Tema 5 –Gestión de fallos

Guía de estudio

Introducción

La transparencia es uno de los objetivos de todo sistema distribuido. Dentro de los diferentes tipos de transparencia que debe facilitar un sistema distribuido destaca la transparencia de fallos. Cuando un sistema ofrezca este tipo de transparencia, los usuarios no deben verse afectados cuando algún recurso del sistema o algún componente de la aplicación utilizada deje de funcionar. Para lograr este tipo de comportamiento habrá que replicar todas las entidades que puedan fallar. De esa manera, cuando alguna réplica no funcione, otra la reemplazará y entre todas se podrá proporcionar la funcionalidad esperada por los usuarios.

En este tema se revisará el concepto de fallo y se describirá qué modelos de fallos se pueden asumir a la hora de escribir un algoritmo distribuido. También se analizará el concepto de robustez y se explicarán los mecanismos necesarios para alcanzarla. La replicación es el mecanismo más importante, pues incrementará la fiabilidad, disponibilidad y mantenibilidad de los componentes de las aplicaciones. Se analizarán los modelos de replicación clásicos, permitiendo seleccionar el modelo más adecuado para cada aplicación, en función del intervalo de cómputo y de la cantidad de información que modifique cada una de sus operaciones.

Por último, cuando se mantienen varias réplicas de un elemento determinado es posible que surjan divergencias entre los estados de estas réplicas (debido al intervalo necesario para propagar y aplicar las modificaciones en ellas). Los diferentes modelos de consistencia caracterizan el grado de divergencia que existirá. En este tema se definirán los modelos más importantes y se analizará el coste que exige llegar a cumplir con cada modelo. En la práctica, si el principal objetivo sigue siendo la escalabilidad, se tendrá que seleccionar un modelo de consistencia que minimice la necesidad de coordinación entre las réplicas. Eso conduce normalmente a una consistencia relajada.

1. Fallos: Concepto y tipos

De manera general se dice que ha habido un fallo cuando algún componente del sistema es incapaz de comportarse de acuerdo con su especificación. Esto ocurre, por ejemplo, cuando un componente no puede proporcionar ninguna respuesta a las peticiones recibidas, cuando la respuesta sea siempre incorrecta o cuando la respuesta se proporcione fuera de plazo.

Revisemos a continuación esta definición y veamos qué modelos de fallos existen.

1.1. Concepto de fallo

La anterior es solo una definición intuitiva. Para entender mejor qué es un fallo y por qué se produce conviene distinguir tres etapas distintas dentro de estos "comportamientos inesperados". Así, se distinguirá entre defecto (o falta), error y fallo propiamente dicho [1].

- *Defecto*: Habrá un defecto o falta cuando se dé alguna condición que no se contempló al diseñar el componente; esto es, una condición anómala. Ante esa condición el componente es incapaz de reaccionar. Ejemplos: Corte de alimentación eléctrica, entrada imprevista, interferencias, error de diseño…
- *Error*: Manifestación de un defecto en el sistema. El estado de alguno de los recursos del sistema o componentes de la aplicación diferirá del que sería previsible según las

entradas recibidas hasta el momento y la ejecución llevada a cabo. Ejemplos: Retornar un resultado incorrecto (que no cuadre con las entradas facilitadas y la operación solicitada), no llegar a proporcionar ningún resultado (porque el ordenador sobre el que se ejecutaba el servicio se ha estropeado y es incapaz de continuar)…

- *Fallo*: Incapacidad para que un elemento desarrolle aquellas funciones para las que fue diseñado debido a errores en el propio elemento o en su entorno, causados por diferentes defectos.
  En ocasiones los errores advertidos son transitorios, debido a algún defecto puntual que se llega a subsanar rápidamente. En esos casos no se llega a hablar de fallo.
  Si el fallo se produce, esta definición nos recuerda que a esa situación se llega por la aparición de un defecto que el sistema no ha podido manejar, que a su vez ha generado un error, dejando algún componente inoperativo. Si ese componente hubiese estado replicado y se emplease un protocolo de replicación adecuado, esa situación no habría llegado a ser observada por el usuario. En ese caso habría un error local en cierta réplica del componente pero no se llegaría al fallo de todo el componente.
  Por tanto, un fallo es el resultado de dos transiciones previas (defecto a error, error a fallo) y habría que intentar que ambas transiciones no llegasen a darse nunca en nuestro sistema. Esto queda reflejado en la Ilustración 1.

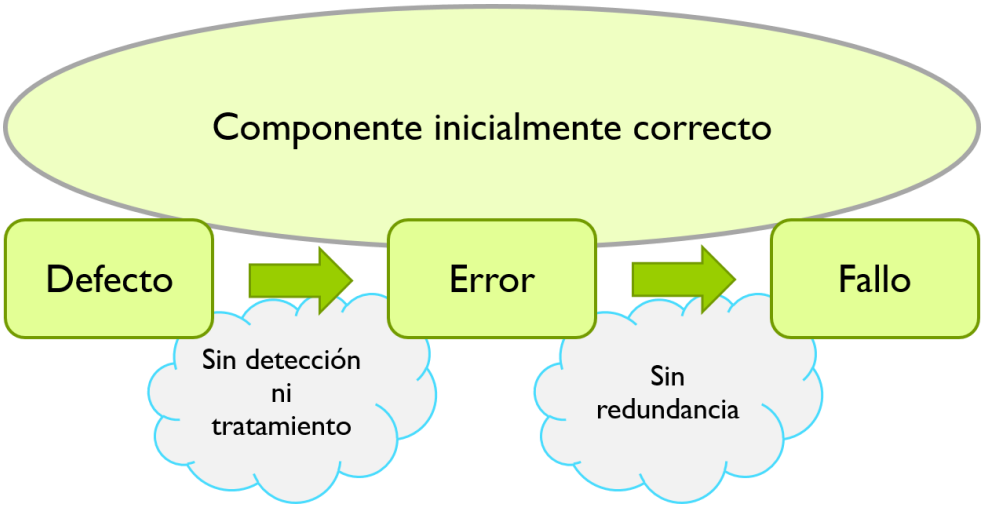

Ilustración 1: Transiciones en una situación de fallo.

Aunque la replicación permita superar las situaciones de error, logrando que el usuario no observe el fallo de un componente, hay que realizar ciertas tareas cuando alguna de las réplicas deje de funcionar. La primera de esas tareas es una detección y diagnóstico adecuados. Es decir, darse cuenta de que ha habido un error en una réplica (detección), averiguar la causa (diagnóstico) y tomar las medidas adecuadas para que no se repita. Esto conduce a la parada de la réplica, la reparación del elemento erróneo, la reconfiguración de la réplica y la recuperación de ésta. Recuperación significa que una vez reparada y reconfigurada (solucionando el problema) hay que facilitarle todas las modificaciones de su estado que se hayan perdido durante ese intervalo de inactividad. Una vez la recuperación termine, la réplica mantendrá ya un estado coherente con el de las demás réplicas y será capaz de atender las nuevas solicitudes de manera normal.

La descripción realizada hasta el momento ha estado centrada en el concepto de fallo. En algunas ocasiones se incide en la capacidad para tolerar defectos (se dice que un sistema o componente es "*fault-tolerant*"). A pesar de que el enfoque parezca distinto, el objetivo sigue siendo el mismo: aunque haya defectos, esos defectos no deben provocar fallos. Por tanto, la tolerancia a defectos y la transparencia de fallos son dos formas de referirse a cómo se gestiona un mismo problema.

Para finalizar, cabe resaltar que un servicio A no podrá ser tolerante a defectos si no lo son también todos los componentes de los que él dependa. Observe que un defecto en alguno de esos componentes se convertiría en un fallo. El servicio A habría solicitado alguna operación a ese componente que ha fallado y no obtendría ninguna respuesta (o quizá obtuviese una respuesta errónea, pero resulta difícil decidir qué es peor) por lo que tampoco podría realizar sus tareas y también generaría errores y fallos.

1.2. Modelos de fallos

Durante la ejecución de una aplicación distribuida podrían darse múltiples tipos de fallos. Dependerá de los defectos que ocurran y del tratamiento que pueda darse a cada uno de ellos. Si hablamos de servicios distribuidos que deban estar siempre disponibles y deban atender a un alto número de usuarios, habrá que desplegarlos sobre un alto número de máquinas. Será bastante común que algunas de esas máquinas lleguen a fallar si consideramos un intervalo de estudio prolongado. Así, habrá múltiples tipos de fallos y los fallos podrán afectar a varios nodos del sistema.

En el campo de la teoría de los sistemas distribuidos ha habido diversas propuestas de clasificación de los fallos, identificando múltiples modelos [1] [2] [3]. ¿Por qué hablamos de "modelos" [4]? Porque un modelo no considera los detalles concretos de cada escenario. Se utiliza un alto nivel de abstracción y se centra la discusión en los aspectos más generales e importantes. El objetivo es caracterizar adecuadamente cada variante.

Schneider [4] propuso una clasificación basada en los siguientes modelos de fallos, en la que se considera que los fallos únicamente afectan a un proceso o a un enlace de comunicación:

- *Parada*: Un proceso falla parando. Una vez parado, permanecerá siempre en ese estado. Este hecho podrá ser detectado por otros procesos.
- *Caída*: Un proceso falla parando. Una vez parado, permanecerá siempre en ese estado. Este hecho puede que no sea detectable por otros procesos.
- *Caída y enlace*: Un proceso falla parando. Una vez parado, permanecerá siempre en ese estado. Un enlace falla perdiendo algunos mensajes, pero no retrasa, duplica ni corrompe los mensajes entregados.
- *Omisión de recepciones*: Un proceso falla recibiendo sólo un subconjunto de los mensajes que se le han enviado o parando y permaneciendo parado.
- *Omisión de envíos*: Un proceso falla enviando sólo un subconjunto de los mensajes que debía enviar o parando y permaneciendo parado.
- *Omisión general*: Se combina la omisión de envíos y la omisión se recepciones.
- *Bizantinos o arbitrarios*: Un proceso falla exhibiendo un comportamiento arbitrario.

