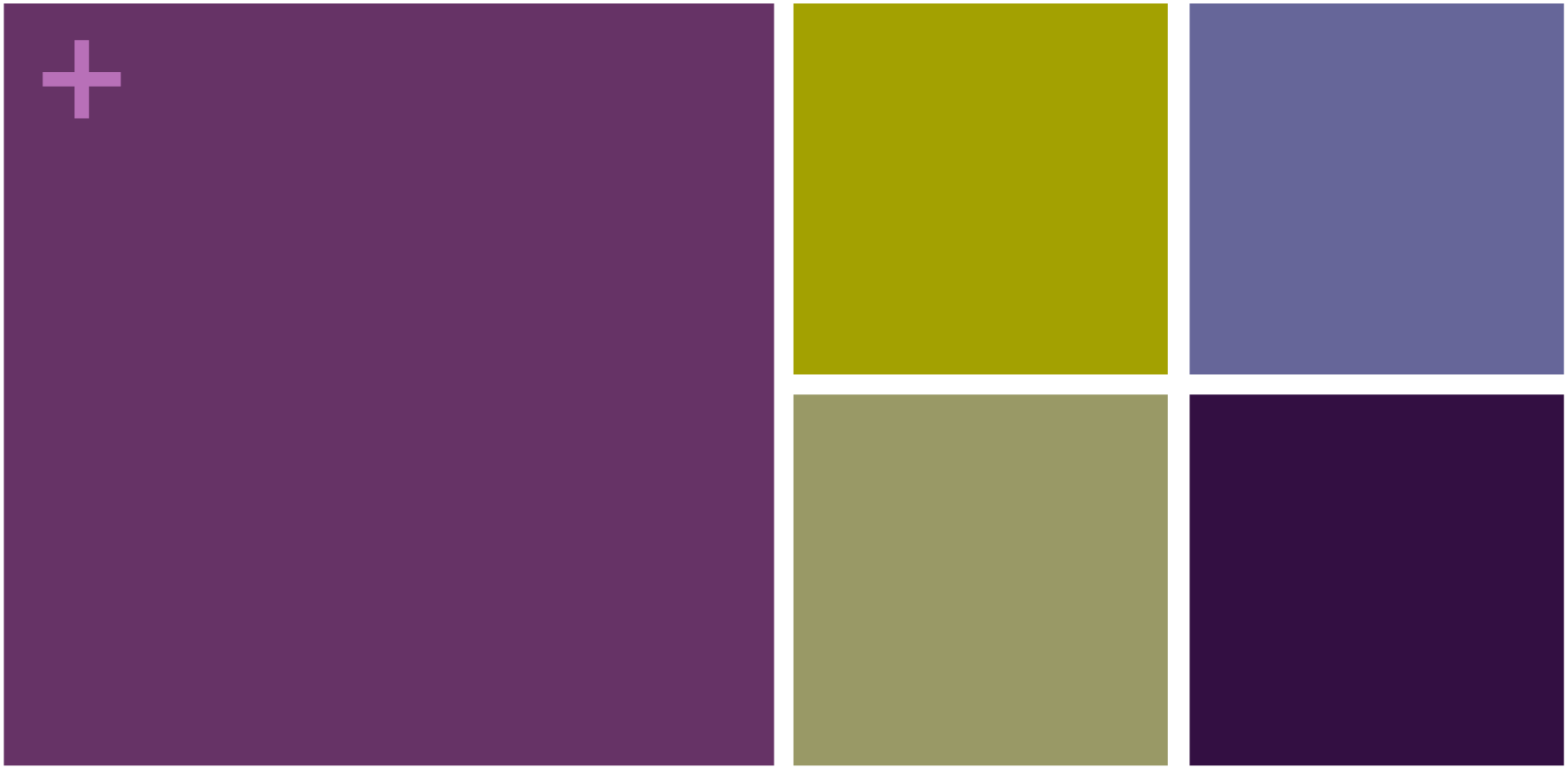


Sistemas Distribuidos
Replicación

Rodrigo Santamaría

Replicación

Introducción

- Mejoras
- Requisitos

Modelo de sistema

Servicios tolerantes a fallos

Servicios de alta disponibilidad

Transacciones con datos replicados

Introducción

- **Replicación**: mantenimiento de copias de la información en múltiples computadores
- Es un recurso utilizado ampliamente en sistemas distribuidos
  - Servidores web: servidores caché y proxies
  - DNS: copias de los mapeos URL-IP, actualizadas diariamente
  - Google: Google Data Centers
- Ayuda a mejorar un SD en distintos aspectos:
  - Rendimiento
  - Disponibilidad
  - Tolerancia a fallos

Introducción

Mejoras del rendimiento

- Principalmente, a través de cachés en clientes o servidores
- Mantener copias de los resultados obtenidos en llamadas anteriores al servicio reduce el coste de llamadas idénticas
  - Evita el tiempo de latencia del cálculo del resultado o de las consultas a otros servidores
- La replicación de datos inmutables es trivial
- La replicación de datos cambiantes (frecuentes en la red) conlleva un coste en protocolos de intercambio y actualización, que pueden limitar la efectividad de la réplica

Introducción

Alta disponibilidad

- La proporción de tiempo que un servicio está accesible con tiempos de respuesta razonables debe ser ~ 100%
- Factores de pérdida de disponibilidad
  - Fallos en el servidor
    - Si un servidor tiene una posibilidad de fallo *p* del 5%, tendrá disponibilidad del 95%
    - Si replicamos *n* veces el servidor, la disponibilidad será $1-p^n$
    - Con $n=2$ servidores: $1 - 0.05^2 = 99.75\%$
  - Particiones de red o desconexiones
    - Desconexión intencionada (p.ej. BitTorrent) o no intencionada (p. ej. conexión inalámbrica viajando en tren)

Introducción

Tolerancia a fallos

- Una alta disponibilidad no implica necesariamente corrección
  - Puede haber datos no actualizados, o inconsistentes (concurrencia)
- Podemos utilizar replicación para ganar tolerancia a fallos
  - Parada o caída
    - Si tenemos *n* servidores, pueden fallar *n*-1 sin alterar el servicio
  - Fallo bizantino
    - Si *f* servidores tienen fallos bizantinos, un sistema con 3*f*+1 servidores proveería un servicio correcto

Introducción

Requisitos

- La replicación debe llevarse a cabo considerando
  - **Transparencia**: los clientes no son conscientes de que hay múltiples copias del recurso al que acceden
    - Para ellos, sólo existen recursos lógicos individuales
  - **Consistencia**: las operaciones sobre un conjunto de objetos replicados deben dar resultados que sigan la especificación de corrección definida para dichos objetos
    - Puede ser más o menos estricta según la aplicación

