3 Sincronización distribuida

Contenido

3.1 Introducción
3.2 Exclusión mutua distribuida
3.2.1 Algoritmos centralizados
3.2.2 Algoritmos distribuidos
3.3 Algoritmos de elección
3.3.1 Algoritmo peleón (*bully*)
3.3.2 Algoritmo del anillo
3.4 Comunicación a grupos
3.4.1 Modelo de sistema
3.4.2 Semánticas de difusión
3.4.3 Gestión de la composición del grupo
3.4.4 Ejemplos de sistemas para comunicación a grupos
3.5 Replicación
3.5.1 Modelo
3.5.2 Criterios de consistencia
3.5.3 Técnicas de replicación
3.6 Transacciones distribuidas
3.6.1 Ejecución de transacciones distribuidas
3.6.2 Acuerdo atómico
3.6.3 Mecanismos de recuperación
3.7 El problema del consenso
3.8 Ejercicios

3.1 Introducción

En el capítulo precedente hemos tratado los problemas y los conceptos fundamentales relacionados con la gestión del estado global en un sistema distribuido. Comprobamos que las limitaciones prácticas en la sincronización de los relojes locales nos llevaba a considerar un modelo de tiempo lógico que permite expresar la causalidad entre eventos y a introducir una definición de estado global consistente a partir del concepto de corte. Sobre esta base vamos a describir aquí una serie de problemas de sincronización que plantean las aplicaciones distribuidas.

- Exclusión mutua. El acceso exclusivo a recursos compartidos, un problema clásico de sincronización en sistemas centralizados, cuando se traslada a un sistema distribuido requiere un nuevo enfoque, que implica comunicar a los procesos involucrados para acordar el orden de acceso.

- Elección. La figura de un proceso coordinador o líder es frecuente en sistemas distribuidos. La posibilidad de fallo del coordinador hace necesarios mecanismos que permitan la promoción de un proceso como nuevo coordinador, lo que implica detectar el fallo del coordinador y la elección de uno nuevo.

- Comunicación a grupos. El *multicast* o difusión de mensajes a un grupo de procesos requiere definir criterios de fiabilidad y orden en la entrega de mensajes e implementar algoritmos que los garanticen.

- Servicios replicados. Cuando un conjunto de servidores redundantes tratan de proporcionar tolerancia a fallos o alta disponibilidad, se requieren modelos que proporcionen la consistencia adecuada para la actualización de las réplicas y para la gestión de las altas y bajas.

- Transacciones distribuidas. En sistemas de reserva o bancarios, la actualización del estado distribuido mediante una operación que involucra a varios procesos (transacción) debe preservar invariantes que implican el acuerdo entre los procesos sobre el resultado de la operación.

Aunque las aplicaciones relacionadas aparentan ser de distinta naturaleza, coinciden en cuanto al contexto (procesos que actualizan el estado global distribuido y que están sujetos a fallo) y en cuanto al objetivo (la coordinación entre los procesos para una actualización consistente del estado global). [COU05] es la referencia más general para este tema.

3.2 Exclusión mutua distribuida

De uso más limitado que en los sistemas operativos centralizados, la exclusión mutua es necesaria en sistemas distribuidos para resolver situaciones particulares en el acceso a recursos compartidos, como en sistemas de ficheros

distribuidos (por ejemplo, en NFS[1]), memoria compartida distribuida, y entrada/salida (por ejemplo, sistemas de ventanas). Los algoritmos de exclusión mutua distribuida pueden ser centralizados o distribuidos.

3.2.1 Algoritmos centralizados

Se basan en la existencia de un proceso coordinador de la sección crítica, que gestiona peticiones de entrar y dejar la sección crítica.

- Petición de entrar a la sección crítica:
  - Si está libre, el coordinador responde con un mensaje de confirmación.
  - Si está ocupada, no responde (o responde con un mensaje de denegación), y encola la petición.
- Petición de dejar la sección crítica:
  - Si la cola de procesos pendientes no está vacía, elige al primero para entrar y le envía la confirmación.

Este algoritmo garantiza exclusión mutua, no interbloqueo y no inanición. El problema de los algoritmos centralizados es que el coordinador se convierte en un cuello de botella, por lo que son poco escalables. Además, ofrecen escasa tolerancia a fallos, ya que un fallo en el coordinador deja a los clientes sin acceso a la sección crítica. Por otro lado, se requiere un tratamiento explícito de los fallos de los clientes, ya que un cliente que no llegase a solicitar la liberación de la sección crítica impediría a otros el acceso.

3.2.2 Algoritmos distribuidos

A diferencia de los centralizados, en los algoritmos distribuidos no existe un proceso diferenciado que coordine el acceso a la sección crítica, sino que se implementa un protocolo que permite establecer un acuerdo entre los procesos para decidir quién entra a la sección crítica.

3.2.2.1 Algoritmo de Ricart y Agrawala

Desarrollado en 1981, se basa en que todo proceso que quiera acceder a la sección crítica obtenga antes el beneplácito del resto de los procesos. Requiere establecer un orden total en los eventos, lo que se consigue con marcas de tiempo y asignando un orden predeterminado para procesos con la misma marca (por ejemplo, a partir del identificador del proceso).

---

[1] En UNIX, NFS utiliza un proceso daemon, *lockd*, que gestiona el acceso exclusivo a los ficheros.

- Petición de entrar a la sección crítica:
  - El solicitante envía a todos los procesos un mensaje con su nombre, la sección crítica y una marca del tiempo.
  - Un receptor:
    - Si no está en la sección crítica, envía un mensaje de confirmación al solicitante.
    - Si está en la sección crítica, no responde (o deniega) y encola la petición.
    - Si está esperando a entrar a la sección crítica, compara la marca de tiempo del solicitante con la de su propia petición. Si la del solicitante es anterior, le envía la confirmación; si no, no responde (o deniega) y encola su petición.
  - Para poder entrar a la sección crítica, un proceso tiene que recibir confirmaciones de todos los procesos.
- Petición de salir de la sección crítica:
  - Un proceso envía confirmaciones a todos los procesos que tiene en su cola.

La denegación mediante la no respuesta ofrece nula tolerancia a fallos (falla uno, no entra ninguno). La denegación explícita permite utilizar time-outs para suponer que un proceso que no contesta está fallando y prescindir de él. Sin embargo, ha de tenerse en cuenta que una sospecha errónea podría conducir a la violación de la exclusión mutua en la sección crítica. Por otra parte, aumentar las duraciones de los plazos para incrementar la certidumbre de las hipótesis de fallo implica mayores retardos, con la consiguiente pérdida de rendimiento.

Si el conjunto de procesos es dinámico, requiere gestionar las altas y bajas de procesos. Como se verá, el soporte para comunicación a grupos, si se dispone de él, facilita la implementación de este tipo de algoritmos. La complejidad del protocolo, medida en función del número de mensajes necesarios, es alta, aunque mejora si se dispone de soporte para *multicast*.

Con respecto a los algoritmos centralizados, en este algoritmo todos los procesos son cuellos de botella, por lo que no tiene interés práctico a partir de unos pocos procesos.

3.2.2.2 Algoritmo del anillo (*token ring*)

Se construye un anillo lógico ordenando los N procesos circularmente. Cada proceso P_k tiene un predecesor P_{k-1} y un sucesor $P_{(k+1) \bmod N}$.

Un testigo (*token*) circula por el anillo de procesos en sentido único, mediante paso de mensajes, de acuerdo al protocolo siguiente:

- Inicialmente, se entrega el testigo a un proceso cualquiera.
- Un proceso P_k fuera de la sección crítica, cuando recibe el testigo lo pasa al proceso $P_{(k+1)\ mod\ N}$.
- Cuando un proceso P_k quiere entrar a la sección crítica, espera a obtener el testigo. Cuando lo recibe, conserva el testigo y entra a la sección crítica.
- Cuando un proceso P_k deja la sección crítica, pasa el testigo al $P_{(k+1)\ mod\ N}$.

Este algoritmo presenta algunos problemas. Por una parte, hay que buscar un compromiso entre latencia y sobrecarga de tráfico en la red. Por otra, el anillo se rompe por el fallo de un proceso. La tolerancia a fallos puede mejorarse si se requiere reconocimiento del sucesor al recibir el testigo. Si el sucesor P_{k+1} no contesta (en un plazo), P_k supone que falla y envía el testigo al P_{k+2}. Otro problema es la pérdida del testigo por el fallo del proceso que lo tiene.

De nuevo, este tipo de algoritmos sólo puede funcionar en modelos de sistema síncronos, es decir bajo la suposición de que, si se cumple un plazo sin recibir un mensaje esperado, es que el emisor ha fallado. En este algoritmo, cuando el testigo no ha llegado pasado un plazo, hay que iniciar un protocolo de introducción de un nuevo testigo. Pero ya que, en general, no es posible acotar el tiempo que cada proceso retiene el testigo (es decir, la duración de las secciones críticas), puede ocurrir que un testigo presumiblemente perdido siga circulando, lo que implica gestionar las situaciones de más de un testigo en el anillo (mediante la utilización de identificadores de testigo).

