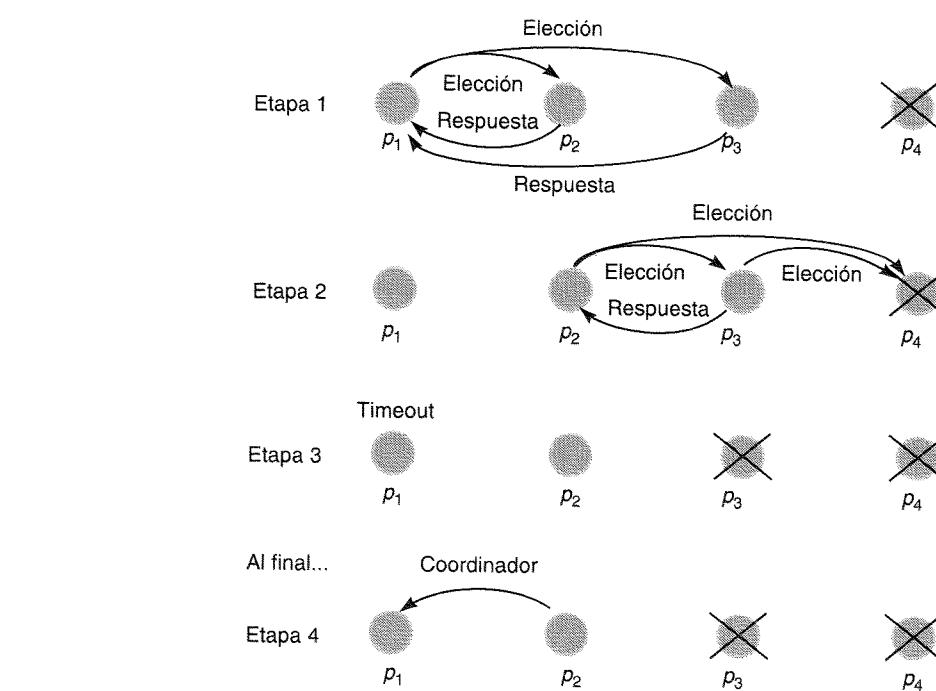
El proceso que sabe que posee el identificador más alto puede elegirse a sí mismo como el coordinador, simplemente enviando un mensaje *coordinador* a todos los procesos con identificadores más bajos. Por otro lado, un proceso con identificador más bajo comienza una elección con un mensaje de *elección* hacia aquellos procesos que tienen un identificador más alto y espera un mensaje de *respuesta*. Si no le llega ninguno dentro de un tiempo T , el proceso se considera a sí mismo el coordinador y envía un mensaje *coordinador* anunciándolo a todos los procesos con un identificador menor. Si ha recibido un mensaje de *respuesta*, el proceso espera otro período T' por que le llegue un mensaje *coordinador* procedente del nuevo coordinador. Si no llega ninguno, comienza otra elección.

Si un proceso p_i recibe un mensaje *coordinador*, fija su variable $elegido_i$ al identificador del coordinador que está contenido en el propio mensaje y trata a ese proceso como el coordinador.

Si un proceso recibe un mensaje de *elección*, devuelve un mensaje de *respuesta* y comienza otra elección a no ser que él ya haya comenzado una.

Si para sustituir a un proceso caído se lanza uno nuevo, éste comienza una elección. Si tiene el identificador de proceso más alto, entonces decidirá que es el coordinador y lo anunciará a otros procesos. Así se convertirá en el coordinador, aunque el coordinador actual siga funcionando. Esta es la razón por la que a este algoritmo se le llama algoritmo «del abusón».

El funcionamiento del algoritmo se muestra en la Figura 11.8. Hay cuatro procesos $p_1 - p_4$. El proceso p_1 detecta el fallo del coordinador p_4 y anuncia una elección (etapa 1 en la figura). Cuando reciben el mensaje de elección proveniente de p_1 , los procesos p_2 y p_3 mandan mensajes de *respuesta* a p_1 y comienzan sus propias elecciones; si bien p_3 envía un mensaje de *respuesta* a p_2 , p_3 no recibe mensaje de *respuesta* del proceso fallido p_4 (etapa 2). Por lo tanto, p_3 decide que es el nuevo coordinador. Pero antes de que pueda enviar el mensaje *coordinador*, él también falla (etapa 3). Cuando el timeout T' de p_1 expira (suponemos que ocurre antes de que expire el tiempo



La elección del coordinador p_2 después del fallo de p_4 y p_3

Figura 11.8. El algoritmo del abusón.

límite de p_2) deduce la ausencia de un mensaje *coordinador* y comienza otra elección. Al final, p_2 es elegido coordinador (etapa 4).

Este algoritmo cumple claramente la condición de pervivencia E2, ya que supone que la entrega de mensajes es fiable. Y en el caso en que no se reemplace proceso alguno, entonces también cumple la condición E1. Es imposible que dos procesos decidan que ambos son el coordinador, dado que el proceso con el identificador menor descubrirá que el otro existe y cederá ante él.

Sin embargo, *no* se puede garantizar que el algoritmo cumpla la condición de seguridad E1 si los procesos que se han caído son reemplazados por procesos con el mismo identificador. Un proceso que reemplaza a otro proceso p caído puede decidir que posee el identificador más alto justo cuando otro proceso, que ha detectado la caída de p , ha decidido que posee el identificador más alto. Los dos procesos se anunciarán de forma concurrente como los coordinadores. Desgraciadamente, no existen garantías en el orden de entrega de los mensajes, y los receptores de los mensajes pueden alcanzar conclusiones diferentes sobre qué proceso es el coordinador.

Además, la condición E1 puede romperse si los valores de los tiempos límites que se han supuesto resultan ser imprecisos, o lo que es lo mismo, si el detector de fallos en los procesos no es fiable.

Volviendo al ejemplo anterior, supongamos que p_3 no había fallado pero estaba funcionando de una forma inusualmente lenta (dicho de otra forma, la suposición de que el sistema es síncrono es incorrecta) o bien que p_3 ha fallado y ha sido reemplazado. Justo cuando p_2 manda su mensaje *coordinador*, p_3 (o su sustituto) hace lo mismo. p_2 recibe el mensaje *coordinador* de p_3 cuando ya ha enviado el suyo y, por lo tanto, determina que $elegido_2 = p_3$. A continuación, debido a los retrasos variables en la transmisión de mensajes, p_1 recibe el mensaje *coordinador* de p_2 tras el mensaje de p_3 y al final decide fijar $elegido_1 = p_2$. Por lo tanto, la condición E1 no se ha cumplido.

En lo que respecta al rendimiento del algoritmo, en el mejor caso, el proceso con el segundo identificador más alto se da cuenta del fallo en el coordinador. En ese caso se puede elegir inme-

diatamente a sí mismo como coordinador y manda $N - 2$ mensajes de *coordinador*. El tiempo necesario para completar la tarea es de un mensaje. Sin embargo, el algoritmo del abusón requiere un número de mensajes de $O(N^2)$ en el peor caso (esto es, cuando el proceso con el identificador menor detecta en primer lugar el fallo en el coordinador). En ese caso serán $N - 1$ procesos quienes comiencen a la vez el proceso de elecciones, cada uno mandando mensajes a los procesos con los identificadores mayores.

11.4. COMUNICACIÓN POR MULTIDIFUSIÓN

La Sección 4.5.1 describía la multidifusión IP, que es una implementación de la comunicación en grupo. La comunicación en grupo, o multidifusión, requiere la presencia de coordinación y acuerdo. Su objetivo es que cada uno de los procesos de un grupo reciba los mensajes enviados al grupo, frecuentemente, con garantías de que han sido entregados. Estas garantías incluyen el acuerdo sobre el grupo de mensajes que todo proceso del grupo debería recibir y sobre el orden de entrega dentro de los miembros del grupo.

Los sistemas de comunicación en grupo son extremadamente sofisticados. Incluso la multidifusión IP, que proporciona unas garantías de entrega mínimas, requiere grandes esfuerzos de realización. Como preocupaciones principales están la eficiencia en el tiempo consumido y en el uso de ancho de banda, que suponen un reto incluso para grupos estáticos de procesos. Estos problemas se multiplican cuando los procesos pueden unirse o dejar los grupos de forma arbitraria.

Aquí se estudiará la multidifusión para grupos de procesos cuyos miembros son conocidos. El Capítulo 14 ampliará este estudio para la comunicación en grupo con todas sus variantes, incluyendo la gestión de grupos que varían de forma dinámica.

La comunicación por multidifusión ha sido el objeto de estudio de varios proyectos entre los que se incluyen el sistema V [Cheriton y Zwaenepoel 1985], Chorus [Rozier y otros 1988], Amoeba [Kaashoek y otros 1989, Kaashoek y Tanenbaum 1991], Trans/Total [Melliar-Smith y otros 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse y otros 1996], Totem [Moser y otros 1996] y Transis [Dolev y Malki 1996], junto con otros trabajos reseñables que se irán citando a lo largo de esta sección.

La característica esencial de la comunicación por multidifusión es que un proceso realiza solamente una operación *multicast* para enviar un mensaje a cada uno de los miembros de un grupo (en Java esta operación es `unSocket.send(unMensaje)`) en lugar de realizar múltiples operaciones *enviar* sobre los procesos individuales. La comunicación a *todos* los procesos de un sistema, en contraposición a un subgrupo de los mismos, se conoce como difusión (*broadcast*).

El uso de una única operación *multicast* en lugar de múltiples operaciones *enviar* implica mucho más que una ventaja para el programador. Permite una implementación eficiente y que ésta proporcione garantías de entrega más fuertes que las que serían posible obtener de otra forma.

Eficiencia: la certeza de que el mismo mensaje va a ser entregado a todos los procesos de un grupo permite a la implementación ser eficiente en el uso del ancho de banda. La implementación puede proceder de tal forma que sólo envíe el mensaje una vez sobre cada enlace de comunicación, mediante el envío de un mensaje sobre un árbol de distribución. Incluso puede valerse del hardware de la red que proporcione multidifusión allí donde esté disponible. Además dicha implementación puede también minimizar el tiempo total empleado en entregar el mensaje a todos los destinos, en lugar de transmitirlo de forma separada y en serie.

Para ver estas ventajas basta comparar la utilización del ancho de banda y el tiempo total de transmisión empleado cuando se envía el mismo mensaje desde un computador en Londres a dos computadores en la misma Ethernet en Palo Alto, en el caso (a) utilizando dos órdenes UDP *enviar*, y en el caso (b) mediante una única operación de multidifusión IP. En el primer

caso, se mandan dos copias del mensaje de forma independiente y la segunda se retrasa en función de la primera. En el segundo caso, un conjunto de encaminadores atentos a la multidifusión envía una única copia del mensaje desde Londres al encaminador en la LAN de destino. El encaminador final utiliza entonces hardware de multidifusión (proporcionado por la Ethernet) para entregar el mensaje a los dos destinos, en lugar de tener que enviarlo dos veces.

Garantías de entrega: si un proceso realiza múltiples operaciones *envía* independientes a procesos individuales, entonces la implementación no tiene medios para proporcionar garantías de entrega que afecten al grupo de procesos en su conjunto. Si el remitente falla en la mitad del proceso de envío, entonces algunos miembros del grupo recibirán el mensaje mientras que otros no lo recibirán. Además, no está definido el orden relativo entre dos mensajes entregados a dos miembros de un grupo. En el caso particular de la multidifusión IP, no se ofrecen garantías de ordenación ni de fiabilidad. Sin embargo, pueden proporcionarse mayores garantías de multidifusión y en breve definiremos algunas.

◊ **Modelo del sistema.** El sistema contiene una colección de procesos que pueden comunicarse entre ellos de forma fiable a través de canales uno-a-uno. Como se había supuesto antes, los procesos sólo pueden fallar por caída.

Dichos procesos son miembros de grupos, que son los destinos de los mensajes enviados en una operación *multicast*. Por lo general, es útil permitir que los procesos sean miembros de distintos grupos de forma simultánea; por ejemplo permitir a los procesos que reciban información de distintas fuentes mediante la inclusión en distintos grupos. Pero, para simplificar la discusión de las propiedades de ordenación, algunas veces se restringirá en cada instante la pertenencia de un proceso como máximo a un grupo.

La operación *multicast*(g, m) envía el mensaje m a todos los miembros del grupo de procesos g . A esta operación le corresponde una operación *entrega*(m) que entrega el mensaje mandado por multidifusión al proceso que ha realizado la petición. Utilizamos el término *entrega* en lugar de *recibe* para dejar claro que un mensaje multidifundido no siempre se entrega a la capa de aplicación dentro del proceso tan pronto como es recibido en el nodo del proceso. Esto será explicado brevemente cuando se discuta la semántica de la entrega por multidifusión.

Todo mensaje m porta el identificador único del proceso *emisor*(m) que lo envía y el identificador único del grupo al que se envía *grupo*(m). Se supone que los procesos no mienten sobre el origen o destino de sus mensajes.