Replicación

Introducción

Modelo de sistema

- Componentes
- Operaciones

Servicios tolerantes a fallos

Servicios de alta disponibilidad

Transacciones con datos replicados

Modelo de sistema

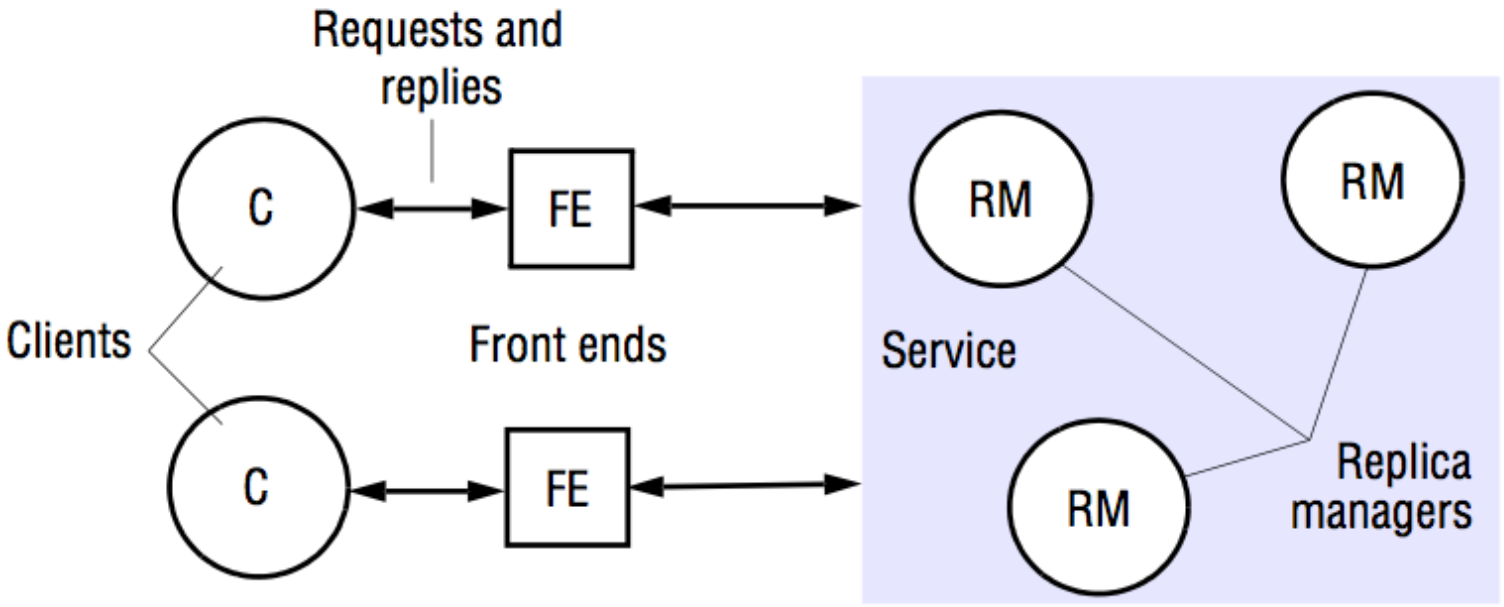
Definiciones

- **Objeto**: cualquier entidad de información a replicar
  - Archivos, objetos Java, etc.
  - **Objeto lógico**: entidad de información visible por el usuario, implementada por distintas copias físicas o **réplicas**
- Las réplicas de un objeto lógico no tienen por qué ser todas iguales en todo momento
  - P. ej. algunas pueden haberse actualizado y otras no
- Asunciones:
  - Sistema asíncrono
  - Los procesos fallan solo por caída
  - No hay particiones de red

Modelo de sistema

Componentes

- **Gestor de réplicas**: componentes que almacenan réplicas de un determinado objeto o servicio y operan sobre ellas
- **Frontal (front end)**: componente que atiende las llamadas de los clientes y se comunica con los gestores de réplicas



Coulouris et al. 2011

Modelo de sistema

Fases de una petición [Wiesmann et al. 2000]

1. **Petición**: el frontal envía la petición a un gestor
   - O bien envía la petición a un gestor y éste reenvía a otros
   - O multidifunde la petición a varios gestores
2. **Coordinación**: los gestores se coordinan para ejecutar la petición de manera consistente
   - Ordenación FIFO, causal o total
3. **Ejecución**: se ejecuta la petición (puede ser de forma tentativa)
4. **Acuerdo**: se llega a un consenso antes de consumar la ejecución
5. **Respuesta**: uno o más gestores de réplicas responden al frontal

Modelo de sistema

Tipos de ordenación

- **FIFO:** si un frontal solicita la petición *r* antes que la petición *r'*, todos los gestores que resuelvan *r'* deben hacerlo después de resolver *r* (ordenación 'local')
  - La mayoría de las aplicaciones sólo requieren una ordenación FIFO
- **Causal**: si *r* -> *r'*, entonces todos los gestores resuelven *r* antes de resolver *r'*
- **Total**: si un gestor resuelve *r* antes que r', todos los gestores resuelven *r* antes que *r'*

Replicación

Introducción

Modelo de sistema

Servicios tolerantes a fallos

- Criterios de corrección
- Replicación pasiva
- Replicación activa

Servicios de alta disponibilidad

Transacciones con datos replicados

Servicios tolerantes a fallos

- Deben proporcionar un servicio correcto aunque fallen *f* procesos, mediante la replicación de datos y la funcionalidad asociada a los gestores de réplicas
- Intuitivamente, un servicio es correcto si
  - Responde a pesar de haber fallos *o*
  - El cliente no distingue la ejecución normal de una con fallos
- Dos modelos principales
  - Replicación pasiva o primario-respaldo
  - Replicación activa
- Criterios de corrección
  - Linealizabilidad
  - Consistencia secuencial

Servicios tolerantes a fallos

Criterios de corrección: linealizabilidad

- Sea una secuencia o_{i0}, o_{i1}, o_{i2}... de operaciones de lectura y actualización del cliente *i*
  - Las operaciones son síncronas para cada *i*: no se ejecuta una hasta que no se recibe respuesta de la anterior (ordenación FIFO)
  - Si dos clientes hacen operaciones, se intercalarán en un determinado orden, p. ej.: o_{20}, o_{21}, o_{10},o_{22}, o_{11}, o_{12}...

- Un servicio es **linealizable** si para alguna de las posibles secuencias de operaciones intercaladas:
  - El orden de las operaciones en el intercalado es **consistente con los tiempos reales** en los que ocurrieron
  - Dicho orden en serie 'real' es consistente con las copias u operaciones realizadas

Servicios tolerantes a fallos

Criterios de corrección: consistencia secuencial

- Un servicio es **consistente secuencialmente** si para alguna de las posibles secuencias de operaciones intercaladas:
  - El orden de las operaciones en el intercalado es **consistente con el orden de programa** que ejecuta cada cliente individual
  - Dicho orden en serie ‘real’ es consistente con las copias u operaciones realizadas
- Cualquier servicio linealizable es consistente secuencialmente, pues el orden real de ejecución refleja el orden del programa
  - El inverso no es cierto
- Linealizabilidad = atomicidad

Servicios tolerantes a fallos

Linealizabilididad y consistencia: ejemplo

a1 b1 b2 a2 b3 a3 -> orden lineal

a1 b1 b2 b3 a2 a3 -> orden consistente secuencial

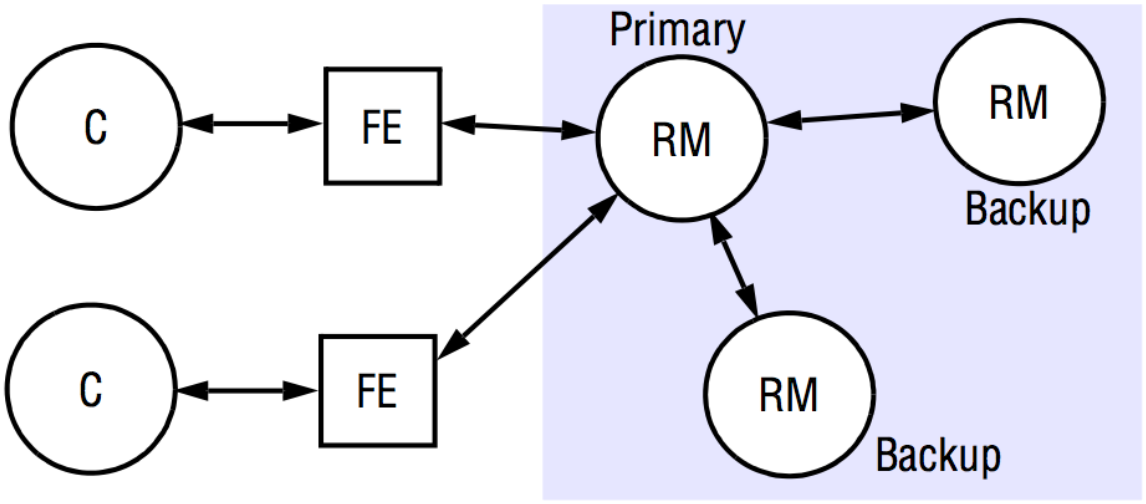
a1 a2 a3 b1 b2 b3 -> orden consistente secuencial

a1 b2 b1 b3 a2 a3 -> orden inconsistente

Servicios tolerantes a fallos

Replicación pasiva (primario-respaldo)