3.3 Algoritmos de elección

Muchas aplicaciones y servicios distribuidos se basan en la existencia de un proceso diferenciado que coordina el trabajo de un conjunto de procesos. Por ejemplo, acabamos de ver que el algoritmo centralizado para exclusión mutua distribuida requiere un proceso coordinador. En este mismo capítulo comprobaremos esta misma necesidad para los algoritmos de acuerdo atómico y en algunos modelos de replicación.

En todas estas situaciones se requiere detectar que el proceso coordinador falla y elegir un nuevo proceso que asuma el papel de coordinador. La elección requiere el acuerdo sobre quién va a ser el nuevo y único coordinador. De nuevo, las decisiones se basan en la existencia de plazos para la recepción de los mensajes de respuesta.

Por otra parte, para determinar el criterio de elección de un proceso como coordinador se requiere definir un orden total entre el conjunto de procesos. Supondremos que los procesos tienen asociados identificadores únicos según los cuales pueden ordenarse.

3.3.1 Algoritmo peleón (*bully*)

Este algoritmo lo propuso Garcia-Molina en 1982 y se basa en la idea de que un proceso que sospecha del fallo del coordinador intente promoverse a coordinador *retando* a los procesos más fuertes que él (es decir, por ejemplo, con identificador más alto).

- Cuando un proceso sospecha que el coordinador falla, inicia una elección y se convierte en *elegible*. Un proceso elegible P ejecuta el siguiente protocolo:
  1. Envía un mensaje de elección (un *reto*) a todos los procesos con un identificador mayor que el suyo.
  2. Si nadie responde, P gana la elección y se convierte en *coordinador*, difundiendo un mensaje para dar a conocer que él es el nuevo coordinador. En caso contrario, no será coordinador y deja de ser elegible.
- Cuando un proceso recibe un mensaje de elección (de un proceso con identificador menor):
  1. Responde al remitente con un *mensaje de confirmación*.
  2. Si no lo era ya, se convierte en *elegible* y ejecuta el protocolo anterior.

Al final quedará un único proceso elegible, que será el nuevo coordinador, y que dará a conocer su nueva condición a todos los procesos.

3.3.2 Algoritmo del anillo

Los N procesos se ordenan circularmente, cada uno con un sucesor. El algoritmo tiene dos fases:

Fase 1:

- Cuando un proceso P_i sospecha que el coordinador falla, envía a su sucesor $P_{(i+1) \bmod N}$ un *mensaje de elección* que contiene el identificador de P_i. Si P_{i+1} no responde, P_i repite el envío a P_{i+2} y así hasta que encuentra un proceso que confirma la recepción.
- Cuando un proceso recibe un mensaje de elección, lo reenvía a su sucesor incluyendo en el mensaje su propio identificador, aplicando el procedimiento anterior.

Fase 2. Tarde o temprano, el mensaje de elección llegará a P_i:

- El mensaje de elección (que contiene una lista con todos los procesos no sospechosos de fallar) se convierte en un *mensaje de tipo coordinador*, con el mismo contenido.
- Tras una vuelta completa, el mensaje de tipo coordinador habrá informado a todos los procesos que el nuevo coordinador es el proceso al que corresponde el identificador mayor de los incluidos en el mensaje.

En otra versión del algoritmo [TAN07], el mensaje de elección contiene únicamente el identificador del proceso elegible, y un proceso lo modifica sólo si su identificador es mayor.

3.4 Comunicación a grupos

En el capítulo anterior considerábamos un modelo de comunicación por paso de mensajes entre pares de procesos donde la determinación de la causalidad era el problema fundamental a resolver para el mantenimiento del estado global. Sin embargo, en sistemas distribuidos, la comunicación 1:N, que denominaremos *multicast* o **difusión** de mensajes a un grupo, introduce nuevos conceptos y la necesidad de ampliar el modelo. Existe un abanico de situaciones donde la comunicación de tipo multicast es aplicable, con diferentes requisitos en cuanto a su semántica. Ejemplos de estas aplicaciones son la notificación de eventos; el registro o búsqueda de servicios remotos, y la replicación de servicios, ya sea por tolerancia a fallos o por disponibilidad y rendimiento.

En general consideraremos una infraestructura (*middleware*) para la gestión adecuada de la comunicación a grupos, con una interfaz concreta y una semántica precisa para la comunicación. Las principales características que hay que definir en un sistema de comunicación a grupos son:

- La identificación de grupos de procesos. El grupo de procesos debe ser accesible como tal, no como la suma de sus miembros.
- Si los grupos son *abiertos* o *cerrados*. En un grupo cerrado, sólo los miembros del grupo pueden difundir mensajes al grupo, mientras que un grupo abierto, puede hacerlo también un proceso ajeno al grupo.
- El direccionamiento de mensajes al grupo, cuya semántica estará determinada por un conjunto de propiedades.
- La gestión de la composición del grupo. Las bajas (en general, por fallo) y las altas (por recuperación) deben ser gestionadas consistentemente.

3.4.1 Modelo de sistema

Definiremos inicialmente un modelo de comunicación a grupos [COU05 §12] [MUL93 §5] donde un grupo *G* se compone de N procesos P_1, P_2, …, P_N. La definición de los grupos como cerrados no supone una restricción, ya que la difusión de un mensaje desde fuera del grupo se puede modelar como un mensaje *enviado* a un proceso del grupo que actúa como front-end y *difunde* el mensaje al resto de los procesos del grupo.

Definiremos las siguientes primitivas para difusión y entrega de mensajes al grupo:

difundir*(G, m)*. Difunde el mensaje *m* al grupo *G*.

entregar*(m)*. Entrega el mensaje *m* al proceso que la ejecuta.

Estas primitivas se implementan sobre la interfaz de comunicaciones del sistema operativo y se construyen mediante protocolos que utilizan *enviar()* y *recibir()*. La difusión puede tener soporte hardware (por ejemplo, con IP multicast). La estructura de este modelo se representa en la Figura 3.1.

Consideraremos que los procesos del grupo pueden fallar. Nos referiremos a un proceso del que no se sospecha que falle como **proceso correcto**. El modelo de fallos es el de fallos de *crash* o de parada, es decir, un proceso que falla no se recupera[2]. En nuestro modelo, consideraremos que los canales de comunicación son *cuasi-fiables*, lo que significa que un proceso correcto terminará recibiendo el mensaje enviado por otro proceso correcto[3].

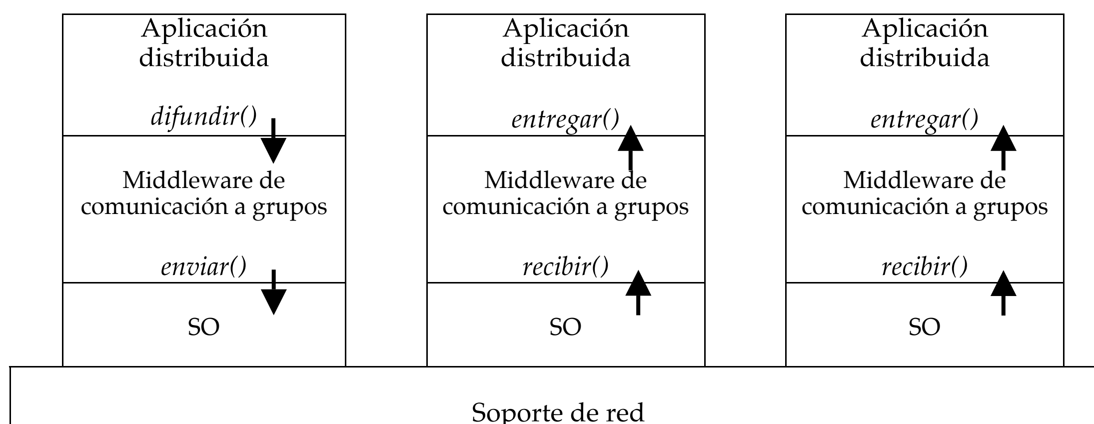


Figura 3.1. Estructura de un modelo de comunicación a grupos.

3.4.2 Semánticas de difusión

Para definir las semánticas de difusión en un grupo vamos a basarnos en dos características de la comunicación a grupos: **fiabilidad** de la difusión y **orden de entrega** de los mensajes.

Consideremos en primer lugar una implementación sencilla de las primitivas *difundir* y *entregar*:

- Cuando un proceso P_i ejecuta *difundir(G, m)*:

  Para todo $P_j \in G$, *enviar*(P_j, *m*).

- Cuando un proceso P_i ejecuta *recibir(m)*:

  entregar(*m*).

[2] Existen otros modelos de fallos (*crash-recovery*, de omisión, bizantinos), que no vamos a considerar aquí.