Se dice que un grupo está *cerrado* si sólo los miembros del grupo pueden multidifundir dentro de él (véase la Figura 11.9). Un proceso dentro de un grupo cerrado se entrega a sí mismo cual-

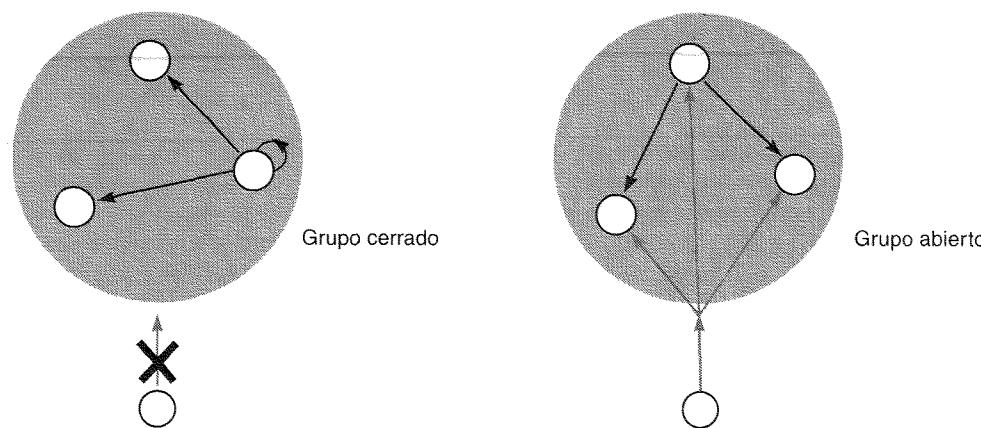


Figura 11.9. Grupos abiertos y cerrados.

quier mensaje que multidifunde al grupo. Un grupo se dice *abierto* si los procesos que no están en el grupo le pueden mandar mensajes (las categorías de «abierto» y «cerrado» también se aplican con significados análogos a las listas de correo). Los grupos cerrados de procesos son útiles, por ejemplo, entre servidores cooperantes para enviar mensajes entre ellos y que sólo ellos deberían recibir. Los grupos abiertos son útiles, por ejemplo, para informar de eventos a grupos de procesos interesados.

Algunos algoritmos suponen que los grupos son cerrados. El mismo efecto que proporciona la apertura se puede conseguir con un grupo cerrado escogiendo a un miembro del grupo y enviándole un mensaje (uno-a-uno) para que multidifunda al resto de su grupo. Rodrigues y otros [1998] discuten la multidifusión para grupos abiertos.

11.4.1. MULTIDIFUSIÓN BÁSICA

Es útil tener a nuestra disposición una primitiva de multidifusión básica que garanticé, a diferencia de la multidifusión IP, que un proceso correcto al final entregará el mensaje siempre que el sistema que realiza la multidifusión no se caiga. La primitiva se llamará *B-multicast* y su correspondiente primitiva de entrega básica se llamará *B-entrega*. Se permitirá a los procesos que pertenezcan a varios grupos y que cada mensaje se destine a un grupo en particular.

Una forma directa de implementar el *B-multicast* es utilizar una operación *envía* fiable uno-a-uno de la siguiente forma:

Para realizar *B-multicast*(g, m): para cada proceso $p \in g$, $g, \text{envía}(p, m)$;

Al *recibir*(m) en $p : B\text{-entrega}(m)$ en p .

La implementación puede usar hilos para realizar las operaciones *envía* de forma concurrente, con el fin de reducir el tiempo total empleado para entregar el mensaje. Desgraciadamente, tal implementación es susceptible de sufrir la denominada *ack-implosion* (colapso por exceso de acuses de recibo) si el número de procesos es grande. Es factible que los acuses de recibo, enviados como parte de una operación *envía* fiable, lleguen aproximadamente al mismo tiempo desde muchos procesos. Los búferes del proceso que están realizando la multidifusión se llenarán rápidamente y es probable que pierdan algún acuse de recibo. En ese caso retransmitiría el mensaje, dando lugar todavía a más acuses de recibo y un mayor desperdicio del ancho de banda. Puede construirse un servicio básico de multidifusión más práctico usando la multidifusión IP y se deja como ejercicio para el lector.

11.4.2. MULTIDIFUSIÓN FIABLE

La Sección 2.3.2 definió los canales de comunicación fiables uno-a-uno entre pares de procesos. La propiedad de seguridad requerida se denominaba *integridad*, esto es, que todo mensaje entregado es idéntico al enviado y que ningún mensaje se entrega dos veces. La propiedad de pervivencia requerida se denominaba *validez*, esto es, que todo mensaje al final sea entregado en su destino, si éste es correcto.

De forma similar a Hadzilacos y Toueg [1996], se define a continuación una *multidifusión fiable* con sus operaciones correspondientes *F-multicast* y *F-entrega*. Queda claro que sería muy conveniente disponer de propiedades análogas a la integridad y a la validez para la entrega en la operación de multidifusión. No obstante, se añade otro requisito: *todos* los procesos correctos en un grupo deben recibir un mensaje si *alguno* de ellos lo recibe. Es importante darse cuenta de que esta propiedad no existe en el algoritmo de *B-multicast* que se basa en una operación fiable de envío uno-a-uno. El emisor puede fallar en cualquier instante del *B-multicast*, por lo tanto algunos procesos pueden recibir un mensaje mientras que otros no.

Un proceso de multidifusión fiable será aquel que satisfaga las siguientes propiedades:

Integridad: un proceso correcto p entrega un mensaje m a lo sumo una vez. Además, $p \in grupo(m)$ y m fue proporcionado a la operación *multicast* por un *emisor(m)* (de la misma forma que con la comunicación uno-a-uno, los mensajes pueden distinguirse por un número de secuencia propio al emisor).

Validez: si un proceso correcto multidifunde un mensaje m , entonces al final m será entregado.

Acuerdo: si un proceso correcto entrega un mensaje m , entonces el resto de procesos correctos en el *grupo(m)* deben al final entregar el mensaje m .

Una vez definidas estas propiedades se procede a explicarlas con más detalle.

La propiedad de integridad es análoga a la de comunicación fiable uno-a-uno. La propiedad de validez garantiza la pervivencia para el emisor. Ésta podría parecer una propiedad poco habitual, ya que es asimétrica (sólo se menciona un proceso en concreto). Pero hay que darse cuenta de que las propiedades de validez y acuerdo conjuntamente permiten obtener el requisito de pervivencia global; si un proceso, el *emisor*, finalmente entrega un mensaje m entonces, puesto que los procesos correctos están de acuerdo en el conjunto de mensajes que entregan, de lo que se deduce que m será entregado, finalmente, a todos los miembros correctos del grupo.

La ventaja de expresar la condición de validez en términos de auto-entrega es la sencillez. Lo que se requiere es que el mensaje sea entregado en último caso por *algún* miembro correcto del grupo.

La condición de acuerdo está relacionada con la atomicidad, que es la propiedad de «todo o nada» aplicada a la entrega de mensajes a un grupo. Si un proceso que está enviando un mensaje mediante multidifusión se cae antes de haberlo entregado, entonces es posible que el mensaje no se entregue a ningún proceso del grupo; pero si lo ha entregado a algún proceso correcto, entonces el resto de procesos correctos lo entregarán. Muchos trabajos en la literatura existente usan el término «atómico» para incluir una condición de ordenación total, que se define a continuación.

◇ **Implementación de la multidifusión fiable sobre *B-multicast*.** La Figura 11.10 proporciona un algoritmo de multidifusión fiable con las primitivas *F-multicast* y *F-entrega*, que permiten a los procesos pertenecer a distintos grupos cerrados de forma simultánea. Para enviar un mensaje mediante *F-multicast*, un proceso envía el mensaje mediante *B-multicast* a los procesos miembros del grupo (incluido él mismo). Cuando el proceso ha sido *B-entregado*, el receptor, a cambio, realiza *B-multicast* del mensaje al grupo (si no es el emisor original) y, a continuación, *F-entrega* el mensaje. Ya que un mensaje puede llegar más de una vez a cualquier nodo, los duplicados del mensaje se detectan y no se reenvían.

En la inicialización

 Recibido := {};

Para que el proceso p realice *R-multicast* de un mensaje m a un grupo g

B-multicast(g, m); // $p \in g$ se incluye como destino

En *B-entrega*(m) a un proceso q con $g = grupo(m)$

 si ($m \notin Recibido$)

 entonces

$Recibido := Recibido \cup \{m\}$;

 si ($q \neq p$) entonces *B-multicast*(g, m); fin si;

F-entrega m ;

 fin si

Figura 11.10. Algoritmo de multidifusión fiable.

Este algoritmo claramente satisface el requisito de validez, ya que un proceso correcto realizará en último caso una *B-entrega* a sí mismo. Además, por la propiedad de integridad de los canales de comunicación subyacentes que se usan en los envíos *B-multicast*, el algoritmo también satisface la propiedad de integridad.

El acuerdo se consigue a partir del hecho de que todo proceso correcto envía mediante *B-multicast* el mensaje a otros procesos una vez que lo ha *B-entregado*. Si un proceso correcto no realiza la *F-entrega* del mensaje, entonces esto se debe a que nunca ha realizado la *B-entrega*. Esto, a su vez, puede deberse solamente a que ningún otro proceso correcto lo ha *B-entregado* antes; por lo tanto, ninguno lo *B-entregará*.

El algoritmo de multidifusión fiable que se ha descrito es correcto en un sistema asíncrono, ya que no se han realizado suposiciones sobre las temporizaciones. Sin embargo, el algoritmo es ineficiente en la práctica, ya que cada mensaje se envía $|g|$ veces a cada proceso.

◇ **Multidifusión fiable sobre multidifusión IP.** Una alternativa para realizar *F-multicast* es combinar la multidifusión IP, acuses de recibo adheridos (*piggy backed*) (esto es, acuses de recibo adjuntos a otros mensajes), y acuses de recibo negativos. Este protocolo para *F-multicast* se basa en la observación de que la comunicación por multidifusión IP casi siempre tiene éxito. En el protocolo, los procesos no envían mensajes separados de acuse de recibo; en su lugar adhieren los reconocimientos en los mensajes que envían al grupo. Los procesos envían un mensaje de respuesta por separado sólo cuando detectan que han perdido un mensaje. La respuesta indicando la ausencia de un mensaje esperado se conoce como *acuse de recibo negativo*.

La descripción supone que los grupos son cerrados. Cada proceso p mantiene un número de secuencia S_q^p para cada grupo g al que pertenece. El número de secuencia es inicialmente cero.

Cada proceso almacena además R_g^p , el número de secuencia del último mensaje que ha sido entregado por el proceso p y que fue enviado desde el grupo g .

Cuando el proceso p emplea *F-multicast* para un mensaje al grupo g , adhiere a dicho mensaje el valor S_g^p . Además adhiere acuses de recibo sobre el mensaje que envía, de la forma $\langle q, R_g^p \rangle$. Dicho acuse de recibo establece el número de secuencia del último mensaje destinado para ese grupo, que le ha sido entregado desde el proceso q desde la última vez que se multidifundió un mensaje. El proceso p que empleó la multidifusión envía a continuación el mensaje mediante multidifusión IP con su número de secuencia y con acuse de recibo adheridos para g , e incrementa S_g^p en uno.

Un proceso *F-entrega* un mensaje destinado para g con el número de secuencia S y proveniente de p si y sólo si $S = R_g^p + 1$, e incrementa R_g^p en uno inmediatamente tras la entrega. Además, retiene cualquier mensaje que no haya podido entregar en una *cola de retención* (véase la Figura 11.11); tales colas se necesitan con frecuencia para que se cumplan las garantías en la entrega de mensajes. Si, por otro lado, un mensaje que llega tiene $S \leq R_g^p$, entonces r ha entregado el mensaje antes y lo descarta. Si $S > R_g^p + 1$ o $R > R_g^p$, para cualquier acuse de recibo $\langle q, R \rangle$ incluido en otro mensaje, entonces r ha perdido uno o más mensajes. En ese momento los solicita mediante el envío de acuses de recibo negativos. Puede enviar la petición al proceso del cual percibió la omisión o al emisor original, si es distinto. En algunas variaciones del protocolo, aquél realiza la petición mediante multidifusión a los procesos de su red vecina, por si acaso lo hubieran recibido.

La propiedad de integridad se consigue con la detección de duplicados y las propiedades subyacentes de la multidifusión IP (que utiliza sumas de comprobación para eliminar mensajes corruptos). La propiedad de validez se cumple porque la multidifusión IP tiene dicha propiedad. Por acuerdo necesitamos, primero, que un proceso siempre pueda detectar mensajes perdidos. Esto a cambio significa que siempre recibirá un mensaje posterior que le permite detectar la omisión. Como establece este protocolo simplificado, garantizamos la detección de los mensajes perdidos sólo en el caso en que se suponga que cada proceso multidifunda de forma indefinida. Segundo, la

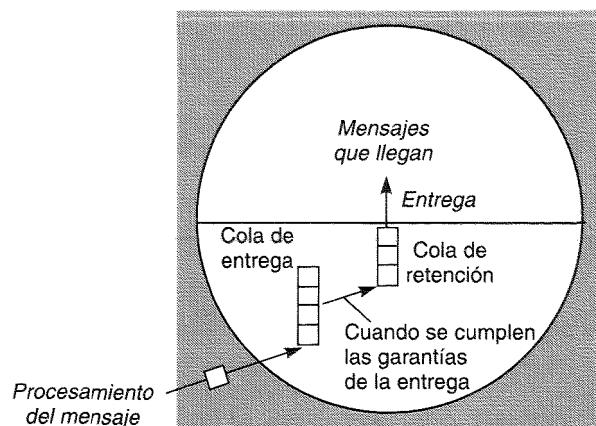


Figura 11.11. Cola de retención para la llegada de mensajes de multidifusión.

propiedad de acuerdo requiere que se garantice que haya una copia disponible para un proceso que no lo haya recibido. Por lo tanto, se supone que los procesos mantienen copias, de forma indefinida, de los mensajes que han enviado.