- Un gestor de réplicas primario y uno o más gestores secundarios (‘respaldos’ o ‘esclavos’)
- Los frontales sólo se comunican con el gestor primario
  - Ejecuta las operaciones y manda copias a los respaldos
- Si el primario falla, uno de los respaldos promociona a primario



Coulouris et al. 2011

Servicios tolerantes a fallos

Replicación pasiva: fases

1. **Petición**: el frontal envía la petición al gestor primario
2. **Coordinación**: el gestor primario ejecuta las peticiones siguiendo una ordenación FIFO
3. **Ejecución**: se ejecuta la petición y se almacena la respuesta
4. **Acuerdo**: si es una petición de actualización, el gestor primario envía la actualización a todos los respaldos, que confirman la recepción
5. **Respuesta**: el gestor primario responde al frontal

Servicios tolerantes a fallos

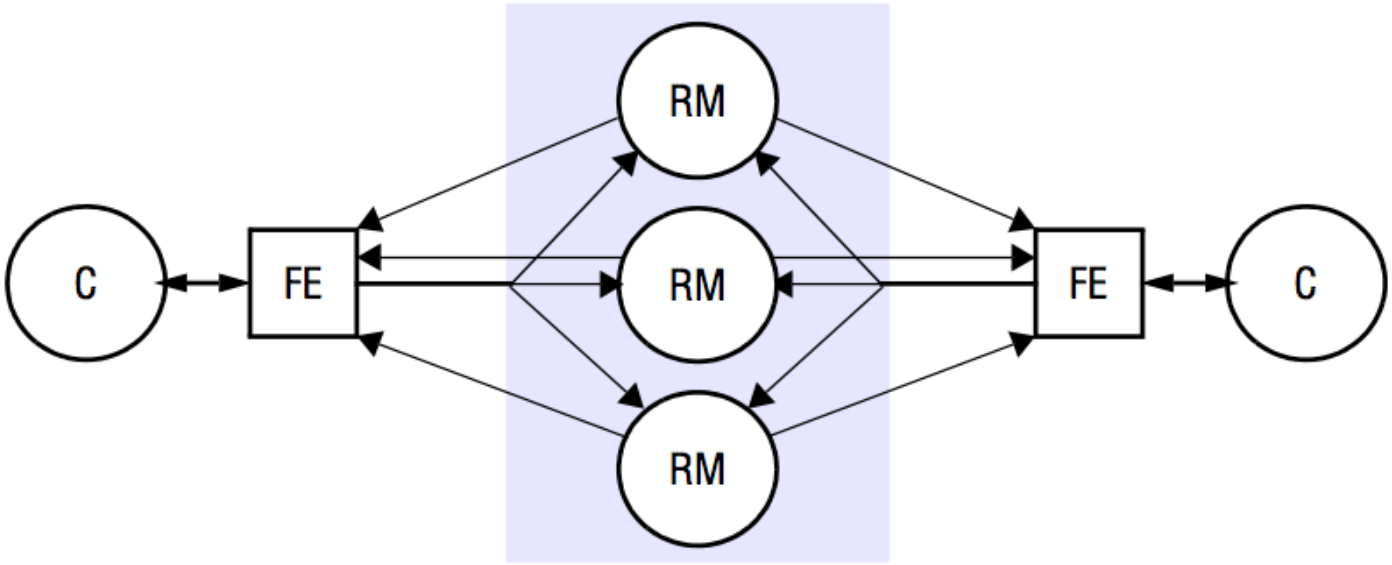
Replicación pasiva: análisis

- Tolera fallos de proceso (gestor primario o respaldos)
- No tolera fallos bizantinos
- El frontal requiere poca funcionalidad
- Al controlar el orden de modificación mediante el gestor primario, mantiene la *consistencia secuencial*
- Problemas de cuello de botella en el gestor primario
  - Actualización de todos los respaldos antes de dar respuesta
  - Los respaldos no dan servicio directo

Servicios tolerantes a fallos

Replicación activa

- Todos los gestores de réplicas tienen el mismo papel
- Los frontales multidifunden las peticiones a todos los gestores
- Todos los gestores de réplicas procesan la petición de manera independiente, pero idéntica



Coulouris et al. 2011

Servicios tolerantes a fallos

Replicación activa: fases

1. **Petición**: el frontal multidifunde la petición a los gestores. Se utiliza multidifusión fiable y de ordenación total*. No envía otra petición hasta que recibe la respuesta a la actual
2. **Coordinación**: el sistema de comunicación entrega la petición a todos los gestores según una ordenación total (multidifusión)
3. **Ejecución**: cada gestor ejecuta la petición
4. **Acuerdo**: no es necesaria, debido al tipo de multidifusión
5. **Respuesta**: cada gestor manda su respuesta al frontal. El nº de respuestas que recoge el frontal depende de las asunciones de fallo y del algoritmo de multidifusión

*Por ejemplo, recordad el algoritmo ISIS visto en coordinación y acuerdo

Servicios tolerantes a fallos

Replicación activa: análisis

- La multidifusión fiable y totalmente ordenada es la que aporta la tolerancia a fallos
- Dicho tipo de multidifusión es equivalente a un algoritmo de consenso, con lo que resuelve fallos bizantinos
  - El frontal recoge $f+1$ respuestas iguales antes de responder al cliente

Servicios tolerantes a fallos

Replicación activa vs pasiva

	Pasiva	Activa
Petición	Se envía sólo al gestor primario	Multidifusión con ordenación total a todos
Coordinación	FIFO	Total
Ejecución	En el primario	En todos los gestores
Acuerdo	Si es una actualización, se envía a todas las réplicas	No es necesario (debido a la ordenación total)
Respuesta	Tras la actualización si es necesaria	El cliente recoge un nº de respuestas

Replicación

- Introducción
- Modelo de sistema
- Servicios tolerantes a fallos
- Servicios de alta disponibilidad
  - Arquitectura cotilla
  - Sistema Bayou
- Transacciones con datos replicados