Este listado también establece cierta gradación entre los modelos. El primer modelo citado (parada) es el que ofrecerá un entorno más fácil de manejar para los algoritmos que se diseñen sobre él. Es el entorno más favorable posible. Por el contrario, también es el modelo más difícil de proporcionar. Resulta complejo que cuando un proceso tenga algún defecto se consiga pararlo antes de que otros procesos interactúen con él y observen un comportamiento distinto al esperado.

En el extremo opuesto, un modelo de fallos arbitrarios es complejo de manejar para los algoritmos que deban diseñarse sobre ese entorno, mientras que no exige nada especial al sistema para que el comportamiento de los procesos en caso de fallo cumpla con lo que establece el modelo.

En ninguno de estos modelos se habla sobre recuperaciones. No es motivo de preocupación. En algún momento un componente que ha fallado tendrá que recuperarse. El modelo de parada parece que lo prohíba pero no es exactamente así. El componente en cuestión podrá ser recuperado. Cuando eso suceda se habrá tenido que generar un nuevo proceso para ofrecer la funcionalidad de ese componente. Ese nuevo proceso tendrá su propia identidad. Por ello, el proceso que falló quedará parado indefinidamente.

Lo que sí debe considerarse con cuidado es de qué forma se ubicarán los procesos en los diferentes ordenadores del sistema y qué dependencias podrán existir entre esos ordenadores a la hora de que haya fallos. Por ejemplo, los procesos a analizar pueden ser los utilizados para replicar un componente de una aplicación distribuida. Cuando haya un fallo, interesaría que ese fallo afectara a un número mínimo de ordenadores y procesos. Así, los ordenadores utilizados deberían ubicarse en segmentos independientes de la red eléctrica gestionados por diferenciales distintos. A su vez, los ordenadores deberían tener más de una tarjeta de red, para que el acceso a la red de interconexión esté replicado y cuando falle un "*switch*" se pueda seguir conectado.

Por último, si los mecanismos de replicación son adecuados, cuando haya algún problema en una réplica, el componente replicado no fallará y de esa manera esa situación de fallo no se propagará a otros componentes. Para ello, la interacción entre réplicas de un servicio A con réplicas de otro servicio B será dinámica. Si alguna réplica de B llega a caer, las réplicas de A que la estuvieran utilizando pasarán a utilizar cualquier otra. Con esta estrategia se logra la contención de los fallos.

Hasta el momento solo se han considerado los fallos en un proceso o en un enlace. Hay que estudiar también qué ocurre cuando consideramos conjuntos de nodos. En algunos casos una misma aplicación distribuida estará desplegada sobre ordenadores ubicados en varias redes locales. En esos escenarios puede ocurrir que haya intervalos en los que la comunicación entre las diferentes redes no sea posible. Es lo que se conoce como una "partición" de la red. Cuando se dé una partición un determinado subconjunto de nodos queda aislado del resto del sistema. Como mínimo el sistema considerado inicialmente se habrá fragmentado en dos (o más) subgrupos de nodos. Dentro de cada subgrupo la comunicación entre los diferentes nodos es posible, pero no lo es entre nodos ubicados en diferentes subgrupos.

Cuando se da una partición en la red, hay dos formas básicas de afrontarla [5]:

- *Sistemas particionables*. Cada uno de los subgrupos aislados puede continuar con su trabajo. Si en cada uno de los subgrupos en que ha quedado dividido el sistema hubiese una réplica de un determinado componente, queda claro que las modificaciones aplicadas en estas réplicas inconexas no se podrían aplicar sobre las demás ubicadas en los otros subgrupos. Por ello se perdería temporalmente la consistencia entre las réplicas.

  Los sistemas particionables necesitan algún protocolo de reconciliación para decidir de qué manera se reintegrarán todas las modificaciones aplicadas en cada réplica cuando los subgrupos recuperen su conectividad. La reconciliación no será compleja si todas las operaciones admitidas durante esta situación de partición de la red son conmutativas. En ese caso no importa en qué orden se apliquen las operaciones (al ser conmutativas entre sí), pero sí que todas las réplicas hayan llegado a recibir el mismo conjunto de solicitudes.
- *Modelo de subgrupo primario* (también llamado *componente primario* o *partición primaria*). En este modelo sólo se admite que continúe aquel subgrupo que tenga una mayoría de los nodos que formaban el sistema. Los demás subgrupos deberán parar. Esto es, no servirán ninguna petición solicitada por los usuarios. De esta manera se asegura que los usuarios que reciban respuesta a sus peticiones observen siempre respuestas consistentes.

  No obstante, en una partición puede ocurrir que no haya ningún subgrupo mayoritario. En ese caso deberían quedar parados todos los subgrupos. Por ejemplo, si el sistema considerado estuviese formado por 40 nodos y la partición generara tres subgrupos de 18, 12 y 10 nodos cada uno, ninguno de ellos sería “mayoritario”. Por tanto, la actividad en este sistema quedaría bloqueada.

2. Replicación

Como ya se ha sugerido en las secciones anteriores al describir el concepto de fallo y los procedimientos de recuperación, la replicación es el mecanismo básico para proporcionar transparencia de fallos y asegurar la disponibilidad de los servicios de un sistema distribuido.

A la hora de replicar deben considerarse algunos aspectos:

- Las réplicas deben ubicarse en ordenadores que no dependan de una misma fuente de fallos. Por ejemplo, la alimentación eléctrica es crítica para mantener a un elemento en ejecución. Las máquinas que soporten a las réplicas no deberían estar ubicadas en un mismo segmento de la instalación eléctrica ni depender del mismo diferencial. A su vez, cada una de esas máquinas debería disponer de al menos dos tarjetas de red, de manera que si hay alguna avería en una de esas redes locales, las demás puedan utilizarse sin problemas.
- Deben revisarse con cuidado las dependencias entre diferentes componentes de nuestro sistema distribuido. Cuando alguna de las réplicas de un componente falle, ese fallo debe aislarse para que el resto de los componentes que utilicen sus servicios no advierta esa situación. Las peticiones que haya en curso en la réplica defectuosa tendrán que ser reasignadas a otra réplica correcta y ser reanudadas sin que deba intervenir ningún operador o administrador.

Además de su aportación en la gestión de fallos, la replicación también tiene efectos positivos sobre el rendimiento y la escalabilidad. Aquellas operaciones que solo necesiten consultar el estado de un determinado servicio podrán ser ejecutadas por una sola réplica, sin necesidad de que las demás intervengan. Por ello, este tipo de consultas obtienen una escalabilidad lineal en un sistema replicado.

Las operaciones que modifiquen el estado del servidor resultarán algo más complejas. Su petición (o bien sus efectos) tendrá que llegar a todas las réplicas del servicio y eso requerirá cierto esfuerzo para propagarla y atenderla. En estos casos la escalabilidad no mejora en gran medida. Afortunadamente, la mayor parte de las operaciones solicitadas suelen ser consultas. Por ello, la replicación mejora generalmente la escalabilidad.

Aunque hay otras opciones que no vamos a describir en este documento, la solución comúnmente aceptada en los sistemas actuales [6] es que las operaciones que únicamente realicen consultas se ejecuten únicamente en una réplica. Por su parte, aquellas operaciones que generen alguna modificación deben ser ejecutadas en todas las réplicas del servicio.

El protocolo utilizado para gestionar las operaciones que impliquen alguna modificación depende del tipo de sentencias utilizadas.

Si la operación es breve y modifica un alto porcentaje del estado, interesará propagar la petición a todas las réplicas, para que todas la reciban y la puedan ejecutar localmente. De esa manera no hay que realizar ninguna propagación posterior. Como la petición suele necesitar un mensaje más pequeño que la transferencia de estado, así se ahorra ancho de banda y tiempo en la gestión.

Por el contrario, cuando la ejecución de la operación es costosa en cuanto a esfuerzo y tiempo de cómputo, pero solo modifica una pequeña parte del estado del servicio, entonces interesará dirigirla a una sola réplica, que la ejecutará localmente, para luego transferir las modificaciones efectuadas al resto de réplicas. Así, esas otras réplicas no tendrán que ejecutar todas las sentencias de la operación sino solo esperar los mensajes de propagación y aplicar esos pocos cambios sobre sus propios objetos.

Habrá que prestar atención a la secuencia seguida en cada réplica para recibir y atender esa lista de operaciones. Si la secuencia no es idéntica en todas ellas surgirán ciertas divergencias entre los estados de las réplicas. El grado de divergencia admisible definirá el modelo de consistencia utilizado. Estos modelos de consistencia serán explicados en la sección 4.

Veamos ahora qué modelos de replicación existen para realizar la gestión de las operaciones en un servicio replicado. Los dos modelos clásicos son el pasivo [7] y el activo [8].

2.1. Modelo de replicación pasivo

En el modelo de replicación pasivo los clientes envían sus peticiones a una réplica especial: la réplica primaria. En un servicio replicado bajo este modelo sólo podrá haber una réplica primaria en todo momento. Esta réplica primaria servirá la petición localmente, ejecutando todas sus sentencias. Al terminar, propagará las modificaciones a las demás réplicas (conocidas como réplicas secundarias) y responderá al cliente.

En caso de operaciones que solo realicen consultas, la especificación original del modelo pasivo [7] exigía que también fueran dirigidas a la réplica primaria. Sin embargo, en múltiples sistemas se ha admitido que las réplicas secundarias puedan atender este tipo de peticiones, pues esto no puede generar inconsistencias y mejora la escalabilidad.