Ninguna de las suposiciones hechas para asegurar el acuerdo es práctica. Sin embargo, el acuerdo se consigue en la práctica en los protocolos de los cuales se deriva éste: el protocolo Psync [Peterson y otros 1989], el protocolo Trans [Melliar-Smith y otros 1990] y el protocolo de multidifusión fiable y ampliable [Floyd y cols. 1997]. Psync y Trans garantizan, además, una entrega ordenada.

◊ **Propiedades uniformes.** La definición de acuerdo antes mencionada sólo se refiere al comportamiento de los procesos *correctos*, que nunca fallan. Considérese qué ocurriría en el algoritmo de la Figura 11.10 si un proceso no es correcto y se cae tras haber *F-entregado* un mensaje. Teniendo en cuenta que cualquier proceso que *F-entrega* el mensaje ha de haber realizado previamente un *B-multicast* del mismo, se concluye que todos los procesos correctos acabarían entregando el mensaje.

Cualquier propiedad que se cumpla tanto si los procesos son correctos como si no se denomina propiedad *uniforme*. Se define acuerdo uniforme de la siguiente forma:

Acuerdo uniforme: si un proceso, correcto o con fallo, entrega un mensaje m , entonces todos los procesos correctos en el grupo(m) entregarán finalmente m .

El acuerdo uniforme permite que un proceso tras entregar un mensaje, y al mismo tiempo asegura que todos los procesos correctos entregarán el mensaje. Ya se discutió que el algoritmo de la Figura 11.10 satisface esta propiedad, que es más fuerte que la propiedad de acuerdo no uniforme definida con anterioridad.

El acuerdo uniforme es útil en aquellas aplicaciones donde un proceso puede realizar una acción que produzca una inconsistencia observable justo antes de caerse. Por ejemplo, considérese el caso en el que los procesos son servidores que gestionan las copias de una cuenta bancaria y que las actualizaciones en la cuenta se mandan al grupo de servidores utilizando multidifusión fiable. Si la multidifusión no satisface la propiedad de acuerdo uniforme, un cliente que acceda a un servidor justo antes de que se caiga puede observar una actualización que ningún otro servidor procesará.

Es importante resaltar que si se invierte el orden de las líneas *F-entrega* m y *si* ($q \neq p$) *entonces* *B-multicast*(g, m); *fin si* en la Figura 11.10, entonces el algoritmo resultante no cumple la propiedad de acuerdo uniforme.

De la misma forma que hay una versión uniforme de acuerdo, hay versiones uniformes del resto de propiedades de la multidifusión, incluyendo validez e integridad además de las propiedades de ordenación que se definen a continuación.

11.4.3. MULTIDIFUSIÓN ORDENADA

El algoritmo de multidifusión básico de la Sección 11.4.1 entrega los mensajes a los procesos en un orden arbitrario, debido a los retrasos arbitrarios asociados a las operaciones subyacentes de envío uno-a-uno. Esta falta de la propiedad de ordenación no es admisible en muchas aplicaciones. Por ejemplo, en una planta nuclear es importante que los eventos asociados a amenazas en las condiciones de seguridad y los eventos que implican acciones por parte de las unidades de control sean observados en el mismo orden por todos los procesos del sistema.

Los requisitos de ordenación más frecuentes son la ordenación total, la ordenación causal, la ordenación FIFO y las ordenaciones híbridas total-causal y total-FIFO. Para simplificar la exposición se definen a continuación estas ordenaciones bajo la suposición de que cualquier proceso pertenece a lo sumo a un grupo. Más adelante se discutirán las implicaciones de permitir que los grupos se solapen.

Ordenación FIFO: si un proceso correcto realiza un *multicast*(g, m) y a continuación un *multicast*(g, m'), entonces todo proceso correcto que entregue m' ha de haber entregado previamente m .

Ordenación causal: si $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, donde \rightarrow indica la relación sucedió-antes inducida por los mensajes enviados solamente entre los miembros de g , entonces cualquier proceso correcto que entregue m' habrá entregado antes m .

Ordenación total: si un proceso correcto entrega el mensaje m antes de que entregue m' , entonces cualquier otro proceso que entregue m' ha de haber entregado m antes.

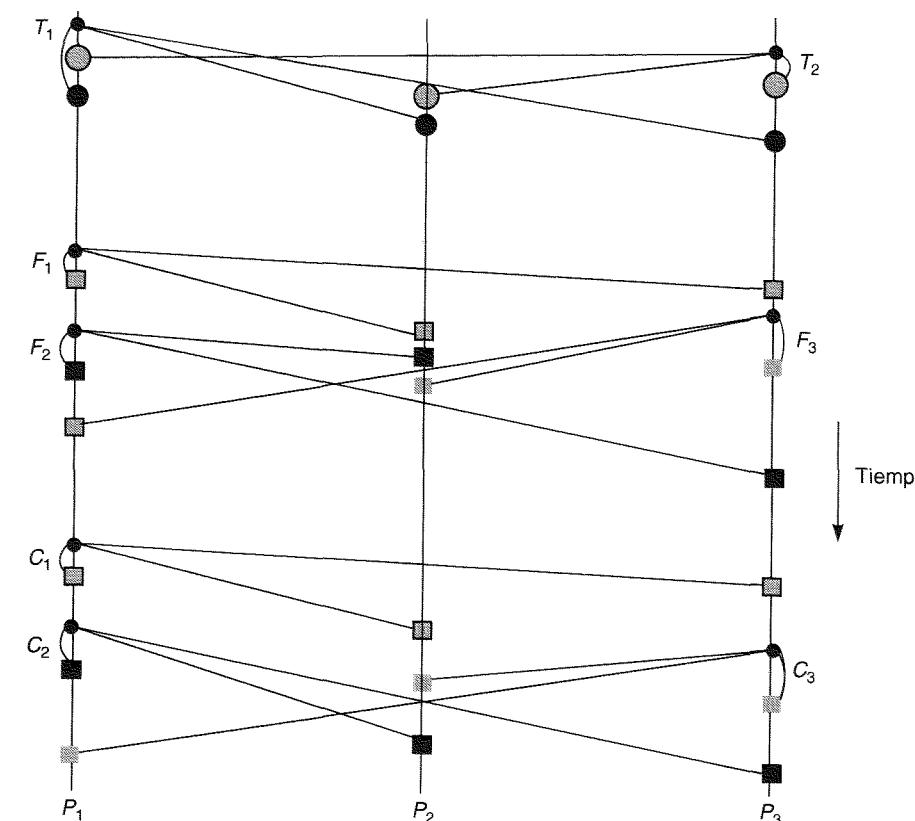
La ordenación causal implica una ordenación FIFO, ya que dos envíos *multicast* cualesquiera dentro del mismo proceso están relacionados en la forma sucedió-antes. Hay que resaltar que tanto la ordenación FIFO como la ordenación causal son ordenaciones parciales: de forma general, no todos los mensajes son enviados por el mismo proceso; de forma similar, algunas multidifusiones son concurrentes (y, por lo tanto, no ordenadas mediante sucedió-antes).

La Figura 11.12 ilustra las ordenaciones en el caso en que existan tres procesos. Un análisis en profundidad de la figura muestra que los mensajes totalmente ordenados se entregan en el orden contrario al tiempo físico en el que fueron enviados. De hecho, la definición de ordenación total permite que la entrega de mensajes se ordene de manera arbitraria, ya que el orden es el mismo en procesos diferentes. Además, dado que la ordenación total no es necesariamente ni una ordenación FIFO ni una ordenación causal, se define la ordenación híbrida *FIFO-total* como aquella en la que la entrega de mensajes obedece a ambos tipos de ordenación; de forma parecida, cuando existe una entrega de mensajes con ordenación *causal-total* se obedecen ambos tipos de ordenaciones.

Las definiciones de multidifusión ordenada ni suponen ni implican fiabilidad. Por ejemplo, el lector podría comprobar que bajo la hipótesis de ordenación total, si un proceso p entrega un mensaje m y a continuación entrega un mensaje m' , un proceso correcto q puede entregar m sin necesidad de entregar m' o cualquier otro mensaje ordenado tras m .

También se pueden formar protocolos híbridos ordenados y fiables. Una multidifusión totalmente ordenada fiable suele denominarse en la literatura como un *multidifusión atómica*. De forma similar se pueden conseguir multidifusión FIFO fiable, multidifusión causal fiable y versiones fiables de los multidifusión con ordenación híbrida.

La ordenación en la entrega de mensajes por multidifusión, tal y como se verá, puede ser costosa en términos de latencia en la entrega y consumo de ancho de banda. La semántica asociada a las



Obsérvese la ordenación consistente de los mensajes con ordenación total T_1 y T_2 , los mensajes relacionados por criterio FIFO, $F_1 > F_2$, y los mensajes relacionados causalmente C_1 y C_2 ; y en el resto de casos, entrega de mensajes con ordenación.

Figura 11.12. Ordenaciones total, FIFO y causal de mensajes de multidifusión.

ordenaciones que se han descrito puede retrasar la entrega de mensajes de forma innecesaria. Esto es, desde el punto de vista de la aplicación, un mensaje puede verse retrasado por otro del cual no depende. Por esta razón, algunos autores han propuesto sistemas de multidifusión que utilizan semánticas de los mensajes específicos de la aplicación para determinar el orden en la entrega de mensajes [Cheriton y Skeen 1993, Pedone y Schiper 1999].

◇ **El ejemplo del tablón de anuncios.** Para concretar más la semántica de la entrega de mensajes considérese una aplicación en la cual los usuarios ponen mensajes en tablones de anuncios. Cada usuario ejecuta un proceso que es una aplicación tablón de anuncios. Cada tema de discusión tiene su propio grupo de procesos. Cuando un usuario manda un mensaje al tablón, la aplicación multidifunde dicho mensaje al resto del grupo del tema en el que él o ella están interesados, de tal forma que el usuario sólo recibirá los anuncios concernientes a ese tema.

Se necesita multidifusión fiable si todo usuario ha de recibir al final cada anuncio. Los usuarios también imponen requisitos de ordenación. La Figura 11.13 muestra los anuncios tal y como le aparecen a un usuario en particular. Como mínimo, se desea tener una ordenación FIFO, de tal forma que cualquier anuncio para un usuario (por ejemplo, A. Pérez) se recibirá en el mismo orden y los usuarios pueden hablar de forma consistente acerca del segundo mensaje de A. Pérez.

El lector debe fijarse en que los mensajes cuyos asuntos son *Re: Microkernels* (25) y *Re: Mach* (27) aparecen tras los mensajes a los que se refieren. Por lo tanto, se necesita una multidifusión

| Tablón de anuncios: os.interesante | | |
|------------------------------------|---------------|------------------|
| Ítem | De | Asunto |
| 23 | A. Pérez | Mach |
| 24 | G. Mayor | Microkernels |
| 25 | A. Pérez | Re: Microkernels |
| 26 | T. L. Heureux | RPC performance |
| 27 | M. Walker | Re: Mach |
| Fin | | |

Figura 11.13. Pantalla de un programa tablón de anuncios.

ordenada causalmente para garantizar estas relaciones. De otra forma, retrasos arbitrarios en los mensajes pueden ocasionar que un mensaje *Re: Mach* aparezca antes que el mensaje original concerniente a Mach.

Si la multidifusión estuviese totalmente ordenada, entonces la numeración en la columna de la izquierda sería consistente entre los usuarios. En ese caso, los usuarios podrían referirse sin ambigüedad al mensaje 24.

En la práctica, el sistema del tablón de anuncios de USENET no implementa ni ordenación causal ni total. Los costes asociados a la comunicación para conseguir estas ordenaciones a gran escala exceden ampliamente las ventajas que aportan.

◇ **Implementación de una ordenación FIFO.** Para conseguir una multidifusión con ordenación FIFO (que tendrá las operaciones *OF-multicast* y *OF-entrega*) se utilizan secuencias de números, de forma similar a como se conseguiría en una comunicación uno-a-uno. Se considerarán sólo grupos que no se solapen. Se deja como ejercicio para el lector que verifique que el protocolo de multidifusión fiable que se definió sobre la base de la multidifusión IP en la Sección 11.4.2, además garantiza la ordenación FIFO. Ahora se mostrará cómo construir multidifusión con ordenación FIFO sobre la base de cualquier multidifusión. Se usarán las variables S_g^p y R_g^q asociadas al proceso p como se hacía en el protocolo de multidifusión fiable de la Sección 11.4.2: S_g^p es un contador del número de mensajes que p ha enviado al grupo g , y para cada q , R_g^q es el número de secuencia del último mensaje que p ha entregado, partiendo del proceso q y que fue enviado al grupo g .