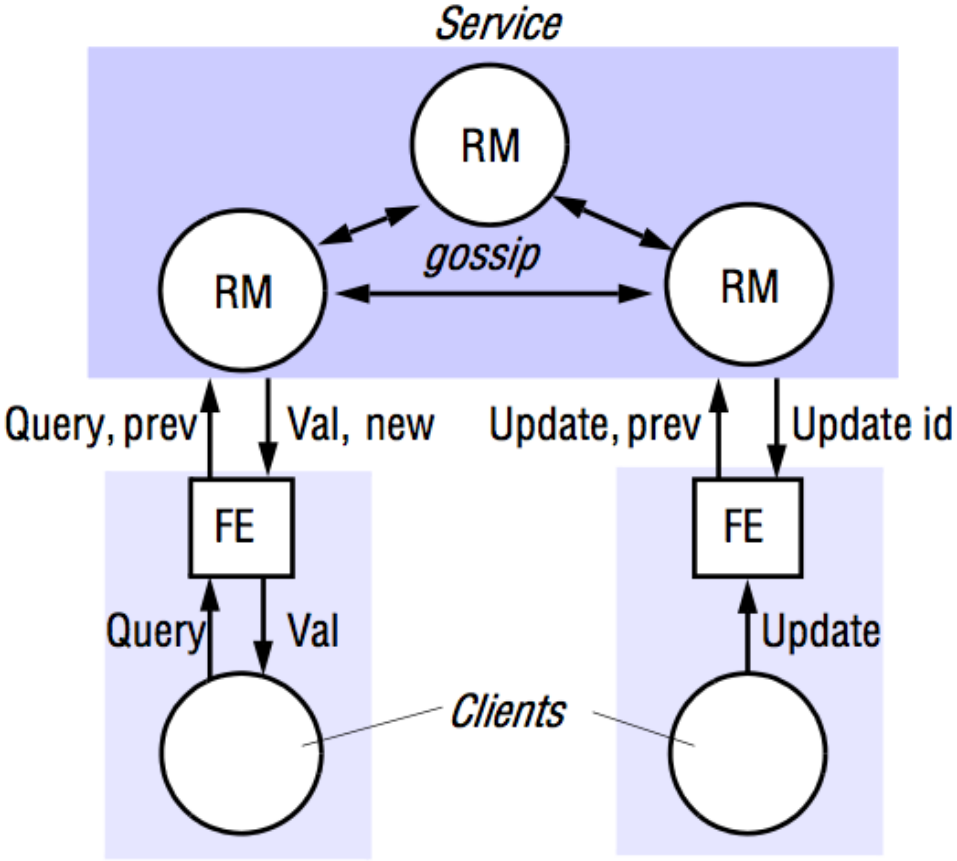
Servicios de alta disponibilidad

- Énfasis: dar a los clientes acceso al servicio – con tiempos de respuesta razonables – durante el mayor tiempo posible
- Tolerancia a fallos VS alta disponibilidad
  - Tolerancia a fallos: prioridad de la actualización de las réplicas
  - Alta disponibilidad: prioridad en la velocidad de respuesta
    - Aunque no todas las réplicas estén actualizadas
    - Menor grado de acuerdo -> menos consistencia

Servicios de alta disponibilidad

Arquitectura ‘cotilla’

- Desarrollada por Ladin et al [1992]
- Dos tipos de operaciones
  - Pregunta (lectura)
  - Actualización (escritura)
- El frontal envía la petición a un gestor a su elección



Servicios de alta disponibilidad

Arquitectura ‘cotilla’: garantías

- Cada cliente obtiene un servicio consistente *a lo largo del tiempo*
  - Los gestores siempre muestran datos con, al menos, todas las actualizaciones que el cliente ha visto hasta ahora
    - Incluso aunque un gestor conozca menos actualizaciones
- Consistencia *relajada* entre réplicas
  - Todos los gestores eventualmente reciben todas las actualizaciones y las aplican con garantías de ordenación, de modo que las réplicas sean suficientemente parecidas como para cumplir con las necesidades de la aplicación
  - Dos clientes pueden ver réplicas distintas, aunque incluyan el mismo conjunto de actualizaciones
  - Un cliente puede observar datos antiguos

Servicios de alta disponibilidad

Arquitectura ‘cotilla’: fases

1. **Petición***:* el frontal manda una petición a un gestor
   - Si el gestor ‘habitual’ falla, elige otro
   - Petición pregunta: el cliente (y frontal) se quedan bloqueados
   - Petición actualización: el cliente continúa tan pronto como la petición pasa al frontal, que propaga la operación en ‘segundo plano’.
2. **Respuesta a actualización**: tan pronto como se recibe
3. **Coordinación**: el gestor no procesa una petición hasta que se cumplan todas las restricciones de orden requeridas
4. **Ejecución**: el gestor ejecuta la petición
5. **Respuesta a pregunta**: el gestor responde ahora
6. **Acuerdo**: los gestores se actualizan entre ellos con mensajes de ‘cotilleo’

la respuesta ocurre *antes* de que haya acuerdo

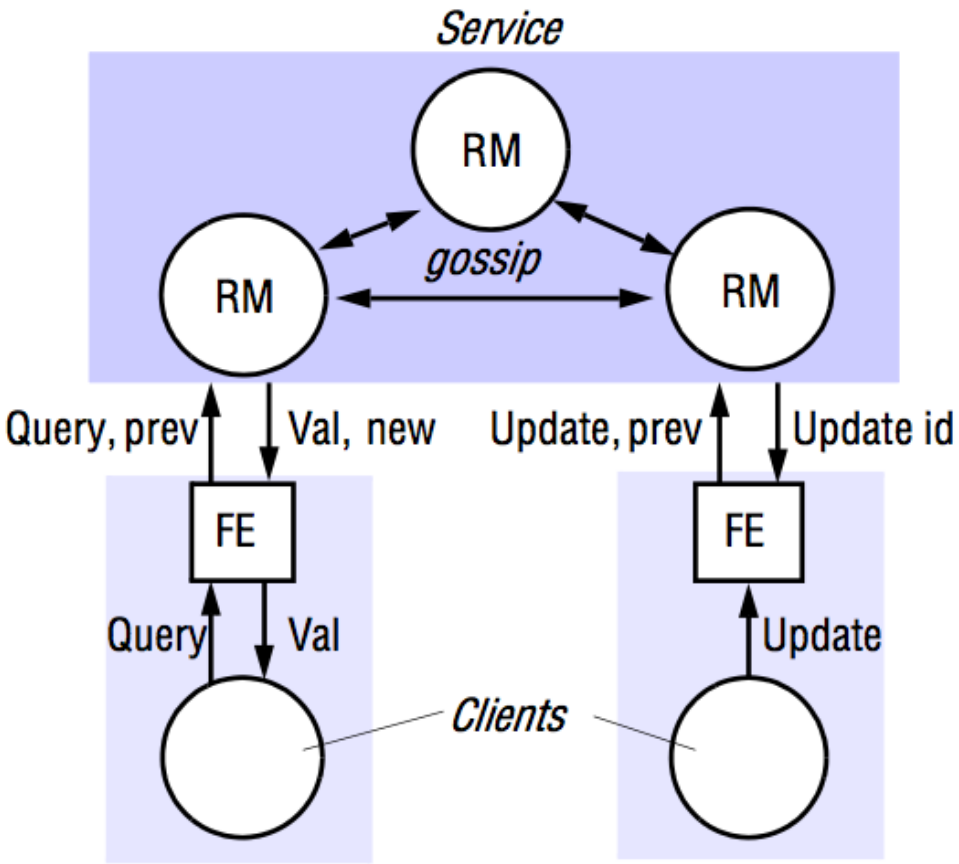
Servicios de alta disponibilidad

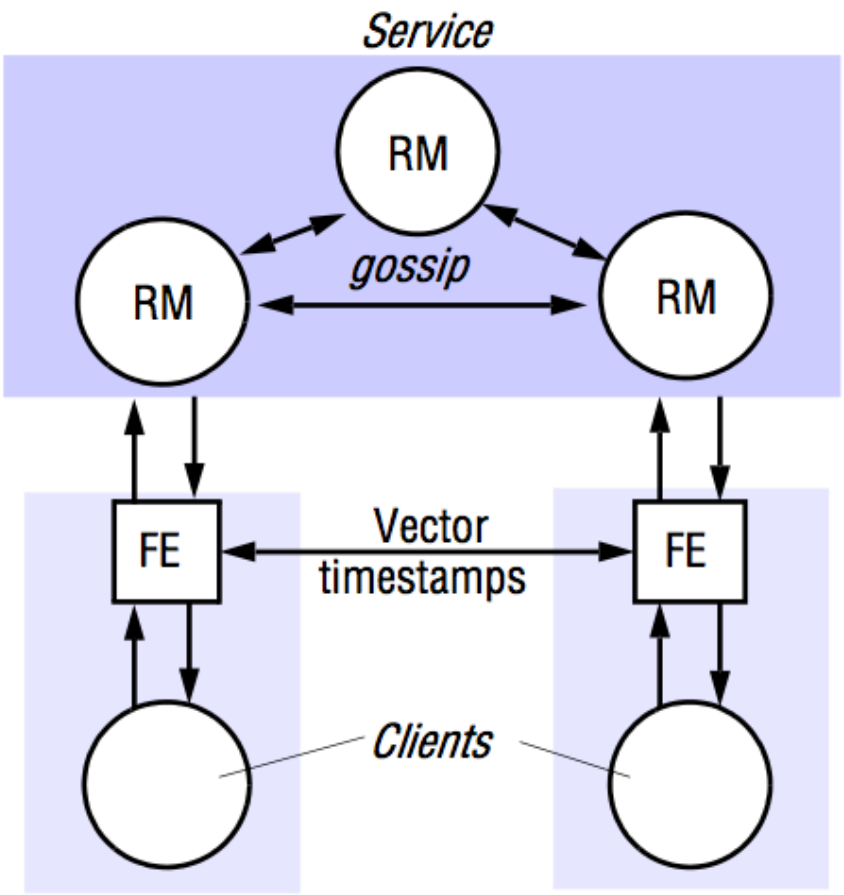
Arquitectura ‘cotilla’: marcas temporales

