

Sincronización, tolerancia a fallos y replicación

Leandro Navarro Moldes
Joan Manuel Marquès i Puig

P07/M2106/02840



Universitat Oberta
de Catalunya

www.uoc.edu

Índice

Introducción	5
Objetivos	7
1. La observación de un sistema distribuido	9
1.1. Observación y relatividad	9
2. Tiempos y relojes	11
2.1. La medida del tiempo	11
2.2. ¿Cómo ajustar el reloj?	12
2.3. Relojes lógicos	15
2.4. Relojes vectoriales	17
3. Exclusión mutua	19
3.1. Algoritmo centralizado	19
3.2. Algoritmo descentralizado	19
3.3. Algoritmo basado en un anillo	20
3.4. Algoritmo distribuido	21
3.5. Comparación de algoritmos	22
4. Algoritmos de elección	24
4.1. El algoritmo de Bully	24
4.2. Algoritmo anillo	24
4.3. Elección en entornos sin hilo	25
5. Tolerancia a fallos	27
5.1. Comunicación fiable en grupo	30
5.2. Entrega de mensajes	32
5.3. Transacciones en presencia de fallos	34
6. Consenso	36
6.1. Algoritmo de Paxos	37
7. Conceptos básicos de replicación	40
7.1. <i>Single</i> frente a <i>multi-masters</i>	40
7.2. Sistemas síncronos frente a sistemas asíncronos	41
7.2.1. Síncronos	41
7.2.2. Asíncronos.....	42
7.3. Modelos de replicación síncronos	42
7.3.1. Replicación pasiva	42
7.3.2. Replicación activa	43
7.3.3. Basados en quórum	44

7.4. Algoritmos para replicación síncrona	45
7.4.1. Reserva en dos fases	45
7.4.2. Confirmación distribuida.....	45
7.4.3. Confirmación en dos fases	46
7.4.4. Confirmación en tres fases	48
7.5. Replicación optimista	50
7.5.1. Pasos seguidos por un sistema optimista para llegar a un estado consistente	50
7.5.2. Algunos ejemplos de sistemas optimistas	52
Resumen	54
Actividades	57
Ejercicios de autoevaluación	57
Solucionario	58
Glosario	59
Bibliografía	60

Introducción

En este módulo didáctico se describen los problemas que presenta y las ventajas que ofrece un sistema informático distribuido formado por procesos y/o máquinas que trabajan de manera independiente, y que tan sólo se comunican mediante el intercambio de mensajes por una red que puede retardar, desordenar, duplicar o perder los mensajes.

Un sistema distribuido está formado por multitud de componentes que interactúan entre sí. El objetivo del *software* intermediario (*middleware*) es tratar con la red y presentar un modelo abstracto de programación a las aplicaciones sencillo y completo, ocultando los inconvenientes y ofreciendo servicios para aprovechar las ventajas. Aunque su complejidad es mayor que la suma de la complejidad de cada componente, es deseable que el sistema no falle cuando falla cualquier componente (fragilidad), sino que funcione mientras algún componente funcione (robustez o tolerancia a fallos).

Un sistema distribuido formado por varias máquinas puede funcionar de manera coordinada como si tuviera una capacidad de proceso suma de la capacidad de cada componente.

El coste de un sistema de capacidad N veces superior a la capacidad de una sola máquina de capacidad media puede costar N veces el coste de la máquina media, mientras que una sola máquina de capacidad N puede ser imposible de construir o puede costar demasiado (C^N).

En realidad, el razonamiento anterior puede aplicarse tanto a la capacidad de procesar peticiones como a la capacidad de tolerar fallos: con sistemas distribuidos diseñados adecuadamente, se pueden construir sistemas más rápidos y que funcionan durante más tiempo que cada uno de sus componentes.

El primer problema que existe en un sistema distribuido es que no nos es posible hacer una foto “instantánea” o una película “global” de todo el sistema para conocer su estado o sacar conclusiones de cómo ha ocurrido algo. El observador es también un proceso y recibe mensajes como cualquier otro componente de un sistema distribuido. Se verá que ocurren fenómenos similares a los que describe la teoría de la relatividad.

El segundo problema es que el tiempo se mide localmente en cada lugar, no está sincronizado globalmente y puede derivar (acelerarse o retrasarse respecto a otros): no hay un reloj global. Se podrían recibir mensajes con marcas de tiempo que parecen venir del futuro o secuencias de mensajes que parecen tener un orden extraño a la vista de las marcas de tiempo que traen de su origen.

Hay protocolos para sincronizar mutuamente los relojes de cada computador de un sistema distribuido mediante el intercambio de mensajes y garantizar que con “relojes lógicos” se puede certificar que los mensajes se entregan a las aplicaciones respetando las relaciones de causalidad (los mensajes llegan en el orden en el que han ocurrido si uno ha podido influir en el otro).

El tercer problema está en cómo caracterizar y tratar un sistema en el que algún componente pueda fallar o algún mensaje pueda perderse, o en el que la demanda de un servicio pueda requerir el trabajo de varios componentes de manera coordinada. Se presentan las formas en las que un sistema puede comportarse en presencia de fallos.

El cuarto problema es cómo garantizar que la compartimentación de recursos se realice de manera coordinada. Se presentarán unos cuantos algoritmos que permiten gestionar este acceso.

Otro aspecto que trataremos son los algoritmos de elección, que son muy importantes en sistemas distribuidos. Muchos algoritmos necesitan que un proceso actúe como coordinador, iniciador o desarrolle un papel especial. En estos casos da igual quién lo haga, pero es necesario que alguien lo haga.

El quinto problema consiste en pensar un modelo de programación que permita diseñar aplicaciones en las que diferentes componentes cooperan para proveer un servicio más rápidamente, con una disponibilidad mayor. Éste es el modelo de comunicación en grupo.

En el módulo también se presentan los conceptos básicos relacionados con la replicación en sistemas distribuidos. La replicación permite aumentar la disponibilidad y el rendimiento de los sistemas distribuidos. También contribuye a mejorar la escalabilidad. Esta replicación puede ser síncrona –las actualizaciones se aplican a todas las copias del objeto como parte de la operación de actualización– o asíncrona –como parte de la operación de actualización no hace falta que se actualicen todas las copias del objeto. Posteriormente, se hace llegar la actualización al resto de réplicas.

Objetivos

Los objetivos de este módulo didáctico son los siguientes:

- 1.** Conocer los problemas y conceptos básicos para observar sistemas distribuidos (tiempo, orden, causalidad, fallos, grupos).
- 2.** Conocer las ventajas que se pueden obtener de explotar las “aparentes” debilidades (sincronización, tolerancia a fallos, alto rendimiento).
- 3.** Conocer las posibilidades que ofrecen algunos entornos para facilitar la programación, presentando el sistema en formato más tratable (comunicación en grupo).
- 4.** Ser capaces de elegir las características relacionadas con la distribución para organizar un sistema que funciona en Internet.
- 5.** Conocer los conceptos básicos de replicación en sistemas distribuidos, especialmente en sistemas optimistas.
- 6.** Conocer algunos algoritmos distribuidos populares para solucionar los aspectos básicos relacionados con la construcción de sistemas distribuidos.

1. La observación de un sistema distribuido

Un sistema distribuido está formado por personas, máquinas, procesos, “agentes” situados en *lugares distintos*. Puede entenderse de manera abstracta como un conjunto de procesos que *cooperan* para solucionar un problema, intercambiando mensajes. Cada proceso es autónomo y a menudo asíncrono: funciona a su propio ritmo. Los mensajes pueden sufrir retrasos arbitrarios o perderse, y no hay un reloj global.

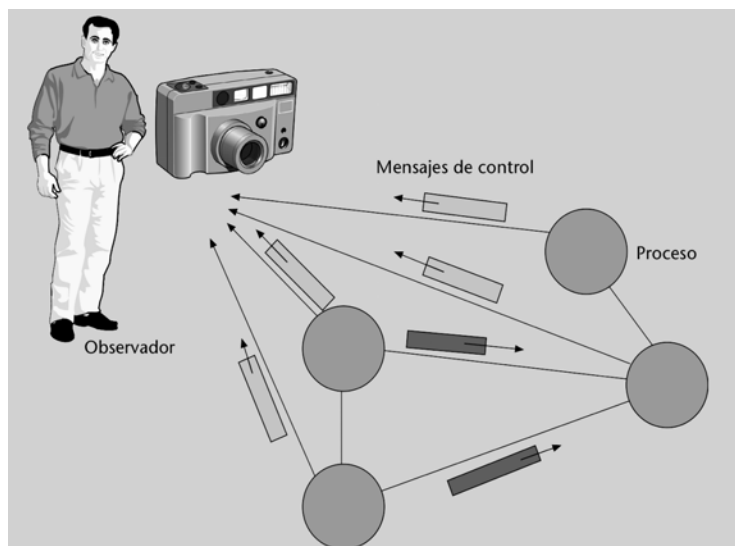
Por ejemplo, un computador con varios procesadores no encajaría en este modelo: hay un reloj único, el sistema es síncrono, los retardos de los mensajes son fijos.

Para tener una idea clara de qué ocurre en un sistema distribuido y de cómo programarlo para aprovechar sus ventajas, primero es necesario ver la forma en la que se puede observar un sistema distribuido y constatar la dificultad de llegar a una conclusión sobre el estado en un cierto momento o el orden en el que ocurrieron las cosas, debido a que el observador está sujeto también a la asincronía del sistema. Ocurre como en el mundo físico, que aparenta regirse por las leyes de Newton, y en realidad se rige por las leyes de la teoría de la relatividad de Einstein. El problema aquí es que la velocidad de propagación de cada mensaje es variable.

1.1. Observación y relatividad

Para observar una computación distribuida hay que recibir mensajes de control de todos los procesos y construir una “imagen” del sistema, pero los mensajes de control tienen distintos tiempos de propagación: varios procesos “nunca” se pueden observar a la vez y, por tanto, no se pueden hacer afirmaciones sobre el estado global.

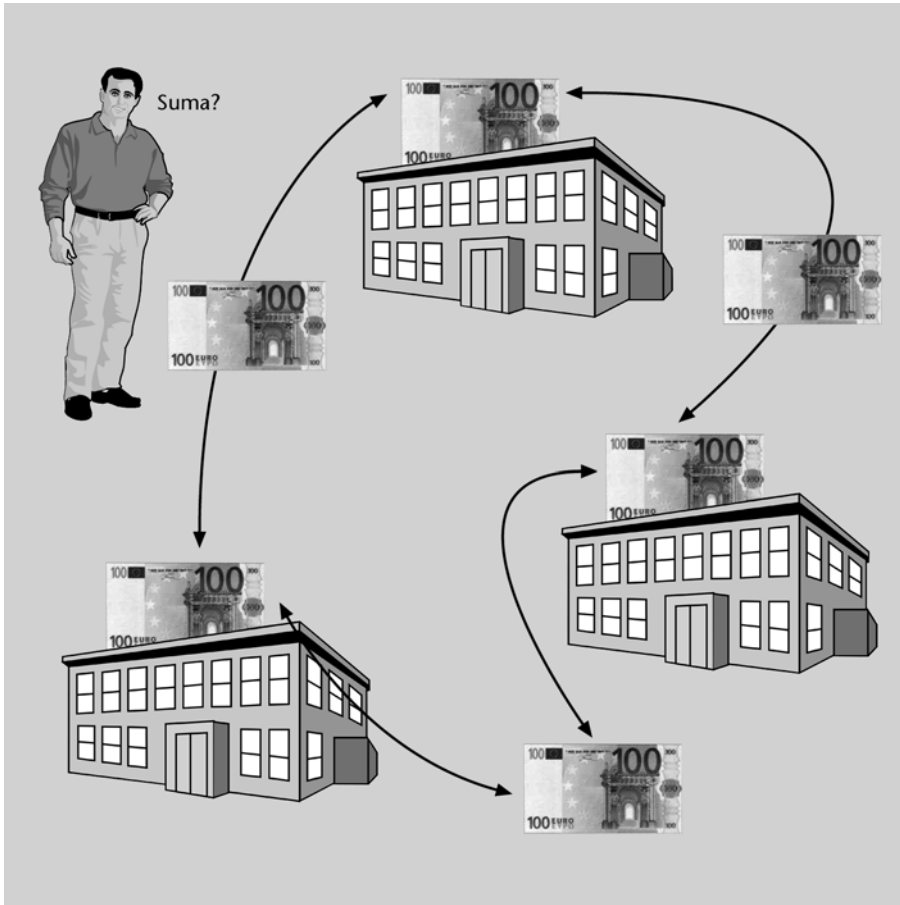
Figura 1



La observación de un mensaje se hace como un proceso más del sistema distribuido.

Por ejemplo, cuando se mira el cielo en una noche clara, se está observando una imagen “distorsionada” del universo: vemos la luz que emitieron hace muchos años las estrellas, más años atrás cuanto más lejos de nosotros estén. Es difícil hacer afirmaciones sobre el estado del universo en este momento mirando al cielo.

Figura 2



Contar el valor global del número de billetes en circulación.

Otro ejemplo: si quisiéramos contabilizar cuánto dinero existe en total (o sólo el dinero almacenado en bancos), tendríamos que contabilizar el dinero que hay en cada uno. Se podría hacer enviando un mensaje a todos a la vez pidiendo que nos devolvieran el importe total en sus cuentas. Mientras contamos, el dinero fluye y nuestro mensaje llega a cada banco en tiempos distintos. No podríamos calcularlo bien, pues podríamos haber dejado de contar o haber contado dos veces ciertas cantidades. ¿Por qué? No hay una vista global (una “foto” global instantánea: sólo un observador omnipresente lo podría saber...).

El efecto “relativista” de la gravedad sobre la luz no es nada comparado con lo que ocurre en Internet todos los días: la velocidad de propagación es mucho menor, los datos que circulan forman colas y sufren retrasos para atravesar routers, pueden perderse porque una cola rebosa o un bit cambia de valor por error, pueden desordenarse, tardan en ser procesados por la carga del servidor, pueden ser almacenados a la espera de ser reenviados, etc.



Figura 3. Carta de Albert Einstein a G.E. Hale de 1913 para pedirle que midiese el efecto de la gravitación en la luz de una estrella. No pudo responder hasta que finalizó la Primera Guerra Mundial.

2. Tiempos y relojes

El tiempo es una propiedad fundamental en sistemas distribuidos. Medir el tiempo y saber con la mayor exactitud el momento en el que ocurrió un evento: es necesario sincronizar el reloj del computador con una fuente de referencia externa.

Una definición de tiempo

Let us consider first what we mean by time. What is time? It would be nice if we could find a good definition of time. Webster defines "a time" as a "a period" and the latter as "a time" which doesn't seem to be very useful. Perhaps we should say: "Time is what happens when nothing else happens." Which also doesn't get us very far. Maybe it is just as well if we face the fact that time is one of the things we probably cannot define (in the dictionary sense), and just say that it is what we already know it to be: it is how long we wait!

Richard Feynman

Cuando se habla del *tiempo psicológico*, hay que diferenciar tres tipos de conceptos: el primero trata orden y simultaneidad; el segundo, intervalos de tiempo y duración; y el tercero es la experiencia que nos permite distinguir entre pasado, presente y futuro.

2.1. La medida del tiempo

Un computador es un sistema síncrono: está regido por una señal de reloj que indica el transcurso del tiempo. En realidad, se trata de un contador de sucesos: se mide el tiempo en el que ocurre algún fenómeno con regularidad.

Hasta 1955, el patrón científico del tiempo, el segundo, se basaba en el periodo de rotación terrestre, y se definía como 1/86.400 del día solar medio. Cuando se comprobó que la velocidad de rotación de la Tierra, además de ser irregular, estaba decreciendo gradualmente, se hizo necesario redefinir el segundo.

En 1955, la Unión Astronómica Internacional definió el segundo como 1/31.556.925,9747 del año solar en curso el 31 de diciembre de 1899. La definición de segundo desde 1967 es: 1 segundo = 9.192.631.770 periodos de la radiación correspondiente a la transición entre dos niveles hiperfinos del estado fundamental del átomo de cesio Cs¹³³.

Los computadores suelen utilizar un reloj de cuarzo (inventado en 1927): un dispositivo mide oscilaciones gracias al efecto piezoeléctrico. Su precisión es elevada: 10^{-6} , es decir, 1 segundo cada 10^6 segundos = cada 11,6 días.

Para medidas de tiempo de referencia más precisa se usan relojes atómicos de cesio que tienen una precisión de 10^{-13} (10^7 veces más preciso que un reloj de cuarzo).

En $1/1.000$ segundo = 1 ms:

- En el sistema de posicionamiento global (GPS), las referencias de tiempo se obtienen a partir de tres o cuatro referencias de satélites con relojes atómicos con una precisión de 1 ms.
- ¡Un reloj de cuarzo (10^{-6}) puede desviarse 1 ms cada 20 minutos!
- La luz, a 3^8 metros/s, viaja a 300 km/ms.
- ¡Un computador con un procesador de 1 GHz (10^9) puede ejecutar hasta 1 millón de instrucciones en 1 ms!

El tiempo universal coordinado (UTC) es un estándar internacional de tiempo. Se basa en el tiempo atómico y se emite por emisoras de radio de onda corta y por satélite, como el sistema de posicionamiento global (GPS) de EE.UU. o pronto el sistema europeo Galileo. Los receptores permiten obtener el tiempo con precisiones de 0,1 a 10 ms.

La consecuencia es que en una red de computadores cada computador tiene un reloj propio, con valor temporal que varía de manera distinta y que requiere un ajuste continuo para que todos los computadores tengan un valor de tiempo aproximadamente igual. Es necesario tener en cuenta que el reloj de un computador puede acelerarse o ralentizarse, pero no puede ni retroceder ni dar saltos, pues se podrían producir problemas (si cierto programa se ejecuta a una hora concreta, al retroceder el reloj el programa se podría ejecutar dos veces, y si el reloj da un salto, podría no ejecutarse).

2.2. ¿Cómo ajustar el reloj?


Puesto que cada reloj da marcas de tiempo a velocidades ligeramente distintas, tienen deriva: velocidad de cambio respecto al reloj ideal por unidad de tiempo. Y puesto que esta deriva puede cambiar con la temperatura, humedad, etc., es necesario sincronizar el reloj con una fuente de referencia o, en su falta, con otros computadores cercanos.

Por medio de intercambiar mensajes, los relojes de varias máquinas se pueden coordinar entre éstas o con una máquina de referencia: se envían mensajes pidiendo la hora. Teniendo en cuenta el tiempo de ida y vuelta del mensaje, se puede ajustar el reloj local, sin dar saltos, acelerándolo o ralentizándolo durante un periodo de tiempo (hacerlo progresar más despacio pero sin hacerlo retroceder), hasta que incorpore el ajuste.

Con los relojes atómicos...

... se han descubierto fluctuaciones de 10^{-3} s en el periodo de rotación de la Tierra, debidas a cambios en las mareas, las corrientes marinas y atmosféricas, fenómenos como "el Niño", las corrientes de convección del núcleo terrestre, las erupciones volcánicas y los terremotos.

En el año 2000 se llegó a precisiones de 10^{-14} con relojes atómicos de rubidio refrigerados hasta cerca de 0 K con láser, y se piensa conseguir 10^{-16} : una incertidumbre de 1 segundo en mil millones de años, con la misma técnica en microgravedad a bordo de la Estación Espacial Internacional (ISS).