[3] Si un proceso falla, el proceso no es correcto. Un canal de comunicación *fiable* garantizaría la recepción por un proceso correcto de todo mensaje enviado al canal aún en el caso de que el emisor sea incorrecto y falle nada más enviar el mensaje.

Una implementación así proporciona **difusión no fiable**. En los siguientes apartados definiremos formalmente comunicación fiable e introduciremos diferentes órdenes de entrega de mensajes. Intuitivamente, diremos aquí que la difusión es no fiable cuando un mismo mensaje difundido al grupo puede ser entregado por algunos procesos correctos y no entregado por otros procesos correctos. Efectivamente, dado el algoritmo anterior puede comprobarse que si el proceso que difunde el mensaje falla mientras ejecuta *difundir()* eventualmente unos procesos recibirán el mensaje y otros no.

3.4.2.1 Difusión fiable

Definiremos como **difusión fiable** la que cumple las siguientes tres propiedades:

- **Validez**. Si un proceso correcto difunde un mensaje *m*, entonces todos los procesos correctos terminarán entregando *m*.

- **Acuerdo**. Si un proceso correcto entrega un mensaje *m*, entonces todos los procesos correctos terminarán entregando *m*.

- **Integridad**. Para cualquier mensaje *m*, todo proceso correcto entrega *m* como mucho una vez, y sólo si *m* ha sido previamente difundido.

La combinación de estas tres propiedades se conoce como **propiedad de todos-o-ninguno**, que implica que un mensaje difundido a un grupo de procesos, o llega a todos los procesos correctos del grupo o no llega a ninguno.

Puede comprobarse que la difusión no fiable definida en el apartado anterior posee las propiedades de validez e integridad, pero no la de acuerdo.

La difusión fiable la expresaremos mediante las primitivas *R-difundir()* y *R-entregar()*[4] y se puede implementar a partir de la primitiva de *difundir()* no fiable con el siguiente algoritmo:

- Cuando un proceso P_i ejecuta *R-difundir(G, m)*: *difundir(G, m)*.

- Cuando un proceso P_j ejecuta *recibir(m)*, si *m* no se ha recibido previamente:
  1. Si $P_j \neq P_i$, *difundir(G, m).*
  2. *R-entregar(m).*

Se puede demostrar que este algoritmo proporciona difusión fiable. Evidentemente, el algoritmo preserva la validez del no fiable. Además, si existe un proceso correcto que entrega *m*, entonces es que lo ha reenviado previamente a todos los procesos; luego, por definición de canal cuasi-fiable y extendiendo el razonamiento, en ese caso todos los procesos correctos terminarán recibiendo *m*, y por lo tanto entregándolo, proporcionando acuerdo.

---

[4] El prefijo *R-* viene de *Reliable*.

La integridad se asegura porque un proceso sólo ejecuta la entrega la primera vez que recibe el mensaje, y sólo se entregan mensajes difundidos.

3.4.2.2 Orden de entrega de los mensajes

La difusión fiable no establece garantías acerca del orden de entrega de los mensajes. Sobre la difusión fiable pueden implementarse diferentes semánticas en cuanto al orden de entrega:

- **Orden FIFO**. Si un proceso difunde un mensaje m_1 antes que un mensaje m_2, entonces ningún proceso correcto entrega m_2 si no ha entregado previamente m_1.
- **Orden causal**. Si la difusión de un mensaje m_1 *ocurre antes que* la difusión de un mensaje m_2, entonces ningún proceso correcto entrega m_2 si no ha entregado previamente m_1.
- **Orden total**. Si dos procesos correctos P_i y P_j entregan dos mensajes m_1 y m_2, entonces P_i entrega m_1 antes que m_2 sí y solo sí P_j entrega m_1 antes que m_2.

Nótese que el orden causal es más fuerte que el orden FIFO, ya que la relación *ocurre antes de* incluye la relación de orden temporal entre eventos de un mismo proceso. La **difusión con orden total**[5] es la base para asegurar consistencia: todos los procesos correctos ven la misma secuencia de mensajes[6]. La difusión fiable se combina con distintos órdenes para dar lugar a diferentes semánticas, como muestra la Tabla 3.1[7].

	Orden FIFO	Orden Causal	Orden Total
Difusión Fiable			
Difusión FIFO	•		
Difusión Causal	•	•	
Difusión con Orden Total			•
Difusión con Orden Total Causal	•	•	•

Tabla 3.1. Diferentes semánticas de difusión fiable en función del orden de entrega de los mensajes.

En general, un algoritmo de difusión con una semántica dada se puede construir a partir de un algoritmo que proporcione una semántica más débil, como se muestra en [MUL93 §5]. Por ejemplo, puede construirse difusión FIFO

[5] Históricamente denominada *difusión atómica*.
[6] En teoría es posible difusión con orden total no causal, así que habría que distinguir entre consistencia total y consistencia causal. Sin embargo, en la práctica es directo obtener orden causal cuando se tiene orden total, así que puede hablarse simplemente de consistencia en este caso.
[7] Obsérvese que la combinación de orden total y orden FIFO no causal no es posible (véase el Ejercicio 3).

sobre difusión fiable numerando los mensajes en el emisor. La difusión causal se consigue incluyendo vectores de tiempos. Hay que hacer notar, sin embargo, que mientras las semánticas de difusión fiable, FIFO y causal no requieren hacer suposición alguna acerca del tiempo, la difusión con orden total, sí. De esta forma, según demostraron Fischer, Lynch y Paterson en 1985, en sistemas asíncronos, donde no se pueden acotar el tiempo de transmisión de los mensajes ni las velocidades relativas de ejecución de los procesos, no es posible una solución determinista de los problemas que requieran un *consenso* distribuido, como es el caso de la difusión fiable con orden total. Trataremos más adelante el problema del Consenso.

Para sistemas síncronos existen diferentes algoritmos que implementan difusión con orden total, que se apoyan en difusión fiable. Cuando se dispone de sincronización interna se puede utilizar el siguiente algoritmo. Supondremos que el mensaje *m* difundido en *R-difundir* está etiquetado con la marca local del tiempo, *t(m)*[8].

- Cuando un proceso P_i ejecuta *TO-difundir(G, m)*:

  R-difundir(G, <t(m), m>).

- Cuando un proceso P_j ejecuta *R-entregar(m)*:

  Planifica *TO-entregar(m)* para el instante de tiempo local $t(m)+\Delta$

Δ es un parámetro que se configura de acuerdo a diferentes parámetros del sistema, como el retardo máximo de los mensajes, el número máximo de procesos que pueden fallar y las garantías ofrecidas por la sincronización interna de los relojes [MUL93 §5]. La planificación de la entrega en $t(m)+\Delta$ implica que si el instante de tiempo local en el que se ejecuta *TO-entregar* es posterior a ese tiempo, la entrega con orden total no puede llevarse a cabo y se considera que el emisor falla. Este algoritmo proporciona también orden causal.

Entre los algoritmos que no utilizan relojes físicos sincronizados podemos distinguir entre los centralizados, que se basan en un *secuenciador*, y los distribuidos, que se basan en acuerdos entre los procesos que reciben los mensajes[9].

Describiremos aquí un algoritmo basado en secuenciador. El secuenciador, *Q*, que puede ser uno de los procesos del grupo, *G*, o un proceso específico, decide el orden en que han de entregarse los mensajes. Cada proceso envía una copia del mensaje *m* al secuenciador, incluyendo un identificador único *id(m)*. El secuenciador les asigna un número de secuencia y comunica el orden de entrega a los procesos del grupo mediante un mensaje especial (*ORDEN*). Cada proceso $P_i \in G$ ejecuta las siguientes tareas. Inicialmente la variable S_i vale 0.

- Cuando P_i ejecuta *TO-difundir(G, m)*:

  R-difundir(G ∪ {Q}, <m, id(m)>)

---

[8] Este tipo de difusión se denomina **difusión fiable temporizada** [MUL93].

[9] El protocolo de difusión con orden total de ISIS es un ejemplo de algoritmo distribuido.

- Cuando P_i ejecuta *R-entregar(<m, id(m)>)*:

  Encola *<m, id(m)>* en la cola de espera para entrega con orden total.

- Cuando P_i ejecuta *R-entregar(<ORDEN, id, S>)*:

  1. Espera a que *<m, id>* esté en la cola y S sea igual a S_i+1
  2. Ejecuta *TO-entregar(m)*
  3. Elimina *<m, id>* de la cola
  4. $S_i = S$

El secuenciador Q ejecuta la siguiente tarea (inicialmente S_G vale 0):

- Cuando Q ejecuta *R-entregar(<m, id(m)>)*:

  1. $S_G = S_G + 1$
  2. Ejecuta *R-difundir(G, <ORDEN, id(m),* S_G*>)*

3.4.3 Gestión de la composición del grupo

Los grupos de procesos pueden ser estáticos o dinámicos. Este último caso es el más general e implica considerar en la gestión del grupo de procesos el hecho de que los procesos pueden dejar el grupo (si fallan) y sumarse a él (una vez que se recuperan del fallo)[10]. Consideraremos dos operaciones sobre un grupo, que se implementan sobre la interfaz de paso de mensajes (Figura 3.2):

join(P_i, G). Alta de un proceso P_i en un grupo G.

leave(P_i, G). Baja de un proceso P_i en un grupo G.