- Cada frontal tiene un vector de marcas temporales con la última versión de los datos a la que accedió (**prev**)
- El frontal adjunta *prev* a cada mensaje de petición a un gestor, junto con el tipo de petición (*update* o *query*)
- Si es una actualización (*update*), el gestor responde al frontal inmediatamente, aunque se encargará de asegurarse de tener una versión adecuada antes de efectuar la actualización
- En caso de consulta (*query*), debe asegurarse de que tiene una versión más actualizada que prev (no necesariamente la más actualizada en el sistema) antes de contestar
- En ambos casos, para actualizar sus versiones, el gestor intercambia mensajes de cotilleo (*gossip*) con otros gestores
- Los clientes también pueden cotillear entre sí -> los frontales se intercambian y fusionan sus marcas temporales

Servicios de alta disponibilidad

Arquitectura ‘cotilla’: Peticiones y cotilleo





Servicios de alta disponibilidad

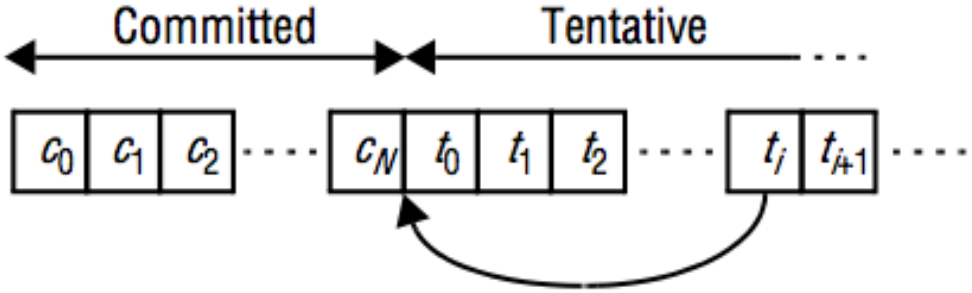
Arquitectura ‘cotilla’: resumen

- Para conseguir alta disponibilidad, respondemos al frontal tan pronto como podamos
  - Aplicando los cambios localmente en cuanto tengamos unos mínimos de consistencia.
  - Dichos cambios se propagan a otros gestores de manera ‘perezosa’
  - Puede ser insuficiente para sistemas que requieren un grado alto de consistencia (p.ej. transacciones bancarias) -> mejor usar un sistema tolerante a fallos o multidifusión con ordenación total
- Necesidad de varias estructuras de datos, y de un tráfico importante de mensajes para mantener a los gestores de réplicas actualizados
  - Problemas de escalabilidad

Servicios de alta disponibilidad

Sistema Bayou [Terry et al. 1995, Petersen et al. 1997]

- Como la arquitectura 'cotilla', da garantías más débiles que la consistencia secuencial
  - Las actualizaciones se aplican *descoordinadamente,* de manera **tentativa**, en cada gestor
  - Eventualmente, se reordenan (deshaciéndose/rehaciéndose) para lograr una aplicación coordinada y se **cometen**



- A veces no es posible una reordenación sin problemas de consistencia
  - Técnicas resolución de conflictos definibles por el usuario*

* Veremos cómo han evolucionado y se han estandarizado estas técnicas: transformaciones operativas y CRDTs

Servicios de alta disponibilidad

Cotilla vs Bayou

	Cotilla	Bayou
Disponibilidad	Las actualizaciones no se aplican hasta que se ha llegado al orden requerido	Las actualizaciones se aplican inmediatamente de manera tentativa
Coordinación	Coordinación mediante *marcas temporales* y actualización '*perezosa*' entre gestores	Coordinación mediante operaciones *undo/redo* y fusión
Transparencia	Transparente	No transparente*

*El desarrollador debe definir reglas específicas del domino. Una vez definidas, sería transparente para el usuario final

Servicios de alta disponibilidad

Conflict-free Replicated Data Type (CRDT)

- Los **CRDT** (Shapiro et al. 2011) son estructuras de datos replicadas usadas en arquitecturas con concurrencia optimista*
  - Aseguran que los posibles conflictos se resuelven de manera *automática*
- Pueden compartir sus actualizaciones (CRDT conmutativos o **CmRDT**) o todo su estado (CRDT convergentes, **CvRDT**)
  - Son teóricamente equivalentes
  - Los CmRDT requieren transferencia con canal de comunicación fiable y ordenación causal
  - Los CvRDT requieren algún tipo de protocolo de cotilleo

* Facebook o Redis (una BBDD distribuida utilizada por Amazon, Twitter, Microsoft o Alibaba) usan CRDTs
LoL usa también una implementación de CRDT para su chat in-game.

Servicios de alta disponibilidad

Grow-only counter (G-counter)

```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = ?i P[i]
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

- Este contador siempre creciente mantiene un vector con un entero para cada proceso en un sistema de *n* procesos
  - El valor del contador es el sumatorio del vector
- Cada proceso incrementa localmente su posición*
- Como CmRDT, cada incremento local se multidifunde al resto de máquinas
- Como CvRDT, la consistencia eventual se garantiza con una operación de fusión (`merge`) entre contadores que toma el valor máximo para cada posición
  - Estas operaciones ocurren al retransmitir en segundo plano los contadores mediante un protocolo de cotilleo

* ¿Te recuerda a algo?

Servicios de alta disponibilidad

G-set y 2P-set

- **G-set** (Grow only set) es un conjunto que solo permite adiciones
  - La fusión es la unión de los elementos de las réplicas
- **2P-set** (Two-phase set) son dos G-set, uno de elementos añadidos (A) y otro de eliminados (R)
  - La fusión es la unión de los elementos de cada conjunto de las réplicas
  - Los elementos eliminados priman sobre las adiciones: una vez un elemento está eliminado (añadido a R), no se considera como elemento del 2P-set, aunque se añada de nuevo posteriormente
- **LWW-set** (Last Write Wins) es una versión con marcas temporales sobre los elementos en A y R que permite reintroducir elementos en el conjunto

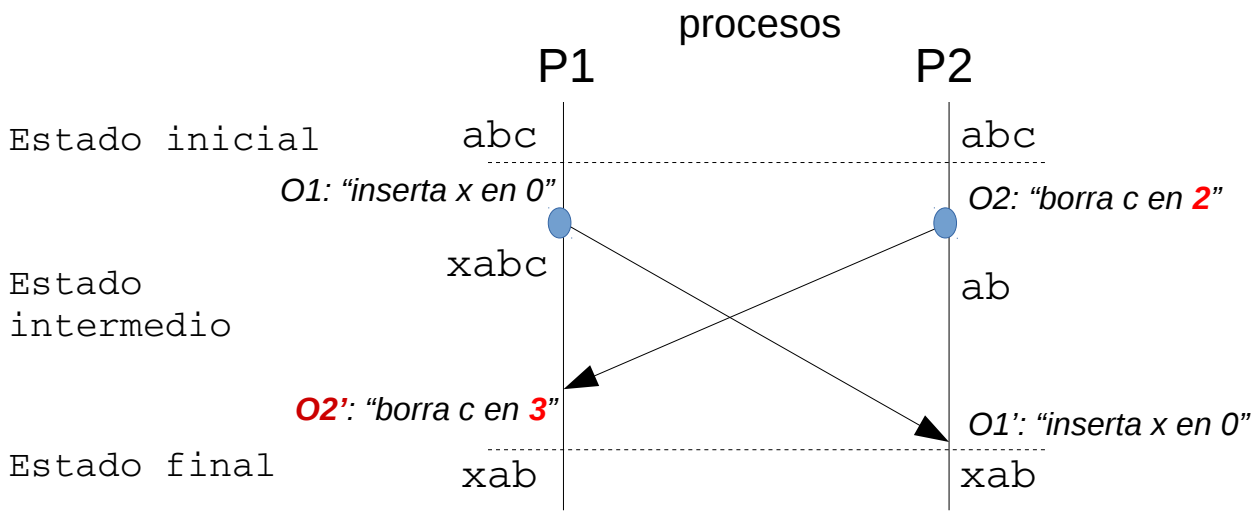
```
payload set A
    initial ∅
update add(element e)
    A := A ∪ {e}
query lookup(element e) : boolean b
    let b = (e ∈ A)
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
```