Esto sirve para que puedan funcionar correctamente algunas aplicaciones que necesitan que los relojes de los computadores que participan estén sincronizados. Por ejemplo, si un servidor web con la hora atrasada modifica un fichero HTML, el cambio podría pasar desapercibido para el cliente que leyó la página web y al cabo de un rato vuelve a visitarla. 

Si $H(t)$ es la hora que indica el reloj de un computador y se le debe añadir un ajuste de $\delta(t)$ segundos para llegar al valor deseable $S(t) = H(t) + \delta(t)$.

$\delta(t)$, como puede verse en la gráfica, es una función lineal del reloj del computador: $\delta(t) = a \cdot H(t) + b$

Por tanto, $S(t) = (1 + a) \cdot H(t) + b$

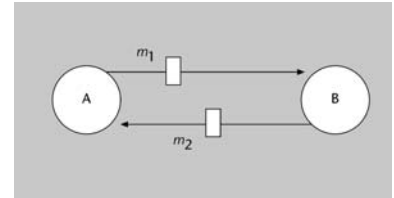
Si $S(t) = T_{despl}$ cuando $H(t) = h$ en $t = T_{real}$, y si se debe ajustar el reloj en N pasos de reloj,

$$T_{despl} = (1 + a)h + b, T_{real} + N = (1 + a)(h + N) + b$$

Por tanto, los valores para ajustar el reloj del computador una diferencia $T_{real} - T_{despl}$ durante N ciclos de reloj son: $a = (T_{real} - T_{despl})/N$, $b = T_{despl} - (1 + a)h$

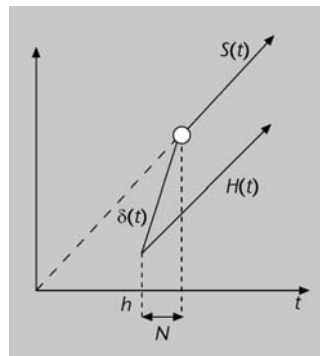
Método Christian para sincronizar relojes intercambiando mensajes (1989):

Figura 4



Ajuste del reloj progresivo en N pasos.

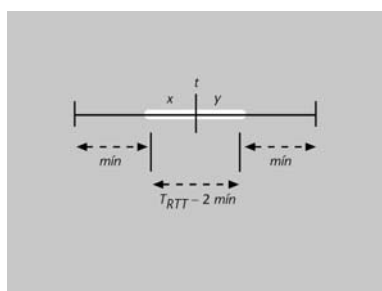
Figura 5



Intercambio de mensajes para pedir marca de tiempo a B.

A envía un mensaje m_1 pidiendo una referencia de tiempo a B; la respuesta llega en T_{IV} (tiempo de ida y vuelta) y m_2 contiene el valor de tiempo t_B que tenía B cuando ha enviado el mensaje, T_{TRANS} atrás. $T_{TRANS} = mín. + x$; con $x \geq 0$ *mín* puede llegar a estimarse mediante medidas históricas, idealmente cuando la red no tenga tráfico, como el valor mínimo de T_{TRANS} .

Figura 6



Márgenes de variación de la medida de tiempo según el retardo.

Puede estimarse en A que la hora actual según B es $t_A = t_B + T_{IV}/2$. Cuanto menor es T_{TRANS} , mejor es la estimación, ya que $T_{TRANS} = (mín + x) + (mín + y)$ y, cuanto menor es T_{TRANS} , x e y tienden a ser más insignificantes.

La precisión es: $\pm(T_{RTT}/2 - mín)$

Por ejemplo, en una red local en la que el tiempo de respuesta varíe entre 1-10 ms se puede conseguir que el desajuste entre las máquinas sea de pocos milisegundos, y que la deriva del conjunto respecto a una referencia externa sea del orden de 10^{-5} .

En realidad, si sólo hay un servidor puede ser problemático en caso de fallos, y por este motivo se suelen usar mecanismos de sincronización con varias referencias de tiempo. Así es como se hace en los algoritmos del protocolo de tiempo en red (NTP, *network time protocol*) o Berkeley (Unix BSD).

En el algoritmo de Berkeley, una máquina pregunta a otras su hora y estima la hora local descontando la propagación por la red como hace el algoritmo de Christian. Calcula una media con tolerancia a fallos (eliminando la influencia de medidas erróneas) y envía a cada máquina del conjunto el ajuste de reloj que necesita.

El protocolo de tiempo en red NTP (RFC 1305, RFC 2030) permite que una máquina pueda ajustar su hora mediante Internet. Las máquinas están organizadas en forma jerárquica, y tienen tres modos de operación.

- *Multicast* o difusión selectiva. El servidor difunde periódicamente información de tiempo.
- *Procedure-call* o invocación: similar al método de Christian.
- Simétrico: se mantiene una asociación entre servidores que intercambian información de tiempo.

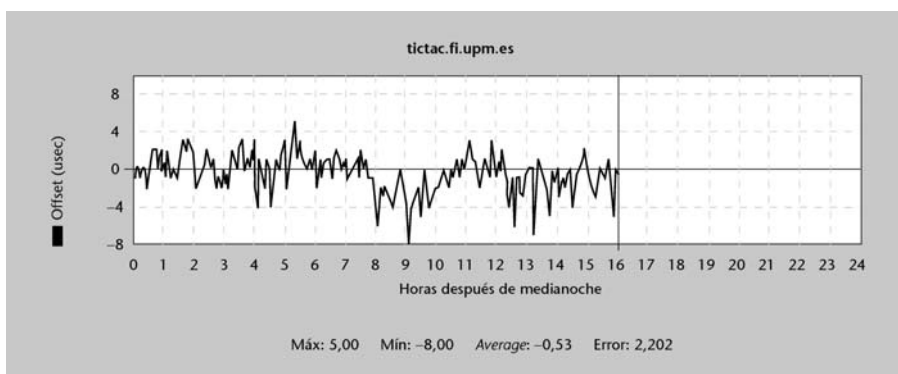
El ajuste de relojes entre máquinas de una red permite que la información de tiempo sea aceptablemente válida entre las máquinas que están sincronizadas. Por ejemplo, cuando se comparte un disco en red y las máquinas que comparten tienen relojes desajustados, se pueden producir situaciones extrañas si al modificar un fichero en una máquina con el reloj retrasado las demás máquinas observan que el fichero aparentemente ha retrocedido en el tiempo.

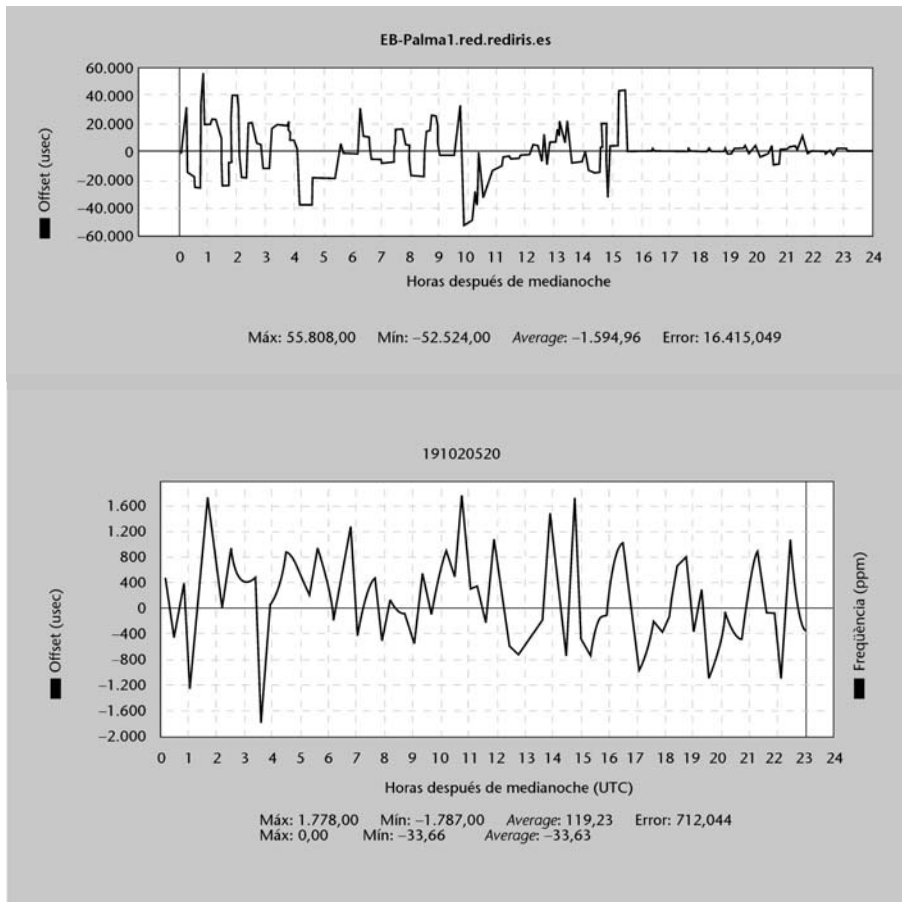
Sincronización de tiempo

En la red académica RedIRIS hay un grupo de trabajo sobre tiempo. Es interesante visitar su web:

<http://www.rediris.es/gtiris-ntp/drafts/>

Figura 7





Variación de distintos servidores NTP en microsegundos a lo largo de un día (20/5/2002).

- "stratum 1": tictac.fi.upm.es, sincroniza con el GPS.
- "stratum 2": eb-palma1.red.rediris.es, sincroniza con servidores "stratum 1" por la red.
- "stratum 3": hora.usc.es, sincronizados por la red con 3 servidores de "stratum 2".

2.3. Relojes lógicos

La teoría de la relatividad muestra que la velocidad de la luz es constante para cualquier observador y que la percepción del tiempo que transcurre es local. El paso del tiempo entre dos eventos desde la Tierra será distinto que dentro de una nave espacial, especialmente cuando hay aceleración: no existe un tiempo físico absoluto al que preguntar para medir intervalos de tiempo.

Por tanto, en lugar de usar la hora (el reloj físico) para determinar el orden relativo de ocurrencia de dos eventos, Leslie Lamport, en 1978, propone los relojes lógicos. También propone distinguir entre relación de precedencia (*happened-before* o causalidad potencial: \rightarrow) y concurrencia (\parallel) entre eventos.

Con un sistema distribuido,...

... puede no haber un reloj para marcar eventos con suficiente precisión que nos permita saber en qué orden ocurrieron ciertos eventos o si ocurrieron simultáneamente. Un mensaje enviado a varios destinatarios seguramente nunca llegará a todos al mismo tiempo. Incluso en una misma máquina, el sistema operativo y la gestión de memoria, disco y procesos pueden introducir retardos inesperados.

Un reloj lógico es un contador de eventos (un número natural) que siempre aumenta y que no tiene relación con el reloj físico. Si un sistema distribuido está formado por procesos separados, y cada proceso P_i tiene un reloj lógico L_i que

se usa para marcar el tiempo (virtual) en el que se ha producido un evento, si el evento e sucede en el proceso P_i , $L_i(a)$ es el valor del reloj lógico para este evento.

Observación causal

El problema no es nuevo:
 “Cuando un espectador observa un batallón haciendo ejercicios desde cierta distancia, ve a las personas correr antes de oír la orden de mando, pero por su conocimiento de las relaciones causales sabe que los movimientos son el resultado de la orden, y que objetivamente la última debe haber precedido a la primera.”
 Christoph von Sigwart (1830-1904). *Logic* (1889).

Figura 8

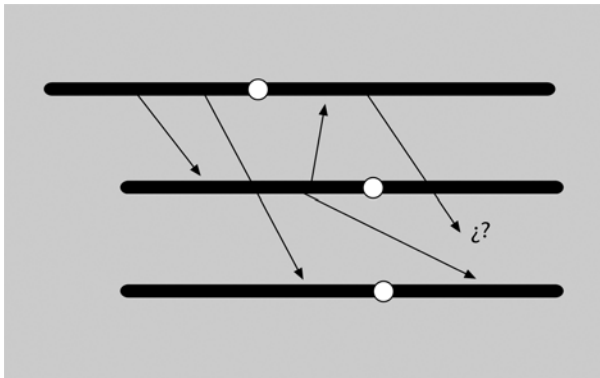


Diagrama de una computación distribuida. Cada línea es un proceso que envía o recibe mensajes en forma de flechas. La inclinación indica la velocidad a la que va el mensaje, que puede variar para cada uno dependiendo de la carga de la red y las máquinas. El mensaje entre ¿? puede ser consecuencia de los anteriores: tiene una posible relación de orden con los eventos de envío o recepción de mensajes en este proceso. ¿Observas algún problema de causalidad?

Cuando se envía un mensaje, se incluye el valor de reloj Lamport de este proceso. Esto permite que en cada proceso se sepa el valor del reloj que había en el proceso origen cuando se envió el mensaje.

El algoritmo es el siguiente:

- L_i se incrementa en cada evento en P_i : $L_i := L_i + 1$.
- Cuando un proceso P_j recibe un mensaje que incluye el reloj (m,t) , se calcula $L_j := \max(L_j, t)$ y luego se aplica el paso anterior para marcar la recepción del mensaje.

Puede verse que si $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Las relaciones importantes son las siguientes.

- Precedencia (*happened-before* o \rightarrow) o causalidad potencial.
 - Si dos eventos e y e' se dan en el mismo proceso P_i , el orden es claro para todo el sistema: $e \rightarrow e'$.
 - El envío de un mensaje ocurre antes que la recepción: $\text{envío}(m) \rightarrow \text{recepción}(m)$.
 - Si e, e' y e'' son eventos que $e \rightarrow e'$ y $e' \rightarrow e''$, entonces: $e \rightarrow e''$.
- Concurrencia (\parallel) entre eventos. Cuando dos eventos no están relacionados por \rightarrow (no hay una relación causal).

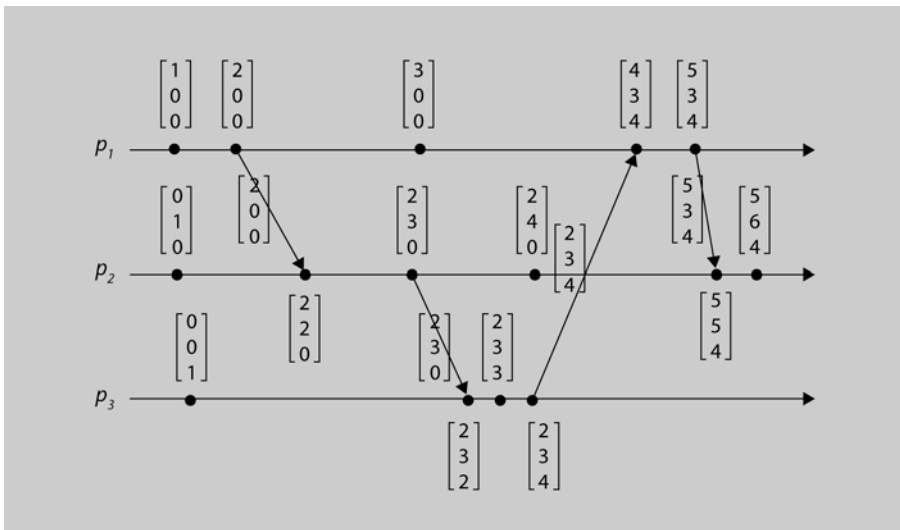
Resulta que la ordenación causal de eventos captura en muchos casos la información esencial para describir una ejecución.

El inconveniente es que hay un reloj Lamport por cada proceso, y comparando su valor en dos mensajes que nos llegan no podemos concluir si uno precede al otro o son concurrentes. Para esto se inventaron los relojes vectoriales.

2.4. Relojes vectoriales

Un reloj vectorial refleja la evolución de todo el sistema, no sólo de un proceso. Su dimensión, el número de elementos del vector, coincide con el número de procesos del sistema distribuido. De esa forma, cada proceso lleva cuenta de su punto de vista del estado global: todo lo que sabe de los demás procesos y de sí mismo gracias a los mensajes que recibe.

Figura 9



Evolución del vector tiempo en un sistema de dimensión 3.

En un sistema con n procesos, cada proceso P_i mantiene un vector $V_i[1..n]$ con su vista de todo el sistema según la información recibida a través de mensajes: la historia causal. Justo antes de que se produzca un evento en un proceso, su propio componente se incrementa:

$$V_i[i] = V_i[i] + 1.$$

Cuando se envía un mensaje, el vector viaja con él: $(m, V[i])$.

Cuando se recibe un mensaje, primero se funden el vector que viene en el mensaje con el vector local, tomando el mayor valor de cada componente:

$$\text{Para } k = 1..n : V_i[k] = \max(V_i[k], V[k])$$

En segundo lugar, se incrementa el componente propio y finalmente se entrega el mensaje.

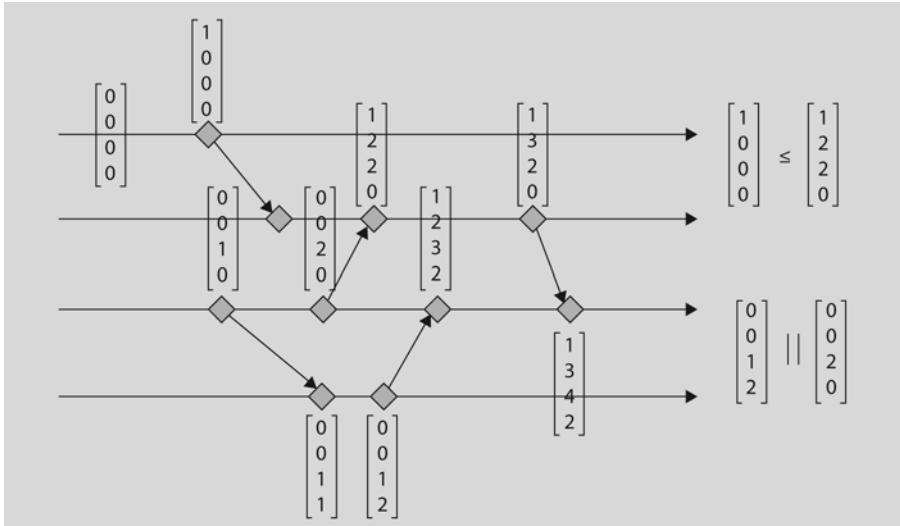
Los relojes vectoriales se pueden comparar entre sí, comparando elemento a elemento. Así:

$V_h = V_k$ si en cada elemento ocurre que $V_h[x] \leq V_k[x]$

$V_h < V_k$ si en cada elemento ocurre que $V_h[x] \leq V_k[x]$ y alguno $V_k[x] < V_h[x]$

$V_h \parallel V_k$ si ni $(V_h < V_k)$ ni $(V_k < V_h)$

Figura 10



Relojes vectoriales: las líneas de puntos indican la frontera de la historia causal.

Comparar vectores es como comparar la historia causal (lo que sabemos del resto), por tanto la relación $V_h \leq V_k$ entre vectores es equivalente a la relación $e_h \rightarrow e_k$ entre eventos y también $V_h \parallel V_k$ entre vectores es equivalente a la relación $e_h \parallel e_k$. Es decir, que la propiedad relevante de los relojes vectoriales, que no tienen los relojes de Lamport es que comparando vectores podemos saber si dos eventos están relacionados causalmente o son concurrentes.

