TECNOLOGÍAS DE LOS SISTEMAS DE INFORMACIÓN DE LA RED

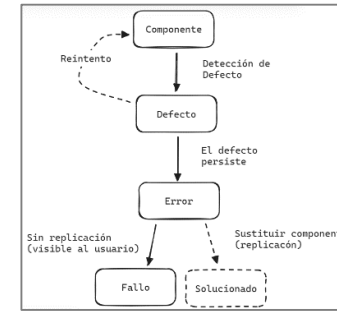
Tema 5

Miquel Gómez Corral Ing. Informática ETSINF, UPV

Tema 5

Gestión de fallos

Fallo: Se produce un fallo cuando **algún componente** del sistema es incapaz de comportarse como toca. Debemos distinguir entre defectos, errores y fallos.



- **Defecto (“*fault*”)**: **Comportamiento/Condición anómala**. Ejemplos: *Error de diseño, interferencias, entrada imprevista, corte del suministro eléctrico…* **HAY QUE REINTENTAR**. Si tras reintentar varias veces sigue fallando se cataloga de error.
- **Error**: **Manifestación de un defecto en un sistema**. El estado de algún componente diferirá de su estado previsto y su error produce que el sistema haga lo que queremos. *Se puede solucionar dándolo de baja y sustituirlo por una copia.* En caso de no haber réplicas estaríamos hablando de Fallo, xq afecta a todo el sistema.
- **Fallos (“*failure*”)**: 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**.

Transparencia/tolerancia a defectos : “*Faul tolerance*”, Tolerante a fallos no sería a fallos, es una mala traducción. Lo que es tolerante es a Defectos, si algo no funciona como toca que se puede **reestructurar el sistema para que el usuario no perciba nada raro y se le proporcione correctamente el servicio**.

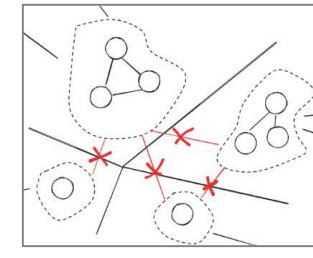
- Para que un sistema tolere defectos, **TODOS sus subservicios** HAN de ser TOLERANTES también. Tiene que usar el sistema que sea, pero que lo sean (replicación, detección de defectos…).

Sistema de detección de defecto: **Casi todas las componentes del sistema pueden fallar**. Hay que tener un sistema de detección de defectos. En **caso de tener este sistema** se puede reintentar rehacer la operación (*por si fuera un defecto transitorio y se puede solucionar apagando y encendiendo*). Si tras reintentar NO se solucionar hay un error. Aquí podemos intentar sustituir el componente (*replicación*), pero si NO tengo replicación no podría seguir ocultando el defecto y hablaríamos de fallos, visibles al usuario.

Tipos de fallos

Los fallos pueden tener muchas causas. **Depende de los defectos** que haya y a **qué componentes afecten** se pueden distinguir múltiples tipos de fallos: *Cuando se diseñan algoritmos distribuidos hay que asumir algún **modelo de fallos**.*

- Ningún modelo puede reflejar todas las situaciones de fallo. Hay casos que se pueden detectar y otros no:
  - **Sí**: **No contesta** (dentro de un plazo) o da una **respuesta absurda**.
  - **No**: **Fallo bizantino**. *Fallo que no se puede detectar*.
- Cuando hay un fallo puede ser que **solo le afecte a un componente**, maybe si es breve o mientras lo soluciono NO salen otros relacionado con él. Pero puede ser que el **defecto de este afecte a otros**.
  - **Simples**: **SOLO** afecta a **un nodo**.
  - **Compuestos**: **VARIOS fallos** en diferentes nodos simultáneamene.



Particiones/Fallos de la red: De normal **fallos Simples Detectables** cubren CASI TODOS LOS CASOS. Pero puede ser que en una red fallen las comunicaciones y varios **nodos queden aislados de otros**. Pueden comunicarse entre los nodos de una partición, pero NO entre particiones. *De normal solo habrán 2 o 3 particiones.*

Dependiendo del sistema y el servicio, puede seguir funcionando en caso de haber particiones. Pero en algunos casos esto mal (*por ejemplo vendiendo butacas, 2 grupos pueden vender la misma sin darse cuenta*).