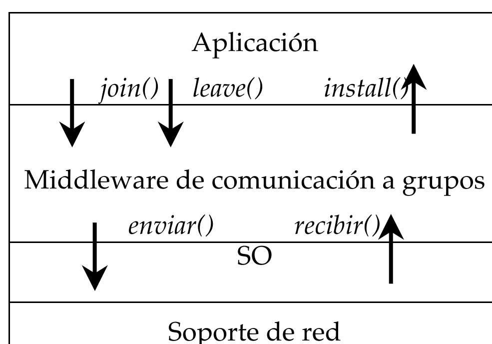


Figura 3.2. Estructura de la gestión de la composición del grupo.

Hay que remarcar que un proceso puede ser excluido del grupo porque se sospecha que falla, lo que implica implementar un mecanismo efectivo de detección de fallos.

[10] Hay que precisar que existen dos alternativas: (a) un proceso que falla se recupera con la misma identidad; (b) un proceso que falla no se recupera, se crea otro nuevo.

La gestión de la composición del grupo se modela en torno al concepto de **vista** v_k del grupo como parte de su estado global. Cuando se produce un cambio en la configuración del grupo (por *join()*, por *leave()*, o por fallo), se instala una nueva vista v_{k+1} mediante una primitiva ***install***(v_{k+1}) que el middelware de comunicación a grupos proporciona a la aplicación.

v_k). Una vista está definida por el conjunto de procesos que pertenecen al grupo cuando se instala la vista. Para garantizar la consistencia entre la instalación de vistas y las operaciones de entrega de mensajes se requiere orden total. La gestión de vistas posee las siguientes propiedades, como extensión de las que definen la difusión fiable:

- **Terminación**. El fallo de un proceso o la ejecución de *join()* o *leave()* conduce tarde o temprano a la instalación de una nueva vista.

- **Acuerdo**. Todos los procesos correctos entregan el mismo conjunto de mensajes en una vista dada.

- **Validez**. Sea P_i un proceso correcto que entrega un mensaje m en una vista v_k. Si algún proceso $P_j \in v_k$ no entrega m en v_k, entonces en la vista v_{k+1} instalada por P_i no está P_j.

A este modelo de comunicación a grupos con gestión de vistas se le denomina **virtualmente síncrono**, término utilizado originalmente en el sistema ISIS.

La gestión de la composición del grupo es también un problema de consenso.

3.4.4 Ejemplos de sistemas para comunicación a grupos

El middleware pionero en dar soporte a comunicación a grupos fue ISIS[11], desarrollado en la Universidad de Cornell a partir de 1983. Incluye un conjunto de herramientas para construir aplicaciones distribuidas sobre UNIX. ISIS derivó en Horus, y este a su vez en Ensemble[12], que es el proyecto que desarrolla actualmente la Universidad de Cornell.

ISIS proporciona un conjunto de primitivas de difusión y de gestión de pertenencia al grupo, fundamentalmente síncronas:

- FBCAST. Proporciona difusión fiable FIFO.

- CBCAST. Proporciona difusión causal. Básicamente sigue el algoritmo de los vectores de tiempos.

- ABCAST. Proporciona difusión con orden total causal (*atomic broadcast*).

- GBCAST. Gestión de pertenencia al grupo. Se basa en el concepto de sincronía virtual. Utiliza difusión con orden total, como ABCAST.

[11] http://www.cs.cornell.edu/Info/Projects/ISIS/
[12] http://www.cs.cornell.edu/Info/Projects/Ensemble/

En otra línea, las ideas de ISIS han derivado en un middleware basado en software libre y construido en Java, JGroups[13] [BAN03]. JGroups proporciona una arquitectura modular basada en pilas de protocolos, con interfaces sencillas y homogéneas (en la pila, los protocolos comunican eventos mediante métodos *up()* y *down()*). Las aplicaciones componen las pilas de protocolos a su medida y se desarrollan sobre éstas como un conjunto de bloques. A una pila de protocolos se le asocia un canal, que es el concepto que permite la asociación de los procesos en grupos. JGroups proporciona un conjunto de protocolos de diferente nivel, desde transporte hasta gestión de grupos, e incluso bloques predefinidos.

3.5 Replicación

Entendida de una manera general, la replicación engloba una serie de técnicas que tratan de proporcionar mayor rendimiento, disponibilidad y escalabilidad mediante el mantenimiento de copias de un recurso (servicio o conjuntos de datos). Más en particular, podríamos distinguir:

Replicación propiamente dicha. Un conjunto de nodos, normalmente constituidos en red local, proporciona un servicio. Cada uno de los nodos mantiene una copia del servicio, siendo capaz cada nodo de realizar todas las funciones del servicio. Las peticiones de acceso al servicio se dirigen a todos los nodos o a un subconjunto de ellos, aunque puedan utilizarse *front-ends* como intermediarios. El mantenimiento de la consistencia es fundamental. El objetivo principal es proporcionar tolerancia a fallos.

Distribución de un servicio. Un conjunto de nodos, normalmente distribuidos geográficamente en diferentes ubicaciones, proporcionan un servicio. Cada uno de los nodos mantiene una copia del servicio, aunque puede ser que no todas las funciones del servicio estén replicadas en todos los nodos. Las peticiones de acceso al servicio se dirigen a ubicaciones determinadas. Los objetivos son aumentar la disponibilidad y la escalabilidad, disminuyendo los costes de comunicación[14]. Se utiliza en internet (*mirroring*) y en ficheros y bases de datos distribuidas.

Caching. Es el almacenamiento *temporal* de subconjuntos de datos en los nodos clientes o en ubicaciones cercanas. El objetivo es aumentar el rendimiento explotando la localidad. Esta técnica es muy habitual en todo tipo de servicios distribuidos. En principio, la gestión del almacenamiento temporal es responsabilidad del cliente, aunque el servidor puede colaborar.

[13] http://www.jgroups.org/javagroupsnew/docs/index.html. Hasta Septiembre de 2003 se denominaba *JavaGroups*.

[14] En este contexto, la disponibilidad se un servicio se entiende no sólo como que el servidor que lo soporta no esté fallando, sino que sea capaz de atender las peticiones de los clientes con latencias acotadas. Por ejemplo, un servidor activo pero sobrecargado podría no estar disponible para muchos clientes.

	Objetivos	Mantenimiento de la consistencia	Ambito espacial	Ambito temporal	Gestión
Replicación	Tolerancia a fallos, alta disponibilidad	Fundamental	Todo el sistema	Permanente	Distribuida
Distribución	Alta disponibilidad, latencias bajas	Secundario	Todo o parte del sistema	Permanente	Centralizada
Caching	Latencias bajas	Importante	Parte del sistema	Temporal	En el cliente

Tabla 2. Comparación entre las diferentes técnicas de replicación.

Una comparación de las diferentes técnicas de replicación se muestra en la Tabla 2. Un mismo servicio puede utilizar una combinación de estas técnicas. Por ejemplo, un servidor de Internet puede estar soportado por una red local de servidores replicados y distribuir copias en *mirrors* por todo el mundo. Además, los clientes que accedan al servicio almacenarán temporalmente copias de las páginas más recientes.

Nos vamos a centrar fundamentalmente en la replicación para tolerancia a fallos, que reúne el conjunto más general de conceptos y métodos. La distribución y el caching recogen aspectos particulares de la replicación propiamente dicha. En concreto, el caching se estudiará dentro del tema dedicado a los sistemas de ficheros distribuidos.

3.5.1 Modelo

Consideraremos un servicio con un conjunto de operaciones claramente especificadas. Los clientes solicitan operaciones al servicio mediante peticiones según el modelo cliente-servidor. De acuerdo a la estructura de la comunicación en este modelo, una operación o_k se compone de un par de eventos invocación-respuesta (i_k, r_k). Los clientes son síncronos, es decir, dada una secuencia de operaciones de un mismo cliente, [o_1, o_2, ..., o_k, ...], i_{k+1} sólo ocurre tras r_k. No haremos ninguna suposición inicial en cuanto a cómo gestiona el servidor las peticiones, y en particular si es replicado o no; lo veremos como una caja negra.