En "Logical time: capturing causality in distributed systems" Raynal y Singhal se plantean los relojes matriciales: vectores de vectores que funcionan de forma similar. El reloj matricial permite conocer el vector de cada proceso y por tanto lo que conoce cada proceso de todo el sistema. Por lo que permite determinar qué eventos ya son conocidos o recibidos por todos los procesos, eventos que no van a volver a circular o reclamarse y se pueden olvidar.

Bibliografía complementaria

M. Raynal; M. Singhal (febrero, 1996). "Logical time: capturing causality in distributed systems". *Computer* (vol. 29, núm. 2, pág. 49-56).

3. Exclusión mutua

Los procesos distribuidos a menudo deben coordinarse. Si esta coordinación implica compartir recursos, será necesario algún mecanismo para evitar interferencias y asegurar la consistencia en el acceso a estos recursos. En este apartado veremos algunos de los algoritmos distribuidos más populares.

3.1. Algoritmo centralizado

La manera más sencilla de conseguir exclusión mutua es que un proceso actúe de coordinador y dé permisos de acceso a una sección crítica. Para acceder a una sección crítica, un proceso envía un mensaje de petición al coordinador y espera una respuesta de éste. Si en este momento no hay ningún proceso accediendo a la sección crítica, el coordinador responde inmediatamente y el proceso puede entrar. En caso de que haya un proceso en la sección crítica, el coordinador no contesta y encola la petición. Cuando el proceso que está en la sección crítica sale, se envía un mensaje al coordinador informándolo. De esta manera, el coordinador ya puede dar acceso a la sección crítica al proceso siguiente.

Si la cola de procesos esperando no está vacía, el coordinador elige la entrada que hace más tiempo que está en la cola, la saca y responde a este proceso (recordemos que el proceso había hecho una petición y estaba bloqueado esperando respuesta).

En esta solución, el coordinador es un punto centralizado de fallo. Si éste falla, todo el sistema deja de funcionar. Un proceso bloqueado en la petición de acceso a la sección crítica no es capaz de distinguir entre un fallo del coordinador y estar esperando que éste le dé acceso a la sección crítica. Además, en un entorno de gran escala, un único coordinador puede ser un cuello de botella. Sin embargo, los beneficios provenientes de la simplicidad del mecanismo a menudo pesan más que estos posibles inconvenientes.

3.2. Algoritmo descentralizado

Tener un único coordinador es una aproximación muy pobre. Un modo de gestionar el acceso a un recurso de manera descentralizada es usar un algoritmo basado en votaciones que se ejecute en un sistema basado en una DHT. La idea es la siguiente: se supone que cada recurso está replicado n veces. Cada réplica tiene su coordinador que controla el acceso de procesos concurrentes.

Lectura recomendada

En el artículo siguiente encontraréis una comparación entre diferentes algoritmos para conseguir exclusión mutua:

P. Saxena; J. Rai (2003, mayo). "A survey of permission-based distributed mutual exclusion algorithms". *Computer Standards and Interfaces* (vol. 25, núm. 2, pág. 159-181).

Lectura recomendada

S. D. Lin; Q. Lian; M. Chen; Z. Zhang (2004). "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems". Proc. Third International Workshop on Peer-to-Peer Systems. *Lecture Notes in Computer Sciences* (vol. 3279, pág. 11-21).

Cuando un proceso quiere acceder a un recurso sólo debe conseguir que una mayoría m de coordinadores esté de acuerdo. Esta mayoría m ha de ser superior a $n/2$, es decir, necesita que más de la mitad de réplicas voten a favor de que pueda acceder. Cada uno de los coordinadores, al recibir la petición de acceso al recurso, contesta al proceso que realiza la petición con un voto para que pueda acceder o con una negación de acceso si ya ha dado permiso a algún otro proceso para acceder al recurso.

Esta solución es menos vulnerable a fallos que la centralizada, ya que soporta fallos puntuales de coordinadores. El problema se presenta cuando el proceso coordinador se recupera del fallo. Durante el fallo, el proceso habrá perdido la información asociada a los accesos que había concedido y al recuperarse puede dar de manera incorrecta acceso al recurso a otro proceso. En el artículo mencionado han llegado a la conclusión de que si los procesos que fallan se recuperan rápidamente, la probabilidad de que k procesos entre los m coordinadores que habían participado en una votación reinicien es muy pequeña y, en cualquier caso, mucho menor que la disponibilidad del recurso que hay que proteger.

La implementación que proponen se basa en una DHT donde cada recurso está replicado n veces. Si el recurso está identificado de manera única por *nombre* y suponemos que cada réplica del recurso se identifica por *nombre-i* donde i es la i -ésima réplica del recurso, entonces cada réplica del recurso se puede identificar de manera única usando una función resumen o *hash*. De esta manera, dado un nombre de recurso, cada proceso puede generar las n claves y buscar el proceso responsable de la réplica.

Si se deniega el acceso al recurso (es decir, la petición obtiene menos de m votos), el proceso que realiza la petición se debe esperar un tiempo aleatorio y volver a intentarlo. En situaciones en las que muchos procesos quieren acceder a un mismo recurso, este sistema pierde rendimiento muy rápidamente, ya que ningún proceso puede conseguir suficientes votos para acceder al recurso y se deja el recurso sin utilizar. En el artículo mencionado al inicio del apartado los autores proponen una solución para este problema.

3.3. Algoritmo basado en un anillo

Una manera sencilla de gestionar la exclusión mutua entre un conjunto de procesos sin que ninguno de ellos haga una función especial o sin requerir ningún proceso adicional es organizar los procesos como si formaran un anillo. La idea que subyace tras esta solución es que se puede acceder a la sección crítica cuando se dispone de un testigo, que es el que da el derecho de acceso. Este testigo es un mensaje que se va pasando de un proceso a

otro según un orden establecido, que se construye teniendo un anillo con tantas posiciones como procesos haya que coordinar. Cada proceso ocupa una posición en este anillo virtual y conoce cuál es el siguiente proceso siguiendo este orden. No es necesario que la topología de este anillo tenga relación con la topología de interconexión de los procesos que forman parte de él.

Si un proceso no tiene que acceder a la sección crítica cuando recibe el testigo, inmediatamente pasa el testigo al siguiente proceso según la orden del anillo. Un proceso que necesite el testigo espera hasta recibirlo y, una vez lo tiene, lo retiene hasta que acabe las operaciones que ha de realizar. Cuando quiere salir de la sección crítica, envía al testigo a su vecino.

Uno de los problemas de este algoritmo es el caso de pérdida del testigo. De hecho, es difícil detectar que se ha perdido el testigo, ya que la cantidad de tiempo que pasa desde que se ve el testigo hasta que se vuelve a ver es totalmente imprevisible. Que haga mucho tiempo que no se ve el testigo no quiere decir que éste se haya perdido, ya que puede ser que alguno de los procesos lo esté utilizando.

Un segundo problema se puede presentar si falla un proceso, pero en este caso la solución es más sencilla. Se puede pedir a un proceso que cuando reciba el testigo confirme su recepción. Se detectará que un proceso ha fallado cuando intente darle el testigo y éste falle. En este momento se puede eliminar del grupo el proceso que ha fallado, y el proceso que tiene el testigo lo envía al siguiente proceso en el orden del anillo, o al siguiente, si procede. Por supuesto, será necesario que todos los procesos rehagan la configuración del anillo.

Otro inconveniente de este algoritmo es que consume ancho de banda continuamente, excepto cuando alguien está dentro de una sección crítica.

3.4. Algoritmo distribuido

Cuando un proceso P_i quiere entrar en una sección crítica genera una nueva marca de tiempo Ts_{P_i} , y envía un mensaje de petición (P_i, Ts_{P_i}) al resto de procesos del sistema.

Cuando un proceso P_j recibe un mensaje de petición, puede contestar inmediatamente o retrasar el envío de la respuesta hasta más adelante.

Cuando el proceso P_i recibe el mensaje de respuesta del resto de procesos del sistema, puede entrar en la sección crítica.

Lectura recomendada

G. Ricart; A. K. Agrawala (1981). "An Optimal Algorithm for Mutual Exclusion in Computer Network Computing". *Communications of the ACM* (vol. 24, núm. 1, pág. 9-17).

Después de salir de la sección crítica, el proceso envía un mensaje de respuesta a todos los que tienen solicitudes retrasadas.

La decisión de si el proceso P_j responde inmediatamente a un mensaje de solicitud (P_i , $T_{s_{P_i}}$) o retrasa la respuesta se basa en tres factores:

- Si P_j está en una sección crítica, entonces retrasa la respuesta a P_i .
- Si P_j no quiere entrar en una sección crítica, envía inmediatamente una respuesta a P_i .
- Si P_j quiere entrar a una sección crítica pero todavía no ha entrado, compara la marca de tiempo de su solicitud ($T_{s_{P_j}}$) con la marca de tiempo $T_{s_{P_i}}$.
 - a) Si la marca de tiempo de su solicitud es mayor que $T_{s_{P_i}}$, entonces envía inmediatamente un mensaje de respuesta a P_i (P_i lo había pedido antes). Si $T_{s_{P_j}}$ es igual a $T_{s_{P_i}}$, entonces la petición se ordena según los identificadores de los procesos. Eso garantiza que haya un orden total.
 - b) De lo contrario, retrasa la respuesta.

De este comportamiento se deduce lo siguiente:

- El sistema está libre de bloqueos indefinidos.
- Ningún proceso estará esperando entrar en la sección crítica infinitamente. La ordenación por marcas de tiempo asegura que los procesos acceden a la sección crítica de acuerdo con un orden.
- El número de mensajes para entrar en la sección crítica es de $2(N - 1)$, donde N es el número de procesos. Son $N - 1$ mensajes para hacer la petición y $N - 1$ para las respuestas. Si se utilizara una primitiva de *multicast*, el número de mensajes sería N .

Algunos inconvenientes de este algoritmo son que obliga a utilizar una primitiva de comunicación en grupos o a que cada proceso conozca la identidad del resto de procesos, lo que hace más complejo añadir o quitar procesos. Además, cuando un proceso falla, el sistema no funciona, lo que obliga a utilizar algún sistema de monitorización. Esto provoca que este algoritmo funcione bien en grupos pequeños y estables de procesos que cooperan entre sí.

Sabed que se han propuesto variaciones del algoritmo que mejoran el rendimiento, pero no las trataremos porque se escapan de los objetivos de la asignatura.

Para acabar, merece la pena decir que este algoritmo es más lento, más complicado, más costoso y menos robusto que la solución centralizada. Sin embargo, está bien ver que se puede construir una solución distribuida.

3.5. Comparación de algoritmos

La tabla siguiente presenta una comparación entre los algoritmos de exclusión mutua vistos.

Algoritmo	Mensajes para entrar/salir	Retraso antes de entrar (en tiempo de mensajes)	Problemas
Centralizado	3	2	Fallo coordinador
Descentralizado	$3mk, k = 1, 2 \dots$	$2mk, k = 1, 2 \dots$	Inanición, baja eficiencia
En anillo	1 a 8	0 en $n - 1$	Pérdida del testigo, fallo proceso
Distribuido	$2(n - 1)$	$2(n - 1)$	Fallo cualquier proceso

En el caso del algoritmo centralizado, necesita tres mensajes para entrar y salir: petición de entrada, autorización de entrar y liberación para salir. El descentralizado necesita tres mensajes para cada uno de los coordinadores. La k corresponde al número de intentos que ha de hacer hasta que consigue entrar. En el algoritmo en anillo, si los procesos quieren entrar constantemente en la sección crítica, cada mensaje corresponderá a una entrada y salida. Pero si no hay nadie interesado en entrar en la sección crítica, el testigo puede estar circulando por el anillo durante horas, lo que hace que el número de mensajes para entrar en la sección crítica sea ilimitado. Finalmente, el algoritmo distribuido necesita $n - 1$ mensajes para la petición de entrada en la sección crítica y $n - 1$ mensajes para las respuestas.

El retraso (en número de mensajes necesarios) desde que un proceso intenta acceder a una sección crítica hasta que lo consigue es de 2 en el caso del algoritmo centralizado; $2mk$ para el descentralizado; $2(n - 1)$ para el distribuido; y un número de mensajes entre 0 (el testigo acaba de llegar) y $n - 1$ (el testigo acaba de salir) en el caso del algoritmo de anillo.

Finalmente, todos los algoritmos presentados, excepto el descentralizado, sufren mucho cuando hay fallos. En la evaluación con respecto a la tolerancia a los fallos, es necesario considerar qué sucede cuando los mensajes se pierden y qué pasa cuando un proceso falla. Hay que añadir medidas especiales y complejidad adicional con el fin de evitar que un fallo haga detener todo el sistema. Si los fallos ocurren raramente estos mecanismos pueden ir bien, sino habrá que tener en cuenta mecanismos que permitan a los procesos coordinarse en presencia de fallos. El algoritmo descentralizado es menos sensible a fallos, pero a cambio puede tener problemas de inanición y de eficiencia.

4. Algoritmos de elección

Muchos algoritmos distribuidos necesitan que alguno de los procesos actúe como coordinador, iniciador o desarrolle algún papel especial. En general, no importa qué proceso lo haga, pero alguno lo debe hacer. A continuación veremos tres algoritmos de elección de coordinador.

4.1. El algoritmo de Bully

El algoritmo de Bully sirve para que un conjunto de procesos elija uno que actúe como coordinador. Un proceso empieza la elección del coordinador en las situaciones siguientes:

- acaba de entrar en el sistema;
- el coordinador actual no responde;
- ha recibido un mensaje de elección de coordinador de otro proceso.

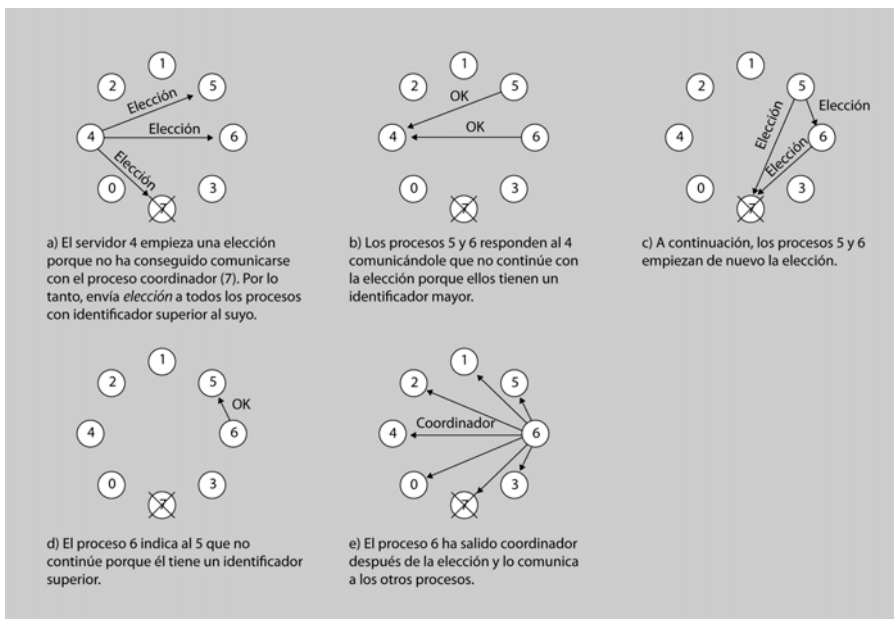
La elección consiste en enviar un mensaje de elección al resto de procesos con identificador superior al suyo. (Se puede optimizar dejando de enviar elección cuando alguno de los servidores con un identificador mayor responde.) Si algún servidor responde, el originador de la elección no hace nada más.

El coordinador será el proceso que no obtenga respuesta a sus mensajes de elección dirigidos a procesos con identificador superior al suyo. Este proceso debe avisar a los otros procesos de su identificador para que sepan que él es el nuevo coordinador.

Lectura recomendada

H. García-Molina (1982, enero). "Election in a Distributed Computing System". *IEEE Trans. Comp.* (vol. 31, núm. 1, pág. 48-59).

Ejemplo



4.2. Algoritmo anillo

Consiste en usar un anillo. Supongamos que cada proceso conoce a su sucesor según algún orden dado. Cuando algún proceso se da cuenta de que el coordinador no está funcionando, envía un mensaje *elección* a su sucesor. Este mensaje contiene la información de su identificador de proceso. Si el sucesor no responde, el proceso que está intentando hacer el envío lo intenta con el sucesor, o el siguiente, hasta que encuentra un proceso que esté funcionando. Cuando un nodo recibe un mensaje de *elección*, añade su identificador a la lista de identificadores y pasa el mensaje a su sucesor.

Cuando el mensaje vuelve a llegar al originador del proceso de elección, se envía un nuevo mensaje *coordinador* donde se informa a todos de quién es el nuevo coordinador (por ejemplo, el miembro de la lista con el identificador mayor) y quiénes son los miembros del nuevo anillo. Cuando este mensaje acaba la vuelta, se elimina del anillo y todos los procesos ya pueden volver a trabajar con el nuevo coordinador.

El algoritmo funciona aunque haya más de un proceso que inicie el proceso de elección; en este caso habrá más de un mensaje *elección*. Cuando estos mensajes acaben la vuelta llegarán a su originador y en todos los casos se convertirán en un mensaje *coordinador* con la misma lista de miembros y en el mismo orden. Cuando hayan dado toda la vuelta se eliminarán del anillo y todos los procesos tendrán el mismo coordinador. Estos mensajes de más no hacen ningún tipo de daño aparte del hecho de consumir un poco de ancho de banda.

4.3. Elección en entornos sin hilo

Las asunciones que hemos hecho en los algoritmos anteriores no las podríamos hacer en un entorno sin hilo. Por ejemplo, suponen que el envío de mensajes es fiable y que la topología de la red no cambia. A continuación explicaremos un algoritmo de elección que funciona en una red *ad hoc*. Este algoritmo puede gestionar nodos que fallan y particiones en la red. Una propiedad importante de esta solución es que elige al mejor líder, no a uno al azar como en los dos anteriores que hemos visto.

Cualquier nodo en la red (lo denominaremos *originador*) puede iniciar una elección de líder enviando un mensaje *elección* a sus vecinos inmediatos (es decir, los nodos que están dentro de su rango). Cuando un nodo recibe un mensaje de *elección* por primera vez, designa quién le ha enviado el mensaje como padre y a continuación envía un mensaje de *elección* a todos sus vecinos inmediatos excepto a su padre. Cuando un nodo recibe un mensaje de *elección* de alguien diferente de su padre, sencillamente confirma la recepción.

Red *ad hoc*

Para simplificar, nos hemos centrado en el hecho de que la red es *ad hoc*. No hemos tenido en cuenta que los nodos se pueden mover.

Cuando un nodo R ha designado a otro nodo Q como padre, envía el mensaje de *elección* a sus vecinos inmediatos (excluyendo Q) y espera a que lleguen las confirmaciones antes de confirmar el mensaje de *elección* recibido del nodo Q .

Los nodos que ya hayan seleccionado un padre contestarán a R muy rápidamente. Si todos los vecinos ya tienen un padre, R es una hoja y responderá a Q muy rápidamente. Al hacerlo, también proporcionará información como la duración de su batería y otras capacidades del recurso.