Cuando p quiere enviar un mensaje mediante *OF-multicast* al grupo g , adhiere en el mensaje el valor S_g^p , realiza el *B-multicast* del mensaje al grupo g e incrementa S_g^p en 1. Cuando p recibe un mensaje de q que lleva el número de secuencia S , comprueba si $S = R_g^q + 1$. Si es así, ése era el mensaje esperado por parte de q y p usa *OF-entrega*, fijando $R_g^q := S$. Si $S > R_g^q + 1$, retiene el mensaje en la cola hasta que los mensajes intermedios hayan sido entregados y $S = R_g^q + 1$.

Dado que todos los mensajes enviados por un emisor se entregan siguiendo la misma secuencia, y dado que se pospone la entrega de un mensaje hasta que se alcance su número de secuencia, se cumple claramente la condición de ordenación FIFO. Sin embargo, esto sólo ocurre si se cumple la condición de grupos que no se solapan.

En este protocolo podría usarse cualquier implementación de *B-multicast*. Además, si se usase una primitiva *F-multicast* en lugar de *B-multicast*, se obtendría una multidifusión FIFO fiable.

◇ **Implementación de la ordenación total.** La manera básica de implementar la ordenación total es asignar identificadores totalmente ordenados a los mensajes que se multidifunden de tal forma que todo proceso realice la misma ordenación basada en esos identificadores. El algoritmo de entrega sería muy similar al descrito para la ordenación FIFO, la diferencia sería que los procesos almacenarían números de secuencia específicos para cada grupo en lugar de números específicos de secuencia para cada proceso. Aquí se considerará sólo cómo ordenar totalmente mensajes

1. Algoritmo para el miembro p del grupo

En la inicialización: $r_g := 0$;

Para hacer OT-multicast de un mensaje m al grupo g
Hacer $B\text{-multicast}(g \cup \{\text{secuenciador}(g)\}, \langle m, i \rangle)$;

En $B\text{-entrega}(\langle m, i \rangle)$ con $g = \text{grupo}(m)$
Coloca $\langle m, i \rangle$ en la cola de retención;

En $B\text{-entrega}(m_{\text{orden}} < "orden", i, S)$ con $g = \text{grupo}(m)$
Espera hasta que $\langle m, i \rangle$ esté en la cola de retención y $S = r_g + 1$;
 $OT\text{-entrega } m$; // (tras eliminarlo de la cola de retención)
 $r_g := S$;

2. Algoritmo para el secuenciador de g

En la inicialización: $s_g := 0$;

En $B\text{-entrega}(\langle m, i \rangle)$ con $g = \text{grupo}(m)$
Haz $B\text{-multicast}(g, \langle "orden", i, s_g \rangle)$;
 $s_g := s_g + 1$;

Figura 11.14. Ordenación total que usa un secuenciador.

que se envían a grupos que no se solapan. Las operaciones de multidifusión se denominarán *OT-multicast* y *OT-entrega*.

Se discuten fundamentalmente dos métodos para asignar los identificadores a los mensajes. El primero consiste en que un proceso denominado *secuenciador* los asigne (véase la Figura 11.14). Cuando un proceso desea enviar un mensaje m con *OT-multicast* a un grupo g adjunta un identificador único $id(m)$ al mismo. Los mensajes destinados a g se envían al secuenciador para g , *secuenciador*(g), al igual que a los miembros de g . (El secuenciador puede ser, incluso, un miembro de g .) El proceso *secuenciador*(g) guarda un número de secuencia específico para el grupo S_g , que utiliza para asignar números de forma creciente y consecutiva a los mensajes que *B-entrega*. El secuenciador anuncia la secuencia de números mediante *B-multicast* al grupo g de mensajes con el *ordenamiento* (para obtener más detalles véase la Figura 11.14).

Un mensaje permanecerá retenido en la cola indefinidamente hasta que pueda efectuarse *OT-entrega* de acuerdo con el correspondiente número de secuencia. Partiendo de que la secuencia de números está bien definida (ya que viene dada por el secuenciador), se observa que se cumple el criterio de orden total. Además, si los procesos utilizasen una variante de ordenación FIFO del *B-multicast*, entonces además de estar totalmente ordenado está también causalmente ordenado. Esta demostración se deja para el lector.

El problema obvio asociado a un esquema basado en un secuenciador es que éste puede convertirse en un cuello de botella y además es un punto de fallo crítico. Existen algoritmos que en la práctica solucionan el problema del fallo. Chang y Maxemchuk [1984] fueron los primeros en sugerir un protocolo de multidifusión que utilizaba un secuenciador (lo que denominaron *lugar del testigo*). Kaashoek y otros [1989] desarrollaron un protocolo basado en un secuenciador para el sistema Amoeba. Estos protocolos aseguran que un mensaje está en la cola de retención en $f + 1$ nodos antes de ser entregado; de esta forma se pueden tolerar f fallos. Al igual que Chang y Maxemchuk, Birman y otros [1991] también utilizan un sitio que retiene el testigo y que actúa como secuenciador. El testigo se puede pasar de unos procesos a otros, de tal forma que sólo un proceso multidifunde todos los mensajes de forma totalmente ordenada dicho proceso puede actuar como un secuenciador, ahorrando el proceso de comunicación.

El protocolo propuesto por Kaashoek y otros usa multidifusión implementada vía hardware (que está disponible, por ejemplo, en Ethernet) en lugar de usar una comunicación fiable punto-a-

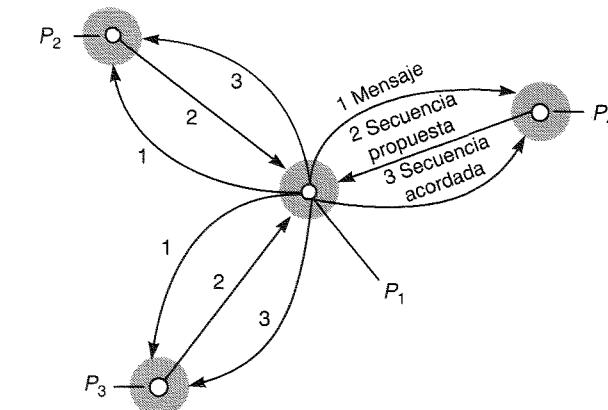


Figura 11.15. El algoritmo ISIS para ordenación total.

punto. En la versión más simple de su protocolo los procesos envían sus mensajes al secuenciador para que realice la multidifusión, y esta comunicación la realizan uno-a-uno. El secuenciador multidifunde junto con el identificador y el número de secuencia. Esto presenta la ventaja de que los otros miembros del grupo reciben sólo un mensaje mediante multidifusión; el inconveniente radica en el incremento en el uso del ancho de banda. El protocolo se encuentra descrito en su totalidad en www.cdk3.net/coordination.

El segundo método examinado para conseguir multidifusión totalmente ordenada es aquel en el que los procesos se ponen de acuerdo de una forma distribuida en la asignación de números de secuencia. En la Figura 11.15 se muestra un algoritmo simple, similar al que fue desarrollado originalmente para implementar la entrega en la multidifusión totalmente ordenada para la herramienta ISIS [Birman y Joseph 1987a]. Una vez más, un proceso envía un mensaje mediante *B-multicast* a los miembros del grupo. Éste puede ser abierto o cerrado. Los procesos que reciben los mensajes proponen números de secuencia según les van llegando y se los devuelven al emisor, que los usa para generar números de secuencia *acordados*.

Cada proceso q en el grupo g mantiene A_g^q , que es el mayor número de secuencia acordado que se ha observado hasta el momento para el grupo g y P_g^q , que es su mayor número de secuencia propuesto. El algoritmo que sigue cada proceso p para multidifundir un mensaje m al grupo g es el siguiente:

1. p usa *B-multicast* $\langle m, i \rangle$ a g , donde i es el identificador único para m .
2. Cada proceso q responde al emisor p con una propuesta del número de secuencia acordado como $P_g^q := \text{Máx}(A_g^q, P_g^q) + 1$. En realidad, habría que incluir los identificadores de procesos en los valores propuestos P_g^q para asegurar el orden total, ya que de otra forma, distintos procesos podrían proponer el mismo valor para mensajes distintos; sin embargo, para facilitar la notación, no se mencionará de forma explícita. Cada proceso asigna al mensaje de forma provisional el número de secuencia propuesto y lo almacena en la cola de retención, que se ordena con el número de secuencia *menor* al comienzo.
3. p recoge todos los números de secuencia propuestos y selecciona el mayor a como el siguiente número de secuencia acordado. A continuación emplea *B-multicast* $\langle i, a \rangle$ para g . Cada proceso q en g fija $A_g^q := \text{Máx}(A_g^q, a)$ y adjunta a al mensaje (que es identificado mediante i). Además reordena el mensaje en la cola de retención si el número de secuencia acordado difiere del propuesto. Cuando al mensaje al principio de la cola de retención se le haya asignado su número de secuencia acordado, se transfiere al final de la cola de entre-

ga. Por el contrario, no se transfieren aún aquellos mensajes a los que ya se les ha asignado su número de secuencia acordado, pero no están al comienzo de la cola de retención.

Si todos los procesos acuerdan el mismo conjunto de números de secuencia y los entregan en el orden correspondiente, se cumple el orden total. Está claro que los procesos correctos al final acuerdan el mismo conjunto de número de secuencia, pero hay que mostrar que éstos se incrementan de forma monotónica y que ningún proceso correcto puede anticiparse al entregar un mensaje.

Supongamos que a un mensaje m_1 se le ha asignado un número de secuencia acordado y ha alcanzado el inicio de la cola de retención. Por la forma en que se construye, un mensaje recibido tras esta etapa tiene que ser y es entregado después de m_1 ; tendrá un número de secuencia propuesto mayor, y por lo tanto un número de secuencia acordado mayor que m_1 . De esta forma, sea m_2 cualquier otro mensaje al que no se le haya asignado su número de secuencia acordado, pero que esté en la misma cola. Se tiene que:

$$\text{secuenciaAcordada}(m_2) \geq \text{secuenciaPropuesta}(m_2)$$

en función del algoritmo utilizado. Y dado que m_1 está al inicio de la cola:

$$\text{secuenciaPropuesta}(m_2) > \text{secuenciaAcordada}(m_1)$$

En consecuencia,

$$\text{secuenciaAcordada}(m_2) > \text{secuenciaAcordada}(m_1)$$

y se asegura la ordenación total.

Este algoritmo tiene mayor tiempo de latencia que el *multicast* basado en secuenciador: se mandan tres mensajes en serie entre el emisor y el grupo antes de que se entregue el mensaje.

Hay que resaltar que la ordenación total elegida por este algoritmo no garantiza ordenación causal ni FIFO: dos mensajes cualesquiera se entregan en un orden total arbitrario, influido por los retrasos en las comunicaciones.

Melliar-Smith y otros [1990], García-Molina y Spauster [1991] y Hadzilacos y Toueg [1994] presentan otras aproximaciones para implementar una ordenación total.

◊ **Implementación de la ordenación causal.** La Figura 11.16 ofrece un algoritmo para grupos cerrados que no se solapan, que está basado en el propuesto por Birman y otros [1991], y en el que se introducen las operaciones de multidifusión ordenadas causalmente: *OC-multicast* y *OC-entrega*. El algoritmo tiene en cuenta la relación sucedió-antes tal y como la establecen los mensajes multidifusión. Si los procesos intercambian mensajes uno-a-uno, estos mensajes no se tienen en cuenta.

Algoritmo para un miembro del grupo p_i ($i = 1, 2, \dots, N$)

En la inicialización

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

Para hacer OC-multicast de un mensaje m al grupo g

$$V_i^g[i] := V_i^g[i] + 1;$$

Hacer *B-multicast*($g, < V_i^g, m >$);

En B-entrega($< V_i^g, m >$) que viene de p_j , con $g = \text{grupo}(m)$

Colocar $< V_i^g, m >$ en la cola de retención;

Esperar hasta que $V_i^g[j] = V_i^g[i] + 1$ y $V_i^g[k] \leq V_i^g[l]$ ($k \neq j$);

OC-entregar m ; // tras eliminarlo de la cola de retención

$$V_i^g[j] := V_i^g[j] + 1;$$

Figura 11.16. Ordenación causal que usa vectores con marcas temporales.

Cada proceso p_i ($i = 1, 2, \dots, N$) mantiene su propio vector de marcas temporales (véase la Sección 10.4). Cada entrada asociada a la marca temporal cuenta el número de mensajes multidifundidos de cada proceso que sucedieron-antes que el próximo mensaje que va a ser enviado mediante multidifusión.

Para enviar un mensaje mediante *OC-multicast* al grupo g , el proceso suma 1 a su entrada en el vector de marcas temporales y envía el mensaje mediante *B-multicast* a g junto con su marca de tiempo.

Cuando un proceso p_i *B-entrega* un mensaje p_j , tiene que colocarlo en la cola de retención antes de realizar la *OC-entrega*: primero tiene que asegurarse de que ha entregado todos los mensajes que le precedían causalmente. Para asegurarse de ello, p_i espera hasta que a) ha entregado todo mensaje enviado anteriormente por parte de p_j y b) ha entregado cualquier mensaje que p_j hubiese entregado hasta el instante de tiempo en el que multidifundió el mensaje. Ambas condiciones pueden detectarse examinando el vector de marcas temporales, como se muestra en la Figura 11.16. Hay que tener en cuenta de que un proceso puede *OC-entregarse* inmediatamente hacia sí mismo un mensaje que él envíe con *OC-multicast*, aunque esto no se describa en la Figura 11.16.