- **Sistema particionable**: **Cada** uno de los **grupos** aislados puede **continuar con su trabajo**. Se necesitará algún protocolo de reconciliación cuando se recupere la conectividad para poner todo en conjunto. ***AP.***
- **Modelo de partición primaria**: **Sólo** se admite que continúe aquel **grupo** que tenga una mayoría de nodos. *No siempre existirá tal grupo*. SOLO un grupo funciona y se evitan inconsistencias. ***CP***.

Teorema CAP: Decía que, entre consistente, Disponible y Particiones solo es como máximo 2 cosas a la vez, NO PUEDE TENER LAS 3 PROPIEDADES A LA VEZ. En caso de particiones tenemos o AP(particionable) o CP (Partición 1).

REPLICACIÓN

Mecanismo básico para **asegurar la disponibilidad de un componente**. Replicación es cuando cada componente tiene lo que le haga falta para que funcionar. *Que no haya varias máquinas el mismo enchufe xq si quito el enchufe caen varios a la vez...*

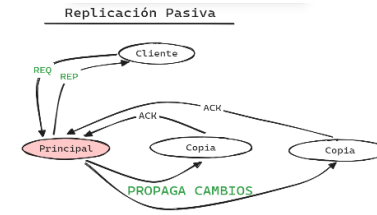
1. Cada **réplica** del componente se ubica en una máquina distinta (a poder ser en diferentes sitios físicos).
2. Cada **máquina** con fuentes diferentes (de red, eléctrica…)
3. Cuando **falle una réplica** las demás no tienen por qué fallar. *Las operaciones en curso en la réplica caída podrán completarse en una de las demás.*
4. La replicación facilita recuperarse de fallos. Las replicas activas se usan para reconfigurar el sistema.

Mejora de rendimiento

- **Lectura (*o no modificación del estado*)**: Las **operaciones no cambian nada** (*son calculadas/la respuesta SOLO depende de los argumentos, es* ***idempotente***) pueden ser ejecutadas por una o todas las réplicas, lo que hace que sea fácil de escalar y que **NO** se **tengan que actualizar** todo el rato las réplicas **comunicándose**.
- **Escritura**: Deben ser aplicadas en todas las réplicas, **dando retardos**. Se puede hacer que, si la operación es sencilla, TODAS las réplicas lo hagan en vez de pasarse los datos, sino que lo haga solo uno y envía a todas.
    - *Los estados de las réplicas pueden divergir, habría que determinar un modelo de consistencia.*

Modelos de replicación

Pasivo: El **cliente envía su petición a la réplica primaria**. Una misma réplica primaria para todos los clientes y todas las peticiones. Según se actúe tenemos dos tipos de aproximación:



- **Pesimista**: Lento, pero con garantías: 1. Petición 2. Operar (el primario) 3. Propagar cambios 4. Esperar ACKs 5. Devolver respuesta.
- **Optimista**: Rápido, pero sin garantías: 1. Petición 2. Opera (el primario) 3. Devolver respuesta 4. Propagar cambios 5. Espera ACKs

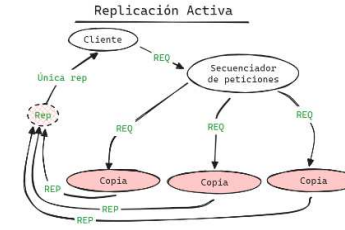
Fallo: En caso de que **falle una copia** no se hace nada. Para saber el **fallo de la copia primaria** se hace que está **mande un latido** cada x tiempo con las actualizaciones. En caso de dejar de recibir el latido es que ha fallado.

Ventajas: Mínima **carga**, **las lecturas pueden distribuirse**. El **orden de peticiones** fácil establecer y **no hacen** falta algoritmos distribuidos para **concurrencia**. Además admite operaciones NO deterministas.

Inconvenientes: Reconfiguración pesada cuando falle el primario, y NO soporta el modelo de fallos bizantinos.

Activo (*o "máquina de estados"*): El **cliente propaga su petición a todas las réplicas servidoras**. **Cada réplica servidora ejecuta directamente la operación** y cuando una réplica **termina**, responde al cliente.

Los mensajes de los clientes se tienen que secuencializar para que a todos les llegue en el mismo orden. Después se ha de devolver solo una respuesta (que al cliente no le lleguen 15 cosas). También se han de usar mecanismos de concurrencia distribuido para acceder a recursos compartidos.