Esta información permitirá a Q comparar las capacidades de R con las de otros nodos, y elegir el mejor nodo para que sea el líder. A su vez, Q había enviado el mensaje de *elección* porque lo había recibido de su padre P , que había hecho lo mismo. Cuando Q confirme el mensaje de *elección* recibido de P , le pasará el mejor candidato para ser elegido. De esta manera, el originador llegará a saber cuál de los nodos es el mejor para ser elegido como líder. Una vez elegido, hará un *broadcast* de esta información a todos los nodos.

Cuando se inicien distintas elecciones, cada nodo decidirá apuntarse sólo a una de las elecciones. Para conseguirlo, cada elección irá etiquetada con un identificador. Los nodos participarán sólo en la elección con el identificador más alto y detendrán cualquier otra participación en otras elecciones.

Lectura recomendada

Podéis encontrar los detalles del algoritmo en:

S. Vasudevan; J. F. Kurose, D. F. Towsley (2004). "Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks". *Proc of the 12th IEEE International Conference Network Protocols. IEEE Computer Society Press* (pág. 350-360).

5. Tolerancia a fallos

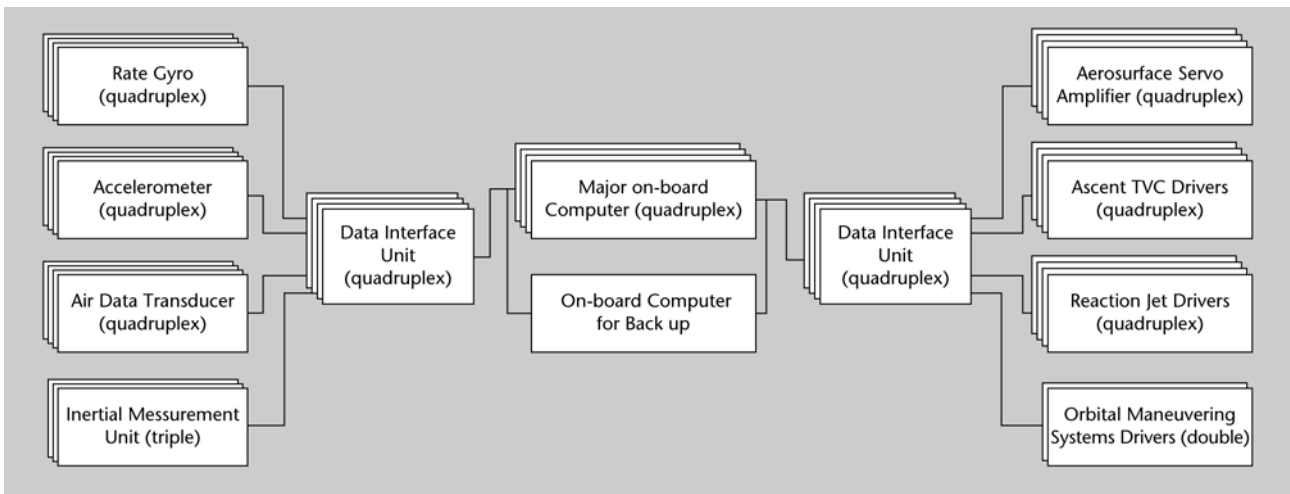
Podría esperarse que la fiabilidad de un sistema distribuido dependiera de la fiabilidad de sus componentes, pero esto no suele ser así. Un sistema distribuido está formado por componentes que interactúan. Esta interacción puede hacer que falle el sistema en cuanto un componente falla, o que siga funcionando sin excesiva degradación a pesar del fallo de algún componente. De esta manera, pueden construirse sistemas con capacidad de “supervivencia”.

El objetivo de funcionamiento es no fallar cuando falla un componente, sino funcionar con componentes averiados o en mantenimiento (fallos parciales), de manera que el sistema “aguante” más que sus componentes.

Un sistema falla cuando no puede cumplir sus promesas de servicio y es consecuencia de algún fallo: una falta, deficiencia o error.

Una forma tradicional para conseguir tolerancia a fallos es mediante *hardware* repetido. Por ejemplo, la lanzadera espacial de la NASA tiene muchas piezas repetidas (muchas cuadruplicadas). Las decisiones se toman por consenso, con tres bastaría para detectar un fallo, con cuatro se permiten dos fallos durante una misión. El computador principal está 4 veces + 1 de respaldo = 5.

Figura 11



Esquema de la redundancia del sistema de vuelo de la lanzadera espacial de la NASA.

Sin embargo, es posible obtener tolerancia a fallos menos crítica a partir de *hardware* “normal” y una capa de programas que ofrezca un modelo de programación y algunos servicios esenciales para programar aplicaciones distribuidas tolerantes a fallos.

De hecho, Internet es una red que tolera fallos en los enlaces debido a su organización redundante. También algunas aplicaciones como DNS y SMTP tie-

Una definición de sistema distribuido...

... desde un punto de vista cínico:
“You know you have one when the crash of a computer you’ve never heard of stop you from getting work done.”
 Leslie Lamport

Sobre la lanzadora espacial

“El programa de a bordo se ejecuta en dos parejas de computadores principales. Una pareja lleva el control mientras sus cálculos simultáneos coinciden. En caso de desacuerdo, pasa el control a la otra pareja. Los cuatro computadores principales ejecutan programas idénticos. Para prevenir fallos catastróficos en los cuales las dos parejas cometen un error (por ejemplo, si el programa tiene un error), la lanzadera tiene un quinto computador que está programado con código distinto por programadores diferentes de una empresa distinta, pero usando las mismas especificaciones y el mismo compilador (HAL/S)”.
 Peter Neuman (1995). *Computer Related Risks*. Addison-Wesley.

nen mecanismos para ofrecer la información y el servicio replicados para evitar que el fallo de una máquina deje a una organización sin servicio.

La tolerancia a fallos trata de cómo se puede depender o contar con el servicio de un sistema.

Hay cuatro aspectos importantes que caracterizan un sistema y su contexto:

- Disponibilidad: el sistema está listo para el uso inmediato.
- Fiabilidad: el sistema puede funcionar de forma continua sin fallo.
- Seguridad: si un sistema falla, nada grave ocurre.
- Mantenibilidad: cuando falla un sistema, que se pueda reparar de forma fácil y rápida (e idealmente, los usuarios no noten el fallo).

Por ejemplo, un sistema que falla durante 1 mseg cada minuto es más disponible que uno que falla durante un minuto cada año, sin embargo, el primero es menos fiable. La seguridad también hay que considerarla, pues el fallo de un servidor de impresión nos deja temporalmente sin poder imprimir, mientras que el fallo de un controlador de una central nuclear o de una sala de operaciones puede tener consecuencias catastróficas.

El fallo de un sistema puede describirse en términos de cuándo falla y de cómo falla. Respecto a cuándo ocurre encontramos tres categorías:

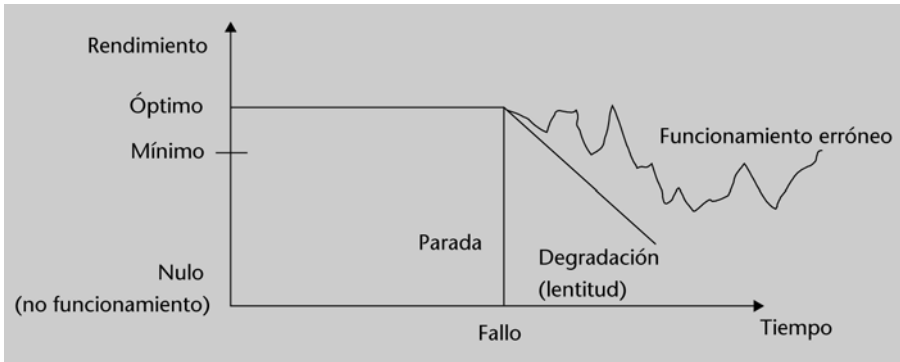
- Fallo transitorio: ocurre una vez y desaparece.
- Fallo intermitente: ocurre, desaparece y vuelve a aparecer más tarde.
- Fallo permanente: cuando aparece, ya no cesa.

Respecto a cómo falla, el comportamiento puede ser:

- Fallo y parada: un proceso funciona bien y de pronto se para. Es el modelo de fallo más fácil de detectar.
- Funcionamiento erróneo: un proceso funciona bien y de pronto empieza a dar algunos resultados erróneos. Es muy difícil detectar la situación, y en un proceso de decisión puede llevar a que el sistema cometa un error antes de verificar que está funcionando mal.
- Funcionamiento lento: el proceso funciona bien, pero empieza a ir cada vez más lento. Puede deberse a que el sistema operativo está paginando, un proceso está reintentando alguna operación o la red está congestionada. Puede llegar a ralentizar el resto del sistema.

Una manera de simplificar los casos es disponer de una capa de software que proporcione el modelo *fallo-parada*: vestir todos los fallos como si fueran paradas. Por tanto, se trata de hacer preguntas a los procesos para verificar que no están funcionando erróneamente, y si se ralentizan o dan errores se les considera parados y se les ignora.

Figura 12



Rendimiento en función del tiempo. El modelo "deseable" es fallo y parada, pero no es el único y esto complica el tratamiento de los fallos.

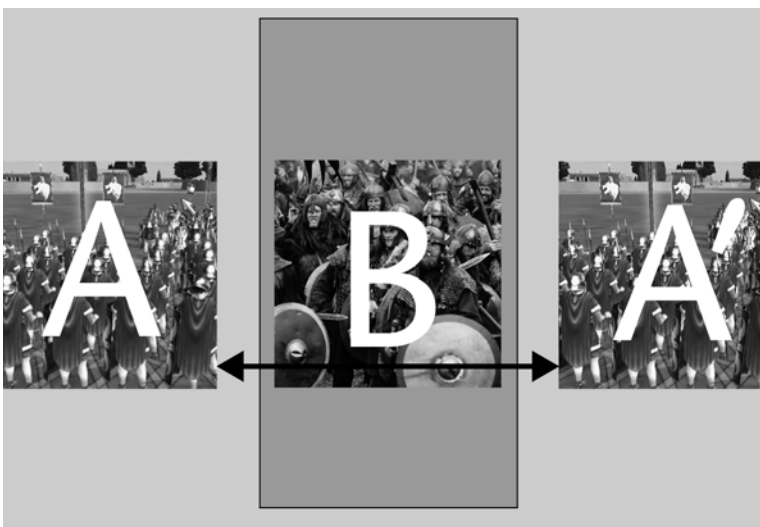
En un sistema donde los mensajes o los participantes pueden fallar, para conseguir fiabilidad hace falta replicación: componentes repetidos; pero si hay componentes repetidos, éstos deben ponerse de acuerdo para realizar una acción o compartir información. Sin embargo, este acuerdo o consenso no es posible si los mensajes se pueden perder o los procesos pueden fallar.

Estos problemas se suelen explicar con el ejemplo de varios generales con sus ejércitos en el campo de batalla. Es el problema de los dos generales y el problema de los generales bizantinos.

1) Imposibilidad de consenso con comunicación no fiable.

El problema de los dos generales explica que dos generales A y A' sólo pueden vencer si acuerdan atacar a la vez al ejército B. Para llegar a un acuerdo, han de intercambiar mensajes. Si los mensajes pueden perderse por el camino (han de atravesar el ejército enemigo), es imposible ponerse de acuerdo: un mensaje con una propuesta de hora de ataque ha de confirmarse, pero también ha de confirmarse la confirmación. Así que un ejército nunca está seguro de que el otro atacará a la vez.

Figura 13



El ejército B sólo puede ser derrotado si los ejércitos A y A' suman sus fuerzas y atacan a la vez. Para ello, han de enviar un mensajero que atravesará el campamento contrario y puede fallar en el camino. Por ejemplo, A envía un mensajero indicando la hora del ataque m (A, A', 12:30 h), pero A' habrá de confirmar si quiere que A esté seguro. Sin embargo, este proceso de confirmación se repite indefinidamente. Es el problema de alcanzar consenso cuando la comunicación no es fiable.

2) Cuál es el número total de participantes necesario para llegar a un acuerdo cuando varios participantes pueden fallar de forma arbitraria.

El problema de los generales bizantinos describe esta situación. Un general bizantino representa un proceso o componente que puede fallar de forma arbitraria: en el peor de los casos podemos suponer que es malicioso y quiere confundir al resto de generales. El nombre viene de las traiciones entre los generales del imperio de Bizancio (la actual Turquía y parte de los Balcanes).

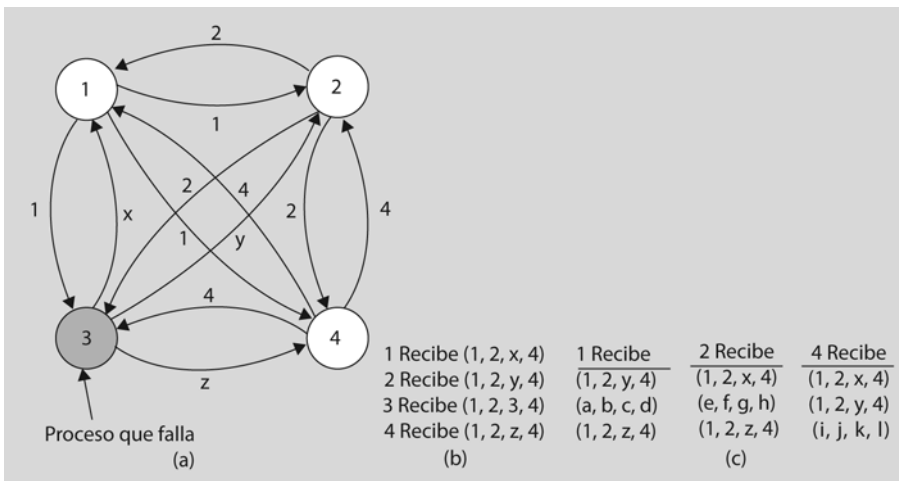
El algoritmo tiene tres fases:

- a) Los generales anuncian su número de soldados (su identificador en este caso) en un mensaje a los demás miembros del grupo.
- b) Cada general construye un vector con los datos recibidos en (a) y su propio número que envía a los demás generales.
- c) Cada general puede determinar quién es el traidor mirando las mayorías en las columnas.

Bibliografía complementaria

Lecturas sobre tolerancia de fallos bizantinos:
L. Lamport; R. Shostak; M. Pease (1982). "Byzantine Fault Tolerance". *ACM Transactions on Programming Languages and Systems*.
L. Lamport; R. Shostak; M. Pease (julio, 1982). "The Byzantine Generals Problem". *ACM Transactions on Programming Languages and Systems* (vol. 4, núm. 3, pág. 382-401).

Figura 14



El problema de los generales bizantinos con tres generales leales y un traidor. Con dos generales leales no sería posible detectar al traidor.

Siguiendo los detalles del algoritmo, se acaba concluyendo que para que el algoritmo detecte los procesos que tienen fallos arbitrarios, más de dos tercios de los procesos han de funcionar correctamente. Es decir: si hay M procesos que fallan, hacen falta $2M + 1$ procesos correctos para poder llegar a un consenso.

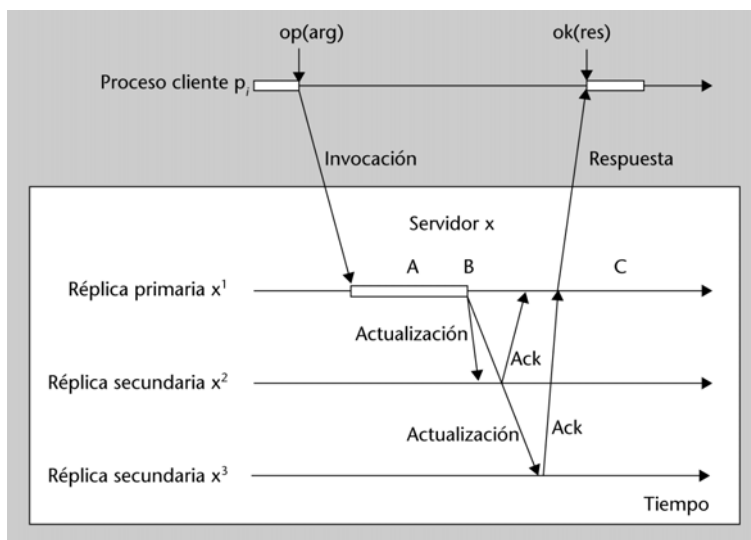
5.1. Comunicación fiable en grupo

Si para conseguir tolerancia a fallos hace falta replicación (varios componentes), cualquier comunicación ha de dirigirse a ese grupo de componentes. La comunicación en grupo (también llamado *groupcast* o *multicast*) fiable es una idea simple pero muy difícil de realizar.

Un aspecto está en cómo enviar mensajes a un grupo de forma fiable por una red. En las redes locales puede aprovecharse la capacidad del hardware para el envío de mensajes a la vez a varios destinatarios. Sin embargo, asegurar la fiabilidad en la entrega de los mensajes a todos los destinatarios y el control de flujo cuando los destinatarios son heterogéneos es un problema. Cuando los destinatarios están en Internet, el problema está en mantener un grafo de conexiones punto a punto entre las diferentes subredes que participan en la comunicación y gestionar la diversidad de redes, terminales y problemas de congestión en la red.

Otro aspecto está en cómo organizar el grupo de servidores. En el modelo primario-secundario (*primary-backup*) las peticiones van al servidor primario y el primario las pasa de forma síncrona a los secundarios. Cuando hay fallos o recuperaciones, se decide quién hará de primario.

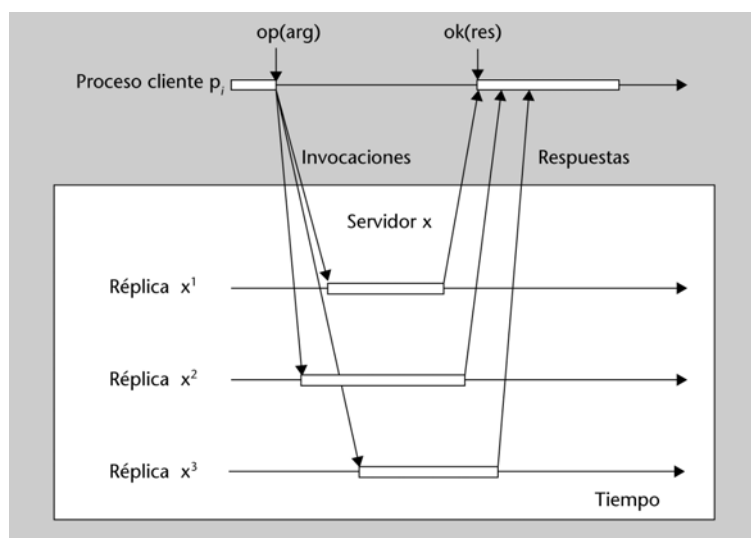
Figura 15



Modelo primario-secundario: el cliente se comunica sólo con el primario. El primario se comunica a su vez con los secundarios.

En el modelo de replicación activa, el cliente tiene que enviar su mensaje a todas las réplicas y éstas simplemente han de responder.

Figura 16



Modelo de replicación activa: el cliente se comunica directamente con todas las réplicas.