Cada proceso actualiza su vector de marcas temporales cada vez que entrega un mensaje, para llevar la cuenta de los mensajes que le preceden causalmente. Esto lo consigue incrementando en 1 la entrada j -ésima en su marca de tiempo. Esto supone una optimización de la operación de fusión que aparece en las reglas de actualización de la Sección 10.4. Esta optimización se consigue gracias a la condición de entrega en el algoritmo de la Figura 11.16, que garantiza que sólo se incrementa la entrada j -ésima.

Sin entrar en detalles, la prueba de corrección del algoritmo sería la siguiente. Supóngase que *multicast*(g, m) \rightarrow *multicast*(g, m'). Sean V y V' los vectores de marcastemporales de m y m' , respectivamente. Mediante inducción se prueba directamente que $V < V'$. En concreto, si un proceso p_k envía m mediante *multicast*, entonces $V[k] \leq V'[k]$.

Considérese qué ocurre cuando algún proceso correcto p_i realiza *B-entrega* de m' (en lugar de *OC-entregarlo*) sin haber *OC-entregado* m . Según el algoritmo, $V_i[k]$ puede incrementarse sólo cuando p_i entrega un mensaje que viene de p_k y lo incrementa en uno. Pero p_i no ha recibido m y por lo tanto $V_i[k]$ no puede aumentar más allá de $V_i[k] - 1$. En consecuencia, no es posible, para p_i realizar la *OC-entrega* de m' ya que requeriría que $V_i[k] \geq V'[k]$, y por consiguiente que $V_i[k] \geq V[k]$.

El lector debería comprobar que si se sustituye la primitiva fiable *F-multicast* en lugar de *B-multicast*, se obtendrá una multidifusión a la vez fiable y con ordenación causal.

Además, si se combina el protocolo para multidifusión causal con el protocolo basado en secuenciador para una entrega totalmente ordenada se obtiene una entrega de mensajes con ordenación total y causal. El secuenciador entrega los mensajes según el orden causal y multidifunde la secuencia de números para los mensajes según el orden en el que los recibió. Los procesos en el grupo de destino no entregan el mensaje hasta que reciben otro mensaje por parte del secuenciador con el *orden* y dicho mensaje sea el próximo en la secuencia de entrega.

Dado que el secuenciador entrega los mensajes según un orden causal y dado que el resto de procesos entregan los mensajes en el mismo orden que el secuenciador, el orden es, por lo tanto, total y causal.

◊ **Grupos que se solapan.** En las definiciones y en la semántica de los algoritmos de ordenación FIFO, causal y total, se han considerado sólo grupos que no se solapaban. Esto simplifica el problema, pero no es satisfactorio, ya que en general los procesos necesitan pertenecer a múltiples grupos que se solapan. Por ejemplo, un proceso puede estar interesado en eventos de múltiples fuentes y, consiguientemente, puede unirse al correspondiente conjunto de grupos de distribución de eventos.

Se pueden ampliar las definiciones de ordenación para cubrir ordenaciones globales [Hadzilacos y Toueg 1994], en las cuales ha de considerarse que si el mensaje m se multidifunde a g , y si el

mensaje m' se multidifunde a g' , entonces ambos mensajes han de enviarse a los miembros de $g \cap g'$.

Ordenación FIFO global: si un proceso correcto realiza $\text{multicast}(g, m)$, y a continuación realiza $\text{multicast}(g', m')$, entonces todo proceso correcto en $g \cap g'$ que entregue m' debe entregar m antes que m' .

Ordenación causal global: si $\text{multicast}(g, m) \rightarrow \text{multicast}(g', m')$, donde \rightarrow es la relación sucedió-antes inducida por cualquier encadenamiento de mensajes *multicast*, entonces cualquier proceso correcto en $g \cap g'$ que entregue m' debe entregar m antes que m' .

Ordenación total por parejas. si un proceso correcto entrega un mensaje m enviado a g antes de entregar m' que fue enviado a g' , entonces cualquier proceso correcto en $g \cap g'$ que entregue m' ha de entregar m antes que m' .

Ordenación total global: sea « $<$ » la relación de ordenación entre eventos asociados a entregas. Se exige que « $<$ » cumpla la ordenación total por parejas y que sea acíclica, ya que bajo la ordenación total por parejas « $<$ » no es acíclica por defecto.

Una forma de llevar a cabo estas ordenaciones sería multidifundir cada mensaje m al conjunto de *todos* los procesos en el sistema. Cada proceso debería rechazar o entregar el mensaje según pertenezca o no al *grupo(m)*. Esta implementación sería ineficiente e insatisfactoria, ya que la multidifusión ha de involucrar al menor número de procesos posibles fuera del grupo de destino. Se han estudiado distintas alternativas en Birman y otros [1991], García-Molina y Spauster [1991], Hadzilacos y Toueg [1994], Kindberg [1995] y Rodrigues y otros [1998].

◊ **Multicast en sistemas síncronos y asíncronos.** En esta sección, hasta el momento, se han descrito algoritmos para conseguir multidifusión fiable y no ordenada, multidifusión fiable con ordenación FIFO, multidifusión fiable con ordenación causal y multidifusión con ordenación total. También se ha indicado cómo conseguir una multidifusión ordenada total y causalmente. Se deja al lector la obtención de un algoritmo para una primitiva *multicast* que garantice las ordenaciones FIFO y total. Todos los algoritmos que se han descrito funcionan correctamente en sistemas asíncronos.

Sin embargo, no se ha descrito un algoritmo que garantice que sea fiable y tenga entrega con orden total. Aunque parezca sorprendente y siendo posible en un sistema *síncrono*, es *imposible* obtener un protocolo con estas garantías en un sistema distribuido asíncrono (incluso en uno que en el peor de los casos sufra una sola caída de un proceso). Se volverá a tratar este tema en la siguiente sección.

11.5. CONSENSO Y SUS PROBLEMAS RELACIONADOS

Esta sección presenta el problema del consenso [Pease y otros 1980, Lamport y otros 1982] y sus problemas asociados: los generales bizantinos y la consistencia interactiva. Estos problemas se denominarán de forma conjunta como *problemas de acuerdo*. Resumiendo, los procesos tienen problemas para ponerse de acuerdo en un valor después de que uno o más de dichos procesos haya propuesto cuál debería ser ese valor.

Por ejemplo, en el Capítulo 2 se describió la situación en la cual dos ejércitos debían decidir, de forma consistente, si atacar o retirarse. De forma parecida, se puede necesitar que todos los computadores correctos que controlan los motores de una nave espacial decidan «proseguir» o que todos decidan «abortar la operación», una vez se haya propuesto una u otra acción. Igualmente, en una transacción para realizar una transferencia de fondos de una cuenta a otra, los computadores involucrados deben acordar de forma consistente cómo actualizar los respectivos crédito y débito. En el caso de la exclusión mutua, los procesos acuerdan cuáles pueden entrar en la sección crítica.

En el caso de una elección, los procesos acuerdan cuál es el proceso elegido. En la multidifusión totalmente ordenada, los procesos acuerdan el orden de entrega de los mensajes.

Existen protocolos que solucionan de forma específica estos tipos de acuerdo. Algunos se han descrito previamente en este capítulo, y los Capítulos 12 y 13 examinarán las transacciones. No obstante, es útil considerar formas de acuerdo más genéricas buscando características y soluciones comunes.

Esta sección define el consenso de forma más precisa y lo relaciona con tres problemas asociados al acuerdo: los generales bizantinos, la consistencia interactiva y la multidifusión con ordenación total. Se procede examinando bajo qué circunstancias se pueden resolver los problemas, y se perfilan algunas soluciones. En particular, se discutirán los conocidos resultados sobre imposibilidad debidos a Fischer y otros [1985], que establecen que en un sistema asíncrono una colección de procesos que contenga un proceso que ha fallado no se puede garantizar que se alcance el consenso. Finalmente, se considerará la existencia de algoritmos que funcionan en la práctica a pesar del resultado de imposibilidad.

11.5.1. DEFINICIÓN DEL MODELO DEL SISTEMA Y DEL PROBLEMA

El modelo del sistema incluye una colección de procesos p_i ($i = 1, 2, \dots, N$) que se comunican mediante el paso de mensajes. En muchos casos prácticos un requisito importante a considerar es que ha de alcanzarse un consenso incluso cuando hay fallos. Como antes, se supone que la comunicación es fiable, pero los procesos pueden fallar. En esta sección se considerarán fallos por caída así como procesos que fallan de formas (arbitrariamente) extrañas. Algunas veces se especificará la suposición de que pueden fallar hasta f de los N procesos, esto es, éstos exhiben algún tipo de fallo especificado, mientras que el resto permanece correcto.

Si pueden ocurrir fallos arbitrarios, ha de considerarse otro factor en la especificación del sistema y es si los procesos firman digitalmente los mensajes que envían (véase la Sección 7.4). Si los procesos lo hacen, entonces se puede limitar el daño que puede ocasionar un proceso que ha fallado. Concretamente, durante la ejecución de un algoritmo de acuerdo dicho proceso no puede hacer afirmaciones falsas sobre los valores que le ha enviado un proceso correcto. De todas formas, la importancia de la firma de mensajes quedará patente cuando se discutan soluciones al problema de los generales bizantinos. Por defecto, se asumirá que no existe dicha firma.

◊ **Definición del problema del consenso.** Para alcanzar el consenso, cada proceso p_i comienza en el estado *no decidido* y *propone* un solo valor v_i de un conjunto de posibles valores D ($i = 1, 2, \dots, N$). Los procesos se comunican entre sí, intercambiando valores. Entonces, cada proceso fija el valor de una *variable de decisión* d_i . Al hacer esto pasa al estado *decidido*, en el cual ya no puede cambiar el valor de d_i ($i = 1, 2, \dots, N$). La Figura 11.17 muestra a tres procesos envueltos en un algoritmo de consenso. Dos procesos proponen «proseguir» y el tercero propone «abortar», pero se cae. Los dos procesos que permanecen correctos deciden «proseguir».

Los requisitos para un algoritmo de consenso son que han de cumplirse las siguientes condiciones cada vez que se ejecute:

Terminación: finalmente, cada proceso correcto ha de fijar su variable de decisión.

Acuerdo: el valor de decisión de todos los procesos correctos es el mismo: si p_i y p_j son correctos y están en el estado *decidido*, entonces $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integridad: si todos los procesos correctos han propuesto el mismo valor, entonces cualquier proceso correcto en el estado *decidido* ha elegido dicho valor.

Pueden aceptarse distintas variaciones en la definición de integridad, en función de la aplicación. Por ejemplo, un tipo de integridad más débil requeriría que el valor de decisión sea igual al pro-

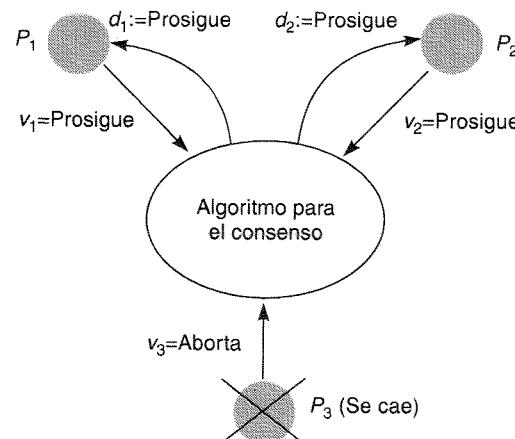


Figura 11.17. El problema del consenso para tres procesos.

puesto por algún proceso correcto, aunque no necesariamente igual para todos. De todas formas, se usará la definición dada anteriormente.

Para facilitar la comprensión de cómo obtener un algoritmo a partir de la formulación del problema, considérese un sistema en el que los procesos no pueden fallar. En ese caso es inmediato resolver el consenso. Por ejemplo, pueden reunirse todos los procesos en un grupo y que cada proceso del grupo multidifunda de modo fiable su valor propuesto al resto de los miembros del grupo. Todos los procesos esperan hasta recibir los N valores propuestos, incluido el suyo. Entonces evalúa la función *mayoría* (v_1, v_2, \dots, v_N), que devuelve el valor más frecuente entre sus argumentos, o el valor especial $\perp \notin D$ si no hay mayoría. Se garantiza la terminación gracias a la multidifusión fiable. Se garantizan el acuerdo y la integridad por la definición de *mayoría* y por la propiedad de integridad de la multidifusión fiable. Cada proceso recibe el mismo conjunto de valores propuestos, y cada proceso evalúa la misma función sobre esos valores. En consecuencia, han de ponerse de acuerdo y si cada proceso propone el mismo valor, entonces todos decidirán ese valor.

Hay que matizar que *mayoría* es sólo una de las posibles funciones a utilizar por los procesos para ponerse de acuerdo en los valores candidatos. Por ejemplo, si los valores están ordenados, las funciones *máximo* y *mínimo* también serían apropiadas.