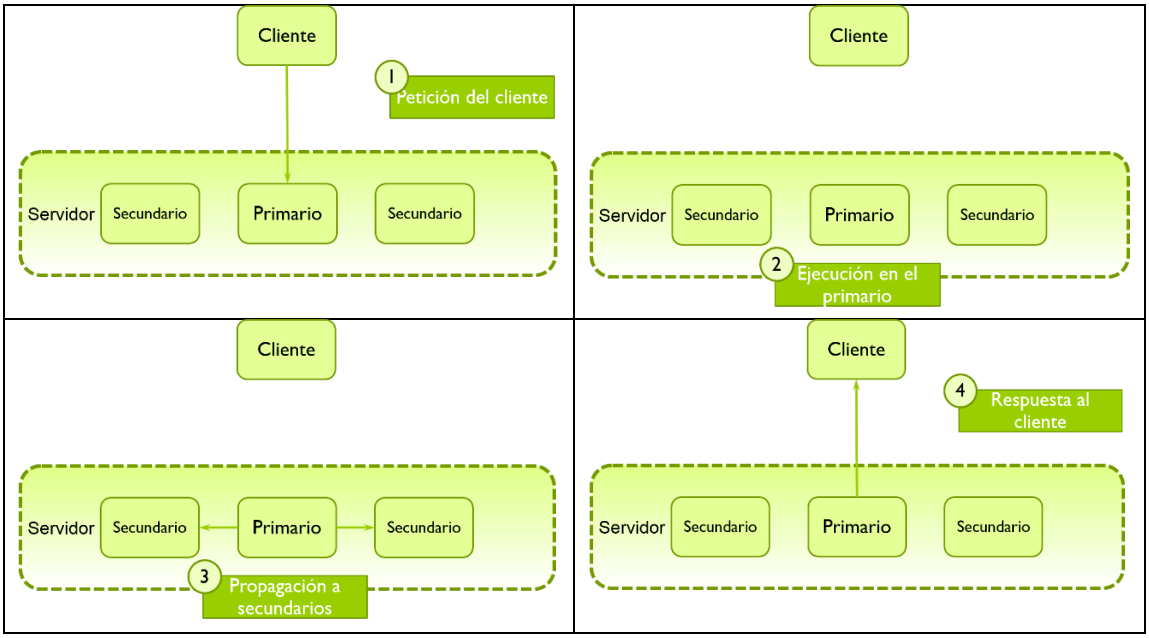

Ilustración 2. Pasos de una operación en el modelo pasivo.

La Ilustración 2 resume los pasos que se distinguen en la ejecución de una operación en este modelo de replicación, según ya se ha explicado anteriormente. Este modelo de replicación es el adecuado para aquellas operaciones "costosas" (con un intervalo de cómputo largo) que solo modifican una pequeña parte del estado del objeto. Así, el primario es el responsable de la ejecución directa (y costosa) de la operación solicitada por el cliente, mientras que los secundarios solo deben recibir el estado modificado (que es poco, según se ha comentado), actualizando su copia local. Esto no sobrecarga en exceso la red y permite que los secundarios finalicen rápidamente esta gestión.

Ventajas

Este modelo de replicación ofrece las siguientes ventajas:

- Mínima carga. La parte pesada de la gestión de una operación recae exclusivamente en la réplica primaria. Las réplicas secundarias tienen que llevar a cabo poco trabajo. Esto permitirá que si en un determinado sistema hay N nodos, se puedan replicar en él N servicios, eligiendo cada uno de estos servicios un nodo distinto como réplica primaria.
- Orden de ejecución de las operaciones fácil de establecer. La réplica primaria debe numerar consecutivamente las difusiones que envíe a las réplicas secundarias. De esa manera se asegura un mismo orden de aplicación de las modificaciones en todas las réplicas, obteniendo una consistencia fuerte.
- Basta con un control de concurrencia local. Este modelo admite ejecución concurrente. Para ello, el primario puede iniciar la ejecución de una nueva operación antes de que haya finalizado la anterior. En ese caso bastará con algún mecanismo de

concurrencia local (“locks”, semáforos, monitores…) para garantizar una ejecución correcta.

- Se admiten operaciones no deterministas. Una operación no determinista es aquella que ante los mismos argumentos de entrada no genera los mismos resultados. Esto suele deberse a que cuando llega a una sentencia donde deba realizarse alguna selección (condiciones, sentencias “switch”…), no basa exclusivamente la decisión tomada en las entradas recibidas.

  Como en el modelo pasivo sólo hay una réplica que ejecuta las operaciones que impliquen modificaciones, los resultados de las decisiones que esta réplica tome serán propagados y aplicados por todas las demás réplicas. Por tanto, las operaciones no deterministas no podrán generar inconsistencias en este modelo.

Inconvenientes

El modelo pasivo presenta estos inconvenientes:

- Cuando falle la réplica primaria hay que realizar algunas tareas. Se debe seleccionar una nueva réplica primaria de entre las secundarias que hubiese antes del fallo. Además, los clientes (en su *stub* cliente o en su *proxy*) deberían recibir algún tipo de excepción notificando la caída del primario y deberían reintentar la operación sobre el primario que se llegue a seleccionar.
- No se soporta el modelo de fallos bizantino. Como el cliente solo recibe una respuesta es incapaz de advertir si esta respuesta “cuadra” con lo esperado o no, pues desconoce el estado actual del servidor. Por tanto, un comportamiento arbitrario por parte de la réplica primaria del servidor no siempre podrá ser detectado.
- En algunos modelos de fallos se necesitará un alto número de réplicas para asegurar la continuidad del servicio y la capacidad de retornar una respuesta. Por ejemplo, en el modelo de fallos de omisión general, si se pueden llegar a dar “f” fallos simultáneos, se necesitará tener al menos 2f+1 réplicas para asegurar un comportamiento correcto [7]. Recuerde que para replicar un servicio bajo el modelo pasivo jamás debería tenerse más de un primario para ese servicio.

  Veamos qué ocurriría si solo hubiese 2f réplicas. Esas réplicas podrían estructurarse en dos conjuntos disjuntos A y B, cada uno con f réplicas.
  - Si en una ejecución del servicio cayesen todas las réplicas de A, entonces se pasaría a tener un primario en B.
  - Si en otra ejecución del servicio cayesen todas las réplicas de B, entonces se pasaría a tener un primario en A.
  - Si en una tercera ejecución del servicio las réplicas de A tuvieran fallos de omisión general en los canales que las comunican con los nodos de B, las réplicas de B pasarían a comportarse como en la primera de las ejecuciones citadas, y se elegiría un primario en B. A su vez, las de A tampoco se podrían comunicar con las réplicas de B y entonces en A también se elegiría otro primario. Como resultado tendríamos dos réplicas primarias en este servicio y se violarían las condiciones dictadas por la especificación del modelo pasivo.

Para evitar que esto ocurra debería haber al menos otra réplica a través de la cual intercomunicar a los nodos de A con los de B en el tercer escenario descrito, evitando que se eligieran simultáneamente dos primarios.

2.2. Modelo de replicación activo

En el modelo de replicación activo [8], los clientes difunden sus peticiones a todas las réplicas del servidor. Cada réplica servidora ejecuta la operación solicitada. Cuando termina, responde al cliente. Las interacciones llevadas a cabo en este modelo de replicación se muestran en la Ilustración 3.

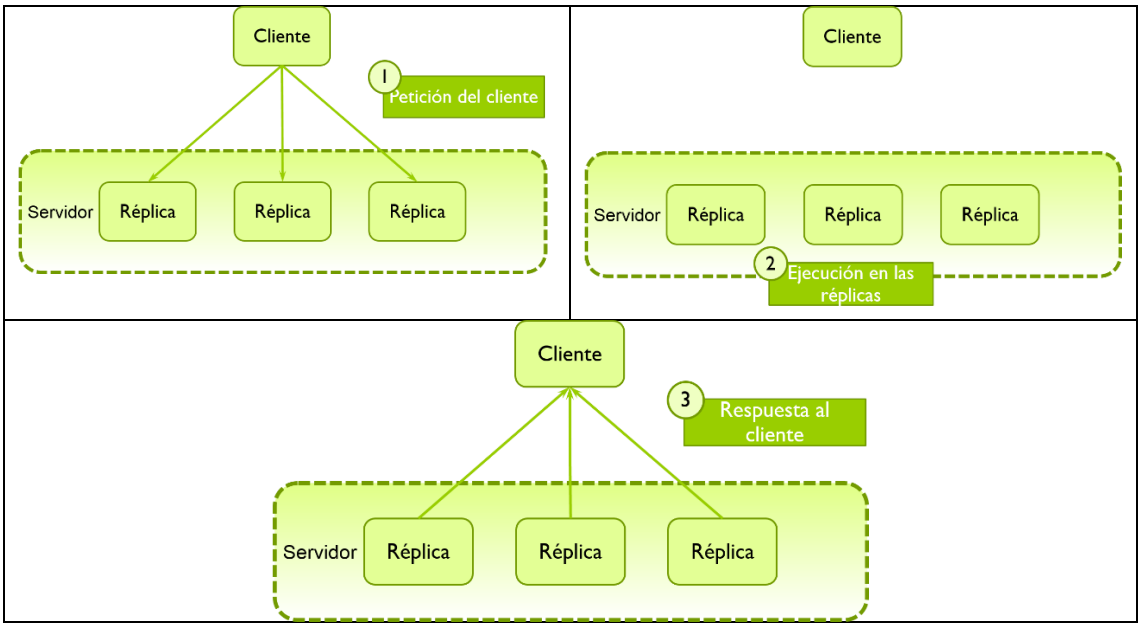

Ilustración 3. Pasos en una operación del modelo activo.

En esta ocasión solo necesitamos tres pasos, mientras que en el modelo pasivo había que realizar cuatro. Sin embargo, el primer paso del modelo activo requiere que la difusión con la que se propague la petición del cliente establezca un orden total con todas las demás peticiones iniciadas por cualquier otro cliente. El protocolo necesario para establecer ese orden total es bastante pesado. Implica resolver un problema de consenso distribuido y este problema no tiene solución en un entorno asincrónico donde los procesos puedan fallar [9].

En el tercer paso, el cliente obtendrá una respuesta por parte de cada réplica servidora. Esto no siempre será un inconveniente, como veremos seguidamente.