Bibliografía complementaria

Mecanismos de replicación de software para tolerancia a fallos:

R. Guerraoui; A. Schiper (1997). "Software-Based Replication for Fault Tolerance". *Computer* (vol. 30, núm. 4, pág. 68-74). [<http://dx.doi.org/10.1109/2.585156>]

Los problemas surgen cuando se producen cambios de estado (operaciones de escritura), fallos o llegan al servidor de operaciones desde varios clientes. La entrega de mensajes al grupo de servidores puede tener diferentes propiedades.

Se ha tratado aquí la replicación síncrona. Más adelante se describen los mecanismos de replicación asíncronos u optimistas.



Sobre los mecanismos de replicación asíncronos u optimistas, podéis ver el subapartado 7.5, "Replicación optimista", de este módulo didáctico.

5.2. Entrega de mensajes

Un mecanismo de entrega de los mensajes es esencial para garantizar ciertas propiedades del sistema, que se respeten las relaciones de orden (por ejemplo, causalidad), que se garantice la entrega de los mensajes y que se respete el tiempo de entrega acordado para un mensaje. Realiza una función similar a un protocolo de transporte como TCP: ofrecer a las aplicaciones un modelo de funcionamiento simple y fiable.

Se trata, pues, de poder determinar:

- *Fiabilidad*. Determina qué procesos pueden recibir una copia del mensaje.
- *Orden*. En qué orden llegan los mensajes.
- *Latencia*. Durante cuánto tiempo puede extenderse la entrega de un mensaje.

Fiabilidad en la entrega: cada participante tendrá que guardar internamente información de estado y una copia de los mensajes, así como recibir confirmaciones de entrega, para ofrecer las siguientes garantías:

- *Atómica*. A todos los miembros de un grupo o a ninguno.
- *Fiable*. A todos los miembros en funcionamiento (los que fallen no recibirán el mensaje, si falla el emisor no hay garantía de entrega).
- *Quorum*. A una fracción del grupo. Si el que envía falla, no hay garantía de entrega.
- *Intent (best effort)*. A cada miembro del grupo, pero ninguno garantiza haber recibido el mensaje.

Ordenación de mensajes (entrega). Cada receptor tendrá una cola de entrega que reordenará los mensajes según las restricciones de orden seleccionadas:

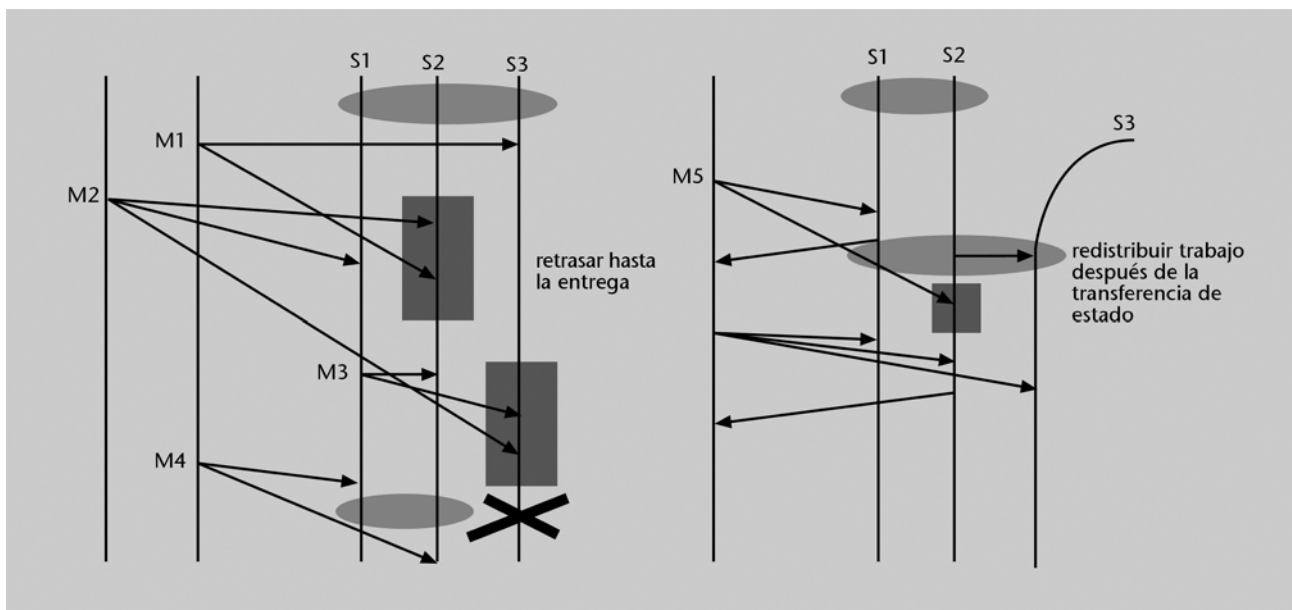
- *Total, causal*. En el mismo orden a cada uno, respetando las relaciones causales.
- *Total, no causal*. En el mismo orden, sin tener en cuenta relaciones causales.
- *Causal*. En orden respecto a potenciales relaciones causales.
- *FIFO*. En orden desde cada uno, pero los mensajes provenientes de otros pueden llegar en cualquier orden.
- *Desordenado*. En cualquier orden (en el orden de llegada).

Latencia en la entrega de mensajes: los procesos que se comunican deberán tener en cuenta la latencia elegida para determinar si un mensaje se ha entregado o ha fallado.

- *Síncrona*. Comienza inmediatamente y se completa en tiempo limitado.
- *Interactiva*. Comienza inmediatamente, aunque puede requerir tiempo finito, pero no limitado, para su entrega completa.
- *Limitada*. Los mensajes pueden ser encolados o retrasados, pero la entrega se realiza en un tiempo límite.
- *Eventual*. Los mensajes pueden ser encolados o retrasados, y la entrega puede requerir un tiempo finito pero no limitado para su entrega completa.

Pueden producirse varios problemas de entrega de mensajes en la comunicación con un grupo de procesos que ofrecen un servicio tolerante a fallos como el siguiente:

Figura 17



Problemas de entrega de mensajes a un grupo de tres procesos servidores.

Problemas de orden total: la entrega de M1 y M2, que son mensajes concurrentes, se hace de manera distinta en S2 que en S1 y S3 (M1, M2; M2, M1; M1, M2). Si se precisa entrega con orden total, en S2 se debe retrasar la entrega de M2 hasta que llegue M1.

Problemas de orden causal: en S3 la entrega de M3 debe retrasarse hasta que llegue M2, pues M2 (precede) → M3.

Problemas de fiabilidad de la entrega: mientras M4 se está enviando, S3 falla. Si se desea una entrega fiable, puede entregarse a S1 y S2, pero si se desea una entrega atómica, entonces hay que cancelar la entrega a S1 y S2, pues S3 ha fallado y no va a recibir M4.

Un problema sutil pero importante sería el que ocurre mientras se entrega M5: S1 recibe el mensaje M5 y, por tanto, su estado cambia. Se incorpora un nuevo servidor y S2 se encarga de transferirle a S3 la información de estado que tenía. S3 comienza a dar servicio después del envío de M5 y, por lo tanto, no recibe

M5. S2 lo recibe justo después de haber transferido su estado a S3. A partir de entonces, S1 y S2 tienen un estado distinto de S3 que no ha visto M5. Para evitar el problema, da la impresión de que durante la operación de transferencia de estado el sistema debería detenerse.

En el ejemplo no se han tenido en cuenta los límites de latencia de entrega de los mensajes que podrían haber hecho que algún mensaje retrasado se eliminara del sistema.

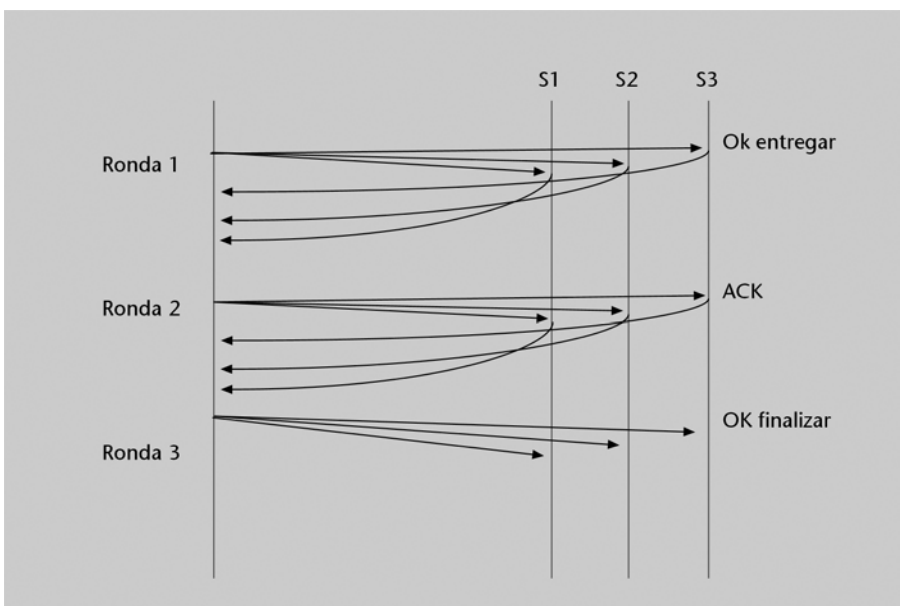
5.3. Transacciones en presencia de fallos

Una operación sobre diferentes procesos puede requerir ciertas garantías para que sea equivalente a la entrega de un solo proceso sin fallo: entrega exactamente una vez y, si falla el emisor, otro toma su lugar, no puede haber *pipelining* o concurrencia.

El protocolo puede requerir dos o tres rondas de comunicación:

- En la primera ronda el emisor propone la operación, y recoge el acuerdo de todos los receptores. Si uno no puede llevarla a cabo, responde negativamente y la operación se cancela.
- En la segunda ronda, el emisor confirma que la operación puede llevarse a cabo (o notifica a todos que la operación se ha cancelado).
- Opcionalmente, en la segunda ronda, los receptores pueden confirmar que cada uno ha podido llevar a cabo la operación y entonces hay una tercera ronda en la que el emisor comunica a todos los receptores que la operación ha ido bien y pueden olvidarse del asunto.

Figura 18



Transacción en tres fases.

Podemos observar que la entrega de un solo mensaje a tres destinatarios con garantías transaccionales (con fiabilidad atómica, como si fuera a uno solo) puede generar mucho más de los tres mensajes que inicialmente podríamos imaginar.

6. Consenso

Cuando se habla de consenso en sistemas distribuidos se hace referencia a un conjunto de procesos que han de ponerse de acuerdo en un valor una vez uno o más de uno de estos procesos han propuesto cuál debería ser este valor.

Ya hemos visto anteriormente en el apartado 5 que los diferentes componentes de la lanzadera espacial deben ponerse de acuerdo antes de realizar ciertas acciones críticas. También hemos visto el problema de los generales bizantinos, en el que dos ejércitos han de decidir de manera consistente si atacan o se retiran. Lo mismo sucedería en el caso de transacciones para transferir fondo de una cuenta a otra: es necesario que los ordenadores implicados se pongan de acuerdo de manera consistente para hacer las respectivas operaciones de débito y crédito. En exclusión mutua, los procesos han de ponerse de acuerdo en qué proceso puede entrar en la sección crítica. En elección, los procesos se ponen de acuerdo en qué proceso eligen. En *multicast* con orden total, los procesos se ponen de acuerdo en el orden de entrega de los mensajes.

Existen protocolos hechos a medida para estos tipos de acuerdos. Ya hemos visto algunos y veremos más en el resto de apartados de este módulo. Lo que veremos en este apartado es cómo se caracterizan tanto el problema como las soluciones, y también un algoritmo de consenso muy utilizado.

Supongamos que tenemos una colección de procesos p_i ($i = 1, 2, \dots, N$) que se comunican por paso de mensajes. Nos interesa llegar a consenso aunque haya fallos. Asumimos que la comunicación es fiable, pero que los procesos pueden fallar.

Para llegar a un consenso, cada proceso p_i empieza en el estado de *no-decisión* y *propone* un valor v_i . Los procesos se comunican unos con otros intercambiando valores. A continuación, cada proceso fija un valor en una *variable de decisión* d_i . Cuando lo hace, entra en el estado *decidido*, en el cual ya no se podrá cambiar d_i .

Cada una de las ejecuciones de un algoritmo de consenso debería satisfacer las siguientes condiciones:

- **Finalización:** tarde o temprano cada proceso correcto acaba asignando un valor a su variable de decisión.
- **Acuerdo:** el valor decidido por todos los procesos correctos es el mismo.
- **Integridad:** si todos los procesos correctos han propuesto el mismo valor, entonces cualquier proceso correcto en el estado de *decisión* ha elegido este valor.

Para entender cómo esta formulación se traduce a un algoritmo, consideramos un sistema en el que los procesos no pueden fallar. En este caso es muy sencillo resolver el consenso. Sólo es necesario agrupar los procesos en un grupo y hacer que cada proceso envíe de manera fiable sus propuestas de valor al resto de miembros del grupo. Cada proceso espera hasta que ha recibido todos los valores de todos los otros procesos y se queda con lo que haya propuesto la mayoría (o no se decide ninguno si no hay mayoría). Esta función *mayoría* es la misma para todos los procesos. La fiabilidad del envío garantiza la finalización. El acuerdo y la integridad se garantizan por la definición de la función *mayoría* y la propiedad de integridad del envío fiable. Todos los procesos reciben el mismo conjunto de valores y todos los procesos aplican la misma función a estos valores y, consiguientemente, han de ponerse de acuerdo.

Si los procesos pueden fallar, la cosa se complica. Hay que detectar los fallos y no está claro que una ejecución del algoritmo de consenso pueda acabar.

Si los procesos pueden fallar de manera *bizantina*, los procesos que fallan pueden comunicar valores aleatorios a los otros. Aunque esto pueda parecer poco probable, no es algo que no pueda suceder si el proceso tiene un error que lo hace fallar de esta manera. Para complicarlo más, este error podría no ser fortuito, es decir, podría ser el resultado de un comportamiento malintencionado o malévol.

Una vez introducido el problema del consenso, veremos uno de los algoritmos de consenso más populares.

6.1. Algoritmo de Paxos

El algoritmo de Paxos es un algoritmo tolerante a fallos que permite llegar a consensos en sistemas distribuidos. Funciona en el modelo de paso de mensajes con asincronía y con menos $n/2$ fallos (pero no con fallos bizantinos). El algoritmo de Paxos garantiza que se llegará a un acuerdo y garantiza la finalización si hay un tiempo suficientemente largo sin que ningún proceso reinicie el protocolo.

Historia del algoritmo de Paxos

El algoritmo de Paxos está considerado uno de los algoritmos más eficientes para conseguir consenso en un sistema de paso de mensajes donde se pueden detectar fallos de procesos. Pero en un primer momento esto no fue así.

El algoritmo de Paxos fue descrito por Lamport por primera vez en 1990 en un informe técnico que fue considerado por muchos como una broma. Si queréis saber la descripción de la historia que hace Lamport, podéis visitar la página <http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos>.

Los procesos se clasifican en *proponentes*, *aceptadores* y *aprendices* (un mismo proceso puede tener los tres roles). La idea es que un proponente intenta ratificar un valor propuesto a fuerza de recoger aceptaciones de una mayoría de

aceptadores, y los aprendices observan esta ratificación. Los aprendices, entonces, intentan comprobar que se ha hecho la ratificación.

La intuición última del funcionamiento del algoritmo es que cualquier proponente puede reiniciar el protocolo generando una nueva propuesta (y así tratar los bloqueos), y que hay un proceso para liberar a los aceptadores de sus votos anteriores si se puede probar que los votos anteriores eran para un valor que próximamente no obtendrá una mayoría.

Para garantizar que el sistema progresa* hay que seleccionar un proponente (líder), que sea quien genere las propuestas. Cada proceso tendrá en todo momento una estimación de quién es el líder. Cuando se quiera realizar una operación, ésta se enviará a quien sea el líder en cada momento. El líder secuencia las operaciones y pone en marcha un algoritmo de consenso para llegar a acuerdos.

* Que el sistema progresa significa que va pasando por las diferentes fases y tarde o temprano acaba.

El algoritmo de consenso se estructura en dos fases: *prepara* y *acepta*. El líder contacta con una mayoría en cada una de las fases. El protocolo permite que en momentos puntuales haya más de un líder de forma concurrente. Aunque más de uno de estos líderes genere una votación, en todo momento se puede distinguir entre los valores propuestos por los diferentes líderes.

Para organizar el proceso de votaciones, se asigna un número de propuesta diferente a cada propuesta. La manera más sencilla de generar estos números de propuesta es que sean parejas formadas por una marca de tiempo (n) y el identificador del originador (p). Un par $\langle n1, p1 \rangle$ es mayor que otro par $\langle n2, p2 \rangle$ si $((n1 > n2) \text{ o } ((n1 = n2) \text{ y } (p1 > p2)))$. El líder elige números de propuesta de manera local, única y monótonamente creciente. Es decir, si el último número de propuesta es $\langle n, q \rangle$ entonces elige $\langle n + 1, q \rangle$.

Marca de tiempo (timestamp, en inglés)

Es un número que crece monótonamente. Inicialmente vale 1 y cada vez que se quiere una nueva marca de tiempo se incrementa su valor en 1. La combinación de la marca de tiempo y el identificador del proceso hará que el número de propuesta sea único.

El algoritmo funciona de la manera siguiente:

Fase 1

- a) Un proponente selecciona un nuevo número de propuesta n y envía una petición $prepara(n)$ a todos los aceptadores.
- b) Si un aceptador recibe una petición de $prepara$ con un número n mayor que cualquier otra petición de $prepara$ que haya respondido hasta ese momento, entonces contesta a la petición con una promesa de no aceptar ninguna otra propuesta con número inferior a n y con el número de propuesta mayor (si hay alguno) que ha aceptado.

Fase 2

- a) Si el proponente recibe una respuesta a su petición de $prepara(n)$ de una mayoría de aceptadores (la mitad más 1), entonces envía una petición de $acepta(n, v)$ a

cada uno de los aceptadores, donde v es el valor del número de propuesta mayor entre todas las respuestas recibidas, o es el nuevo valor que hay que aceptar.

b) Si un aceptador recibe una petición de *acepta* por un número de propuesta n , lo acepta a no ser que ya haya respondido a una petición de *prepara* por un número mayor que n .

La aceptación es un fenómeno local. Son necesarios mensajes adicionales para detectar cuáles, si es que ha habido alguna, de las propuestas han sido aceptadas por una mayoría de aceptadores.

Progreso

Sería fácil construir un escenario en el que haya dos procesos que generen una secuencia de propuestas con números que se incrementen de manera monótona, pero que ninguno de estos números se acabe eligiendo nunca. El proponente p completa la fase 1 para un número de propuesta n_1 . Otro proponente q completa la fase 1 para un número de propuesta $n_2 > n_1$. Se rechaza la petición de *acepta* de la fase 2 del proponente p para el número n_1 porque todos los aceptadores han prometido no aceptar ninguna nueva propuesta con un número menor que n_2 . De esta manera, el proponente p empieza y completa la fase 1 para una nueva propuesta con número $n_3 > n_2$, lo que causa que se rechace la segunda petición de *acepta* de la fase 2 del proponente q . Y así para siempre.