Presentaremos el modelo con ayuda de un ejemplo. Considérese un sistema donde las operaciones posibles son *Leer* y *Escribir* un valor en una variable, con la especificación semántica habitual: *Escribir(x, v)* asigna a la variable *x* el valor *v*, y *Leer(x)* devuelve el último valor asignado a *x*. La Figura 3.3 muestra una ejecución de cuatro operaciones de dos procesos clientes de ese sistema, P_i y P_j. Cada proceso especifica una secuencia de operaciones, P_i: [o_a, o_b] y P_j: [o_c, o_d]. Para cada operación, en el cronograma se indica el valor a escribir o el leído.

Si se consideran globalmente las operaciones de todos los procesos, se obtiene una *secuencia*, que denotaremos τ, donde las secuencias particulares de cada proceso se encuentran entrelazadas. Al solaparse las operaciones, en la *secuencia entrelazada* hay que especificar el orden de los eventos de invocación y

respuesta. En el ejemplo, la secuencia entrelazada resultante es $\tau = [i_a, r_a, i_c, i_b, r_c, i_d, r_b, r_d]$.

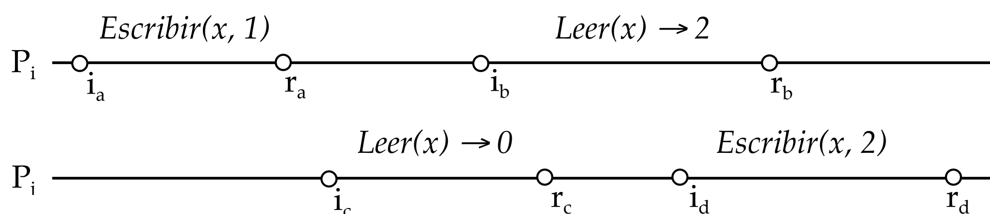


Figura 3.3. Ejemplo de ejecución de una secuencia de operaciones. Inicialmente x=0.

A partir de una secuencia entrelazada se pueden obtener diferentes ejecuciones. Una **ejecución** es la aplicación de las operaciones en un orden determinado. Una **ejecución legal**, σ, es aquélla que se atiene a la especificación de las operaciones. En el ejemplo, es legal la ejecución $[o_c, o_a, o_d, o_b]$, es decir[15]:

$$[Leer_j(x)\rightarrow 0,\ Escribir_i(x, 1),\ Escribir_j(x, 2),\ Leer_i(x)\rightarrow 2]$$

y no lo es

$$[Escribir_i(x, 1),\ Leer_j(x)\rightarrow 0,\ ...]$$

ya que la especificación de las operaciones establece que tras $Escribir_i(x, 1)$, x vale 1, y por lo tanto $Leer_j(x)\rightarrow 1$.

Nos interesa ahora definir los criterios de consistencia que hay que plantear como objetivo en un servidor replicado.

3.5.2 Criterios de consistencia

Los criterios de consistencia se derivan de las restricciones que impongamos en el orden de las operaciones para obtener ejecuciones legales a partir de una secuencia entrelazada. Un primer criterio de consistencia de una ejecución σ con una secuencia entrelazada τ dada es:

Consistencia secuencial. Una ejecución legal σ es secuencialmente consistente con τ si, para todo proceso P_i el orden de las operaciones de P_i en σ coincide con el orden de los eventos del proceso P_i en τ.

En el ejemplo de la Figura 3.3, la ejecución

$$[Leer_j(x)\rightarrow 0,\ Escribir_i(x, 1),\ Escribir_j(x, 2),\ Leer_i(x)\rightarrow 2]$$

es secuencialmente consistente con la secuencia entrelazada definida, ya que es legal y respeta el orden de las secuencias de cada proceso, P_i: $[o_a, o_b]$ y P_j: $[o_c, o_d]$.

[15] Aquí, los subíndices de las operaciones expresan el proceso que las realiza.

Como se ve, una ejecución secuencialmente consistente no tiene porqué respetar el orden temporal en que se producen las operaciones. Para hacer legal la secuencia ha habido que interpretar o_c de P_j como anterior a o_a de P_i, aunque, como se aprecia en el cronograma, el orden temporal[16] es el inverso (es decir, r_a *precede* a i_c en τ). Esta interpretación, que puede parecer poco intuitiva, puede explicarse en determinadas implementaciones de sistemas replicados. Por ejemplo, después de que P_j escriba el valor 1 en la variable x, P_i puede leer x con valor 0 si accede a una réplica no actualizada.

Sin embargo, en muchas aplicaciones la consistencia secuencial no es suficiente. Por ejemplo, en un sistema de gestión de cuentas bancarias donde se ha especificado una operación de ingreso de una cantidad en cuenta y otra operación posterior de cargo (por una cantidad menor) en la misma cuenta, la alteración del orden podría conducir a un estado diferente si el cargo deja la cuenta en números rojos. La propiedad de transparencia en la replicación requiere una consistencia más fuerte:

Linealización. Una ejecución legal σ es linealizable si es secuencialmente consistente con τ y además, $\forall k, h$, si o_k precede a o_h en σ, entonces r_h no precede a i_k en τ.

Es decir, la linealización obliga a respetar el orden temporal *global* en el que suceden los eventos de comunicación. En el ejemplo *Escribir*$_i(x, 1)$ debe ejecutarse antes de *Leer*$_j(x)$, ya que r_a precede a i_c en τ, por lo tanto la ejecución

$$[Leer_j(x) \rightarrow 0, Escribir_i(x, 1), Escribir_j(x, 2), Leer_i(x) \rightarrow 2]$$

no es linealizable con τ. Sí lo sería, en cambio[17]:

$$[Escribir_i(x, 1), Leer_j(x) \rightarrow 1, Escribir_j(x, 2), Leer_i(x) \rightarrow 2]$$

Obsérvese que la consistencia de linealización no establece nada en cuanto al orden en que deban ejecutarse operaciones que suceden concurrentemente en el tiempo. Por ejemplo, también sería linealizable la ejecución

$$[Escribir_i(x, 1), Leer_j(x) \rightarrow 1, Leer_i(x) \rightarrow 1, Escribir_j(x, 2)]$$

La linealización es el criterio de consistencia que garantiza transparencia en sistemas replicados. Las siguientes dos condiciones son suficientes para asegurar linealización:

- **Orden**. Dadas dos peticiones sobre un objeto en un sistema replicado todas las réplicas del sistema tratan estas peticiones en el mismo orden.

- **Atomicidad**. Dada una petición sobre un objeto en un sistema replicado, si una réplica del sistema trata esta petición, entonces todas las réplicas correctas[18] del sistema tratan esta petición tarde o temprano.

[16] Es decir, tal como lo percibiría un observador global.
[17] Nótese que, para obtener una ejecución linealizable, hay que modificar en el cronograma los valores de respuesta.

3.5.3 Técnicas de replicación

Las técnicas de replicación buscan un equilibrio entre consistencia, rendimiento y complejidad. Algunos modelos (**replicación pasiva** o copia primaria, *primary back-up*) son fuertemente síncronos, mientras que otros más elaborados (**replicación activa**) se basan en un soporte de comunicación a grupos más fuerte para relajar la sincronía y proporcionar una gestión más transparente de las copias.

La linealización es el criterio de consistencia general en **aplicaciones tolerantes a fallos**. Otras veces, la consistencia se puede relajar para minimizar el *overhead*, como es el caso de las **aplicaciones de alta disponibilidad** en Internet o en sistemas de ficheros distribuidos, donde es muy complicado (o imposible) proporcionar comunicación fiable.

3.5.3.1 Replicación pasiva

Se define un nodo como **primario** y el resto como **secundarios**. El cliente envía la petición al primario, que eventualmente actualiza su copia y envía la actualización síncronamente a los secundarios. Estos envían un mensaje de reconocimiento al primario una vez que han actualizado su copia. Como se aprecia en la Figura 3.4, la actualización de los secundarios se produce secuencialmente con la del primario.

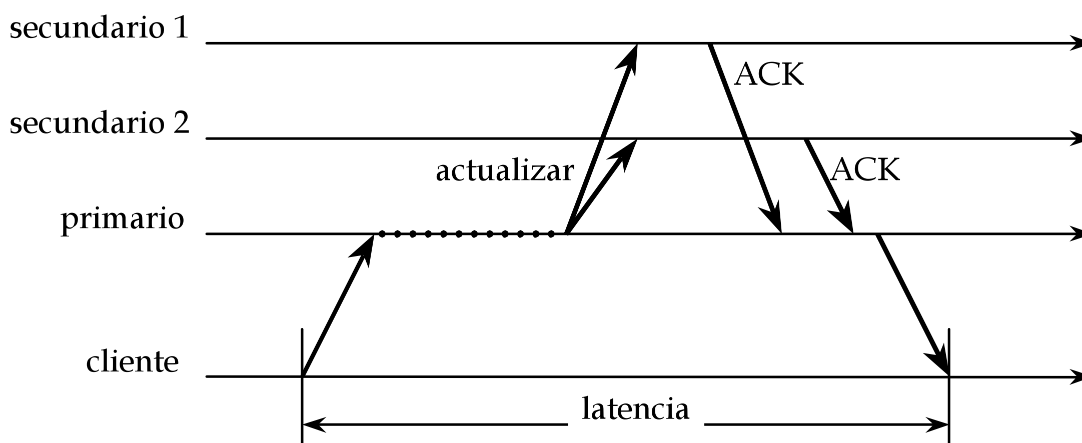


Figura 3.4. Replicación pasiva. Esquema de sincronización entre el cliente y el grupo de servidores.