Ventajas

Comparado con el modelo pasivo, este modelo de replicación ofrece dos ventajas importantes:

- La reconfiguración del servicio replicado en caso de fallo es inmediata. Mientras queden suficientes réplicas vivas, no hay que hacer nada. Como todas las réplicas son idénticas no hay que hacer ningún tipo de promoción o activación. Todas las que queden activas seguirán comportándose de igual forma. Por ello, no habrá ningún intervalo de indisponibilidad causado por fallos.
- Este modelo de replicación sí que tolera los fallos bizantinos. En un entorno favorable (parcialmente sincrónico y donde pueda acreditarse la identidad de los emisores de

cada mensaje) para tolerar "f" fallos de este tipo basta con tener 2f+1 réplicas. Cuando un cliente reciba las respuestas, si no hay fallos, todas esas respuestas serán idénticas. Si hubiera un fallo arbitrario, la réplica defectuosa estaría devolviendo un resultado distinto. Si es previsible que como máximo haya "f" fallos simultáneos, instanciaremos 2f+1 réplicas del servidor y el cliente esperará respuestas hasta haber recibido f+1 respuestas idénticas.

Si únicamente pudieran darse fallos de parada, bastaría con esperar la primera respuesta y bastaría con tener f+1 réplicas.

Inconvenientes

No obstante, hay también algunos inconvenientes serios:

- Como se ha sugerido arriba, cuando se deba proporcionar una consistencia fuerte, las peticiones efectuadas por los clientes deben recibirse en orden total en las réplicas del servicio. Es decir, todas las réplicas deben observar la misma secuencia de peticiones.
- Las operaciones no deterministas generarán inconsistencias en este modelo. Como cada réplica debe ejecutar todas las operaciones y las no deterministas llegan a generar resultados que no dependen exclusivamente de los argumentos de entrada recibidos, cada réplica podría generar un resultado diferente para una misma operación no determinista. Esto provocaría divergencias entre los estados de las réplicas de un mismo servicio.
- Si un servicio A replicado bajo el modelo activo invoca una operación de otro servicio B replicado de manera activa, habrá que filtrar en B las N réplicas de la petición de una misma operación que habrán llegado a cada réplica de B. Es decir, eliminar N-1 de esas repeticiones y asegurar que cada petición sólo se atienda una vez. En caso contrario se ejecutaría múltiples veces cada operación en cada réplica y eso no debe ocurrir.

2.3. Comparativa

Analizaremos seguidamente qué modelo de replicación trata mejor cada uno de los aspectos que debe considerarse a la hora de replicar un servicio. Posteriormente, ampliaremos lo que se había comentado previamente sobre qué modelo proporciona mejores prestaciones dependiendo de lo que dure el procesamiento de cada operación, la cantidad de estado modificado y el esfuerzo que implica la propagación de ese estado.

Aspecto	Modelo pasivo	Modelo activo
Réplicas procesadoras	**1**	Todas
Evita la ordenación distribuida de las peticiones	**Sí**	No
Evita propagación de modificaciones	No	**Sí**
Admite indeterminismo	**Sí**	No
Tolera fallos arbitrarios	No	**Sí** (Necesita que cada réplica procese también las lecturas)
Consistencia	Al menos, secuencial	Al menos, secuencial
Recuperación en caso de fallo	Elección y reconfiguración cuando falla el primario	**Inmediata**

Tabla 1.Comparativa de modelos de replicación

Aspectos generales

Las principales ventajas e inconvenientes de cada modelo pueden resumirse en la Tabla 1, donde se resalta en negrita la alternativa más recomendable.

Además de esa comparativa entre los aspectos a tener en cuenta, es también conveniente presentar algunos casos de estudio que permitan establecer finalmente una recomendación general sobre qué modelo de replicación permitirá obtener un mejor rendimiento a la hora de desplegar un conjunto de servicios que utilicen replicación.

Caso 1. Escenario 1

Supongamos un servicio S1 con estas características:

- 5 réplicas
- Ancho de banda de la red: 1 Gbps
- Tiempo de propagación en los canales de comunicación: 2 ms. Válido como tiempo de transferencia de los mensajes pequeños.
- El tiempo medio de respuesta para las operaciones es 200 ms, pero no se asume concurrencia en este tiempo base. Cuando se procesan N peticiones concurrentemente por los hilos de un mismo proceso, entonces el tiempo de respuesta pasa a ser 200*N.
- El tamaño medio de las modificaciones realizadas en cada operación es 10 KB, que expresado en bits sería 80 Kb = 0.08 Mb = 0.00008 Gb. Con ello, el tiempo de transferencia de las modificaciones pasa a ser (considerando que la transferencia incluye tanto la propagación como la transmisión de los mensajes y que esta última depende del ancho de banda) 0.002 + 0.00008 s = 2.08 ms
- El tiempo de aplicación de modificaciones en las réplicas secundarias es 3 ms.

Con estos datos, ¿qué modelo de replicación tendrá menor tiempo de respuesta? Calculémoslo seguidamente...

Aspecto	Modelo pasivo		Modelo activo
	Réplica primaria	Réplica secundaria	
Entrega de la petición	2 ms	--	4 – 6 ms (con un algoritmo de difusión de orden total basado en secuenciador)
Procesamiento de la petición	200 ms	--	200 ms
Transferencia de modificaciones	2.08 ms		0 ms
Aplicación de modificaciones	--	3 ms	0 ms
Respuesta a modificaciones	2 ms		0 ms
Respuesta al cliente	2 ms	--	2ms
TOTAL:	211.08 ms		206 – 208 ms

En este ejemplo, el modelo pasivo necesita un promedio de 211.08 ms para gestionar cada petición recibida. Inicialmente, se necesitan 2 ms para hacer llegar el mensaje de petición a la réplica primaria. Asumimos que el cliente se encuentra en la misma red que los servidores. Ese intervalo inicial podrá ser más prolongado en otros casos. Posteriormente, la réplica primaria invierte 200 ms en procesar localmente esa petición. A continuación transfiere las modificaciones que haya generado esa petición al resto de réplicas. Asumimos que hay diferentes canales punto a punto entre el primario y los secundarios, con lo que resultará posible transferir esas modificaciones en paralelo. En las réplicas secundarias se invertirán 3 ms para aplicar esos cambios sobre su estado local. Posteriormente, se invertirán 2 ms en enviar un reconocimiento a la réplica primaria. Tras haber recibido todos esos reconocimientos, la réplica primaria responde al cliente, necesitando también 2 ms para ello. En total, se han utilizado 211.08 ms para completar toda esta secuencia de pasos.

En el modelo activo, el mensaje de petición puede enviarse inicialmente a una réplica bien conocida dentro del conjunto, que posteriormente la reenviará a todas las demás para dar por iniciado el procesamiento de la petición. Ese tipo de difusión de orden total se llama difusión basada en secuenciador y es la más rápida de entre todas las alternativas. Ese secuenciador actúa como punto central de ordenación para todos los mensajes que se deban difundir. Existen otras variantes que utilizan tres rondas de propagación de mensajes en lugar de las dos que acabamos de describir. Por ello, esta difusión inicial necesita entre 4 y 6 ms para ser completada. Posteriormente, cada réplica invierte 200 ms en el procesamiento de la petición y 2 ms más para responder al cliente. Con ello, el tiempo total invertido en la gestión de una petición oscila entre 206 y 208 ms en promedio (asumiendo una carga mínima, al igual que en el modelo pasivo).

Caso 1. Escenario 2

A primera vista, los resultados del escenario anterior son similares en ambos modelos. Sin embargo, debemos considerar que cada réplica ha sido desplegada en un ordenador diferente. Si necesitáramos desplegar cinco servicios S1 en esos ordenadores, cada servicio podría tener su réplica primaria en un ordenador diferente en caso de utilizar el modelo de replicación pasivo, pero el activo debería soportar una carga cinco veces mayor en todas sus réplicas.

Con ello, los nuevos resultados serían ahora:

Aspecto	Modelo pasivo		Modelo activo
	Réplica primaria	Réplica secundaria	
Entrega de la petición	2 ms	--	4 – 6 ms
Procesamiento de la petición	200 +4*3 = 212 ms	--	200*5=1000 ms
Transferencia de modificaciones	2.08 ms		0 ms
Aplicación de modificaciones	--	3 ms	0 ms
Respuesta a modificaciones	2 ms		0 ms
Respuesta al cliente	2 ms	--	2 ms
TOTAL:	223.08 ms		1006 – 1008 ms

Así, el modelo pasivo es claramente más eficiente en este segundo escenario.

Caso 2. Escenario 2

Supongamos servicios S2 con estas características:

- 5 réplicas
- Ancho de banda de la red: 1 Gbps
- Tiempo de propagación en los canales de comunicación: 2 ms. Válido como tiempo de transferencia de los mensajes pequeños.
- El tiempo medio de respuesta para las operaciones es 5 ms, pero no se asume concurrencia en este tiempo base. Cuando se procesan N peticiones concurrentemente por los hilos de un mismo proceso, entonces el tiempo de respuesta pasa a ser 5*N.
- El tamaño medio de las modificaciones realizadas en cada operación es 10 MB, que expresado en bits sería 80 Mb = 0.08 Gb. Con ello, el tiempo de transferencia de las modificaciones pasa a ser (considerando que la transferencia incluye tanto la propagación como la transmisión de los mensajes y que esta última depende del ancho de banda) 0.002 + 0.08 s = 82 ms
- El tiempo de aplicación de modificaciones en las réplicas secundarias es 3 ms.

Con estos datos, ¿qué modelo de replicación tendrá menor tiempo de respuesta? Calculémoslo seguidamente asumiendo que van a desplegarse cinco servicios diferentes de esta clase S2 en un mismo conjunto de cinco máquinas...

Aspecto	Modelo pasivo		Modelo activo
	Réplica primaria	Réplica secundaria	
Entrega de la petición	2 ms	--	4 – 6 ms (con un algoritmo de difusión de orden total basado en secuenciador)
Procesamiento de la petición	5 + 4*3=17 ms	--	5*5=25 ms
Transferencia de modificaciones	82 ms		0 ms
Aplicación de modificaciones	--	3 ms	0 ms
Respuesta a modificaciones	2 ms		0 ms
Respuesta al cliente	2 ms	--	2ms
TOTAL:	108 ms		31 – 33 ms

En este segundo caso, el modelo activo es el claro vencedor.