Con el fin de garantizar el progreso, hay que elegir a un proponente “distinguido” que sea el único que intente generar propuestas. Si el proponente distinguido se puede comunicar con una mayoría suficiente de aceptadores, y si utiliza un número de propuesta mayor que cualquier número usado anteriormente, entonces tendrá éxito al generar una propuesta que sea aceptada. En caso de que el proponente aprenda que hay números de propuesta mayores que el que está proponiendo, sólo ha de abandonar la propuesta que esté haciendo e ir intentando números mayores hasta que llegue a un número de propuesta suficientemente grande.

De todo esto se concluye que si hay un número suficiente de procesos que funcionan correctamente, eligiendo a un líder, el sistema funcionará.

Bibliografía sobre el algoritmo de Paxos

Para más información del algoritmo de Paxos podéis ver:

L. Lamport (1998). “The part-time parliament”. *ACM Transactions on Computer Systems* (vol. 16, núm. 2, pág. 133-169).

Artículo original de Paxos. Puede parecer un poco largo, pero está muy bien explicado. Expone el funcionamiento del algoritmo situándolo en la isla de Paxos, en la antigua Grecia. Esta contextualización puede despistar en un primer momento, pero a medida que se va leyendo el artículo ayuda a entender los problemas y cómo los soluciona.

L. Lamport (2001). “Paxos made simple”. *SIGACT News* (vol. 32, núm. 4, pág. 18-25).

Versión reducida del artículo sin la contextualización en la isla de Paxos.

7. Conceptos básicos de replicación

La replicación de datos consiste en mantener diferentes copias de objetos de datos en distintos almacenes de datos. Un objeto es la unidad mínima de replicación en un sistema. Por ejemplo, un objeto puede ser un fichero XML, un registro de una base de datos o una tabla de una base de datos.

La replicación de datos es muy importante en los sistemas distribuidos por dos motivos principales:

- a) mejora la disponibilidad por el hecho de eliminar puntos únicos de fallo (ya que los objetos se pueden acceder en diferentes ubicaciones);
- b) mejora el rendimiento del sistema porque los objetos se pueden ubicar más próximos a los usuarios que los tienen que acceder y porque el mismo objeto lo puede servir más de un almacén.

Un efecto secundario muy interesante de estas dos propiedades es que contribuye a mejorar la escalabilidad del sistema porque puede soportar el crecimiento manteniendo unos tiempos de respuesta aceptables. La contrapartida es que la gestión de las diferentes réplicas de un objeto es compleja.

Hay muchas técnicas para gestionar la replicación. Éstas se pueden clasificar de diferentes maneras. En este módulo nos fijaremos en dos parámetros para presentar dos posibles clasificaciones:

- 1) qué réplica se cambia y
- 2) cuándo se propagan las modificaciones al resto de réplicas.

Según el primer parámetro, los protocolos de replicación se pueden clasificar en *single-master* y *multi-master*. Según el segundo parámetro, en síncronas (*eager*) o asíncronas (*lazy*).

7.1. *Single* frente a *multi-masters*

En el caso del *single-master* hay una copia principal de cada objeto, que se llamaremos primaria. Cuando hay una modificación, ésta se aplica primero a la copia primaria. Después se propaga al resto de copias (que son las secundarias). En este modelo se puede leer cualquier réplica de un objeto, pero sólo se puede modificar la primaria. Un ejemplo de este tipo de sistemas es el DNS.

En la aproximación *multi-master* hay varios almacenes que contienen una copia primaria de un mismo objeto. Todas estas copias se pueden actualizar de

forma concurrente. En este caso se puede acceder por lectura a cualquiera de las copias y por modificación a cualquiera de las primarias. Unos ejemplos de ello son el CVS o los sistemas que utilizan las PDA para sincronizar los datos.

La aproximación *multi-master* reduce los cuellos de botella y los puntos de fallo, así como incrementa su disponibilidad. Por contra, con el fin de garantizar la consistencia de los datos se requieren mecanismos de coordinación y reconciliación entre las diferentes réplicas de ésta.

CVS

Concurrent Version System es un sistema de control de versiones que permite a los usuarios editar ficheros de manera colaborativa. También permite obtener versiones antiguas. Es un sistema muy popular para el desarrollo del software.

7.2. Sistemas síncronos frente a sistemas asíncronos

7.2.1. Síncronos

Los sistemas de propagación **síncronos** aplican las actualizaciones a todas las réplicas de un objeto como parte de la operación de modificación. Esto hace que cuando la operación de actualización acaba, todas las réplicas tienen el mismo estado.

Estos mecanismos se pueden implementar utilizando algoritmos como reserva en dos fases –en el que cuando se quiere hacer una actualización primero se bloquean todas las réplicas, después se actualizan y finalmente se liberan–, o basados en marcas de tiempo, confirmación en dos fases proporcionada, además, atómica (o bien todas las transacciones acaban o bien no se aplica ninguna).

Con este tipo de técnicas se consigue que, aunque haya varias copias de un mismo objeto, el usuario perciba que el comportamiento es como si sólo hubiera una. Este criterio de consistencia se conoce como *one-copy seriality*.

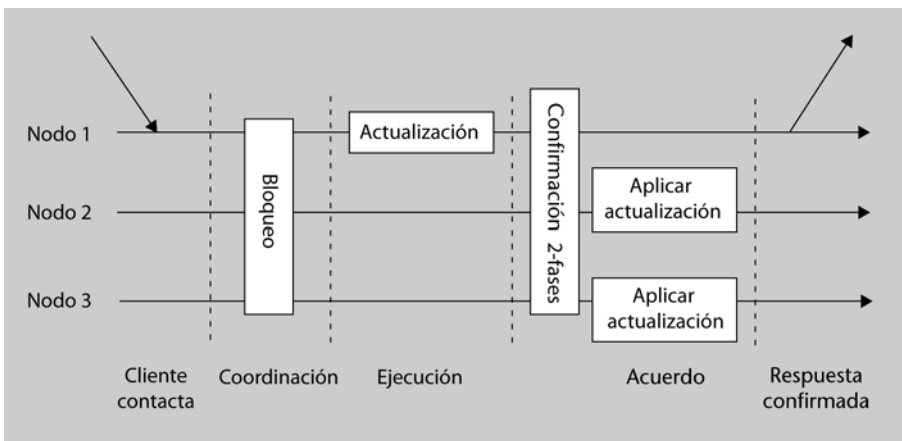
Los protocolos síncronos más sencillos de implementar utilizan *single-master*. También existen *multi-master*, por ejemplo, ROWA (*read-one/write-all*), en el que se puede leer cualquier copia; pero para que la operación de escritura se complete hace falta que se actualicen todas las copias. Tiene el problema de que no es tolerante a fallos, porque si una de las copias no está disponible la escritura se tiene que abortar; ROWAA (*read-one/write-all available*) aborda esta limitación actualizando sólo las copias disponibles. Otros protocolos utilizan quórum. En éstos, una operación de escritura tiene éxito siempre que haya un número determinado de copias (quórum) que ejecutan la operación.

También existen protocolos que aprovechan las ventajas de los sistemas de comunicación en grupo para evitar algunos problemas de rendimiento.

!

Sobre las ventajas de los sistemas de comunicación en grupo, podéis ver el apartado 5.1, "Comunicación fiable en grupo", de este módulo didáctico.

Figura 19



Ejemplo protocolo síncrono

La figura 19 muestra las diferentes etapas que sigue un sistema distribuido síncrono que utiliza un algoritmo de confirmación en dos fases para actualizar un objeto. La operación de actualización se inicia en el nodo 1. Como parte de ésta, se bloquean todas las réplicas del dato. Seguidamente se hace la modificación en el nodo 1 y, utilizando un algoritmo de confirmación en dos fases, la actualización se propaga al resto de nodos. Finalmente, la operación de actualización acaba. En este momento todas las réplicas del objeto tienen el mismo valor. Es importante destacar que la operación no retorna hasta que todas las réplicas han aplicado la actualización.

La gran ventaja de los protocolos síncronos es que evitan la divergencia entre réplicas de un mismo dato. El gran inconveniente es que cualquier escritura tiene que actualizar muchas o todas las réplicas antes de finalizar. Eso es un inconveniente para sistemas cuyos nodos sean dinámicos –sistemas de igual a igual o sistemas que permiten el trabajo en desconectado– o en entornos de gran alcance –donde a causa de la latencia es costoso en tiempo actualizar todas las réplicas. Además, tiene grandes limitaciones de escalabilidad debidas al tiempo necesario para actualizar todas las réplicas.

7.2.2. Asíncronos

En los sistemas **asíncronos** no hace falta que se actualicen todas las copias de un objeto como parte de la operación que inicia la actualización. En este tipo de sistemas la operación de actualización acaba en cuanto puede, y posteriormente el cambio se hace llegar al resto de réplicas.

Los sistemas asíncronos pueden ser optimistas o no optimistas. Los primeros hacen la suposición de que habrá pocas actualizaciones que puedan ser conflictivas. Así, los sistemas optimistas propagan las actualizaciones en *background*. En caso de que haya algún conflicto, éste se resuelve una vez ha ocurrido, lo cual propicia que se requieran mecanismos para detectar y resolver conflictos, así como mecanismos para estabilizar definitivamente los datos. Los sistemas no optimistas, por otra parte, consideran que es probable que pueda haber conflictos, lo que hace que implementen mecanismos de propagación que evitan actualizaciones que puedan provocar conflictos. En este módulo nos centraremos en los sistemas asíncronos optimistas.

Conflicto

Conjunto de actualizaciones originadas en diferentes nodos que conjuntamente violan la consistencia del sistema. Por ejemplo: cuando hay dos actualizaciones concurrentes sobre un mismo dato.

7.3. Modelos de replicación síncronos

7.3.1. Replicación pasiva*

Es un modelo para ofrecer replicación tolerante a fallos caracterizado por el hecho de que en todo momento hay un servidor de réplicas primario y uno o más secundarios (*backups* o esclavos). En el modelo puro, los clientes se comunican sólo con el primario. El primario ejecuta las operaciones y envía copias

* También denominado *primario-secundario* o *primary-backup*, en inglés.

del dato actualizado a los secundarios. Si el primario falla, se elige uno de los secundarios para que pase a ser el primario y éste continúa a partir del punto donde el primario lo había dejado.

Cuando un cliente pide que se ejecute una operación ocurre la secuencia de acontecimientos siguiente:

- 1) El cliente hace la petición al primario.
- 2) El primario toma cada petición de manera atómica, en el orden en el que las recibe.
- 3) El primario ejecuta la petición y almacena la respuesta.
- 4) Si la petición es una actualización, entonces el primario envía el estado actualizado, la respuesta y el identificador único a todos los secundarios. Los secundarios envían un acuse de recibo.
- 5) El primario contesta al cliente.

Este modelo tiene el inconveniente de que introduce sobrecarga. Proporcionar una vista síncrona de los datos implica tener que actualizar los secundarios antes de responder, y si falla el primario la latencia todavía aumenta más mientras el grupo se pone de acuerdo en el estado del sistema y elige un nuevo primario.

Una variante del modelo es que los clientes puedan enviar peticiones a los secundarios, y así aliviar carga al primario.

7.3.2. Replicación activa

Los modelos de replicación activa suelen enviar las operaciones de actualización a todas las réplicas (*multimaster*), y no sólo a una, como hace el modelo primario-secundario. Cada réplica, al recibir la operación de actualización, la ejecuta y responde. También se puede hacer enviando la modificación (en lugar de la operación).

Un problema que tiene la replicación activa es que las operaciones han de ejecutarse en todas partes en el mismo orden, y esto hace que sea necesario un mecanismo de *multicast* con ordenación total.* Estos mecanismos, sin embargo, tienen el inconveniente de que no escalan bien por sistemas distribuidos de gran escala. Otra manera de conseguir esta ordenación total es mediante la utilización de un coordinador, también denominado **secuenciador**. El cliente primero envía la operación al secuenciador, que le asignará un número único secuencial y, posteriormente, se envía la operación a todas las réplicas. Las operaciones se ejecutan en la orden del número de secuencia. Esta solución con secuenciadores se parece mucho al modelo primario-secundario.

* Esto lo hemos visto en el apartado 5.1. "Comunicación fiable en grupo".

El uso de secuenciadores no soluciona los problemas de escalabilidad. Hay propuestas que abordan este problema, pero están fuera del alcance de esta asignatura.

7.3.3. Basados en quórums

En el modelo basado en quórums la operación que ha de realizarse se envía a una mayoría de réplicas, en lugar de enviarla a todas las réplicas, como hace la replicación activa. Más concretamente, no es necesario que la operación se ejecute en todas las réplicas, sino que sólo se pide que un subgrupo del grupo de réplicas complete con éxito la ejecución de la operación. El resto de las réplicas tendrán una copia antigua del dato.

Se pueden utilizar números de versión o marcas de tiempo para saber qué datos están desactualizados. Si se utilizan números de versión, el estado inicial del dato es la primera versión, y después de cada cambio tenemos una nueva versión. Cada copia de un dato tiene un número de versión, pero sólo las versiones que están actualizadas tienen el número actual de versión, mientras que las versiones que no están actualizadas tienen números de versión anteriores. Las operaciones sólo han de realizarse sobre las copias con el número de versión actual.

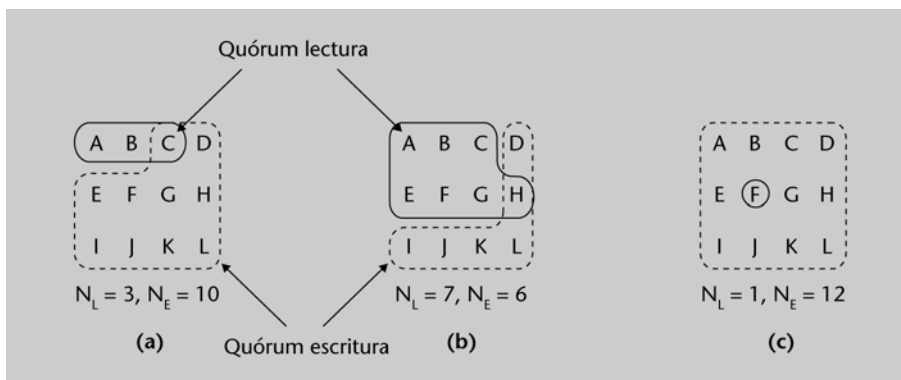
Explicaremos el funcionamiento de los sistemas basados en quórum a partir de un sistema de ficheros que desarrolló Gifford. En este sistema de ficheros, que tiene N copias de un fichero, para poder leer un fichero hay que obtener un quórum de lectura, o sea, un conjunto cualquiera de N_L servidores o superior. De la misma manera, para modificar un fichero hace falta un quórum de escritura de como mínimo de N_E servidores.

Los valores N_L y N_E están sujetos a las siguientes restricciones:

1. $N_L + N_E > N$
2. $N_E > N/2$

La primera restricción se utiliza para evitar conflictos lectura-escritura, la segunda evita conflictos escritura-escritura. Un fichero sólo se puede leer o escribir una vez un número suficiente de servidores han aceptado participar en él.

Figura 20



El ejemplo de la figura 20 nos ayudará a entender el funcionamiento del algoritmo. En la figura *a* el quórum de escritura es 10. La última escritura se ha hecho en las réplicas de la C a la L. Cualquier lectura deberá acceder a alguna de estas diez copias, ya que el quórum de lectura es de tres.

Lectura recomendada

D. K. Gifford (1979). "Weighted Voting for Replicated Data". *Proceedings of the seventh ACM symposium on Operating systems principles* (pág. 150-162).

En la figura *b* pueden ocurrir conflictos escritura-escritura porque N_E es menor o igual que $N/2$. Podemos tener el caso de que un cliente elija $\{A,B,C,E,F,G\}$ como conjunto de escritura y otro cliente $\{D,H,I,J,K,L\}$.

Finalmente, la figura *c* es interesante porque fija el quórum de lectura en 1. En este caso se puede leer cualquier réplica a cambio de que las actualizaciones se realicen en todas las copias. Este esquema se denomina *read-one write-all (ROWA)*.

7.4. Algoritmos para replicación síncrona

7.4.1. Reserva en dos fases*

* *two-phase locking* o 2PL, en inglés.

La reserva en dos fases es un mecanismo sencillo para garantizar seriabilidad y *one-copy seriability*.

Seriabilidad: propiedad que provoca que el resultado de ejecutar una operación sea el mismo que si el resultado se hubiera ejecutado de manera secuencial (sin superposiciones debidas a la concurrencia).

One-copy seriability: propiedad que genera que un usuario perciba el comportamiento de un conjunto de copias de un dato replicado como si sólo hubiera uno.

El protocolo de reserva en dos fases gestiona las reservas (*locks*, en inglés) durante la ejecución de la transacción en las dos fases siguientes:

- 1) Se adquieren todas las reservas y no se libera ninguna.
- 2) Se liberan las reservas y no se adquiere ninguna.

Se garantiza la seriabilidad para ejecuciones que sigan este orden en la gestión de las reservas.

7.4.2. Confirmación distribuida*

* *distributed commit*, en inglés.

Los algoritmos de confirmación distribuida son útiles para situaciones en las que interesa garantizar que todos los procesos de un grupo ejecutan una operación o que ninguno de ellos la ejecuta. En el caso de *multicast* fiable, la operación que se ejecuta sería la entrega del mensaje. En el caso de transacciones distribuidas, la operación sería la realización de la transacción.

Generalmente, las operaciones de confirmación distribuida se basan en un coordinador que notifica al resto de procesos que ya pueden realizar (o no) la operación en cuestión (en local). Claramente ya se ve que si uno de los procesos no puede hacer la operación, no hay manera de notificarlo al coordinador. Por este motivo son necesarios unos mecanismos más sofisticados para poder hacer estas confirmaciones distribuidas. A continuación presentamos la confirmación en dos fases y la confirmación en tres fases.

7.4.3. Confirmación en dos fases*

* *two-phase locking* o *2PL*, en inglés.

Es un algoritmo muy popular para hacer la confirmación distribuida. Se basa en tener un coordinador y el resto de procesos. Si suponemos que no hay fallos, el funcionamiento del algoritmo sería el siguiente:

1. Fase petición de confirmación

- El coordinador envía una **petición de confirmación** al resto de participantes.
- El coordinador espera hasta que recibe un mensaje de cada uno del resto de participantes.
- Cuando un participante recibe un mensaje de **petición de confirmación** contesta al coordinador un mensaje indicando si **está de acuerdo en hacer la confirmación** local o **de aborto** si no la puede hacer.

2. Fase de confirmación

a) Éxito

Si el coordinador ha recibido un mensaje de todos los participantes en la fase de petición de confirmación con la indicación de que están **de acuerdo en hacer la confirmación**:

- El coordinador envía un mensaje de **confirmación** a todos los participantes.
- Cada participante completa la transacción, y libera todos los recursos y las reservas adquiridas durante la transacción.
- Cada participante envía un mensaje de acuse de recibo (*acknowledgement*, en inglés) al coordinador.
- El coordinador acaba la transacción una vez ha recibido todos los acuses de recibo.

b) Fracaso

Si alguno de los participantes contesta un mensaje de **aborto** durante la fase de petición de confirmación:

- El coordinador envía un mensaje a todos los participantes para que deshagan las operaciones que hayan podido realizar o liberen los recursos o las reservas que tengan ocupadas.
- Cada participante deshace las posibles operaciones que ya haya hecho y libera las reservas y los recursos que tenga ocupados.
- Cada participante envía un mensaje de acuse de recibo al coordinador.
- El coordinador acaba la transacción una vez ha recibido todos los mensajes de acuse de recibo.