Ya que los secundarios no tienen que tratar peticiones concurrentes, la semántica de la comunicación primario-secundarios a implementar basta con que sea fiable. Efectivamente, el nodo primario asegura la condición de orden al secuencializar las peticiones y la de atomicidad al esperar el reconocimiento de todos los secundarios antes de responder al cliente.

---

[18] La definición de réplica correcta equivale a la de proceso correcto introducida en difusión fiable.

El fallo de un nodo secundario lo detecta el primario a partir del incumplimiento de un time-out. Es preciso gestionar las bajas y las altas consistentemente y recuperar el estado de un servidor cuando se suma al grupo tras recuperarse del fallo. Un fallo del primario implica que uno de los nodos secundarios se promueva a primario mediante un algoritmo de elección de los estudiados en la Sección 3.3. En general, un cliente reenviará la petición al nuevo primario19 cuando se haya cumplido un plazo sin recibir respuesta. Los secundarios pueden detectar el fallo en el primario, por ejemplo mediante un protocolo de latido (mensaje periódico difundido por primario indicando que está funcionando). Hay que considerar diferentes situaciones de fallo en el nodo primario:

(a) El fallo se produce fuera del intervalo (invocación, respuesta) del tratamiento de una petición. Un secundario detecta el fallo e inicia el protocolo de elección.

(b) El fallo se produce antes de difundir la actualización a los secundarios. Un secundario detecta el fallo e inicia el protocolo de elección. El cliente deberá repetir la petición tras un plazo sin recibir respuesta.

(c) El fallo se produce después de que el primario haya difundido la actualización a los secundarios. La difusión fiable garantiza que todos los secundarios han recibido la actualización o no lo ha hecho ninguno. En general, no es posible para los secundarios determinar si el cliente ha recibido la respuesta o no, por lo que una ulterior petición del mismo cliente podría ser tanto una nueva petición como una reinvocación. En este último caso, si los secundarios llegaron a actualizar el estado, simplemente se envía la respuesta al cliente. El problema se soluciona etiquetando en el cliente las peticiones con un número de orden y almacenando en cada nodo los pares (invocación, respuesta) de cada cliente.

Una variante de la replicación pasiva, que relaja la consistencia para descargar al primario, es que los secundarios reciban directamente de los clientes las peticiones de lectura, que eventualmente no devolverán la última versión. Esta solución tiene interés en aplicaciones (por ejemplo, distribución de contenidos en Internet) donde las lecturas son mayoritarias y no se requiere consistencia de linealización. Otra alternativa de consistencia relajada, orientada a mejorar la latencia, consiste en que el servidor responda al cliente sin esperar los mensajes de reconocimiento de los secundarios.

3.5.3.2 Replicación activa

La petición de un cliente se difunde a todas las réplicas, que la tratan concurrentemente. El cliente se sincroniza con la primera respuesta que obtiene y desecha las que le llegan después, que serán idénticas[20]. La Figura 3.5 sugiere que la concurrencia introducida en los accesos a las réplicas permite disminuir

---

[19] La identidad del nodo primario se resuelve habitualmente en el servicio de nombres, que hace de front-end para el cliente. El nuevo primario se registrará allí tras ser elegido.

[20] Se considera que el servicio es determinista. Si no lo fuera, como por ejemplo en un servidor de tiempos replicado, se devolverían diferentes valores.

la latencia. Sin embargo, hay que considerar en contra el alto coste de soportar la semántica de comunicación (orden total) y el hecho de que las actualizaciones se procesan en todas las réplicas (líneas punteadas en las Figuras).

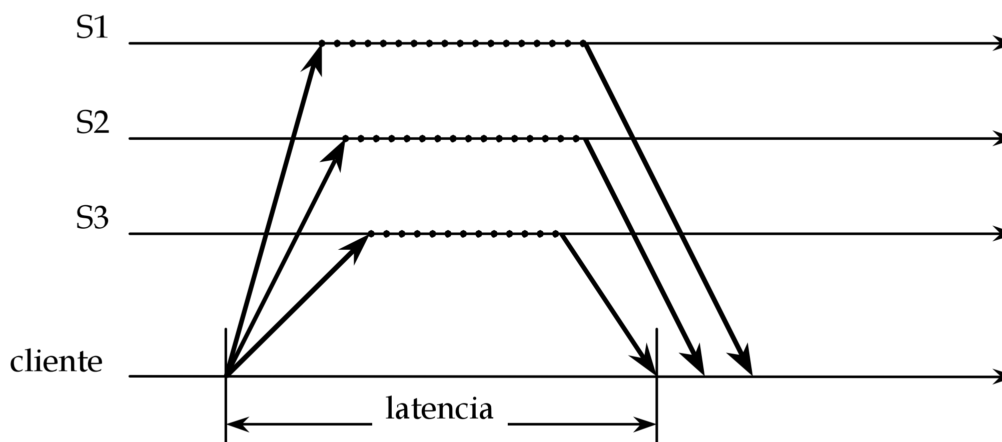


Figura 3.5. Replicación activa. Esquema de sincronización entre el cliente y el grupo de servidores.

La difusión con orden total garantiza las propiedades de orden[21] y atomicidad requeridas por la linealización. A diferencia de la replicación pasiva, donde el cliente debe detectar y gestionar el fallo del nodo primario, en replicación activa, gracias a la semántica de difusión con orden total, los fallos en los nodos son transparentes al cliente (siempre que no fallen todas las réplicas).

3.5.3.3 Técnicas de consistencia débil

En entornos donde los costes de comunicación son elevados, o el número de nodos es muy grande, los modelos de replicación descritos arriba no ofrecen la escalabilidad necesaria y las latencias se disparan. Por otra parte, determinadas aplicaciones no requieren criterios de consistencia tan estrictos, como es el caso de las aplicaciones de alta disponibilidad, usualmente en el ámbito de Internet.

Un ejemplo de estas aplicaciones es el de un tablón de anuncios distribuido geográficamente y accesible en Internet. Los usuarios pueden darse de alta o baja, y pueden leer o actualizar noticias. Los usuarios acceden al sistema distribuido mediante un *front-end* que dirige su petición a una réplica local. Para las operaciones de lectura es admisible no disponer de la última versión, pero debería garantizarse un orden causal en las operaciones de actualización.

El requisito de alta disponibilidad y latencias acotadas[22] en un entorno de gran escala y costes de comunicación elevados, como es el caso de Internet, implica que las actualizaciones se anticipen localmente, siguiendo esquemas de

[21] Hay que recordar que en nuestro modelo los clientes son síncronos.
[22] En este contexto, alta disponibilidad y latencia acotada vienen a significar lo mismo: si el sistema que ofrece un servicio no es capaz de atender las peticiones en un tiempo razonable, se considera que no está disponible para esos clientes.

replicación sencillos que no involucren el acuerdo previo de todas las réplicas (**replicación optimista**), lo que impide proporcionar garantías de consistencia estricta. La propagación de las actualizaciones se hace perezosamente, en función de criterios de vecindad. Un nodo realiza una operación de sincronización con cada uno de los de su vecindad siguiendo un esquema *peer-to-peer* y aplicando un algoritmo que establece el orden en que deben aplicarse las actualizaciones, normalmente basado en vectores de tiempo (por ejemplo, los algoritmos de antientropía [GOL92]).

El criterio de propagación de las actualizaciones debe ser tal que cada actualización llegue, tarde o temprano, a todo el sistema (*eventual consistency* o **consistencia tarde-o-temprano**). Para proporcionar esta propiedad se pueden utilizar algoritmos deterministas, como los basados en *Tablas Hash Distribuidas* (*DHTs*) [BAL03], o algoritmos probabilistas, también llamados *epidémicos* [EUG04].

Por su propia naturaleza, la replicación optimista conduce a conflictos (réplicas que divergen porque las actualizaciones se aplican en diferente orden. Cuando una operación de sincronización entre dos réplicas detecta un conflicto, trata de solucionarlo de acuerdo a reglas preestablecidas, pudiéndose requerir intervención manual en última instancia.

Ejemplos de sistemas de alta disponibilidad son la arquitectura *Gossip* (Landin et al, 1992), Bayou (Terry et al, 1995) [COU05 §15] y el sistema de ficheros Coda [SAT90].