Análisis

Si tenemos en cuenta sus prestaciones, el modelo de replicación elegido depende principalmente de dos factores:

- El tiempo promedio de procesamiento de las peticiones. El modelo pasivo es apropiado cuando el intervalo de procesamiento sea prolongado, pues solo afecta a su réplica primaria, mientras que en el modelo activo afecta a todas sus réplicas.

- El tamaño de las modificaciones. El modelo activo es especialmente apropiado para aquellos servicios que modifiquen una gran cantidad de espacio, pues esas modificaciones no necesitan ser empaquetadas en mensajes, transferidas y aplicadas en otras réplicas. Por el contrario, el modelo pasivo solo será asumible si las modificaciones realizadas afectan a un estado muy pequeño, pues en este modelo hay que empaquetar esas modificaciones, transferirlas y aplicarlas en las réplicas secundarias.

3. Consistencia

Cuando se replica información en múltiples procesos, un modelo de consistencia especifica qué divergencias se admiten entre los valores de las réplicas de un mismo elemento.

A la hora de especificar un modelo de consistencia se suele asumir que:

- Uno de los procesos realiza la escritura inicial.
- Esa escritura se propaga posteriormente al resto de réplicas.
- Cada uno de los demás procesos leerá ese nuevo valor si lo necesitara. Basta con leer su copia local.

Las diferencias entre un modelo y otro se deben a los retardos en la propagación de las escrituras y a las condiciones de corrección que exige cada modelo.

Aunque existen varias propuestas para clasificar los modelos de consistencia, en esta asignatura sólo estudiaremos dos de esas alternativas:

- Los modelos de consistencia centrados en los datos [10] [11] [12], limitando su estudio a aquellos que no requieran herramientas de sincronización.
- El concepto de consistencia final ("*eventual consistency*") [13].

3.1. Consistencia centrada en datos

Si no se asume el uso de herramientas de sincronización los modelos más importantes son, empezando por los más exigentes y terminando por los más relajados: estricto, lineal, secuencial, causal, procesador, FIFO y caché. Veamos seguidamente una breve descripción de cada uno de ellos.

Consistencia estricta

En la consistencia estricta [14] se asume que la propagación de las escrituras es inmediata y que mientras se ejecute una escritura no se puede estar ejecutando ninguna otra escritura. Es el modelo que se obtendría en una máquina con un solo procesador y un solo núcleo.

Su implantación práctica en un entorno distribuido requiere excesivos bloqueos, por lo que el sistema resultante sería ineficiente.

Para establecer una guía sobre qué implica esta consistencia estricta se utilizarán intervalos temporales. Cada vez que algún proceso realice alguna acción de escritura (por ejemplo, para escribir el valor V sobre cierta variable "x") cerrará el intervalo existente hasta ese momento (en el que esa variable "x" había tenido el valor W) y abrirá un nuevo intervalo asignado a ese

nuevo valor V. Todas las lecturas que se realicen mientras perdure ese intervalo actual (es decir, hasta que empiece la siguiente escritura sobre "x") tendrán que devolver el valor V.

Un ejemplo de ejecución bajo el modelo de consistencia estricta sería el mostrado en la Ilustración 4.

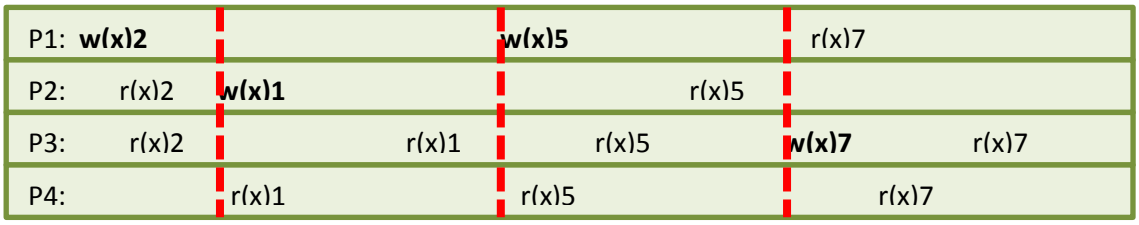

Ilustración 4. Ejecución bajo el modelo estricto.

En este ejemplo, la secuencia global de escrituras ha sido P1:w(x)2, P2:w(x)1, P1:w(x)5 y P3:w(x)7. Los valores leídos siempre coinciden con el último valor escrito en todo el sistema. Esto se debe a que el tiempo necesario para propagar los valores escritos se asume que es cero.

Consistencia secuencial

En la consistencia secuencial [15], el resultado de una ejecución es el mismo que si las operaciones de todos los procesos se ejecutaran en algún orden secuencial, y las operaciones de cada proceso aparecieran en dicha secuencia en el orden que dicta su respectivo programa.

Esto implica que todos los procesos ven todas las escrituras en un mismo orden y, además, ese orden respeta lo que ha escrito originalmente cada proceso en su nodo. A diferencia del modelo estricto, esto no exige que todos los procesos avancen a un mismo ritmo. Cada uno de ellos puede ver las escrituras de esa secuencia común antes o después que algunos de los demás. Además, ese orden global acordado entre todos no tiene por qué coincidir siempre con el orden real en que se han realizado las escrituras en diferentes nodos.

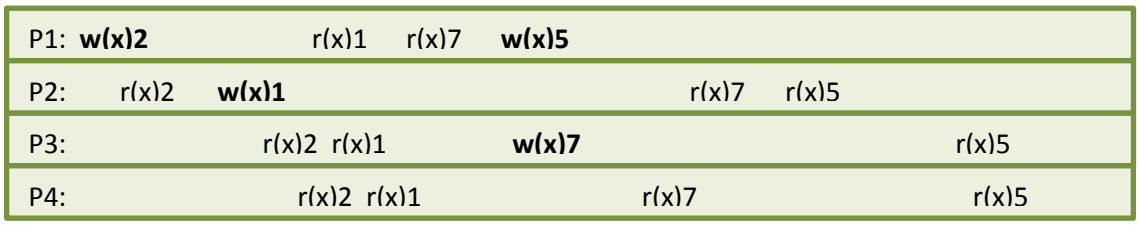

Ilustración 5. Ejemplo de ejecución bajo el modelo secuencial.

Así, por ejemplo, aunque físicamente las escrituras de la Ilustración 5 han sucedido en el orden P1:w(x)2, P2:w(x)1, P1:w(x)5 y P3:w(x)7, el orden acordado ha sido P1:w(x)2, P2:w(x)1, P3:w(x)7 y P1:w(x)5. Además, los procesos P3 y P4 han sido capaces de leer el valor 2 cuando el valor 1 ya había sido escrito. Tanto esa alteración del orden global (entre los valores 5 y 7) como esas inversiones de lectura (leer el valor 2 cuando ya se había escrito el valor 1) no se admitirían en el modelo de consistencia estricto.

Consistencia causal

Este modelo de consistencia asume que las operaciones de escritura (que deben propagarse a otros nodos) pueden verse como eventos de envío de ciertos mensajes y que las operaciones de lectura (que solo podrán suceder cuando las escrituras hayan sido recibidas en el proceso

lector) pueden verse como eventos de recepción. En base a esto, la consistencia causal respeta la relación "happens before" definida por Leslie Lamport en [16] y que ya fue estudiada en CSD para definir los relojes lógicos.

Por ello, dos escrituras "a" y "b" estarán ordenadas como "a < b" (o "a --> b") si ambas han sido generadas por un mismo proceso en ese mismo orden, o bien la escritura "b" ha sucedido en un proceso P1 distinto al escritor de "a" pero tras haber leído en P1 el valor de "a". Es decir, ha ocurrido algo así: P1:w(x)a, P2:r(x)a, P2:w(x)b. Además, esta relación de orden cumple la propiedad transitiva.

Así, por ejemplo, en la ejecución mostrada en la Ilustración 6, la escritura del valor 2 precede causalmente a la escritura del valor 1 y éste precede causalmente tanto al valor 5 como al valor 7. Sin embargo, los valores 5 y 7 son concurrentes entre sí. Por tanto, los procesos tendrán libertad para ordenar los valores 5 y 7 como prefieran. De esta manera, P1 y P2 observan el valor 5 antes que el valor 7, mientras que los procesos P3 y P4 han visto el valor 7 antes que el valor 5. Como la secuencia no es idéntica en todos los procesos, esta ejecución incumple las condiciones de la consistencia secuencial.

P1:	**w(x)2**			r(x)1	**w(x)5**	r(x)7		
P2:	r(x)2	**w(x)1**				r(x)5	r(x)7	
P3:			r(x)2	r(x)1	**w(x)7**			r(x)5
P4:			r(x)2	r(x)1		r(x)7		r(x)5

Ilustración 6. Ejemplo de ejecución bajo el modelo causal.

Consistencia FIFO

En esta consistencia (también llamada pRAM) se requiere que las escrituras realizadas por cada proceso sean obtenidas en el orden en que su escritor las generó. Sin embargo, no impone ninguna restricción sobre la forma de "mezclar" las escrituras generadas por procesos diferentes.

P1:	**w(x)2**			r(x)1	**w(x)5**	r(x)7		
P2:	r(x)2	**w(x)1**				r(x)7	r(x)5	
P3:			r(x)1		**w(x)7**		r(x)2	r(x)5
P4:			r(x)1	r(x)2		r(x)7		r(x)5

Ilustración 7. Ejemplo de ejecución bajo el modelo FIFO.

Por ejemplo, en la ejecución mostrada en la Ilustración 7 hay cuatro escrituras, pero solo dos de ellas han sido generadas por un mismo proceso (los valores 2 y 5 han sido escritos por el proceso P1). Por tanto, lo único que se exige en este ejemplo es que en cualquiera de los procesos la escritura del valor 2 se reciba antes que el valor 5. Los valores 1 y 7 pueden intercalarse libremente en la secuencia que observe cada proceso. De hecho, en este ejemplo las secuencias observadas en todos los procesos son distintas.