La gran desventaja del protocolo de confirmación en dos fases es el hecho de que es un protocolo que bloquea. Un participante se bloqueará mientras espere un mensaje. Eso quiere decir que otros procesos que estén compitiendo por la reserva de un recurso que tiene ocupado el proceso bloqueado deberán esperar a que éste libere la reserva. Un proceso continuará esperando incluso cuando el resto de procesos han fallado. Si el coordinador falla de manera permanente, algunos participantes no resolverán nunca la transacción. Esto provoca que los recursos estén ocupados para siempre. El algoritmo se puede bloquear indefinidamente en los casos siguientes: cuando un participante envía un mensaje de acuse de recibo al coordinador hace que se quede bloqueado hasta que recibe un mensaje de confirmación o de deshacer las operaciones realizadas. Si el coordinador falla y no se vuelve a recuperar, los participantes se quedarán bloqueados hasta que el coordinador se recupere o, en el peor de los casos, indefinidamente, ya que éste no puede decidir por su cuenta abortar o confirmar. Lo único que se podría intentar es averiguar qué mensaje había enviado al coordinador.

Por otro lado, el coordinador se quedará bloqueado una vez haya hecho la petición de confirmación hasta que todos los participantes le contesten. Si un participante está indefinidamente parado, el coordinador decidirá abortar cuando salte el temporizador del nodo que está parado. Esta decisión también es una desventaja del protocolo porque está sesgado hacia el aborto en lugar de estarlo hacia la finalización.

A continuación tenéis un esquema del código que ejecutan tanto el coordinador como los participantes en el que se han tenido en cuenta fallos del coordinador y de los participantes.

a) Acciones coordinador

```

escribe PRINCIPIO en el log local
envía a todos los participantes PETICIÓN_CONFIRMACIÓN
mientras no se han recibido todos los votos
  espera llegada voto
  si timeout
    escribe ABORTO_GLOBAL en el log local
    envía a todos los participantes ABORTO_GLOBAL
    exit
  registra respuesta
si todos participantes enviado DE_ACUERDO_CONFIRMACIÓN
y el coordinador vota CONFIRMAR
  escribe CONFIRMACIÓN_GLOBAL en el log local
  envía a todos los participantes CONFIRMACIÓN_GLOBAL
sino
  escribe ABORTO_GLOBAL en el log local
  envía a todos los participantes ABORTO_GLOBAL

```

b) Acciones de cada participante

```

escribe INICIO en el log local
espera PETICIÓN_CONFIRMACIÓN del coordinador
si timeout
  escribe ABORTO en el log local
  exit
si el participante vota CONFIRMACIÓN
  escribe DE_ACUERDO_CONFIRMACIÓN en el log local
  envía DE_ACUERDO_CONFIRMACIÓN al coordinador
  espera la DECISIÓN del coordinador
  si timeout
    envía al resto de participantes PETICIÓN_DECISIÓN
    espera hasta que se reciba DECISIÓN // continúa bloqueado
    escribe DECISIÓN en el log local
  si DECISIÓN == CONFIRMACIÓN_GLOBAL
    escribe CONFIRMACIÓN_GLOBAL en el log local
  sino si DECISIÓN == ABORTO_GLOBAL
    escribe ABORTO_GLOBAL en el log local
sino
  escribe ABORTO en el log local
  envía ABORTO al coordinador

```

c) Thread que gestiona peticiones de decisión (es un thread diferente. La ejecutan los participantes)

```

mientras cierto
  espera que llegue alguna PETICIÓN_DECISIÓN
  lee el ESTADO más reciente guardado en el log local
  si ESTADO == CONFIRMACIÓN_GLOBAL
    envía CONFIRMACIÓN_GLOBAL
  sino si ESTADO == INICIO
    o ESTADO == ABORTO_GLOBAL
    envía ABORTO_GLOBAL
  sino
    no hacer nada // el participante continúa bloqueado

```

Puede suceder que un participante necesite quedarse bloqueado hasta que un coordinador se recupere de un fallo. Esta situación se puede dar cuando todos los participantes han recibido y procesado una PETICIÓN_CONFIRMACIÓN del coordinador y, en paralelo, el coordinador falla. En este caso, los participantes no pueden resolver de manera cooperativa qué decisión tomar. Hay varias soluciones para evitar el bloqueo. Una solución, descrita por Babaoglu y Toueg, es (usando primitivas de *multicast*): inmediatamente después de recibir un mensaje, el receptor hace *multicast* del mensaje a todos los otros procesos. Esta aproximación permite que un participante llegue a tomar una decisión aunque el coordinador no se haya recuperado. Otra solución es el protocolo de confirmación en tres fases que comentaremos a continuación.

Lectura recomendada

O. Babaoglu; S. Toueg (1993). "Non-blocking atomic commitment". En: S. Mullender (ed.). *Distributed Systems* (2.ª ed., pág. 147-168). Addison-Wesley.

7.4.4. Confirmación en tres fases*

Un problema del protocolo de confirmación en dos fases es que cuando el coordinador falla, los participantes pueden no ser capaces de llegar a una decisión final. Eso puede provocar que los participantes se queden bloqueados hasta que el coordinador se recupere. Aunque el protocolo de confirmación en tres fases sea no bloqueante y esté muy tratado en la literatura, no se utiliza demasiado en la práctica, ya que las condiciones en las que el protocolo de confirmación en dos fases se bloquea ocurren raramente. Más concretamente, el protocolo de confirmación en tres fases fija un umbral superior en la cantidad de tiempo necesario antes de que una transacción o bien confirme o bien aborte. Esta propiedad asegura que si una transacción intenta confirmar vía el protocolo de confirmación en tres fases y toma alguna reserva de un recurso, liberará la reserva después de expirar un temporizador asociado.

* *Three-phase commit* o *3PC*, en inglés.

Coordinador

- 1) El coordinador recibe una petición de transacción. Si hay algún fallo en este punto, el coordinador aborta la transacción (es decir, cuando se recupere, considerará la transacción abortada). De lo contrario, el coordinador envía un mensaje de inicio de transacción a los participantes y se pone en estado de espera.
- 2) Si hay un fallo, expira el temporizador, o si el coordinador recibe un mensaje de algún participante que avisa de que no está en disposición de iniciar la transacción durante la fase de espera, el coordinador aborta la transacción y

envía un mensaje de aborto a todos a los participantes. De lo contrario, el coordinador recibirá mensajes de aceptación del inicio de la transacción de los participantes dentro de la ventana de tiempo, y enviará mensajes de confirmación a todos los participantes. A continuación, se pone en estado preparado.

3) Si el coordinador falla en el estado de preparado, se moverá al estado de confirmación. En cambio, si el temporizador del coordinador expira mientras está esperando la confirmación de uno de los participantes, abortará la transacción. En caso de que se reciban todos los acuses de recibo, el coordinador también cambia al estado de confirmación.

Participante

1) El participante recibe un mensaje de inicio de transacción del coordinador. Si el participante está de acuerdo, contesta con un mensaje indicando que está en disposición de iniciar la transacción y se cambia al estado preparado. De lo contrario, envía un mensaje en el que indica que no está en disposición de iniciar la transacción y aborta. Si hay un fallo, se mueve al estado de abortar.

2) En el estado de preparado, si el participante recibe un mensaje de aborto del coordinador, falla, o si expira el tiempo de espera para una confirmación, aborta. Si el participante recibe un mensaje de confirmación, contesta un mensaje de acuse de recibo y confirma.

La principal desventaja de este algoritmo es que no se puede recuperar de un fallo de partición de la red. Es decir, si los nodos se separan en dos mitades iguales, cada mitad continuará por su cuenta.

7.5. Replicación optimista

La replicación optimista hace referencia a un conjunto de técnicas para compartir datos de manera eficiente en un entorno de gran alcance o móvil. La característica principal que diferencia la replicación optimista de otros enfoques es que las operaciones de actualización se hacen sobre una réplica cualquiera del dato a actualizar. Una vez hecha ésta, el iniciador de la actualización ya da el dato por actualizado. Posteriormente, la actualización se propagará en *background* al resto de réplicas del dato. Este funcionamiento se basa en la asunción “optimista” que sólo ocurrirán problemas muy esporádicamente. Su ventaja principal es que aumenta la disponibilidad y el rendimiento del sistema.

Estas técnicas son de uso corriente tanto en Internet como en la computación móvil porque Internet continúa siendo lenta y no fiable. Además, está creciendo mucho el uso de dispositivos móviles con conectividad intermitente (por ejemplo, ordenadores portátiles o PDA). Otro factor que ha contribuido a ello es que algunas actividades humanas se adaptan muy bien a la comparación optimista, por ejemplo en el desarrollo cooperativo de software.

Lectura asociada

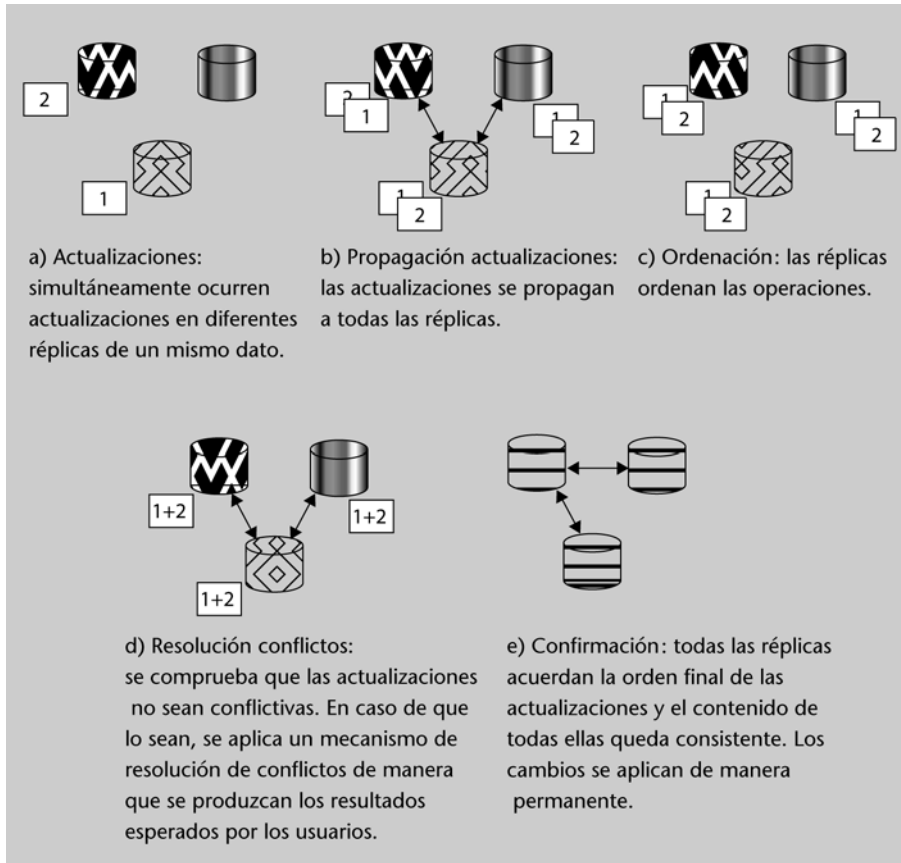
Este apartado sólo ha pretendido presentar algunos conceptos básicos de la replicación optimista. Para conocer las técnicas utilizadas para implementar este comportamiento os recomendamos que consultéis:

Y. Saito; M. Shapiro (2005, marzo). “Optimistic replication”. *ACM Computing Surveys* (vol. 37, núm. 1, pág. 42-81).

7.5.1. Pasos seguidos por un sistema optimista para llegar a un estado consistente

En la figura siguiente se presentan los pasos que sigue un sistema que funciona utilizando replicación optimista para llegar a un estado consistente.

Figura 20



En este módulo no entraremos en los detalles de cada uno de estos pasos –si alguien está interesado, puede consultar la referencia que se proporciona al inicio del apartado anterior. Sin embargo, hay unos aspectos que consideramos interesante comentar:

Las operaciones de actualización del paso *a* pueden ser operaciones en las que se cambia todo el estado del objeto (*transferencia de estado*) u operaciones en las que se especifica qué transformación hay que hacer en el objeto (*transferencia de operación*). La transferencia de estado hace que la construcción del sistema sea más sencilla porque la propagación de la actualización sólo implica la transferencia de todo el contenido a las réplicas no actualizadas. El DNS es un ejemplo de sistema que utiliza transferencia de estado. La transferencia de operaciones es más compleja porque hace falta que todas las réplicas se pongan de acuerdo en la orden de ejecución de las actualizaciones. Por otra parte, puede ir bien cuando los objetos son muy grandes. Además, permite una resolución de conflictos más flexible. La transferencia de operaciones también comporta que cada nodo tengan que mantener un log de operaciones por si

hace falta deshacer y reordenar la ejecución de las mismas. El CVS utiliza transferencia de operaciones.

La propagación de operaciones (paso *b*) se acostumbra a hacer utilizando técnicas epidémicas. De esta manera todos los nodos acaban recibiendo las actualizaciones aunque no se puedan comunicar directamente.

Cada nodo puede recibir las operaciones en un orden diferente. En el paso *c* cada nodo intenta reconstruir un orden de ejecución que produzca un resultado equivalente al del resto de nodos y que encaje con los resultados intuitivos que espera el usuario. Así, en un primer momento, una operación se considera como propuesta de ejecución. Un nodo puede tener que reordenar de manera repetida o transformar operaciones de manera repetida mientras no se acabe decidiendo, entre todos los nodos, cuál es el orden final de las operaciones.

Las políticas de ordenación se pueden clasificar en sintácticas y semánticas. Los métodos sintácticos ordenan las operaciones sólo según la información de cuándo, dónde y quién fue el originador. Una manera muy usada para conseguirlo son las marcas de tiempo. Los métodos semánticos intentan sacar partido a las propiedades semánticas de la aplicación, como la conmutatividad o la idempotencia de las operaciones, para reducir los conflictos y la frecuencia en la que hay que deshacer operaciones. Los métodos semánticos sólo se usan en sistemas de transferencia de operaciones, ya que en sistemas que usan la transferencia de estado es muy complejo hacerlo.

Los métodos sintácticos son más sencillos, pero pueden detectar falsos conflictos. Los métodos semánticos no siempre son fáciles de construir porque hay que tener en cuenta las relaciones semánticas entre las operaciones. Esto suele ser complejo y las soluciones no son generales.

Los usuarios de un sistema optimista pueden hacer actualizaciones de un dato sin saber que hay otros usuarios actualizando el mismo objeto. Además, puede pasar que los usuarios no estén disponibles cuando se detecte que ha ocurrido el conflicto. Algunos sistemas no hacen ningún tratamiento de conflictos y sencillamente se quedan con uno de los datos y descartan el resto. Una manera mejor de hacerlo es disponer de mecanismos de detección y resolución de conflictos, que es el paso *d*. Entre éstos los hay que sencillamente avisan a los usuarios de que ha habido un conflicto y éstos los resuelven de manera manual. Otros sistemas implementan sistemas de automáticos de resolución de conflictos. Estos resolvedores de conflictos tienen el inconveniente de que acostumbran a ser complejos de construir. Además, son muy dependientes de cada aplicación. La detección y resolución de conflictos acostumbra a ser la parte más compleja de los sistemas de replicación optimista.

Para acabar, el paso *e*, corresponde al mecanismo utilizado por los nodos para ponerse de acuerdo en el orden de las actualizaciones y en los resultados de la

La propagación epidémica...

... consiste en lo siguiente: cuando dos nodos se comunican, intercambian sus operaciones locales así como las operaciones que han recibido de otros nodos.

resolución de conflictos de manera que las actualizaciones se puedan aplicar de modo definitivo a los objetos, sin miedo a que haya reordenaciones futuras.

7.5.2. Algunos ejemplos de sistemas optimistas

El DNS, el CVS y las Palm (PDA) son tres sistemas muy populares que utilizan replicación optimista.

El DNS es un sistema *single-master* que utiliza transferencia de estado. Los nombres de una zona están gestionados por un nodo principal, que es quien tiene la copia autorizada de la base de datos de la zona y unos nodos secundarios que copian la base de datos del principal. Tanto el servidor principal de la zona como los secundarios pueden contestar las consultas de los clientes y servidores remotos. Las actualizaciones de la base de datos, en cambio, se hacen únicamente en el nodo principal. Periódicamente, los nodos secundarios consultan el principal por si ha habido cambios en la base de datos. Los servidores DNS recientes soportan un comportamiento proactivo del servidor principal.

El CVS es un sistema *multi-master* con transferencia de operaciones que centraliza la comunicación a través de un almacén único en una topología en estrella. Hay un almacén central que contiene la copia autorizada de los ficheros, así como los cambios que han ocurrido en el pasado. Para hacer una actualización, un usuario crea una copia local de los ficheros y los edita. Puede haber varios usuarios trabajando de manera concurrente, cada uno en sus copias locales de los ficheros. Cuando un usuario acaba, confirma su copia local al almacén. Si ningún otro usuario ha modificado el fichero, la confirmación acaba inmediatamente. Sino, si las modificaciones afectan a líneas diferentes, las combina de manera automática y confirma la versión fusionada (esta resolución automática puede no ser correcta desde el punto de vista semántico, por ejemplo, puede ser que un fichero así fusionado no compile). En otro caso, se informa al usuario de que ha habido un conflicto y éste lo tiene que resolver.

Las agendas electrónicas o PDA son ejemplos de sistemas *multi-master* con transferencia de estado. Los usuarios las utilizan para gestionar su información personal, como la agenda de reuniones o de contactos. De vez en cuando sincronizan la PDA con el ordenador y los datos viajan bidireccionalmente. Si un mismo dato se ha modificado en los dos lados, el conflicto se resuelve usando un resolvidor específico de la aplicación o manualmente por el usuario.

Otros sistemas optimistas en Internet son el *caching* de WWW, los *mirroring* de FTP y los sistemas de noticias Usenet-news.

Resumen

En este módulo se han descrito los problemas que presenta y las ventajas que ofrece un sistema distribuido formado por procesos y/o máquinas que se comunican por mensajes que viajan por una red, como Internet.

Los sistemas distribuidos tienen comportamientos difíciles de observar: no se pueden hacer “fotos instantáneas” del estado del sistema, y cualquier observador tendrá una imagen distinta, dependiendo de la ubicación y de lo que tardan los mensajes en llegarle por la red.

Si existiera un tiempo común para todos los procesos, una marca de tiempo nos daría información de qué ocurrió y en qué orden exacto. Como esto no es posible, se presentan:

1) algoritmos de sincronización de relojes para el intercambio de mensajes, como el algoritmo de Cristian, NTP o Berkeley;

2) algoritmos para abstraerse del paso del tiempo y quedarse sólo con la ocurrencia de eventos y sus relaciones: relojes lógicos, relación \rightarrow (precedencia o causalidad potencial), concurrencia \parallel .

La relación $e \rightarrow e'$ indica que el evento e ha precedido e' , y que e está en la historia causal de e' : e podría haber sido causa de e' . La alternativa es la relación \parallel tal que si $e \parallel e'$ podemos decir que no están relacionados por una relación de precedencia. Los relojes Lamport y sus extensiones vectoriales o matriciales permiten caracterizar lo esencial de una ejecución en un sistema distribuido.