3.6 Transacciones distribuidas

Las transacciones (o transacciones atómicas) se usan desde hace tiempo en el mundo de las bases de datos para controlar los accesos concurrentes. Son también de aplicación en sistemas de ficheros distribuidos y bases de datos distribuidas (transacciones distribuidas), donde cada nodo ejecuta una parte de la transacción. Hay que tener en cuenta que en el mundo de las transacciones priman los criterios de rendimiento por encima de otras consideraciones, como pueda ser la transparencia.

Una transacción es una secuencia de operaciones que modifica consistentemente el estado del sistema, de acuerdo a las siguientes propiedades[23]:

- **Atomicidad**. Una transacción o termina completa y satisfactoriamente (con acuerdo) o no tiene efecto alguno sobre el estado del sistema.
- **Consistencia**. La ejecución de una transacción preserva los invariantes del estado del sistema.[24]

[23] Conocidas en inglés por el acrónimo *ACID*.
[24] La propiedad de consistencia no está garantizada por el sistema transaccional, sino por el programador de las transacciones.

- **Serialización** (*Isolation*). La ejecución de una transacción es independiente de la ejecución concurrente de otras transacciones.

- **Permanencia** (*Durability*). Los efectos de una transacción sobreviven a fallos futuros.

El cliente especifica una transacción mediante una secuencia de operaciones que incluye un conjunto de primitivas específicas, que en parte dependen del tipo de aplicación a la que esté dirigido (memoria, ficheros, bases de datos). Podemos considerar el siguiente conjunto de primitivas abstractas:

- *Comenzar transacción*. Delimita el principio de una transacción.

- *Finalizar transacción*. Delimita el final de una transacción. Origina el intento de acuerdo.

- *Abortar transacción*[25]. Abandona la transacción. Requiere que se restaure el estado global definido antes del comienzo de la transacción.

- Operaciones adicionales para la manipulación de objetos, por ejemplo: *Leer de un objeto*, *Escribir en un objeto*.

En lo que sigue nos centraremos en el ámbito específico de las transacciones distribuidas.

3.6.1 Ejecución de transacciones distribuidas

Una transacción distribuida asegura la actualización consistente del estado global del sistema o la restauración del estado previo si la transacción no termina. Un sistema de transacciones distribuidas pretende garantizar las propiedades transaccionales pese al fallo de alguno de sus componentes. Estos sistemas, a diferencia de los basados en comunicación a grupos, asumen siempre un modelo de sistema síncrono; si un proceso no responde en un plazo prefijado, se considera que el proceso falla.

Consideraremos un conjunto de servidores replicados que realizan la transacción distribuida y que denominaremos **participantes**. Cuando un cliente envía una petición de acceso a un servicio transaccional, ésta es recibida por uno de los participantes (**iniciador**), que comunica la transacción y su inicio al resto de los participantes. Uno de los participantes, el **coordinador**, iniciará un protocolo para intentar alcanzar un **acuerdo** sobre el resultado de la transacción. Si no hay acuerdo, la transacción se **aborta** y se pone en marcha un mecanismo de **recuperación** para restablecer un estado estable.

Los participantes suelen estar especializados en funciones específicas. En unos sistemas el iniciador es siempre el mismo nodo; en otros, un nodo cualquiera.

[25] La transacción puede ser abortada por iniciativa del cliente, al ejecutarse esta primitiva, o por iniciativa de un servidor, como veremos.

Los participantes intercambian mensajes para realizar la transacción e intentar un acuerdo sobre el resultado. Consideraremos un algoritmo básico para ejecutar una transacción distribuida. Cada participante establece un voto (genéricamente SI o NO) sobre el resultado de la ejecución de la transacción en su nodo. El coordinador promueve un acuerdo a partir de los votos de los participantes para garantizar que todos actualicen o que no lo haga ninguno. El algoritmo para ejecutar la transacción es el siguiente:

- El iniciador envía a todos los participantes un mensaje de inicio con la transacción y un plazo de tiempo para realizarla.
- Cada participante (incluido el iniciador):
  1. Recibe el mensaje de inicio de la transacción.
  2. Ejecuta las operaciones especificadas en la transacción.
  3. Dependiendo del resultado, establece su voto, SI o NO, para la actualización del estado global.
  4. Ejecuta un protocolo de acuerdo atómico.

3.6.2 Acuerdo atómico

El objetivo del acuerdo atómico es garantizar que el sistema transaccional quede en un estado global consistente aun en presencia de fallos. Básicamente, un algoritmo de acuerdo atómico pretende que cada proceso correcto decida entre dos alternativas:

- *Acuerdo* (*commit*). En este caso, todos los procesos participantes en la transacción actualizan su estado.
- *Abortar* (*abort*). En este caso, ningún proceso participante en la transacción actualiza su estado.

La decisión posee las siguientes propiedades:

1. Todos los participantes que decidan lo harán en el mismo sentido.
2. Si un participante decide *Acuerdo*, todos los participantes votan SI.
3. Si todos los participantes votan SI y no hay fallos, todos los participantes deciden *Acuerdo*.
4. Cada participante decide como mucho una sola vez (las decisiones son irreversibles).

Un protocolo para resolver el acuerdo atómico requiere dos fases: en la primera fase, el coordinador recibe los votos de los participantes; en la segunda fase, el coordinador comunica la decisión a los participantes. El siguiente es un **protocolo de acuerdo atómico en dos fases** (*2PC*) genérico:

Fase 1 (votación)

- El coordinador envía a todos los participantes una petición de voto
- Cada participante espera el mensaje del coordinador. Si el mensaje no llega en un time-out establecido, el participante supone que el coordinador falla y decide *Abortar*
- Los participantes responden enviando el valor de su voto[26]
- Si todos los participantes (incluido el coordinador) responden SI, el coordinador decide *Acuerdo*; si no, el coordinador decide *Abortar*. Si algún participante no responde en el time-out establecido, el coordinador decide *Abortar*

Fase 2 (comunicación de la decisión)

- El coordinador envía a todos los participantes su decisión
- Cada participante espera un mensaje con la decisión del coordinador. Si el mensaje no llega en un time-out establecido, el participante puede iniciar un protocolo para intentar concluir la transacción con la ayuda de los otros participantes[27]
- Si el mensaje contiene la decisión de *Acuerdo*, la decisión del participante es *Acuerdo*; si no, la decisión del participante es *Abortar*

El algoritmo 2PC está expuesto a interbloqueos en el caso de fallo del coordinador durante la segunda fase, por lo que se proponen algoritmos más elaborados que evitan este problema. [MUL93 §6] ofrece una descripción más formal y exhaustiva de los algoritmos de acuerdo atómico, proporcionando un análisis de cómo se establecen los time-outs. Los algoritmos de acuerdo atómico son siempre síncronos. El acuerdo atómico es un problema de consenso, y como tal su resolución está sujeta a las restricciones que estudiaremos más adelante.

3.6.3 Mecanismos de recuperación

Un sistema de transacciones debe ser capaz de restablecer el estado anterior ante una decisión de *Abortar* (propiedad de atomicidad) y de reflejar permanentemente la actualización en caso de *Acuerdo* (propiedad de permanencia). Por otra parte, en bases de datos es habitual trabajar, por razones de rendimiento, sobre almacenamiento cache volátil. Ante un fallo, las actualizaciones no respaldadas en memoria permanente se perderían. Un

[26] Si un participante ha votado NO, ya ha decidido *Abortar*.

[27] Recuérdese que la decisión ya está tomada y no puede cambiarse. Incluso la decisión podría ser de *Acuerdo*.

mecanismo general es llevar un historial (*logging*) en almacenamiento permanente, donde, para cada transacción se guarda una **lista de intenciones** que refleja las operaciones a realizar sobre el conjunto de datos afectado con su estado actual.

En transacciones distribuidas es necesario además asegurar la recuperación de una transacción cuando el nodo falla durante la ejecución del algoritmo de acuerdo atómico. Se define un nuevo estado intermedio, *incierto*, que se almacena en el historial cuando un participante ha votado SI y no ha recibido la decisión del coordinador. Más información sobre este tema puede encontrarse por ejemplo en [COU05].

3.7 El problema del consenso

Problemas como la difusión con orden total, la gestión de la pertenencia a grupo, la elección de líder y el acuerdo atómico tienen en común el que se pueden especificar como problemas de **consenso**. En un modelo de consenso, se toma y se difunde una decisión después de recoger las propuestas de los procesos participantes. En términos abstractos, el problema del consenso se define en función de dos primitivas:

proponer*(v)*. El proceso propone un valor *v*.

decidir*(v)*. El proceso decide *v*.

El consenso posee las siguientes propiedades:

- **Terminación**. Todo proceso correcto tarde o temprano decide algún valor.
- **Validez**. Si un proceso decide *v*, entonces *v* es un valor propuesto por algún proceso.
- **Acuerdo**. Si un proceso correcto decide un valor *v*, entonces todos los procesos correctos tarde o temprano deciden *v*.