Si los procesos pueden malograrse se introduce la complicación de tener que detectar los fallos y no queda claro que una ejecución del algoritmo de consenso termine. De hecho, si el sistema es asíncrono puede no hacerlo; se volverá a este punto en breve.

Si los procesos pueden fallar de formas (extrañamente) *arbitrarias*, aquellos que fallen pueden, en principio, comunicar valores aleatorios al resto. Esto puede parecer improbable en la práctica, pero no hay que descartar la posibilidad de que un proceso con un error falle de esa forma. Además, el fallo puede no ser accidental, sino el resultado de operaciones maliciosas o malévolas. Alguien podría deliberadamente forzar a un proceso a enviar diferentes valores a distintos procesos parejos con el fin de frustrar sus operaciones cuando están intentando alcanzar un consenso. En caso de que exista inconsistencia, los procesos correctos deben comparar lo que han recibido con lo que otros procesos dicen haber recibido.

◇ **El problema de los generales bizantinos.** En la definición informal del *problema de los generales bizantinos* [Lamport y otros 1982], tres o más generales han de ponerse de acuerdo en si atacar o retirarse. Uno de ellos, el comandante, cursa la orden. Los otros, los tenientes del comandante, deben decidir si atacar o retirarse. Pero uno o más de los generales puede ser un «traidor», o lo que es lo mismo, pueden fallar. Si el comandante es el traidor, manda atacar a un general y

retirarse a otro. Si el traidor es un teniente, informa a uno de sus iguales que el comandante le mandó atacar mientras que a otro le dice que le ordenó retirarse.

El problema de los generales bizantinos se distingue del de consenso en que un proceso destacado proporciona un valor en el que los otros han de ponerse de acuerdo, en vez de que cada uno proponga un valor. Los requisitos son:

Terminación: al final, cada proceso correcto ha de fijar su variable de decisión.

Acuerdo: el valor de decisión de todos los procesos correctos es el mismo; si p_i y p_j son correctos y han entrado en el estado *decidido*, entonces $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integridad: si el comandante es un proceso correcto, entonces todos los procesos correctos han de decidir el valor propuesto por el comandante.

Hay que fijarse que en el problema de los generales bizantinos la integridad implica acuerdo cuando el comandante es correcto: pero el comandante no tiene por qué estarlo.

◇ **Consistencia interactiva.** Este problema es otra variante del consenso, en el cual todo proceso propone un solo valor. El objetivo del algoritmo es que los procesos correctos se pongan de acuerdo en un *vector* de valores, uno para cada proceso. A éste se le llamará el *vector de decisión*. Por ejemplo, el objetivo para cada conjunto de procesos podría ser obtener la misma información acerca de sus estados respectivos.

Los requisitos para la consistencia interactiva son:

Terminación: al final, cada proceso correcto fija su variable de decisión.

Acuerdo: el vector de decisión de todos los procesos correctos es el mismo.

Integridad: si p_i es correcto, entonces todos los procesos correctos deciden que v_i es la i -ésima componente de su vector.

◇ **Relación del consenso con otros problemas.** Aunque lo más habitual es considerar el problema de los generales bizantinos con fallos arbitrarios en los procesos, el hecho es que cada uno de los tres problemas, el consenso, los generales bizantinos y la consistencia interactiva, tiene cabida en el contexto de los fallos arbitrarios o por caída. De forma similar, cada uno de ellos puede enmarcarse bien en sistemas síncronos o asíncronos.

Algunas veces es posible hallar una solución para un problema utilizando la solución del otro. Esta propiedad es muy útil, por un lado porque aumenta nuestra comprensión de los problemas y, por otro lado, porque reutilizando soluciones potencialmente podemos ahorrar en esfuerzo computacional y en complejidad.

Supongamos que existe una solución para el consenso (C), para los generales bizantinos (GB) y para la consistencia interactiva (CI) como sigue:

$C_i(v_1, v_2, \dots, v_N)$ devuelve el valor de decisión p_i en su camino a la solución del problema del consenso, donde v_1, v_2, \dots, v_N son los valores propuestos por los procesos.

$GB_i(j, v)$ devuelve el valor de decisión p_i en su camino a la solución del problema de los generales bizantinos, donde p_j , el comandante, propone el valor v .

$CI_i(v_1, v_2, \dots, v_N)$ devuelve el j -ésimo valor en el vector de decisión p_i en su camino a la solución del problema de consistencia interactiva, donde v_1, v_2, \dots, v_N son los valores propuestos por los procesos.

Las definiciones de C_i , GB_i y CI_i suponen que un proceso con fallo propone una única noción de valor, aunque haya propuesto diferentes valores a cada uno de los otros procesos. Esto es sólo un convenio: las soluciones no deben depender de esa noción de valor.

Es posible formular soluciones a partir de las soluciones de otros problemas. Se proponen tres ejemplos:

CI a partir de GB. Se crea una solución para CI a partir de GB ejecutando GB N veces, una por cada proceso p_i ($i = 1, 2, \dots, N$) que actúe como comandante:

$$CI_i(v_1, v_2, \dots, v_N)[j] = GB_i(j, v_j) \quad (i, j = 1, 2, \dots, N)$$

C a partir de CI. Creamos una solución para C a partir de CI ejecutando CI para producir un vector de valores en cada proceso; a continuación se aplica una función apropiada sobre los valores del vector para determinar un único valor:

$$C(v_1, \dots, v_N) = \text{mayoría}(CI_i(v_1, \dots, v_N)[1], \dots, CI_i(v_1, \dots, v_N)[N])$$

($i = 1, 2, \dots, N$), donde *mayoría* se corresponde con su definición anterior.

GB a partir de C. Se crea una solución para GB a partir de C de la siguiente forma:

- El comandante p_j envía su valor propuesto v a sí mismo y a cada uno de los procesos restantes.
- Todos los procesos ejecutan C con los valores v_1, v_2, \dots, v_N que han recibido (teniendo en cuenta que p_j puede haber fallado).
- Obtienen $GB_i(j, v) = C_i(v_1, v_2, \dots, v_N)$ ($i = 1, 2, \dots, N$).

Queda como ejercicio para el lector comprobar que las condiciones de terminación, acuerdo e integridad se cumplen en cada caso. Fischer [1983] muestra los tres problemas con más detalle.

Solucionar el consenso equivale a resolver la multidifusión fiable y con ordenación total: dada una solución para uno, se puede resolver el otro. La implementación del consenso con una operación de multidifusión fiable y con ordenación total *FOT-multicast* es inmediata. Se reúnen todos los procesos en un grupo g . Para conseguir el consenso cada proceso p_i realiza *FOT-multicast*(g, v_i). Entonces, cada proceso p_i elige $d_i = m_i$, donde m_i es el *primer* valor que p_i ha *FOT-entregado*. La propiedad de terminación se deduce de la fiabilidad de la multidifusión. Las propiedades de acuerdo e integridad se deducen de la fiabilidad y ordenación total de la multidifusión. Chandra y Toueg [1996] han demostrado cómo se puede conseguir multidifusión fiable y con ordenación total a partir del consenso.

11.5.2. CONSENSO EN UN SISTEMA SÍNCRONO

Esta sección describe un algoritmo que utiliza un protocolo de multidifusión básico para resolver el problema del consenso en un sistema asíncrono. El algoritmo supone que un máximo de f de N procesos pueden sufrir fallos por caída.

Para alcanzar el consenso, cada proceso correcto recoge valores propuestos por los otros procesos. El algoritmo realiza $f + 1$ vueltas, en cada una de las cuales los procesos correctos hacen *B-multicast* de los valores entre ellos mismos. Se supone que pueden fallar un máximo de f procesos. En el peor de los casos, las f caídas ocurrieron durante las vueltas, pero el algoritmo garantiza que al final de las mismas todos los procesos correctos que han sobrevivido están en disposición de llegar a un acuerdo.

El algoritmo, que se muestra en la Figura 11.18, se basa en el propuesto por Dolev y Strong [1983] y en su presentación por parte de Attiya y Welch [1998]. La variable $Valores_i^r$ almacena el conjunto de valores conocidos para el proceso p_i al comienzo de la ronda r . Cada proceso multidifunde el conjunto de valores que no ha enviado en rondas anteriores. A continuación recoge las entregas de mensajes multidifusión similares por parte de otros procesos y guarda los nuevos valores. Aunque no se muestra en la Figura 11.18, la duración de cada ronda está delimitada por un timeout basado en el tiempo máximo que necesita un proceso correcto para realizar la multidifusión. Tras $f + 1$ rondas, cada proceso ha elegido el valor mínimo que haya recibido y su valor de decisión.

Algoritmo para los procesos $p_i \in g$; el algoritmo lo realiza en $f + 1$ rondas

En la inicialización

$$Valores_i^0 := \{v_i\}; Valores_i^0 = \{\}$$

En la ronda r ($1 \leq r \leq f + 1$)

Hacer *B-multicast*($g, Valores_i^r - Valores_i^{r-1}$); // Enviar sólo valores que no hayan sido enviados

$$Valores_i^{r+1} := Valores_i^r;$$

mientras (se está en la ronda r)

{

 En *B-entrega*(V_j) por parte de p_j hacer

$$Valores_i^{r+1} := Valores_i^{r+1} \cup V_j;$$

}

Tras ($f + 1$) rondas

Asignar $d_i = \min(Valores_i^{f+1})$

Figura 11.18. El problema del consenso en un sistema síncrono.

La terminación es obvia ya que el sistema es síncrono. Para comprobar la corrección del algoritmo hay que demostrar que cada proceso llega al mismo conjunto de valores al final de la última ronda. El acuerdo y la integridad vendrán dados porque los procesos aplican la función *mínimo* a su conjunto de valores.

Supóngase, por el contrario, que dos procesos difieren en su conjunto final de valores. Sin pérdida de generalidad, algún proceso correcto p_i tiene un valor v que otro proceso correcto p_j ($i \neq j$) no tiene. La única explicación para que p_i tenga al final un valor propuesto v que p_j no tiene es que cualquier tercer proceso, por ejemplo p_k , que consiguió enviar v a p_i se cayó antes de que v pudiese ser entregada a p_j . A su vez, cualquier proceso que hubiese enviado v en la ronda anterior debería haber caído, para explicar por qué p_k tiene v en esa ronda pero p_j no lo tiene. Procediendo de esta forma hemos de suponer al menos una caída en cada una de las rondas precedentes. Sin embargo, se supuso que podían ocurrir a lo sumo f caídas y hubo $f + 1$ rondas. Hemos llegado a una contradicción.

Se concluye que *cualquier* algoritmo que quiera alcanzar el consenso a pesar de f fallos por caída requiere al menos $f + 1$ rondas de intercambio de mensajes con independencia de cómo está construido [Dolev y Strong 1983]. Este límite inferior también se cumple en el caso de fallos extraños [Fischer y Lynch 1980].

11.5.3. EL PROBLEMA DE LOS GENERALES BIZANTINOS EN UN SISTEMA SÍNCRONO

Se ha discutido previamente este problema en un sistema asíncrono. A diferencia del algoritmo para el consenso descrito en la sección anterior, se supone que los procesos pueden presentar fallos arbitrarios. Esto es, un proceso que ha fallado puede enviar un mensaje con cualquier valor en cualquier momento; y puede decidir no enviar ningún mensaje. Un máximo de f de N procesos pueden fallar. Los procesos correctos pueden determinar la ausencia de mensajes mediante un tiempo límite, pero no pueden concluir que el emisor se ha caído, ya que puede estar en silencio por un tiempo y después volver a enviar mensajes.

Se supone que los canales de comunicación entre pares de procesos son privados. Si un proceso pudiera examinar todos los mensajes enviados por otros procesos, podría detectar las inconsistencias enviadas a diferentes procesos por parte de un proceso con fallo. La suposición por defecto de fiabilidad en el canal implica que ningún proceso con fallo puede injectar mensajes en el canal de comunicación de dos procesos correctos.

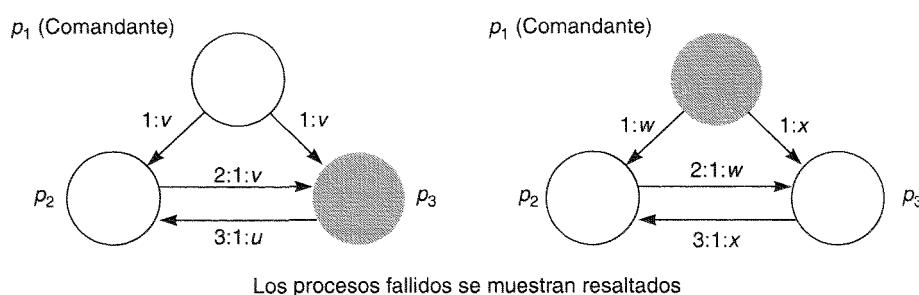


Figura 11.19. Tres generales bizantinos.

Lamport y otros [1982] consideraron el caso de tres procesos que enviaban mensajes no firmados entre ellos. Mostraron que no existe solución que garantice que se cumplan las condiciones del problema de los generales bizantinos si se permite que uno de los procesos falle. Además, generalizaron sus resultados para mostrar que no existe solución si $N \leq 3f$. Aquí se demostrarán estos resultados de forma resumida. Además, Lamport y otros obtuvieron un algoritmo que soluciona el problema de los generales bizantinos en un sistema sincrónico si $N \geq 3f + 1$ para el caso de mensajes sin firma (que ellos denominaron «orales»).