```
payload set A, set R
    initial ∅, ∅
query lookup(element e) : boolean b
    let b = (e ∈ A ∧ e ∉ R)
update add(element e)
    A := A ∪ {e}
update remove(element e)
    pre lookup(e)
    R := R ∪ {e}
merge (S, T) : payload U
    let U.A = S.A ∪ T.A
    let U.R = S.R ∪ T.R
```

Servicios de alta disponibilidad

Transformación operativa [Ellis y Gibbs, 1989]*

- TO es una estrategia para aplicar control de concurrencia optimista en entornos de edición colaborativa**
- *Transforma* las *operaciones* concurrentes para que se apliquen a estados intermedios (posiblemente inconsistentes) para obtener un estado final consistente



*Ver Kagorskii, 2018 para una explicación muy buena de esta técnica
**Adoptada por Apache Wave y Google Docs en 2009

Servicios de alta disponibilidad

Transformación operativa: requisitos

- **Convergencia**: todas las réplicas del documento tienen que ser iguales tras aplicar todas las operaciones
- **Consistencia**: las operaciones deben aplicarse en el mismo orden en todos los nodos
  - Basándose en la relación ‘sucede antes que’
  - No lo garantiza TO → uso de ordenación causal mediante relojes vectoriales, por ejemplo.
- **Intención**: el efecto de ejecutar una operación debe ser el mismo que el efecto buscado en la réplica local sobre la que se aplica inicialmente
  - Para ello, las operaciones deben transformarse en cada nodo según las operaciones que hayan ocurrido antes

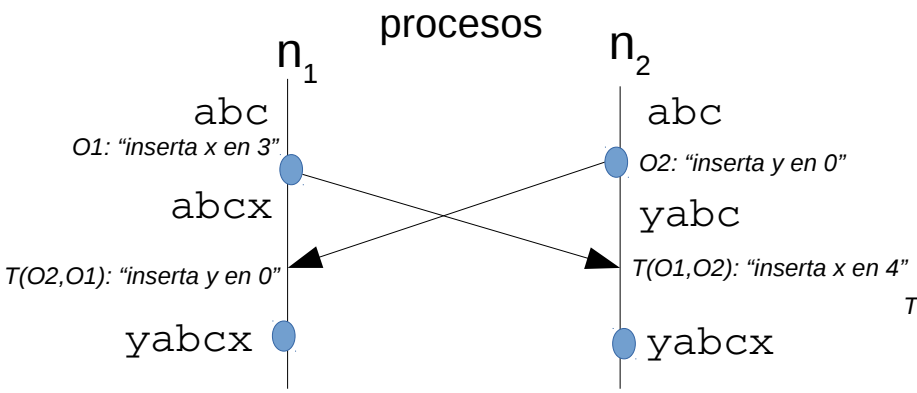
Servicios de alta disponibilidad

Transformación operativa: ejemplos

- Sea *T(op1,op2)* la transformación necesaria para aplicar *op1* depués de *op2*
- Sea *ins(p,c,n)* la operación de insertar un caracter *c* en la posición *p* en el nodo *n*
- Sea *del(p,n)* la operación de borrar la posición *p* en el nodo *n*

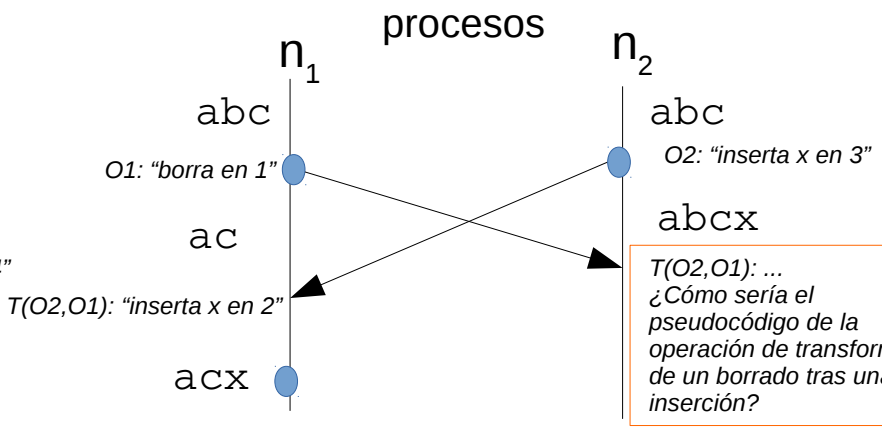
Transformación ante dos inserciones

```
T(ins(p1,c1,n1),ins(p2,c2,n2)):
  if(p1<p2 or (p1=p2 and n1<n2))
    return ins(p1,c1,n1)
  else
    return  ins(p1+1,c1,n1)
```



Transformación ante inserción tras borrado