También se puede comprobar que esta ejecución no cumple con ninguno de los modelos anteriores (causal, secuencial y estricto). No es causal porque todos deberían obtener el valor 2 antes que el 1 y eso no sucede en P3 ni en P4. No es secuencial porque no todos los procesos

respetan una misma secuencia de valores escritos en sus lecturas. No es estricta porque algunos procesos han leído el valor 2 (por ejemplo, P4) cuando ya se había leído en los demás el valor 1 que fue escrito posteriormente.

El orden FIFO también afecta a las escrituras realizadas sobre variables diferentes. En la Ilustración 8 se observa que la escritura del valor 5 sobre 'z' realizada por P1 siempre se recibirá después de la escritura del valor 2 sobre 'x' realizada por ese mismo proceso.

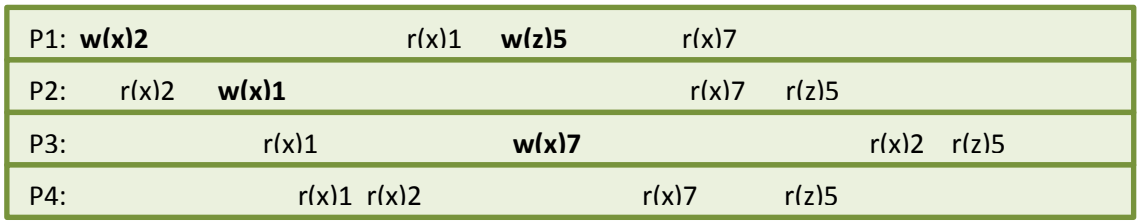

Ilustración 8. Otro ejemplo de ejecución bajo el modelo FIFO.

Consistencia caché

La consistencia caché exige que las escrituras realizadas sobre una misma variable sean obtenidas en un mismo orden por todos los procesos lectores. Además, si varias escrituras sobre una misma variable han sido realizadas por un mismo proceso, ese orden del proceso escritor será utilizado globalmente. No se impone ninguna restricción a la hora de mezclar las escrituras sobre diferentes variables.

Por ejemplo, asumamos un sistema donde los procesos P1 y P2 modifican las variables 'x' y 'z'. Las escrituras deben respetar la restricción 'z>=x+2'. La consistencia caché no sería suficiente para mantener esta imagen en todo el sistema. La Ilustración 9 muestra que aunque P1 y P2 usen instrucciones que respetan la restricción cuando son ejecutadas, reciben posteriormente los valores escritos por el otro proceso que violan esa condición. Al final de la ejecución, ningún proceso mantiene valores que respeten esa restricción impuesta en el programa. Así, todos terminan este fragmento de su ejecución con el valor 2 para 'x' y el valor 3 para 'z'.

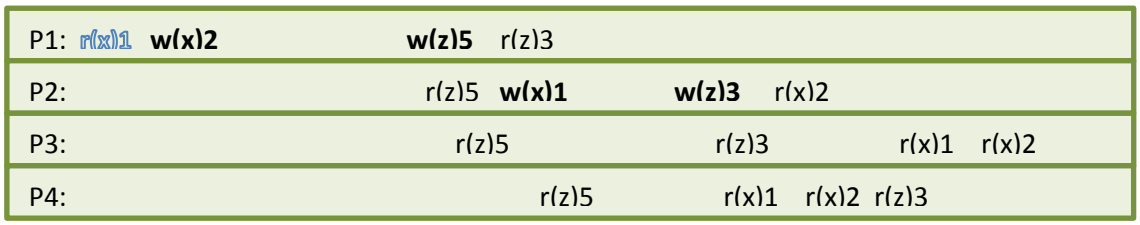

Ilustración 9. Ejecución en el modelo caché.

Para que se respete esa restricción se necesitaría una consistencia secuencial, como muestra la Ilustración 10. En la consistencia secuencial todos los procesos del sistema llegan a un mismo orden para todas las lecturas. Además ese orden común de lectura debe ser consistente con lo que han escrito localmente P1 y P2 (sobre cualquier variable; esto es, consistencia FIFO) y con las dependencias entre P1 y P2 (consistencia causal).

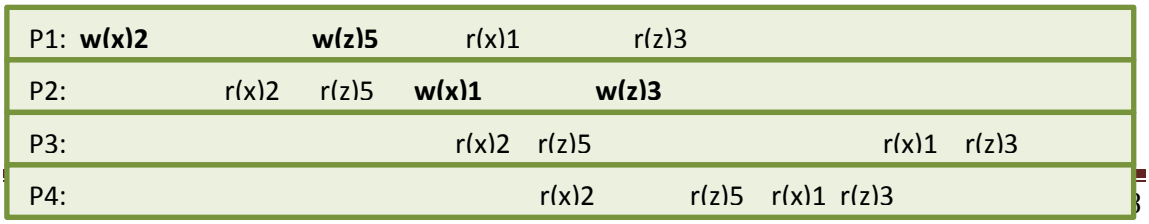

Ilustración 10. Ejecución bajo el modelo secuencial del ejemplo presentado en la ilustración anterior.

Consistencia procesador[1]

Este modelo exige que se respeten simultáneamente las condiciones de los modelos FIFO y caché. Por tanto, se debe llegar a un acuerdo sobre el orden de escritura sobre cada una de las variables (tal como exige el modelo caché), con libertad para intercalar las escrituras sobre diferentes variables (siempre que no incumpla las condiciones del modelo FIFO), y también que se respete el orden de escritura de cada proceso (modelo FIFO), de nuevo con libertad para intercalar las escrituras de diferentes procesos (siempre que no incumpla las condiciones del modelo caché).

P1:	**w(x)2**	**w(z)5**	r(x)1	r(z)3
P2:	r(x)2	r(z)5	**w(x)1**	**w(z)3**
P3:	r(x)2	r(x)1	r(z)5	r(z)3
P4:	r(x)2	r(z)5	r(x)1	r(z)3

Ilustración 11. Ejecución en el modelo procesador del ejemplo de la ilustración 9.

Si tomamos como referencia el ejemplo presentado en la Ilustración 9, así como las restricciones que se comentaron en aquella sección, veremos que el modelo procesador evita las incoherencias que mostraron los procesos en aquella Ilustración 9. De hecho, con este modelo procesador se habrían podido generar las mismas secuencias ofrecidas por los procesos P3 y P4 en la Ilustración 10, que eran correctas en cuanto a las restricciones que imponía el ejemplo. No obstante, este modelo procesador es más relajado que el secuencial. P3 y P4 no presentan ambos la misma secuencia de lecturas.

Jerarquía de modelos

Los modelos de consistencia que se han presentado en este documento pueden ordenarse según su grado de exigencia. El modelo estricto es el modelo que impone una consistencia más fuerte (o estricta), mientras que los modelos FIFO y caché solo son capaces de asegurar unas consistencias muy débiles (o relajadas). El resultado es un orden parcial que se muestra en la Ilustración 12.

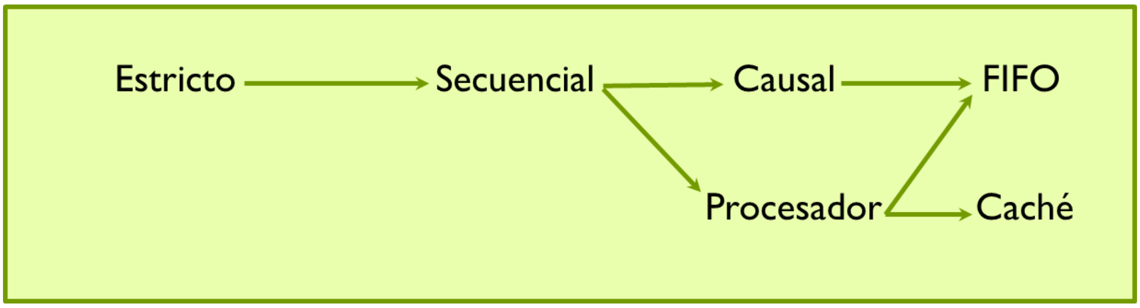

Ilustración 12. Jerarquía de modelos de consistencia.

[1] El modelo de consistencia procesador no será objeto de examen durante el presente curso. Se incluye en esta guía por completitud.

Esto indica que cuando una ejecución satisfaga el modelo estricto, también será una ejecución secuencial; por ser secuencial será causal y por ser causal también será FIFO. Que se cumpla el modelo secuencial también implica que se respetará el modelo procesador. Si se respeta el procesador, se respetarán también tanto el FIFO como el caché.

Sin embargo, hay parejas de modelos que no se pueden comparar entre sí. Los modelos causal y procesador son incomparables. Esto implica que ni el causal es más exigente que el procesador ni el procesador más exigente que el causal. Habrá ejecuciones que cumplirán alguno de esos dos modelos pero no satisfarán el otro. Algo similar ocurre entre el modelo causal y el caché y entre los modelos FIFO y caché.

Cuándo dos modelos serán comparables puede quedar más claro si representamos gráficamente cada modelo como un conjunto de ejecuciones que respetan sus condiciones. Esa representación se muestra en la Ilustración 13. Como podemos ver, las flechas utilizadas en la Ilustración 12 para expresar que un modelo A es más fuerte que otro modelo B (A --> B) se muestran ahora definiendo a A como un subconjunto de B. Por ejemplo, todas las ejecuciones del modelo estricto están contenidas en los demás conjuntos, pues el modelo estricto es el más fuerte de todos. Por otra parte, hay algunas ejecuciones secuenciales que no son estrictas, pero todas las ejecuciones secuenciales son también causales, caché y FIFO; eso implica que el modelo secuencial sea un subconjunto de los conjuntos causal, FIFO y caché. Esas relaciones de inclusión también se dan entre los modelos causal y FIFO, procesador y caché y procesador y FIFO.

El modelo de consistencia procesador no ha podido mostrarse explícitamente en la figura. Debe entenderse que es la intersección entre los modelos caché y FIFO. Por ello, el conjunto procesador incluye como uno de sus subconjuntos al secuencial.