◇ **Imposibilidad con tres procesos.** La Figura 11.19 muestra dos escenarios en los cuales sólo uno de los procesos ha fallado. En la configuración de la izquierda, uno de los tenientes p_3 ha fallado; en la de la derecha, ha fallado el comandante p_1 . Cada escenario en la Figura 11.18 muestra dos rondas de mensajes: los valores que manda el comandante y los valores que los tenientes se mandan a continuación entre ellos. Los números permiten especificar el origen del mensaje e indicar las distintas rondas. Los «::» en los mensajes han de leerse como «dice que»; por ejemplo, «3:1:u» identifica el mensaje «3 dice que 1 dice que u».

En el escenario de la izquierda, el comandante envía correctamente el mismo valor v a cada uno de los otros procesos, y p_2 lo reproduce de forma correcta hacia p_3 . Sin embargo, p_3 envía un valor $u \neq v$ a p_2 . En ese momento, todo lo que p_2 conoce es que ha recibido distintos valores; no puede decir cuál fue enviado por el comandante.

En el escenario de la derecha, el comandante ha fallado y envía valores distintos a los tenientes. Una vez que p_3 ha reproducido el valor x que ha recibido, p_2 está en la misma situación que cuando p_3 falló; ha recibido dos valores distintos.

Si existe una solución, entonces p_2 está forzado a decidir el valor v , cuando el comandante es correcto, por la condición de integridad. Si se acepta que ningún algoritmo puede posiblemente distinguir entre los dos escenarios, p_2 debería elegir también el valor enviado por el comandante en el escenario de la derecha.

Si se sigue el mismo razonamiento para p_3 , suponiendo que es correcto, ha de concluirse, por simetría, que p_3 también elige el valor enviado por el comandante como su valor de decisión. Pero esto contradice la condición de acuerdo (ya que el comandante envía valores distintos si ha fallado). Por lo tanto, no hay solución posible.

Hay que fijarse en que este argumento se apoya en la intuición de que nada puede hacerse para mejorar el conocimiento de un general que está correcto más allá de la primera fase, donde no puede decidir qué proceso ha fallado. Es posible comprobar la corrección de esta intuición [Pease y otros 1980]. El acuerdo bizantino *puede* ser alcanzado por tres generales, con uno de ellos fallido, si los generales firman digitalmente los mensajes.

◇ **Imposibilidad con $N \leq 3f$.** Pease y otros generalizaron el resultado básico de imposibilidad para tres procesos, para probar que no hay solución posible si $N \leq 3f$. Brevemente, el argumento

puede resumirse de la siguiente forma. Supóngase que existe una solución para $N \leq 3f$. Utilícese la solución de tres procesos p_1, p_2 y p_3 para simular el comportamiento de n_1, n_2 y n_3 generales, respectivamente, donde $n_1 + n_2 + n_3 = N$ y $n_1, n_2, n_3 \leq N/3$. Se supone, además, que uno de los tres procesos falla. Aquéllos entre p_1, p_2 y p_3 que están correctos simulan generales correctos: ellos simulan las interacciones de sus propios generales de forma interna y envían mensajes de parte de sus generales a aquellos simulados por otros procesos. Los generales simulados por procesos fallidos se suponen con fallo: los mensajes que envían como parte de la simulación pueden ser espurios. Dado que $N \leq 3f$ y $n_1, n_2, n_3 \leq N/3$, a lo sumo f generales simulados tienen fallos.

Dado que el algoritmo que ejecuta los procesos se supone correcto, la simulación termina. Los generales correctamente simulados (en los procesos correctos) están de acuerdo y satisfacen la propiedad de integridad. Pero ahora es necesario encontrar el consenso para dos de un conjunto de tres procesos: cada uno decide según el valor elegido por sus generales simulados. Esto contradice el resultado de imposibilidad para tres procesos con uno de ellos fallido.

◇ **Solución de un proceso que falla.** Por problemas de espacio no se describe completamente el algoritmo de Pease y otros que resuelve los problemas de los generales bizantinos en un sistema sincrónico con $N \geq 3f + 1$. En su lugar, se muestra el funcionamiento del algoritmo para el caso $N \leq 4, f = 1$ y se da el ejemplo con $N = 4, f = 1$.

Los generales que están correctos alcanzan el acuerdo en dos rondas de mensajes:

- En la primera ronda, el comandante envía su valor a cada uno de sus tenientes.
- En la segunda ronda, cada uno de los tenientes envía el valor que ha recibido a sus iguales.

Un teniente recibe un valor de su comandante y $N - 2$ valores de sus iguales. Si el comandante ha fallado, entonces todos los tenientes están correctos y cada uno habrá recibido exactamente el conjunto de valores que les envió el comandante. En otro caso, uno de los tenientes ha fallado; cada uno de sus iguales que estaban correctos recibirá $N - 2$ copias del valor enviado por el comandante junto con un valor enviado por el teniente que ha fallado.

En ambos casos, el teniente que está correcto sólo necesita aplicar una función de mayoría simple al conjunto de valores que ha recibido. Ya que $N \geq 4, (N - 2) \geq 2$. Por lo tanto, la función *mayoría* ignorará cualquier valor enviado por el teniente que ha fallado y producirá el valor enviado por el comandante, si éste está correcto.

A continuación se ilustra el algoritmo que se ha esbozado para el caso de cuatro generales. La Figura 11.20 muestra dos escenarios similares a los de la Figura 11.19, pero en este caso hay cuat-

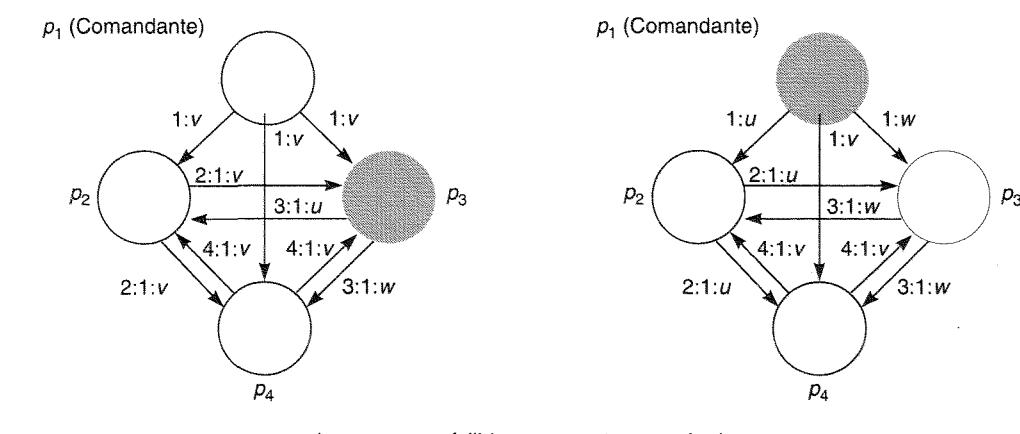


Figura 11.20. Cuatro generales bizantinos.

tro procesos y uno de ellos falla. Al igual que en la Figura 11.19, en la configuración de la izquierda ha fallado p_3 , uno de los tenientes; en la de la derecha, ha fallado el comandante p_1 .

En el caso de la izquierda, los dos tenientes que están correctos llegan a un acuerdo, decidiendo en función del valor del comandante:

$$\begin{aligned} p_2 &\text{ decide por } \text{mayoría}(v, u, v) = v \\ p_4 &\text{ decide por } \text{mayoría}(v, v, w) = v \end{aligned}$$

En el caso de la derecha, el comandante ha fallado, pero los tres procesos correctos alcanzan un acuerdo:

$$p_2, p_3 \text{ y } p_4 \text{ deciden por } \text{mayoría}(u, v, w) = \perp \text{ (valor especial que se da cuando no hay mayoría).}$$

El algoritmo tiene en cuenta el hecho de que un proceso que ha fallado puede no emitir un mensaje. Si un proceso correcto no recibe un mensaje en un tiempo límite apropiado (ya que el sistema es síncrono), el proceso sigue adelante como si el proceso que ha fallado le enviase el valor \perp .

◊ **Discusión.** Se puede medir la eficiencia de la solución al problema de los generales bizantinos, o de cualquier otro problema de acuerdo, preguntando:

- ¿Cuántas rondas de mensajes se llevan a cabo? (Éste es el factor que determina en cuánto tiempo terminará el algoritmo.)
- ¿Cuántos mensajes se mandan y de qué tamaño? (Esto permite medir la utilización del ancho de banda y tiene efecto en el tiempo de ejecución.)

En el caso general ($f \geq 1$), el algoritmo de Lamport y otros para el caso de mensajes no firmados realiza $f + 1$ rondas. En cada ronda, un proceso envía a un subconjunto del resto de los procesos los valores que ha recibido en la ronda previa. El coste del algoritmo es muy elevado, ya que implica enviar $O(N^{f+1})$ mensajes.

Fischer y Linch [1982] probaron que cualquier solución determinista al consenso suponiendo que puede haber fallos extraños (y que, por lo tanto, sirve para el problema de los generales bizantinos, tal y como se mostró en la Sección 11.5.1), necesitará al menos $f + 1$ rondas de mensajes. En consecuencia, ningún algoritmo puede funcionar más rápido que el de Lamport y otros, aunque ha habido mejoras en la complejidad de los mensajes, como en el ejemplo de Garay y Moses [1993].

Distintos algoritmos, como el de Dolev y Strong [1983], sacan ventaja de los mensajes firmados. Su algoritmo también requiere $f + 1$ rondas, pero el número de mensajes enviados es solamente $O(N^2)$.

La complejidad y el coste de las soluciones sugiere que serán aplicados sólo cuando la amenaza sea grande. Si el origen de esta amenaza es un fallo en el *hardware*, entonces la posibilidad de conducta verdaderamente arbitraria es baja. Las soluciones que se basan en un conocimiento más detallado de los fallos pueden ser más eficientes [Barborak y otros 1993]. Si el origen de la amenaza son usuarios malintencionados y un sistema quisiera hacerles frente, probablemente utilizaría firmas digitales; una solución sin firmas digitales no es práctica.

11.5.4. IMPOSIBILIDAD EN SISTEMAS ASÍNCRONOS

Se han proporcionado soluciones para los problemas del consenso y de los generales bizantinos (y, por lo tanto, pueden derivarse para la consistencia interactiva) en sistemas síncronos. Y todas las soluciones necesitan que el sistema sea síncrono. Los algoritmos suponen que el intercambio de mensajes se hace por rondas, y que los procesos han de contar el *timeout* y suponer que un proceso fallido no les enviará un mensaje dentro de una ronda, ya que se excedería el retraso máximo permitido.

Fischer y otros [1985] demostraron que ningún algoritmo puede garantizar que se encuentre un consenso en un sistema asíncrono, incluso cuando sólo falle uno de los procesos. En un sistema asíncrono, los procesos pueden responder a los mensajes en tiempos arbitrarios y, en consecuencia, no se puede distinguir un proceso lento de uno que se ha caído. Su demostración, que está fuera del alcance de este libro, implica la demostración de que siempre hay alguna continuación de la ejecución de los procesos que evita que se alcance el consenso.

A partir de este resultado de Fischer y otros se deduce que no se puede garantizar una solución en un sistema asíncrono para el problema de los generales bizantinos, ni para el de la consistencia interactiva ni para la multidifusión fiable y totalmente ordenada. Si hubiese una solución, a partir de los resultados de la Sección 11.5.1, se encontraría una solución para el consenso, contradiciendo el resultado de imposibilidad.

Obsérvese el término «garantizar» en la sentencia del resultado de imposibilidad. El resultado no significa que *nunca* se pueda alcanzar el consenso distribuido en un sistema asíncrono. Permite que el consenso se alcance con una probabilidad mayor que cero, confirmándose la observación en la práctica. Por ejemplo, a pesar del hecho de que nuestros sistemas frecuentemente son asíncronos en la práctica, los sistemas de transacciones han estado llegando a consensos durante varios años.

Una aproximación para trabajar a pesar del resultado de imposibilidad es considerar a los sistemas como *parcialmente síncronos*. Estos sistemas son lo suficientemente más débiles que los sistemas síncronos como para ser útiles como modelos de sistemas en la práctica, y lo suficientemente más fuertes que los sistemas asíncronos como para que pueda resolverse en ellos el consenso [Dwork y otros 1988]. Esta aproximación va más allá del alcance de este libro; no obstante, se van a esbozar otras técnicas que permiten trabajar bajo el resultado de imposibilidad. Estas técnicas son 1) el enmascaramiento de fallos, 2) alcanzar el consenso aprovechándose de los detectores de fallos y 3) alcanzar el consenso mediante la aleatoriedad en los aspectos del comportamiento de los procesos.