La exclusión mutua evita interferencias y asegura la consistencia en el acceso a recursos cuando hay más de un proceso que quiere acceder a un recurso.

Los algoritmos de elección permiten elegir un coordinador o proceso que desarrolle un rol especial. Hay muchos algoritmos que necesitan un proceso que tenga este papel.

A base de repetir componentes, se pueden construir sistemas distribuidos con mayor rendimiento y tolerancia a fallos que cada componente por separado, y a menor coste que una única máquina.

Los fallos pueden caracterizarse en términos de cuándo falla (transitorio, intermitente, permanente) o de cómo falla (fallo-parada, erróneo, lento). Para simplificar, es necesario introducir mecanismos que hagan aparecer a todos los fallos como del tipo fallo-parada.

La replicación es la solución, pero también es un problema: las réplicas han de coordinarse entre sí, han de llegar a un consenso. El problema de los dos ejércitos ilustra la imposibilidad de consenso cuando la comunicación no es fiable. El problema de los generales bizantinos permite determinar el número de componentes necesarios para soportar cierto número de fallos arbitrarios en los componentes.

Los componentes replicados precisan de mecanismos de comunicación a grupo o *groupcast*. Hay dos modelos principales de comunicación en grupos: primario-secundario y replicación activa.

La entrega de mensajes puede caracterizarse por su fiabilidad (a quiénes se entrega), el orden (orden relativo a otros mensajes) y la latencia (durante cuánto tiempo puede extenderse la entrega de los mensajes).

Para entregar un único mensaje de manera fiable a varios destinatarios, pueden ser necesarias varias interacciones y mensajes entre todos: transacciones. Por tanto, ciertas garantías tienen claramente un coste en tiempo, carga de las máquinas y tráfico en la red.

Una vez vistos estos problemas relacionados con la replicación, hemos presentado los conceptos básicos de una manera más general, para acabar haciendo énfasis en la replicación optimista.

Por consenso se hace referencia a un conjunto de procesos que deben ponerse de acuerdo en un valor una vez uno o más de uno de estos procesos han propuesto cuál habría de ser este valor.

Los sistemas de replicación se pueden clasificar de muchas maneras, nosotros lo hemos hecho según dos parámetros: según qué réplica se modifica (*single-master* o *multi-master*) y según cuándo se propagan las réplicas (síncronos a asíncronos).

Se han presentado tres modelos para replicación síncrona y tres algoritmos populares que se utilizan en estos modelos.

Dado que las técnicas de replicación optimista se usan mucho en Internet y la computación móvil, hemos presentado los pasos que siguen este tipo de sistemas para llegar a un estado consistente y hemos comentado tres ejemplos de sistemas que usan estas técnicas: CVS, DNS y las PDA.

Actividades

1. El coste de un sistema único de capacidad N veces superior a la de un solo PC suele costar más caro que N PC cooperando en un sistema distribuido. Mirad en Internet precios de un PC para hacer de servidor web y calculad a partir de qué valor de N es más barato construir un servidor web distribuido. También podría mirarse para el grado de fiabilidad (el tiempo medio entre fallos), aunque éste es un dato más difícil de conseguir.

Por ejemplo, un servidor web de capacidad $N \cdot C$ puede construirse fácilmente utilizando varios PC de capacidad C haciendo que los servidores se repartan la carga entre sí y que el servicio de nombres DNS resuelva el nombre del sitio a una dirección IP distinta cada vez, o devuelva una lista de direcciones (Round Robin DNS).

2. Sincronizad el reloj de vuestro PC usando algún programa para sincronizar relojes. Anotad las variaciones diarias de tiempo que puedan medirse y comparad con la hora de un reloj de pulsera durante varios días.

Se pueden usar programas como Dimension 4:

<http://www.thinkman.com/dimension4/> para Windows,

<http://www.ua.es/es/servicios/si/ntp/configuracion.html>,

y usar un servidor de tiempo como ntp.upc.es u otros.

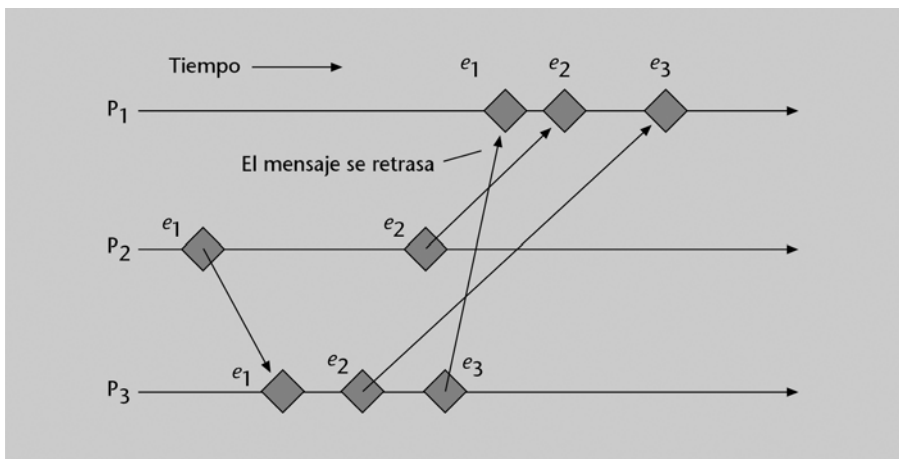
Podéis ver más información sobre el proyecto piloto de sincronización de tiempo en la red académica española RedIRIS: <http://www.rediris.es/gt/iris-ntp/> y <http://www.rediris.es/gt/iris-ntp/drafts/>

3. Visitad varios sitios web, averiguad su hora y comparad si está o no bien ajustada: podríamos llegar a obtener contenido aparentemente futuro, o contenido actual que aparenta ser viejo debido a desajustes en el reloj.

Ejercicios de autoevaluación

1. En una sociedad primitiva en la que los hombres no saben producir fuego, cuando en una aldea el fuego se extingue (observación local) hay que enviar un mensajero con una antorcha a buscarlo a las aldeas vecinas. Si en ninguna aldea vecina queda fuego (observación global), toca esperar hasta que en la próxima tormenta caiga un rayo que prenda fuego a un árbol. Dibujad la trayectoria de varios mensajeros en busca de fuego, y cómo podrían llegar a la conclusión errónea de que el fuego se ha agotado en las tres aldeas de la región.

2. Escribid los valores que toma el reloj de Lamport en un sistema distribuido formado por tres procesos que intercambien algunos mensajes según el diagrama siguiente. Haced una lista de los eventos que tienen relación de precedencia y de concurrencia. Verificad que si $e \rightarrow e' \Rightarrow L(e) < L(e')$.



3. En un servicio formado por un grupo de servidores replicados, haced una tabla comparando cómo actuaría el modelo de gestión *primario-backup*, copia disponible y votación, para estos casos: consultas frecuentes, modificaciones frecuentes, un servidor con fallo-parada, funcionamiento erróneo y funcionamiento lento.

4. Según la fiabilidad/orden/latencia en la entrega de mensajes, clasificad los sistemas siguientes: correo electrónico, web, vídeo y una transacción bancaria.

5. Dibujad el caso de tres procesos con un fallo bizantino y el caso con fallos *fail-stop*. Justificad si puede funcionar o no y cuántos procesos podrían hacer falta.

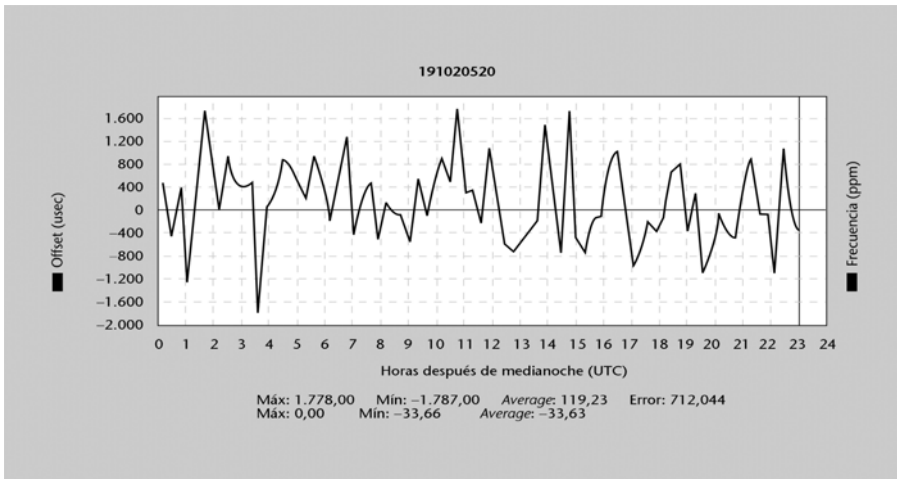
6. Haced una tabla con las diferencias fundamentales entre servidores replicados organizados como primario-secundario y replicación activa.

Solucionario

1. El valor de N por el que un sistema distribuido es preferible a uno centralizado depende de las características del sistema elegido y del momento. En general, los precios de procesadores no crecen linealmente sino exponencialmente con la velocidad, al igual que la capacidad de almacenamiento. Valores de N 2 o 3 pueden ser habituales.

2. Se trata de familiarizarse con los conceptos de sincronización de tiempo para conseguir sincronizar el reloj de nuestro ordenador.

Se trata también de averiguar cuál es el margen de variación de nuestro reloj respecto a la referencia de tiempo oficial. Dependiendo del programa elegido, idealmente podría llegarse a dibujar una gráfica similar a la que se muestra a continuación, con esta u otra escala de tiempo mayor.

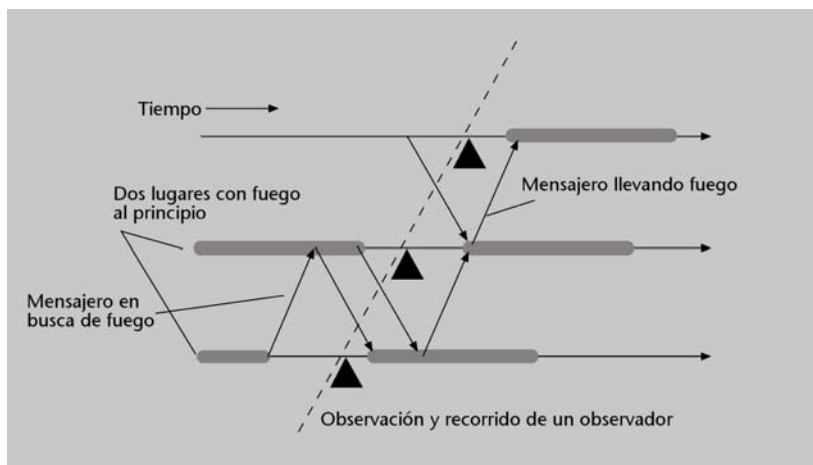


3. Por ejemplo, el código siguiente se obtiene de visitar la página <http://www.apache.org> con el comando Telnet en el puerto 80 del servidor y pidiendo información sobre la página principal mediante el comando HEAD de HTTP (en negrita lo que se escribe):

```
telnet www.apache.org 80
HEAD / HTTP/1.1
Host: www.apache.org
```

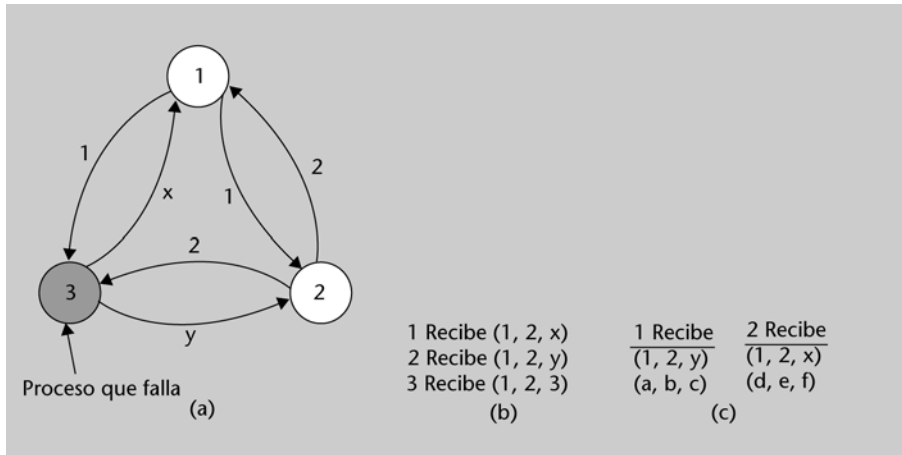
```
HTTP/1.1 200 OK
Date: Sun, 02 Jan 2005 23:51:01 GMT
Server: Apache/2.0.52 (Unix)
Cache-Control: max-age=86400
Expires: Mon, 03 Jan 2005 23:51:01 GMT
Accept-Ranges: bytes
Content-Length: 11795
Content-Type: text/html
```

4. En el diagrama siguiente puede verse cómo un observador que viajase por las aldeas llegaría a una conclusión equivocada.



5. Puede verse que si el proceso 3 es bizantino (x e y tienen valores arbitrarios), los procesos 1 y 2 no pueden decidir.

En cambio, si el proceso 3 falla y se detiene en lugar de dar respuestas incorrectas, y si asumimos que sólo hay fallos de tipo fallo-parada, el sistema podría funcionar correctamente sólo con que funcionara un proceso.



Las fases del algoritmo son (a), (b) y (c).

6.

Característica	Primario-secundario	Replicación activa
Envío mensajes al grupo	1 único mensaje (como si sólo hubiera un servidor)	Mensaje al grupo (<i>multicast</i> o varios mensajes a cada réplica)
Respuestas del grupo	1 única respuesta (por tanto, idéntico a tener un único servidor desde el punto de vista del cliente)	Tantas respuestas como réplicas (el cliente ha de gestionar esto: puede tomar sólo la primera y descartar el resto, tomar una mayoría o esperar a que lleguen todas)
Organización del grupo	Heterogénea (el primario es diferente y hay que elegirlo si falla)	Homogénea (todos hacen lo mismo)
Sencillez (como si no hubiera replicación)	En el cliente	En cada servidor (réplica)

Glosario

causalidad *f* Relación de causa y efecto entre dos fenómenos. Un evento que ha ocurrido antes que otro y pertenece a su historia (eventos relacionados con el actual por proceso o mensajes) mantiene con éste una relación de causalidad potencial. Las situaciones no causales son difíciles de tratar, y un sistema que ordene la entrega de mensajes para preservar las relaciones de causalidad facilita la programación.

commit Véase confirmación.

confirmación *f* Aplicar de manera irreversible una operación.
en commit

conflicto *m* Conjunto de actualizaciones originadas en diferentes nodos que conjuntamente violan la consistencia del sistema.

fallo *m* Un sistema o proceso puede dejar de funcionar correctamente de manera inesperada. El fallo puede hacer que el sistema o proceso responda más lentamente, devuelva respuestas erróneas o se pare.

fiabilidad *f* Probabilidad de que una máquina, un aparato, un dispositivo, etc. cumpla una determinada función bajo ciertas condiciones durante un determinado tiempo. Las necesidades determinan el número de veces o el tiempo que puede durar un fallo en un periodo de tiempo.

grupo *m* Abstracción necesaria para agrupar varios procesos que cooperan para proporcionar un servicio y al que se pueden dirigir mensajes. Una capa de *software* se encarga de repartir los mensajes dirigidos a un grupo entre los procesos que forman parte del mismo.

master *m* Réplica que tiene la capacidad de aceptar actualizaciones.

multi-master *m* Sistema que soporta varios primarios por objeto.

propagación epidémica *f* Modo de propagación en la que cuando dos nodos se comunican intercambian sus operaciones locales, así como las operaciones que han recibido de otros nodos.

reloj *m* Instrumento para medir el tiempo. En cada computador es necesario un dispositivo electrónico que emita señales periódicas para controlar el tiempo de duración de las operaciones. Este reloj tiene imperfecciones y variaciones respecto a un patrón de referencia que hacen que la medida de tiempo sólo sea válida dentro de la máquina. En un sistema distribuido, cada computador tendrá un reloj con valores de tiempo ligeramente distintos.

replicación asíncrona *f* En los sistemas de replicación asíncrona no hace falta que se actualicen todas las copias de un objeto como parte de la operación que inicia la actualización. Posteriormente, la actualización se hace llegar al resto de réplicas.

replicación síncrona *f* En los sistemas de propagación síncronos las actualizaciones se aplican a todas las copias del objeto como parte de la operación de actualización.

sincronía *f* Relacionado con los procesos que ocurren bajo la dirección de una señal de reloj. En un sistema distribuido se puede llegar a obtener una sincronía virtual: un sistema distribuido asíncrono que ofrece propiedades equivalentes a que cada proceso estuviera sincronizado perfectamente a un hipotético reloj global.

single-master *m* Sistema que soporta un primario por objeto.

transferencia de estado *f* Técnica que propaga las operaciones recientes enviando el valor del objeto.

transferencia de operación *f* Técnica que propaga actualizaciones en forma de operaciones (transformación que hay que hacer sobre el objeto).

Bibliografía

Coulouris, G.; Dollimore, J.; Kindberg, T. (2005). *Distributed systems: Concepts and design* 4/E. Londres: Addison-Wesley. (trad. cast.: *Sistemas distribuidos: conceptos y diseño*, 3/E. Pearson, 2001).

Es un libro que trata de los principios y diseño de los sistemas distribuidos, incluyendo los sistemas operativos distribuidos. En este módulo interesan los capítulos 11: "Tiempo y estados globales"; 12: "Coordinación y acuerdo"; 13: "Transacciones y control de concurrencia"; 14: "Transacciones distribuidas"; 15: "Replicación". Se trata de un texto de profundización, con un tratamiento muy exhaustivo de cada tema.

Tanenbaum A.; Steen, M. (2007). *Distributed Systems: Principles and Paradigms*, 2/E. Prentice Hall.

Este libro es una buena ayuda para programadores, desarrolladores e ingenieros con el fin de entender los principios y paradigmas básicos de los sistemas distribuidos. Relaciona los conceptos explicados con aplicaciones reales basadas en estos principios. Es la segunda edición de un libro que ha tenido mucho éxito tanto por los aspectos que trata como por el tratamiento que hace de ellos. En este módulo interesan los capítulos 6: "Synchronization"; 7: "Consistency and replication"; 8: "Fault tolerance".

Birman, K. (2005). *Reliable Distributed Systems. Technologies, Web Services, and Applications*. Nueva York: Springer Verlag.

Es un libro que trata de los conceptos, principios y aplicaciones de las arquitecturas y sistemas distribuidos. En este módulo interesa la parte III: "Reliable Distributed Computing" y los capítulos 23: "Clock synchronization and Synchronous Systems", 24: "Transactional Systems" y 25: "Peer-toPeer Systems and Probabilistic Protocols" de la parte V. También puede ser interesante ver cómo estos aspectos están implementados en los sistemas y aplicaciones que se comentan en el apartado IV: "Applications of Reliability Techniques".

Saito, Y.; Shapiro, M. (marzo, 2005). "Optimistic replication". *ACM Computing Surveys* (vol. 37, núm. 1, pág. 42-81).

Es un estudio con las técnicas más habituales de replicación optimista.