```
T(ins(p1,c1,n1),del(p2,n2)):
  if(p1<p2 or (p1=p2 and n1<n2))
    return ins(p1,c1,n1)
  else
    return  ins(p1-1,c1,n1)
```



T(O2,O1): ...
¿Cómo sería el pseudocódigo de la operación de transformación de un borrado tras una inserción?

Servicios de alta disponibilidad

TO y CRDT

- Son dos métodos de solucionar de manera automática conflictos en entornos de réplicas OCC
- Son equivalentes, pero con distinto enfoque
  - TO se centra en las operaciones
  - CRDT se centra en los datos
- En el fondo, son reinterpretaciones de estrategias conocidas
  - Relojes vectoriales para garantizar la consistencia/ordenación
  - Marcas temporales para identificar las actualizaciones
  - Cotilleo para diseminar actualizaciones de manera perezosa

Servicios de alta disponibilidad

MediaWiki

- Sistema de edición vía HTTP de Wikipedia
  - No es propiamente un sistema para replicación
  - Pero debe lidiar con problemas de varias réplicas
    - Dos usuarios editando el mismo texto a la vez
    - Problema común a muchos sistemas.
- Solución similar al Sistema Bayou:
  - Control de la Concurrencia Optimista (OCC)*: se dejan hacer todas las modificaciones, si dos ocurren concurrentemente hay un conflicto de edición:
    - El último usuario en escribir debe resolverlo.**
    - La resolución de conflictos requiere intervención humana

*Ver el tema de sistemas de archivos

** Algo similar ocurre en Git: `git checkout` antes de `git commit`, o `git fetch` antes de `git push`

Replicación

Introducción
Modelo de sistema
Servicios tolerantes a fallos
Servicios de alta disponibilidad
Transacciones con datos replicados

- Replicación de copia primaria
- Uno lee – todos escriben
- Consenso por quórum
- Commit de dos fases

Transacciones con replicación

- **Transacción**: secuencia de una o más operaciones que tienen que ocurrir de manera atómica
- Los objetos en sistemas transaccionales pueden estar replicados, para mejorar la disponibilidad y el rendimiento
  - El cliente no debe tener conocimiento de la replicación
- Esquemas de replicación
  - Replicación de copia primaria
  - Uno lee, todos escriben
  - Consenso por quórum
  - ...

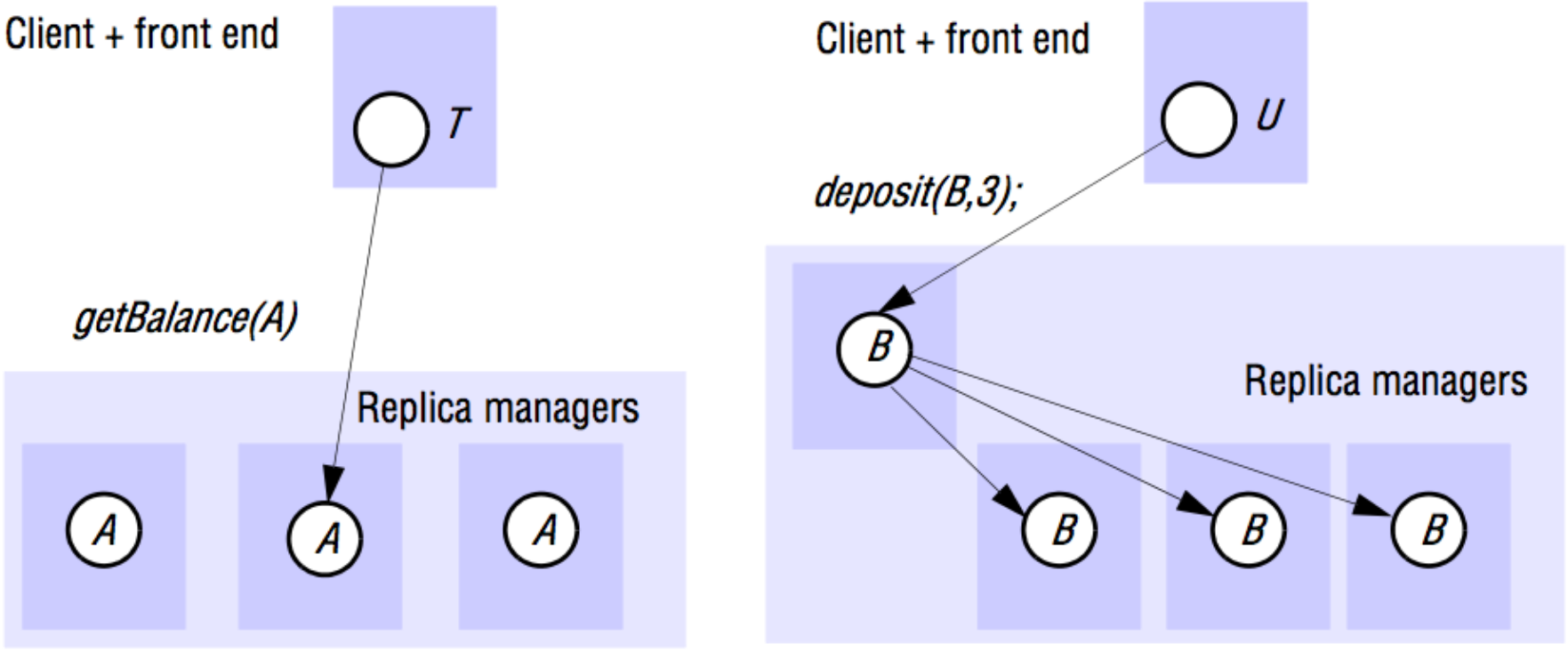
Transacciones con replicación

Replicación de copia primaria

- Similar a la replicación pasiva
  - Un gestor primario se encarga de la concurrencia (FIFO)
  - Para consumar una transacción, el primario comunica con las réplicas y después responde al cliente (estrategia exhaustiva)
    - Implica un mayor tiempo de respuesta
  - O bien, responde inmediatamente y realiza las copias en segundo plano (estrategia perezosa)
    - Si el primario cae, el reemplazo puede no tener la última versión

Transacciones con replicación

Uno lee, todos escriben



Coulouris et al. 2011

Transacciones con replicación

Consenso por quórum

- **Quórum**: subgrupo de gestores de réplicas autorizado a efectuar las operaciones
- Ejemplo [Gifford 1979]: sean *n* copias de un archivo F
  - Requisito de lectura: al menos *r* copias (quórum de lectura) de F deben ser consultadas
  - Requisito de escritura: al menos *w* copias (quórum de escritura) de F deben ser escritas
  - Restricción: $r+w>n$ (intersección no nula entre w y r)
    - Hay al menos una copia actualizada en cualquier quórum

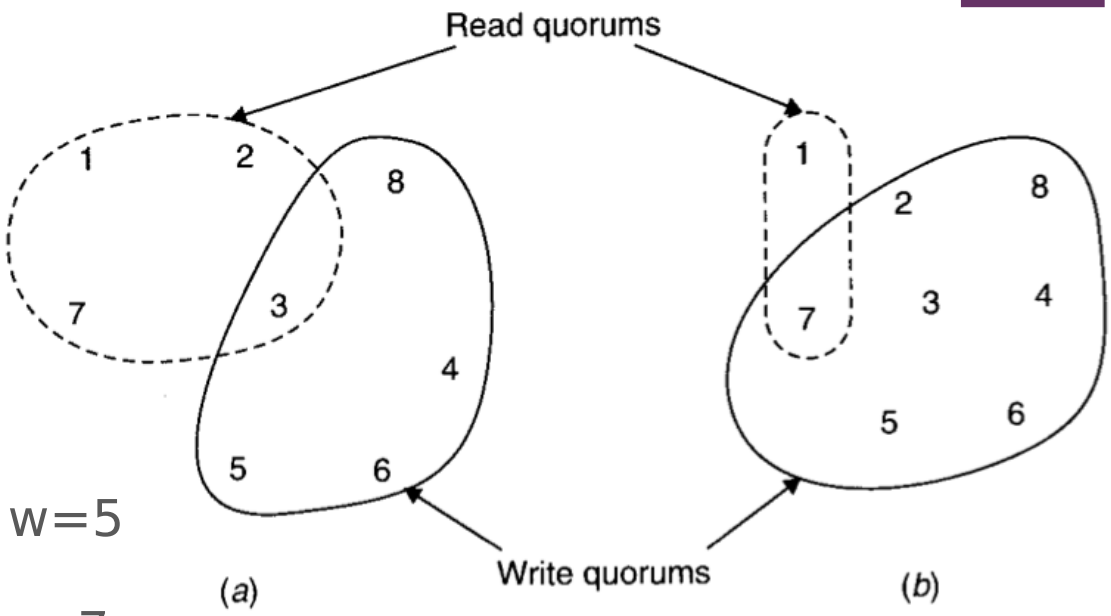
Transacciones con replicación

Consenso por quórum

- Operación de lectura:
  - Recuperar un quórum de lectura (cualquier conjunto de *r* copias)
  - De las *r* copias, seleccionar la copia con el nº de versión más alto
  - Retornar el valor de dicha copia
- Operación de escritura:
  - Tomar un quórum de escritura (cualquier conjunto de *w* copias)
  - De las *w* copias, obtener el nº de versión más alto
  - Incrementar el nº de versión
  - Escribir el nuevo valor y el nuevo nº de versión en todas las *w* copias del quórum de escritura
- Es una esquema diseñado para reducir el número de réplicas que deben ejecutar las actualizaciones
  - Al coste de incrementar el número de réplicas que deben ejecutar las lecturas