◊ **Enmascaramiento de fallos.** La primera técnica consiste en evitar el resultado de imposibilidad en su conjunto mediante el enmascaramiento de los fallos que puedan ocurrir en un proceso (véase la Sección 2.3.2 para una introducción al enmascaramiento de fallos). Por ejemplo, los sistemas de transacciones utilizan el almacenamiento persistente, que sobrevive a la caída de los procesos. Si un proceso se cae, a continuación se reinicia (automáticamente o por medio de un administrador). El proceso guarda en los puntos críticos de su programa suficiente información en el almacenamiento persistente, de tal forma que si se cae y es reiniciado, encontrará suficientes datos como para continuar de forma correcta con la tarea interrumpida. Dicho de otra forma, se comportará como un proceso que está correcto, pero que de vez en cuando toma un largo tiempo largo procesar un paso.

Por supuesto, el enmascaramiento de fallos generalmente se puede aplicar durante el diseño del sistema. El Capítulo 13 tratará cómo los sistemas de transacciones aprovechan el almacenamiento persistente. El Capítulo 14 describirá cómo los fallos en los procesos también pueden ser enmascarados mediante la replicación de componentes software.

◊ **Consenso que utiliza detectores de fallos.** Otro método para evitar el resultado de imposibilidad es emplear detectores de fallos. En la práctica, algunos sistemas utilizan detectores de fallo «perfectos por diseño» para alcanzar el consenso. No obstante, ningún detector de fallo en un sistema asíncrono que funcione exclusivamente en base a paso de mensajes puede realmente ser perfecto. Sin embargo, pueden ponerse de acuerdo en *juzgar* un proceso que ha fallado si no ha respondido en un tiempo delimitado. El hecho de que un proceso no responda no significa que haya fallado, pero el resto de los procesos se comportará como si lo hubiese hecho. En estos casos de «fallo-por-silencio», los procesos descartan, a partir de ese momento, cualesquiera de los mensajes que reciban de ese proceso «fallido». Dicho de otra forma, en la práctica se ha

transformado un sistema asíncrono en uno síncrono. Ésta es la técnica usada en el sistema ISIS [Birman 1993].

Este método requiere que el detector de fallos sea exacto de forma habitual. Cuando no lo es, el sistema ha de seguir adelante sin un miembro del grupo que, de otra forma, habría contribuido potencialmente a la eficacia del sistema. Desgraciadamente, para conseguir un detector de fallos razonablemente exacto, habría que utilizar grandes *timeouts*, forzando a los procesos a esperar durante grandes espacios de tiempo (y, por lo tanto, no realizar trabajo útil en ese tiempo) para concluir que un proceso ha fallado. Otro aspecto que surge de esta aproximación es el particionamiento de la red, que se discutirá en el Capítulo 14.

Una alternativa distinta es usar detectores de fallos imperfectos y llegar al consenso permitiendo procesos sospechosos que se comporten de forma correcta en lugar de excluirlos. Chandra y Toueg [1996] analizaron las propiedades que debía tener un detector de fallos en un sistema asíncrono para poder resolver el problema del consenso. Mostraron que podía conseguirse el consenso en un sistema asíncrono, incluso cuando el detector de fallos no era fiable, si no se caían más de $N/2$ procesos y la comunicación era fiable. El tipo más débil de detector de fallos con el que se consigue se denomina *detector de fallos eventualmente débil*. Será aquel que es:

Eventualmente débilmente completo: cada proceso que ha fallado será sospechoso finalmente y de forma permanente para algún proceso correcto.

Eventualmente débilmente exacto: pasado un cierto tiempo, al menos un proceso correcto no será nunca sospechoso para cualquier otro proceso correcto.

Chandra y Toueg muestran que no se puede implementar uno de estos detectores de fallos en un sistema asíncrono utilizando exclusivamente paso de mensajes. Sin embargo, se ha descrito en la Sección 11.1 un detector de fallos basado en mensajes que adapta sus tiempos límite en función de los tiempos de respuesta observados. Si un proceso, o la conexión hasta él, es muy lenta, el valor de su tiempo límite aumentará para que apenas se produzcan sospechas falsas del proceso. En muchos sistemas reales, el algoritmo se comporta en la práctica de modo suficientemente parecido a un detector de fallos eventualmente débil.

El algoritmo de consenso de Chandra y Toueg permite que los procesos, de los que se sospechó de forma equivocada, que continúen con sus operaciones habituales, y que los procesos que sospecharon de ellos reciban sus mensajes y los procesen con normalidad. Esto complicará la vida del programador de aplicaciones, pero tiene la ventaja de que no se desperdician procesos correctos que fueron excluidos de forma errónea. Además, los tiempos límite para detectar fallos pueden fijarse de forma menos conservadora que en la aproximación ISIS.

◊ **Consenso que utiliza la aleatoriedad.** El resultado de Fischer y otros depende de qué puede considerarse como un «adversario». Éste es un «personaje» (en realidad, una colección de sucesos aleatorios) que puede aprovecharse de los fenómenos propios de las vistas asíncronas para frustrar los intentos de los procesos para alcanzar el consenso. El adversario manipula la red para retrasar los mensajes de tal forma que lleguen en un instante de tiempo erróneo y, de forma similar, ralentiza o acelera los procesos para que estén en el estado «equivocado» cuando reciban el mensaje.

La tercera técnica que solventa el resultado de imposibilidad consiste en introducir en el comportamiento del proceso un elemento de posibilidad, de tal forma que el adversario no pueda llevar a cabo su estrategia de frustración de una forma eficaz. En algunos casos no puede alcanzarse el consenso, pero este método permite a los procesos que lo alcancen en un tiempo finito *esperado*. En Canetti y Rabin [1993] se puede encontrar un algoritmo probabilístico que soluciona el consenso incluso con fallos extraños.

11.6. RESUMEN

El capítulo comenzó discutiendo la necesidad de que los procesos accedan a recursos compartidos bajo condiciones de exclusión mutua. Los servidores que gestionan los recursos compartidos no siempre incorporan la posibilidad de realizar bloqueos, y se necesita un servicio separado de exclusión mutua distribuida. Se trataron tres algoritmos para conseguir la exclusión mutua: uno que emplea un servidor central, otro basado en un anillo y el tercero basado en la multidifusión que usa relojes lógicos. Ninguno de estos mecanismos puede soportar fallos en la forma en la que fueron descritos, aunque pueden modificarse para que toleren algunos fallos.

A continuación, el capítulo consideró un algoritmo basado en anillo y el algoritmo del abusón, cuyo objetivo común es elegir exclusivamente un proceso dentro de un conjunto, incluso si tienen lugar distintos procesos de elección de forma concurrente. El algoritmo del abusón podría usarse, por ejemplo, para elegir un nuevo maestro servidor de tiempos o un nuevo servidor de bloqueos cuando el existente falla.

El capítulo continuó describiendo la comunicación por multidifusión. Se discutió la multidifusión fiable, en el cual los procesos correctos se ponen de acuerdo en el conjunto de mensajes a entregar, y la multidifusión con ordenación FIFO, causal y total, en la entrega. Se dieron algoritmos para obtener multidifusión fiable y para los tres tipos de ordenación en la entrega.

Finalmente, se describieron los problemas del consenso, los generales bizantinos y la consistencia interactiva. Se establecieron las condiciones para poder solucionarlos y se mostraron las relaciones existentes entre ellos, incluyendo la relación entre el consenso y la multidifusión fiable y con ordenación total.

Pueden encontrarse soluciones en un sistema síncrono, como se describió para algunos de los problemas. De hecho, existen soluciones incluso con fallos arbitrarios. Se esbozó parte de la solución propuesta por Lamport y otros para el problema de los generales bizantinos. Aunque algoritmos más recientes tienen una menor complejidad, en principio, ninguno puede evitar las $f + 1$ rondas de dicho algoritmo, a menos que los mensajes se firmen digitalmente.

El capítulo terminó describiendo el resultado fundamental de Fischer y otros que concierne a la imposibilidad de garantizar el consenso en un sistema asíncrono. Se discutió el hecho de que, a pesar de ello, los sistemas llegan regularmente al acuerdo en ese tipo de sistemas.

EJERCICIOS

- 11.1. ¿Es posible implementar un detector de procesos con fallo, fiable o no, utilizando un canal de comunicación no fiable?
- 11.2. Si todos los procesos cliente usan una comunicación de un único hilo, ¿es relevante la condición EM3 de exclusión mutua (que especifica la entrada según ordenación sucedió-antes)?
- 11.3. Obténgase una fórmula para la máxima capacidad de procesamiento de un sistema de exclusión mutua en términos del retardo de sincronización.
- 11.4. En el algoritmo con servidor central para conseguir exclusión mutua, descríbase una situación en la cual dos peticiones no son procesadas con ordenación sucedió-antes.
- 11.5. Adáptese el algoritmo con servidor central para conseguir exclusión mutua para tratar el fallo por caída de cualquier cliente (en cualquier estado), suponiendo que el servidor está correctamente y que se dispone de un detector de fallos fiable. Comentar si el sistema re-

sultante es tolerante a fallos. ¿Qué podría suceder si un cliente que posee el testigo es sospechoso erróneamente de haber caído?

- 11.6. Dése un ejemplo de la ejecución del algoritmo basado en anillo que muestre que no se garantiza que los procesos entran en la sección crítica según la ordenación sucedió-antes.
- 11.7. En un cierto sistema, por lo general cada proceso entra en su sección crítica varias veces antes de que otro proceso lo pida. Explíquese por qué el algoritmo de Ricart y Agrawala para la exclusión mutua basada en anillo no es eficiente en este caso, y describáse cómo mejorar su rendimiento. ¿Cumple la adaptación propuesta la condición de pervivencia EM2?
- 11.8. En el algoritmo del abusón, un proceso que se está recuperando de un fallo comienza un proceso de elección y se convierte en el nuevo coordinador si tiene un identificador mayor que el actual ocupante del cargo. ¿Es necesaria esta característica en el algoritmo?
- 11.9. Sugiérase cómo adaptar el algoritmo del abusón para tratar con particiones temporales en la red (lo que ocasiona una comunicación lenta) y para tratar con procesos lentos.
- 11.10. Desarróllese un protocolo para conseguir un servicio de multidifusión básico sobre la multidifusión IP.
- 11.11. ¿Cómo han de cambiarse, si es que han de hacerlo, las definiciones de integridad, acuerdo y validez para la multidifusión fiable con grupos abiertos?
- 11.12. Explíquese por qué invirtiendo el orden de las líneas *F-entrega m; y si (q ≠ p) entonces B-multicast (q, m); fin si* en la Figura 11.10 se consigue que el algoritmo ya no satisface la propiedad de acuerdo uniforme. ¿Satisface el algoritmo de multidifusión fiable basado en IP la propiedad de acuerdo uniforme?
- 11.13. Explíquese si el algoritmo de multidifusión fiable sobre la multidifusión IP funciona tanto para grupos abiertos como cerrados. Dado cualquier algoritmo para grupos cerrados, ¿cómo, de una forma sencilla, pueden obtenerse algoritmos para grupos abiertos?
- 11.14. Considérese cómo solventar las suposiciones poco prácticas que se hicieron para conseguir las propiedades de validez y acuerdo para el caso de multidifusión fiable basada en multidifusión IP. Pista: añádase una regla para borrar los mensajes retenidos cuando hayan sido entregados en todas partes; y considérese añadir un mensaje falso de «latido» que nunca se entrega a la aplicación, pero que el protocolo manda si la aplicación no tiene que enviar mensaje alguno. ☺
- 11.15. Muéstrese que el algoritmo multidifusión con ordenación FIFO no funciona para grupos que se solapan, considerando los mensajes enviados por la misma fuente a dos grupos que se solapan y considerando un proceso en la intersección de ambos grupos. Adáptese el protocolo para que funcione en ese caso. Pista: los procesos han de incluir junto con los mensajes los últimos números de secuencia de mensajes enviados a *todos* los grupos.
- 11.16. Muéstrese por qué si la multidifusión básica usada en el algoritmo de la Figura 11.14 también cumple la ordenación FIFO, entonces la multidifusión resultante con ordenación total está también ordenada causalmente. En ese caso, ¿cuálquier multidifusión que tenga ordenación FIFO y total, en consecuencia, estará ordenada causalmente?
- 11.17. Sugiérase cómo adaptar el protocolo de multidifusión con ordenación causal para tratar grupos que se solapan.
- 11.18. Cuando se discutió el algoritmo de exclusión mutua de Maekawa, se vio un ejemplo de tres subconjuntos de un conjunto de tres procesos que podrían llevar a un bloqueo mutuo.

Utilíicense estos subconjuntos como grupos de destino de multidifusión para mostrar cómo una ordenación total por parejas no es necesariamente acíclica.

- 11.19. Constrúyase una solución a la multidifusión fiable con ordenación total en un sistema síncrono utilizando multidifusión fiable y una solución para el problema del consenso.
- 11.20. Se ofreció una solución para el problema del consenso a partir de una solución para la multidifusión fiable y totalmente ordenada, que involucraba la selección del primer valor a entregar. Explíquese, utilizando primeros principios, por qué en un sistema asíncrono no podemos obtener una solución utilizando un servicio de multidifusión fiable pero sin ordenación total y la función «mayoría» (ha de notarse que, si se pudiese, ¡esto estaría en contradicción el resultado de imposibilidad de Fischer y otros!) Pista: ténganse en cuenta los procesos lentos o que han fallado.
- 11.21. Muestre que se puede alcanzar el acuerdo bizantino por tres generales, con uno de ellos fallando, si los generales firman digitalmente sus mensajes.