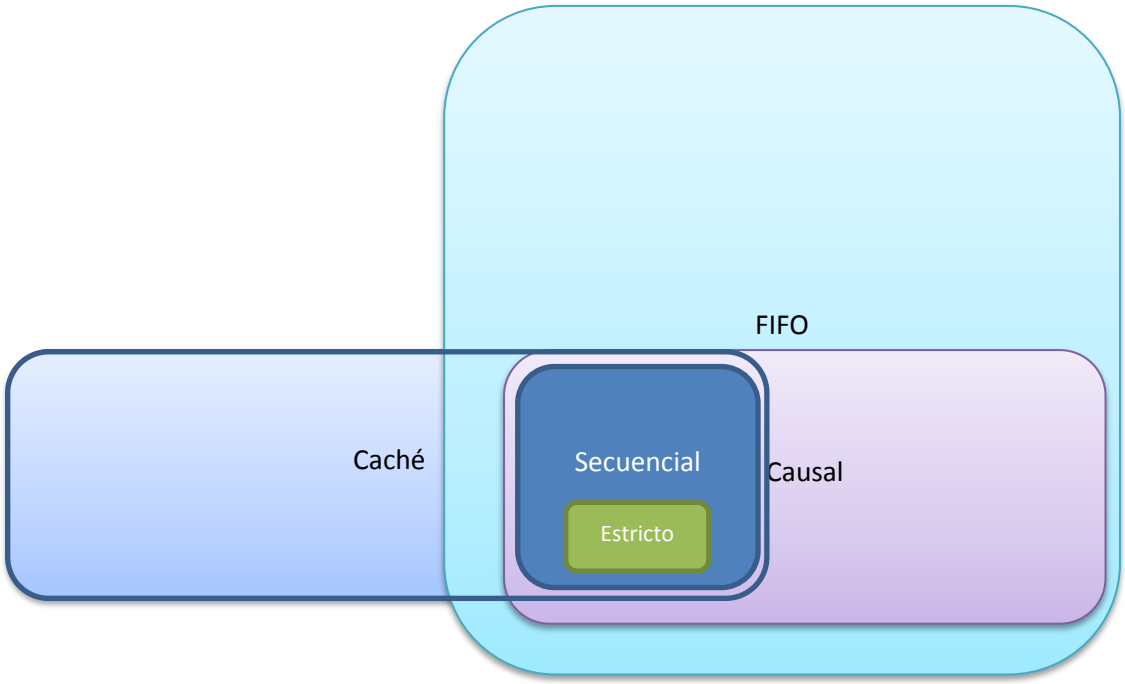

Ilustración 13. Representación alternativa de la jerarquía de modelos de consistencia.

Identificaremos seguidamente algunos ejemplos de ejecución que ilustrarán por qué algunos modelos son incomparables con otros; es decir, no son subconjuntos ni superconjuntos de algún otro. Esos ejemplos también nos ayudarán a entender las diferencias entre esos modelos. Para empezar, la ejecución E1 mostrada en la Ilustración 14 pertenece al subconjunto del modelo caché que no está incluido en el conjunto del modelo FIFO. Es una ejecución caché, pues solo se escribe un valor en cada variable. Por tanto, trivialmente respeta la consistencia caché (pues todos los procesos han observado la misma secuencia en cada variable: una secuencia formada por un solo valor). Sin embargo, no respeta la consistencia FIFO pues P1 escribió el valor 2 en "x" antes de escribir el valor 5 en "z" pero P2 recibe esos dos valores en el orden contrario. Si E1 no es FIFO, tampoco será procesador, causal, secuencial ni estricta.

P1:	**w(x)2**								**w(z)5**										
P2:											r(z)5			r(x)2					

Ilustración 14. Ejecución E1: caché y no FIFO.

La Ilustración 15 muestra otra ejecución E2 que es FIFO pero no respeta los modelos causal ni caché. En este caso, P1 había escrito un valor inicial (2) en "x" que fue propagado a P2 y leído antes de que P2 escribiera el valor 1 en esa misma variable. Sin embargo, ambos valores llegan en el orden contrario a P3. Eso rompe la consistencia causal, pues el valor 1 pudo estar causado por el valor 2 (es decir, hay una secuencia de comunicación que empieza con w(x)2 y termina con w(x)1). La consistencia FIFO todavía se mantiene (trivialmente) pues cada escritor ha generado únicamente un valor.

Como E2 no es causal, tampoco podrá ser secuencial o estricta. No es secuencial porque P3 no ha visto la misma secuencia de valores que P1 y P2. P3 recibe el valor 1 antes que el valor 2, mientras P1 y P2 han visto el valor 2 antes que el 1.

P1:	**w(x)2**												r(x)1						
P2:					r(x)2			**w(x)1**											
P3:										r(x)1							r(x)2		

Ilustración 15. Ejecución E2: FIFO, pero sin ser causal ni caché.

La Ilustración 16 muestra una ejecución E3 que es causal pero no es secuencial. Es causal porque las dos condiciones del modelo causal se respetan. P1 escribe dos valores (1 en "x" y 5 en "z", en ese orden) y esos valores son recibidos en ese orden por los demás procesos (solo P2 en este ejemplo). Además, hay un camino de comunicación entre el valor 1 escrito por P1 y el valor 2 escrito por P2. El valor 1 precede causalmente al valor 2. Esa dependencia es respetada por ambos procesos, pues P1 también recibe el valor tras escribir el valor 1.

P1:	**w(x)1**								**w(z)5**		r(x)2								
P2:			r(x)1				**w(x)2**										r(z)5		

Ilustración 16. Ejecución E3: causal y caché, pero no secuencial.

Los valores 2 y 5 no guardan dependencias causales. Son concurrentes. Por ello, no importa (para el modelo causal) en qué orden esas escrituras sean leídas por los procesos. De hecho, P1 observa el valor 5 antes que el 2 mientras P2 los ve en el orden contrario. Esto incumple la consistencia secuencial, pues en ella se exige que todos los procesos vean una misma secuencia de valores. Como E3 no es secuencial, tampoco será estricta.

Esta ejecución E3 respeta el modelo caché. Ha habido dos escrituras en la variable "x" y sus valores han sido vistos en el mismo orden en todos los procesos (1 antes que 2). Solo ha habido un valor en "z". Por tanto, E3 respeta las condiciones del modelo caché. Como E3 es causal y todas las ejecuciones causales son también FIFO, E3 también respeta la consistencia procesador, pues ese modelo es la unión de las consistencias caché y FIFO.

P1:	**w(x)1**		**w(x)5**	r(x)2
P2:	r(x)1	**w(x)2**		r(x)5

Ilustración 17. Ejecución E4: causal, pero sin ser caché ni secuencial.

Finalmente, la ejecución E4 (mostrada en la Ilustración 17) es una variación de la ejecución E3 que sigue respetando los modelos causal y FIFO, pero no el caché y tampoco el secuencial. Su tercera escritura ha sido reemplazada por una escritura en "x". Eso provoca que se incumpla la consistencia caché, pues P1 ha observado la secuencia 1, 5, 2 para esa variable mientras P2 ve una secuencia diferente: 1, 2, 5.

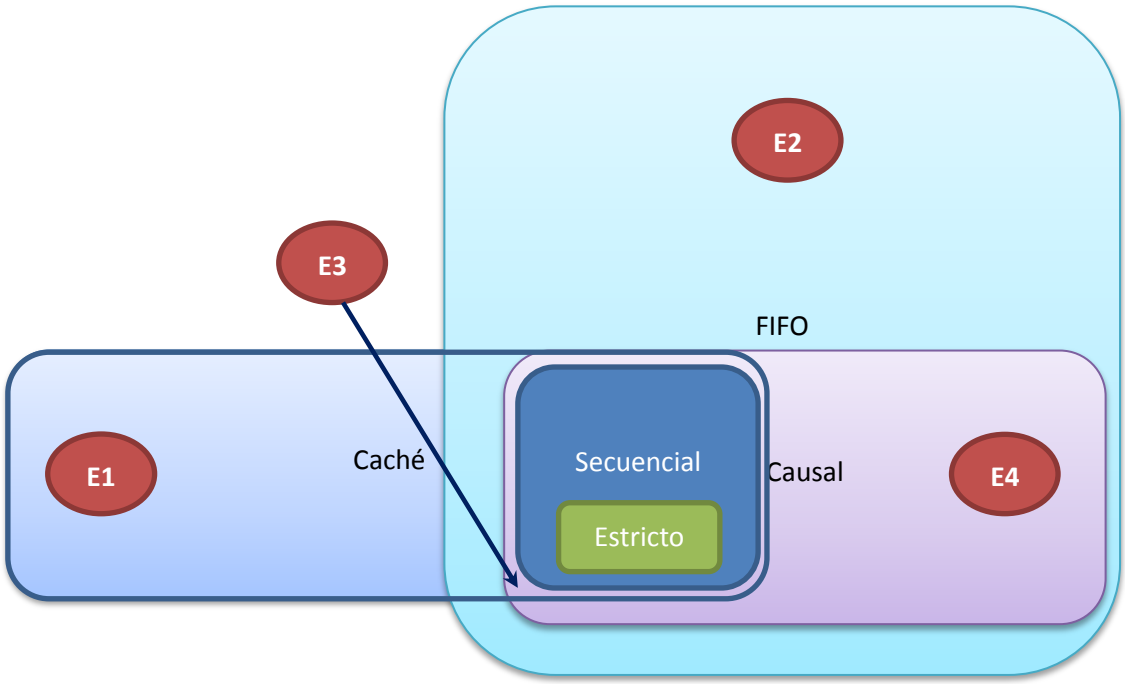

Ilustración 18. Representación gráfica de las ejecuciones E1, E2, E3 y E4 en la jerarquía de modelos.

Estas cuatro ejecuciones (E1, E2, E3 y E4) han presentado ejemplos concretos de las relaciones entre los modelos de consistencia descritos en este documento. También proporcionan una guía para discutir qué modelos de consistencia son respetados en una ejecución. La Ilustración

18 muestra sus posiciones en la colección de conjuntos inicialmente presentada en la Ilustración 13.