Transacciones con replicación

Consenso por quórum: ejemplos



- Ejemplo (a): n=8, r=4, w=5
- Ejemplo (b): n=8, r=2,w=7
- Esquema uno lee, todos escriben: r=1, w=n
- Consenso por mayoría: r ~ w > n/2

Transacciones con replicación

Resumen

- Número de réplicas que deben realizar la operación
  - Uno lee, todos escriben: r=1 vs w=n
  - Quórum: r~w (r+w>n)
- Estrategia de difusión de la operación
  - Estrategia **exhaustiva**: difunde a todas las réplicas en cuanto se produce la operación, antes de consumarla y retornar al cliente
    - Implica un retardo relevante en la respuesta al cliente
  - Estrategia **perezosa**: se reduce el número de réplicas que reciben antes de consumar, difundiendo al resto más tarde
    - Es necesario implementar controles de concurrencia

Commit de dos fases

Two Phase Commit (2PC)

- Generalización de la gestión de transacciones con réplicas a cualquier tipo de transacción
  - Especialización de un protocolo de consenso tipo Paxos
- Dos roles:
  - **Coordinador**: nodo designado según un determinado criterio
  - **Compañeros**: resto de nodos
- Filosofía exhaustiva / conservadora / bloqueante

Commit de dos fases

Fase 1 (voto)

- El coordinador multidifunde una petición de compromiso (*query*) a todos los compañeros y espera por sus respuestas
- Cada compañero ejecuta la transacción hasta el punto en el que le han pedido comprometerse (registrando las acciones en un *log*)
- Responden afirmativamente si sus acciones tuvieron éxito (*agreement*) o negativamente en caso contrario (*abort*)

Commit de dos fases

Fase 2 (resolución)

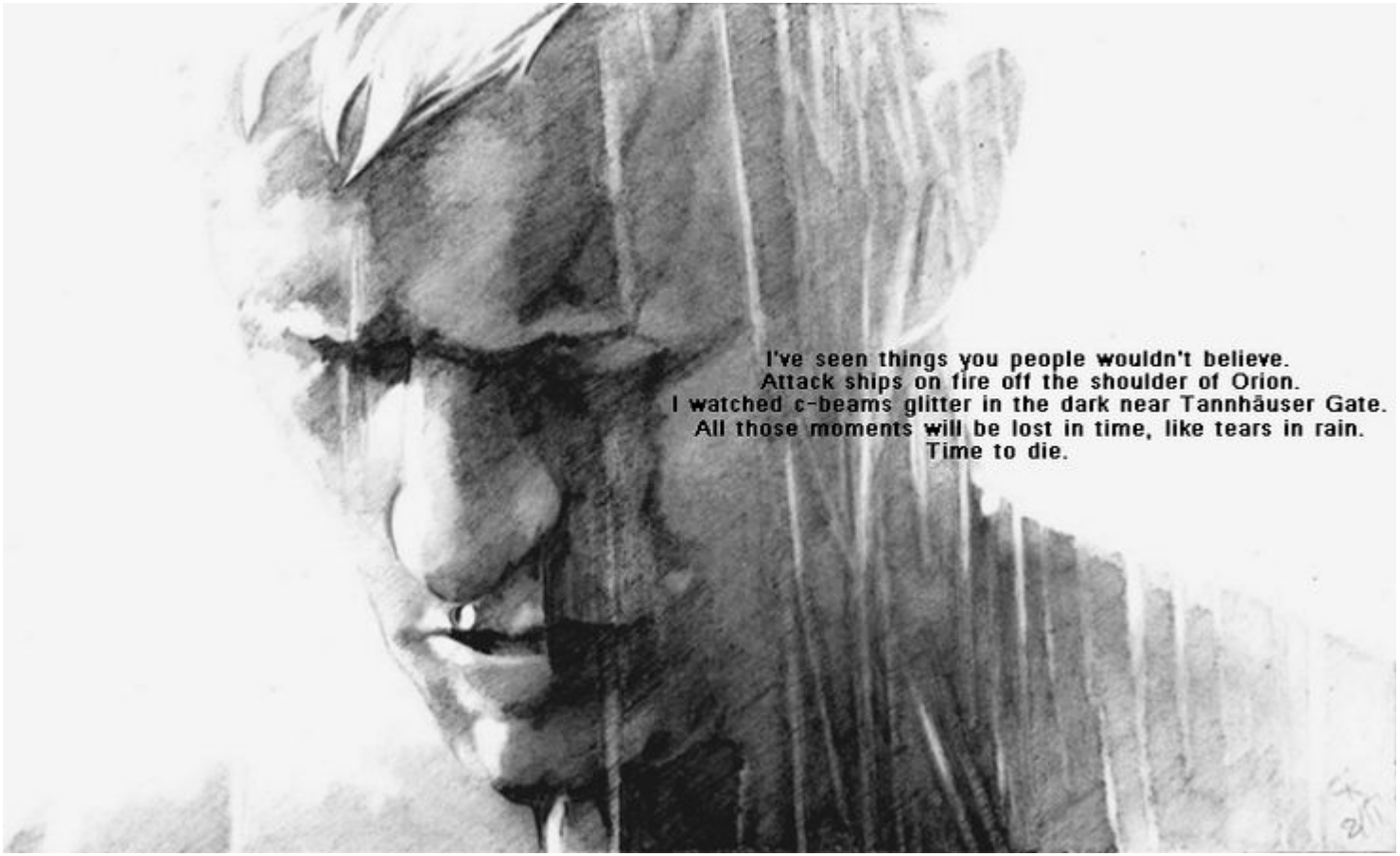
- **Éxito:** Si el coordinador recibe *agreement* de todos los compañeros
  - Multidifunde un mensaje de compromiso *commit*
  - Cada nodo completa la operación
- **Fallo:** Si el coordinador recibe al menos un *abort*
  - Multifidunde un mensaje de retirada *rollback*
  - Cada nodo deshace la transacción usando el *log*
- Cada nodo manda un mensaje de confirmación (*ack*) al coordinador
- El coordinador completa/deshace la transacción cuando recibe todos los *ack*

Replicación - resumen

- La replicación es un medio para incrementar el **rendimiento**, la **disponibilidad** y la **tolerancia a fallos**
- La replicación debe ser **transparente** a los clientes. Involucra sólo a los frontales y los gestores de réplicas
- La tolerancia a fallos debe garantizar un **mínimo de consistencia** (idealmente linealizabilidad pero frecuentemente sólo consistencia secuencial).
- La estrategia **pasiva** consigue la tolerancia mediante una configuración maestro-esclavos. La estrategia **activa** involucra a todos los gestores de réplicas a la vez. Ambas se pueden implementar mediante comunicación por grupos
- El **cotilleo** y **Bayou** son sistemas de **alta disponibilidad**. Ambos consiguen dan una respuesta rápida e independiente de la conexión relajando la consistencia causal. Bayou es más consistente al precio de no ser totalmente transparente
- En **sistemas de transacciones** tenemos problemas similares en caso de replicación. Las estrategias también se parecen: actualizar subgrupos vs actualizar todos; actualización inmediata vs actualización demorada

Referencias

- G. Coulouris, J. Dollimore, T. Kindberg and G. Blair. *Distributed Systems: Concepts and Design (5th Ed)*. Addison-Wesley, **2011**
  - Ch. 18
- Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., & Alonso, G. (**2000**). Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems* (pp. 464-474). IEEE.
- Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., & Demers, A. J. (**1997**). Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (pp. 288-301).
- Ladin, R., Liskov, B., Shrira, L., & Ghemawat, S. (**1992**). Providing high availability using lazy replication. *ACM Transactions on Computer Systems,* 10(4), 360-391.
- Kagorskii, A. (**2018**) Operational Transformations as an algorithm for automatic conflict resolution. *Medium.com*



https://www.pinterest.com/pin/491385009312966635