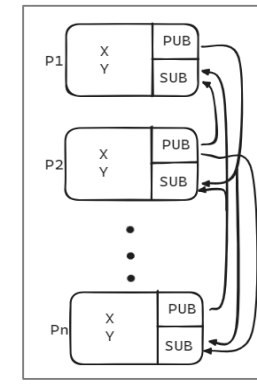
Ventajas: Reconfiguración trivial en caso de fallo ya que no haces nada, solo ignoras a la replica fallida y soporta el modelo de fallos bizantinos.

Inconvenientes: Si se necesita consistencia fuerte, las peticiones deben difundirse a todas las réplicas en orden total. No se toleran las operaciones no deterministas ya que cada réplica puede obtener soluciones diferentes.

- **Cuando interactúen servicios replicados** hay que filtrar las peticiones para evitar rehacer algunas operaciones múltiples veces (pagos, cobros…). Ejemplo:
  - El servicio A invoca al servicio B. Ambos utilizan replicación activa.
  - B tendrá que filtrar las múltiples peticiones recibidas.
  - Para evitar que B ejecute N veces la misma operación.
  - Habrá que propagar las respuestas a todas las réplicas de A.

CONSISTENCIA

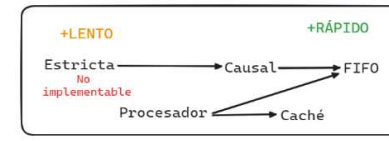
Cuando se replica información en múltiples nodos, un **modelo de consistencia** especifica qué divergencias se admiten entre los valores de las réplicas de un mismo elemento.



Lo que se hace es cuando un cliente o proceso realiza una escritura en un nodo, este **propaga los cambios del resultado a los demás**.

- La **consistencia** obtenida depende del retardo de esa propagación y las esperas que introduzca ese retardo en otros procesos.
- Para **controlar escritura** y lectura se utiliza un algoritmo de consistencia.
- Para la consistencia tendremos que ver qué hacemos: *Si nada más modifico algo, publico y sigo a mía es rápido, pero poco fiable. Pero si difundo, me espero y compruebo cosas tengo más garantías, aunque más lento.*

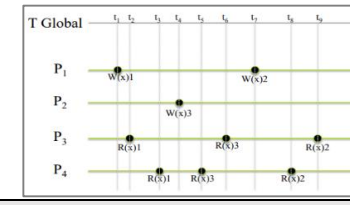
Modelos de consistencia: Estricta (lento): Solo es teórico, es ideal pero no se puede usar. Secuencial (lento), Procesador (lento), Caché (lento), Causal (rápido) y FIFO (rápido).



Consistencia final (“*eventual consistency*”): **Sistemas donde se admitan particiones**. Si admito particiones NO se puede usar los modelos de consistencia. Ahora, si hay particiones y priorizo el servicio, Disponibilidad (*que todos puedan seguir usando el servicio a pesar de las particiones*) NO tengo consistencia (**Teorema CAP**). Pero se puede hacer que cada uno vaya haciendo sus cosas y, a largo plazo, haremos que todo converja/unifique cuando todo se vuelva a restaurar: Cada uno hace sus cambios y luego **cuando se arregle todo** (al final) que todo sea consistente, **que se decida el nuevo estado conjunto**.

Consistencia estricta

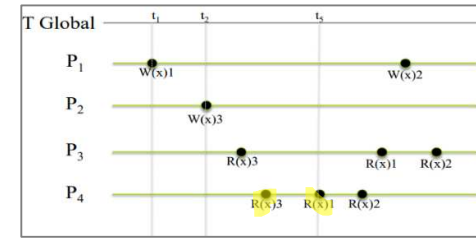
Cómo trabaja una **consistencia fuerte**. Asume un reloj global, capaz de etiquetar cada evento: **No pueden suceder dos escrituras a la vez en todo el sistema**, se asume **propagación inmediata** de las escrituras, latencia 0 (todos actualiza el estado EXACATAMENTE en el mismo modelo) y Cada lectura siempre devuelve el valor de la última escritura realizada sobre esa variable: IMPOSIBLE DE IMPLEMENTAR



Si en un momento dado una copia accede a su X, *al haberse propagado el momento,* Cualquier otro nodo lee el último valor actualizado.