Modelos rápidos

Como ya se ha avanzado en secciones anteriores, necesitamos los modelos de consistencia para identificar qué grado de divergencia admitiremos entre las réplicas de un determinado elemento. Esas réplicas necesitan algún protocolo de replicación que controle cómo se propagan las modificaciones de estado entre todas ellas. El protocolo de replicación intentará implantar alguno de los dos modelos de replicación clásicos o bien alguna variante intermedia.

Cuando el protocolo de replicación utilizado permita que tanto las operaciones de escritura como las de lectura sobre un dato local puedan darse por terminadas sin necesidad de intercambiar ningún mensaje con las demás réplicas, diremos que el modelo de consistencia que se está soportando es un modelo "rápido" [17]. Los modelos rápidos de consistencia propagan las modificaciones de manera diferida y jamás bloquean los accesos de lectura. Los accesos de lectura serían bloqueantes cuando el protocolo de replicación suspendiera al proceso lector mientras no hubiese llegado un mensaje desde otras réplicas.

Los modelos de consistencia estricto, secuencial, procesador y caché[2] no son rápidos puesto que requieren acuerdos entre todos los procesos que repliquen algún dato sobre el orden en que deben recibirse sus modificaciones. Los modelos FIFO y causal son rápidos pues existen implantaciones de protocolos de replicación para estos modelos donde tanto las operaciones de lectura como escritura retornan el control inmediatamente. Esos protocolos no incurren en ningún bloqueo para las lecturas (causado por esperar algún mensaje remoto) ni esperan a la transmisión del valor hacia otras réplicas en las operaciones de escritura.

3.2. Consistencia final

La consistencia final (del inglés "*eventual consistency*") fue definida en [13] como una condición que llegará a cumplirse cuando haya un intervalo suficientemente largo en el que no haya escrituras. Esa condición de corrección podría enunciarse de la siguiente manera: "las réplicas de un determinado elemento llegarán a tener un mismo valor cuando haya un intervalo suficientemente largo sin nuevas escrituras".

Es decir, en un sistema con consistencia final hay libertad completa para que las escrituras se realicen en cualquier réplica y se propaguen en el orden y con el retardo que cada escritor crea conveniente. Sin embargo, cuando se llegue a tener todo el conjunto de escrituras generadas por todos los nodos del sistema, cada nodo sabrá cómo aplicarlas para obtener un mismo valor final. La clave para que esto suceda es sencilla: en lugar de especificar las escrituras como la asignación de un valor, cada escritura debería convertirse en una operación conmutativa con todas las demás.

[2] Existe un algoritmo [21] que implanta el modelo de consistencia caché con operaciones rápidas que no llegan a bloquear al proceso que las realiza (tanto en escrituras como en lecturas), pero no tolera fallos en los procesos ni situaciones de particionado de la red. Además, propaga sus difusiones de manera sincrónica, evitando que se realicen nuevas difusiones mientras no se haya entregado la actual. No obstante, de manera general, en un entorno donde pueda haber fallos de los procesos y particiones en la red, el modelo de consistencia caché no debería considerarse rápido.

Por ejemplo, supongamos que el valor inicial de la variable 'x' fue 530 y, por orden, diferentes procesos han realizado ciertas operaciones que habrían conducido a esta secuencia de nuevos valores en caso de haber tenido un sistema con consistencia estricta: 520, 600, 650, 640, 700. En lugar de realizar directamente esas asignaciones, cada proceso debería haber consultado el valor local que tenía la variable 'x' en ese momento, convirtiendo así esas asignaciones en las siguientes operaciones: x:=x-10, x:=x+80, x:=x+50, x:=x-10, x:=x+60. Tras realizar esa conversión, cada escritor debe propagar la operación efectuada a todas las demás. No importará para nada el orden en que esas operaciones se reciban en cada uno de los nodos. Todos sabrán que una vez hayan recibido todas las operaciones el valor mantenido en su copia de la variable será idéntico al que tendrán las demás, siempre que haya un intervalo suficientemente largo en el que no se apliquen nuevas operaciones sobre esa variable. En el ejemplo propuesto, todas las réplicas llegarían al mismo valor final: x=700. Sin embargo, si cada réplica recibió las operaciones en diferente orden, cada una habrá visto una secuencia de valores intermedios distinta. Durante esa etapa de propagación y aplicación la consistencia entre esas réplicas ha sido muy relajada.

Analicemos con mayor detenimiento esa ejecución, para que quede más claro este ejemplo.

- Valor inicial de la variable 'x': 530.
- Operaciones efectuadas:
    - op1: x:=x-10;
    - op2: x:=x+80;
    - op3: x:=x+50;
    - op4: x:=x-10;
    - op5: x:=x+60;
- Réplicas existentes: P1, P2, P3, P4, P5.
- Orden de ejecución de las operaciones en cada réplica. Las cinco primeras operaciones aplicadas fueron: P1:op1, P2:op2, P3:op3, P4:op4 y P5:op5, respetando lo que se sugería en la descripción de este ejemplo. Sin embargo, ninguno de estos cinco procesos fue capaz de recibir las modificaciones realizadas por los demás antes de aplicar la suya propia. Finalmente, el orden de ejecución de cada proceso es el que aparece a continuación:
    - P1: op1, op2, op3, op4, op5.
    - P2: op2, op4, op3, op5, op1.
    - P3: op3, op5, op2, op4, op1.
    - P4: op4, op1, op3, op2, op5.
    - P5: op5, op4, op3, op2, op1.
- Secuencia de valores observados por cada réplica:
    - P1: 530, 520, 600, 650, 640, 700.
    - P2: 530, 610, 600, 650, 710, 700.
    - P3: 530, 580, 640, 720, 710, 700.
    - P4: 530, 520, 510, 560, 640, 700.
    - P5: 530, 590, 580, 630, 710, 700.

Además, esos valores parciales se habrían podido dar en diferentes instantes. Esto es, no todos los procesos vieron el segundo valor de esas secuencias a la vez. Cada réplica pudo avanzar a

una velocidad distinta a la de los demás. Sin embargo, al terminar esta secuencia de modificaciones, todos los procesos vuelven a obtener un mismo valor final: 700. Así se recupera la consistencia.

La consistencia final es una de las claves para mejorar tanto el rendimiento como la escalabilidad en los sistemas distribuidos [18].

Bibliografía

[1] V. P. Nelson, «Fault-Tolerant Computing: Fundamental Concepts,» *IEEE Computer,* vol. 23, nº 7, pp. 19-25, 1990.

[2] F. Cristian, «Understanding Fault-Tolerant Distributed Systems,» *Commun. ACM,* vol. 34, nº 2, pp. 57-78, 1991.

[3] V. Hadzilacos y S. Toueg, «Fault-Tolerant Broadcasts and Related problems,» de *Distributed Systems*, Addison-Wesley, 1993, pp. 97-145.

[4] F. B. Schneider, «What Good Are Models and What Models Are Good?,» de *Distributed Systems*, Addison-Wesley, 1993, pp. 17-26.

[5] G. Chockler, I. Keidar y R. Vitenberg, «Group Communication Specifications: A Comprehensive Study,» *ACM Comput. Surv.,* vol. 33, nº 4, pp. 427-469, 2001.

[6] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso y B. Kemme, «Are Quorums an Alternative for Data Replication?,» *ACM Trans. Database Syst.,* vol. 28, nº 3, pp. 257-294, 2003.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider y S. Toueg, «Optimal Primary-Backup Protocols,» de *6th International Workshop on Distributed Algorithms and Graphs (WDAG)*, Haifa, Israel, Springer, 1992, pp. 362-378.

[8] F. B. Schneider, «Implementing Faul-Tolerant Services using the State-Machine Approach: A Tutorial,» *ACM Comput. Surv.,* vol. 22, nº 4, pp. 299-319, 1990.

[9] M. J. Fischer, N. A. Lynch y M. Paterson, «Impossibility of Distributed Consensus with One Faulty Process,» *J. ACM,* vol. 32, nº 2, pp. 374-382, 1985.

[10] D. Mosberger, «Memory Consistency Models,» *Operating Systems Review,* vol. 27, nº 1, pp. 18-26, 1993.

[11] R. C. Steinke y G. J. Nutt, «A Unified Theory of Shared Memory Consistency,» *J. ACM,* vol. 51, nº 5, pp. 800-849, 2004.

[12] V. Cholvi y J. M. Bernabéu-Aubán, «Relationships between Memory Models,» *Inf. Process.*

Lett., vol. 90, nº 2, pp. 53-58, 2004.

[13] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer y B. B. Welch, «Session Guarantees for Weakly Consistent Replicated Data,» de *3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, EE.UU., IEEE-CS Press, 1994, pp. 140-149.

[14] L. Lamport, «On Interprocess Communication,» *Distributed Computing,* vol. 1, nº 2, pp. 77-101, 1986.

[15] L. Lamport, «How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,» *IEEE Trans. Computers,* vol. 28, nº 9, pp. 690-691, 1979.

[16] L. Lamport, «Time, Clocks, and the Ordering of Events in a Distributed System,» *Commun. ACM,* vol. 21, nº 7, pp. 558-565, 1978.

[17] H. Attiya y R. Friedman, «Limitations of Fast Consistency Conditions for Distributed Shared Memories,» *Inf. Process. Lett.,* vol. 57, nº 5, pp. 243-248, 1996.

[18] W. Vogels, «Eventually Consistent,» *Commun. ACM,* vol. 52, nº 1, pp. 40-44, 2009.

[19] H. Kopetz y P. Veríssimo, «Real Time and Dependability Concepts,» de *Distributed Systems*, 2ª ed., S. J. Mullender, Ed., Addison-Wesley, 1993, pp. 411-446.

[20] H. Attiya y R. Friedman, «Limitations of Fast Consistency Conditions for Distributed Shared Memories,» *Inf. Process. Lett.,* vol. 57, nº 5, pp. 243-248, 1996.

[21] E. Jiménez, A. Fernández y V. Cholvi, «A parametrized algorithm that implements sequential, causal and cache memory consistencies,» *Journal of Systems and Software,* vol. 81, pp. 120-131, 2008.