La validez asegura que, si no hay unanimidad, el valor decidido esté entre los propuestos. [28]

Se puede demostrar la equivalencia entre el problema del consenso y la difusión con orden total en un sistema sujeto a fallos. Los otros problemas citados también se pueden reducir[29] al problema del consenso[30].

---

[28] El enunciado de estas propiedades se puede modificar para representar modelos particulares. Así, se suele definir la propiedad de *integridad*, que introduce la posibilidad de decidir un valor especial no propuesto cuando no haya unanimidad. Por otra pare, la propiedad de acuerdo enunciada es *no uniforme*. Se habla de *acuerdo uniforme* cuando la decisión afecta a todos los procesos, no sólo a los correctos. Consúltese [MUL93 §5] para más detalles.

[29] La *reducción* de un problema A a otro B se define como la resolución de A en términos de B. Dos problemas son *equivalentes* cuando se reducen recíprocamente.

Según el resultado de imposibilidad de Fischer, Lynch y Paterson (1985), en sistemas asíncronos sujetos a fallo, el problema del consenso no se puede resolver con un algoritmo *determinista*. Esto implica la no resolución en este tipo de sistemas de los problemas reducibles al problema del consenso. La cuestión reside en que en un sistema asíncrono no se pueden establecer límites ni para la velocidad de los procesos ni para el retardo de comunicación de un mensaje, de modo que no es posible determinar si un proceso que tarda en responder está fallando.

Los sistemas síncronos no tienen este problema, ya que se puede determinar un límite de tiempo D para la transmisión del mensaje y en la velocidad relativa de los procesos (ningún proceso puede ejecutar más de un número finito S de pasos mientras el proceso más lento ejecuta 1 paso). De hecho, en este capítulo hemos presentado algoritmos síncronos para la elección de líder, la difusión con orden total y el acuerdo atómico.

El modelo asíncrono es el más general. Se puede pretender modelar cualquier sistema como síncrono introduciendo suposiciones acerca de la velocidad relativa de los procesos y los retardos de comunicación. El problema es dónde establecer estos límites. Si se es muy conservador, por ejemplo definiendo plazos muy largos, las aplicaciones se resentirían en cuanto a la latencia ante el fallo de algún nodo, lo que podría llegar a ser inaceptable. Si, por el contrario, se establecen límites muy estrictos, posiblemente el sistema asíncrono no se estará modelando bien.

En los últimos años se han estudiado alternativas al problema del consenso en sistemas asíncronos que consisten en evitar el resultado de imposibilidad, bien mediante soluciones no deterministas pero con un alto grado de certidumbre, bien introduciendo algún tipo de sincronía en el sistema. Por ejemplo, dentro de este segundo enfoque, Chandra y Tueg (1991) abordaron el problema extendiendo el sistema asíncrono mediante la introducción de **detectores de fallos** locales y no fiables, que monitorizan el estado de los procesos remotos y establecen hipótesis de fallo. Mediante la utilización de time-outs que se incrementan tras una sospecha de fallo, los detectores de fallos encapsulan la sincronía, lo que permite modelar el resto sistema como asíncrono. Cada proceso consulta la lista de procesos sospechosos proporcionada por su detector de fallos asociado y resuelve consenso en base a ella. Al no ser fiables, las sospechas de fallo pueden ser incorrectas, pero si se considera un modelo de sincronía con retardos desconocidos pero finitos, tarde o temprano (es decir, en un tiempo finito pero desconocido) todos los time-outs establecidos para los procesos correctos se respetarán.

[MUL93 §5] ofrece más detalles y referencia los trabajos sobre el tema.

[30] El acuerdo atómico es un problema más fuerte que el consenso y exige suposiciones adicionales.

3.8 Ejercicios

1 Comparar los algoritmos de exclusión mutua de acuerdo al número de mensajes que requiere el entrar y el salir de la sección crítica. Comparar el algoritmo de Ricart-Agrawala con el centralizado: ¿con qué número de procesos resulta rentable el de Ricart-Agrawala?

2 Comparar los algoritmos de elección de líder de acuerdo al número de mensajes intercambiados.

3 Dibujar un diagrama de tiempos que muestre un ejemplo de difusión con orden total no causal.

4 Considerar un servicio de listas de distribución sobre Internet y suponer que la difusión de los mensajes es fiable, pero sin garantía en cuanto al orden de entrega. Discutir cómo el reenvío (forwarding) sistemático de los mensajes proporciona difusión causal.

5 Una familia posee una cuenta corriente on-line compartida por sus miembros. El hijo de la familia, H, opera desde Donostia, usualmente realizando reintegros de la cuenta. Los padres, P, que viven en Bilbao, ingresan cantidades en la cuenta para financiar los estudios de informática del hijo. La entidad bancaria ofrece tres tipos de operaciones on-line:

ingreso *(cc, q): cc = cc + q*

reintegro *(cc, q): if (cc ≥ q) cc = cc – q; else ABORTAR*

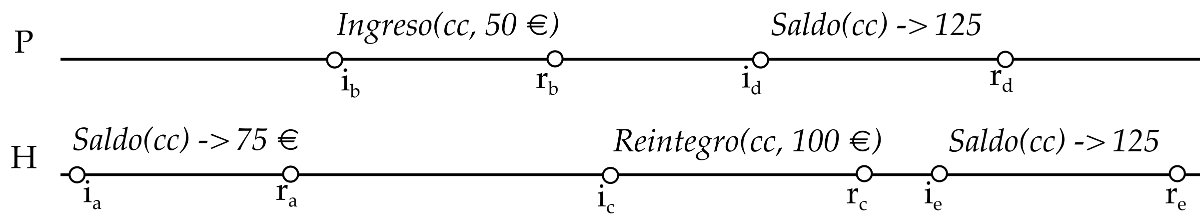
saldo *(cc): Devuelve el valor de cc*

Cada operación posee propiedades transaccionales. En particular, una vez modificado el valor de la cuenta corriente (cc), éste permanece.

A principios de mes, el hijo necesita dinero para pagar su parte de alquiler del piso, 100 €, y en la cuenta hay solamente 75 €, por lo que este telefonea a los padres para que le ingresen 50 €, lo que estos hacen de inmediato, comunicándoselo al hijo. Este, recibido el aviso, transfiere inmediatamente la cantidad del alquiler mediante una operación de reintegro. Después, consulta el saldo y, viendo que es de 125 €, deduce que la operación de reintegro ha abortado, por lo que sospecha que el sistema informático replicado que gestiona la cuenta on-line no cumple alguna de las propiedades que garantizan la semántica de consistencia deseable.

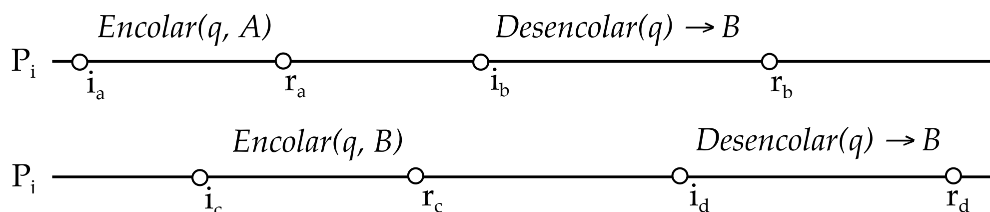
(a) Dado el cronograma adjunto, que muestra la secuencia de eventos asociados a las operaciones, ¿qué ejecuciones son legales?

(b) Respecto a la secuencia de eventos, ¿las ejecuciones del apartado anterior son secuencialmente consistentes? ¿son linealizables?



(c) Nos hemos enterado de que el servidor de cuentas on-line tiene dos réplicas. Como su comportamiento no es el esperado (la operación de reintegro no debiera haberse abortado), parece claro que no se gestiona mediante un esquema de replicación activa ni de replicación pasiva. Plantear cada esquema como una hipótesis y demostrar que con cada esquema los resultados mostrados en el cronograma no son posibles.

6 Dado el siguiente cronograma, en el que se especifican operaciones sobre una cola FIFO implementada sobre un sistema replicado con dos réplicas, ¿hay alguna ejecución legal posible para esta secuencia? Si la hay, ¿es secuencialmente consistente? ¿es linealizable? Encontrar una interpretación para este resultado.



7 En un esquema de replicación pasiva, analizar las diferentes circunstancias de fallo del servicio, precisando en qué caso(s) la transparencia a los fallos, desde el punto de vista del cliente, se ve comprometida. ¿Cómo se comporta ante esos fallos un servicio basado en replicación activa?

8 Considerar las posibles optimizaciones del esquema de replicación pasiva: (a) las operaciones de consulta son tratadas por los secundarios (a1) las de un mismo cliente por el mismo secundario, (a2) indistintamente; (b) el primario responde al cliente antes de recibir los acks de los secundarios. Para cada combinación posible de estas optimizaciones, analizar qué criterio de consistencia se garantiza y en qué condiciones.