Consistencia secuencial

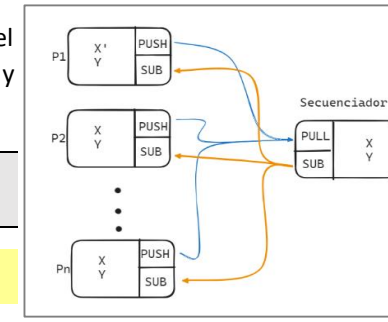
Si hay 2 modificaciones, el n1 escribe x = 3 y un poco después n1 escribe de nuevo x = 2, A TODOS les va a llegar PRIMERO x=3 antes que x=2. **Las modificaciones de un mismo nodo llegan en orden a todos los demás**.



Sin embargo, cuando hay varios nodos modificando, puede ser que los mensajes directamente NO lleguen en orden (xq uno está por medio de otro y se le mete antes de que reciba…). Pues entre todos los nodos, **SE LLEGA A UN ACUERDO** del orden a **ejecutar/ver los cambios en el mismo orden**. Todos aceptarán este orden, pero cada uno irá a su ritmo.

Esta traza es secuencial y no estricta xq SÍ que ven el mismo orden TODOS, pero NO leen siempre el valor del ÚLTIMO WRITE, pero siempre el orden de lectura es 3 1 2.

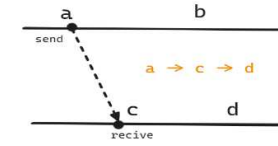
La menara de implementarlo es, todo tienen un socket PUSH y otro SUB. Con el PUSH mandan el mensaje a nodo que hace de **secuenciador**, este tiene un PULL y un PUB, será él que mande a TODOS los nodos CADA CAMBIO.



Cuando un nodo hace un cambio se espera a recibir la contestación del secuenciador de que se ha hecho pública. Es más lento, pero da garantías.

Consistencia causal

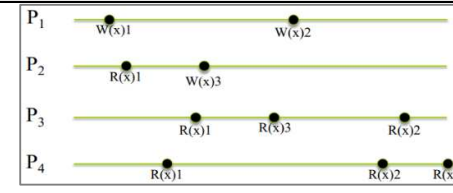
Menos estricto que el secuencial, en vez de que TODOS los eventos se acuerden de que pasen en un orden, seguras que **DENTRO DE CADA NODO va en orden**, y que **cuando se manda** un mensaje, SIEMPRE, el **evento** "envío" es **antes** que el **evento** "llegada": Se establece un orden entre ellos mediante la propiedad transitiva "-->"



SOLO establecemos un orden si se puede extrapolar con la propiedad transitiva, SI NO, se entiende que **esos eventos SON concurrentes**. Por ejemplo, no puedes establecer un orden entre b y d, pues SON concurrentes.

Aquí NO se cumple Estricta xq p3 al final está leyendo R3 pero su último valor escrito es W2. También NO es secuencial xq P3 lee (1,3,2) mientras que P4 lee (1,2,3)

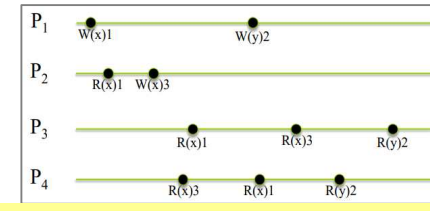
RELOJES LÓGIOS DE LAMPORT, si puede establecer "->" se respeta.



Consistencia FIFO

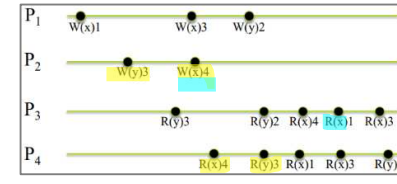
Requiere que las **escrituras realizadas por un mismo proceso sean leídas en orden de escritura por todos los demás procesos**. Pero **no impone ninguna restricción a la hora de "mezclar"** lo que han hecho diferentes escritores.

Escribir y leer en el que te llegue, ese el que pillas, aunque no sea en orden.



Consistencia caché

Lo **mismo que secuencial** pero varia en cada variable: *Las operaciones sobre x se hacen SIEMPRE en el mismo orden, las operaciones sobre y se hacen SIEMPRE en el mismo orden*, pero ENTRE ELLAS se pueden mezclar. El orden de lectura de las escrituras sobre "x" sea 4, 1, 3. El valor 1 debe estar antes que el 3 puesto que ambos fueron escritos por P1. El orden de lectura de las escrituras sobre "y" sea 3, 2. Cómo se intercalen las lecturas de "x" e "y" no es relevante.



Sería como el modelo secuencial que usa el secuenciador pero usa un secuenciador por variable y NO uno global.