uc3m | Universidad Carlos III de Madrid

OpenCourseWare

Sistemas Paralelos y Distribuidos

Félix García Carballeira

Alejandro Calderón Matos

Tema 2. Algoritmos distribuidos

Principales características de un sistema distribuido

- Múltiples procesos
- Comunicación y sincronización entre procesos mediante paso de mensajes
- Espacio de direcciones de memoria disjuntos
- Ausencia de reloj global
- Procesos deben interactuar y cooperar para alcanzar un objetivo común

Ejemplos

- World Wide Web
- Servidores de ficheros en red / distribuidos
- Aplicaciones bancarias
- Redes peer to peer
- Sistemas de control de procesos
- Redes de sensores
- Grid computing
- …

¿Por qué es difícil la coordinación en sistemas distribuidos?

- El emisor no puede saber:
  - Si el mensaje fue recibido
  - Si el receptor falló antes o después de procesar el mensaje

Impossibility of distributed consensus with one faulty process
Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson
Journal of the ACM (JACM)
Volume 32 , Issue 2 (April 1985) , Pages: 374 – 382

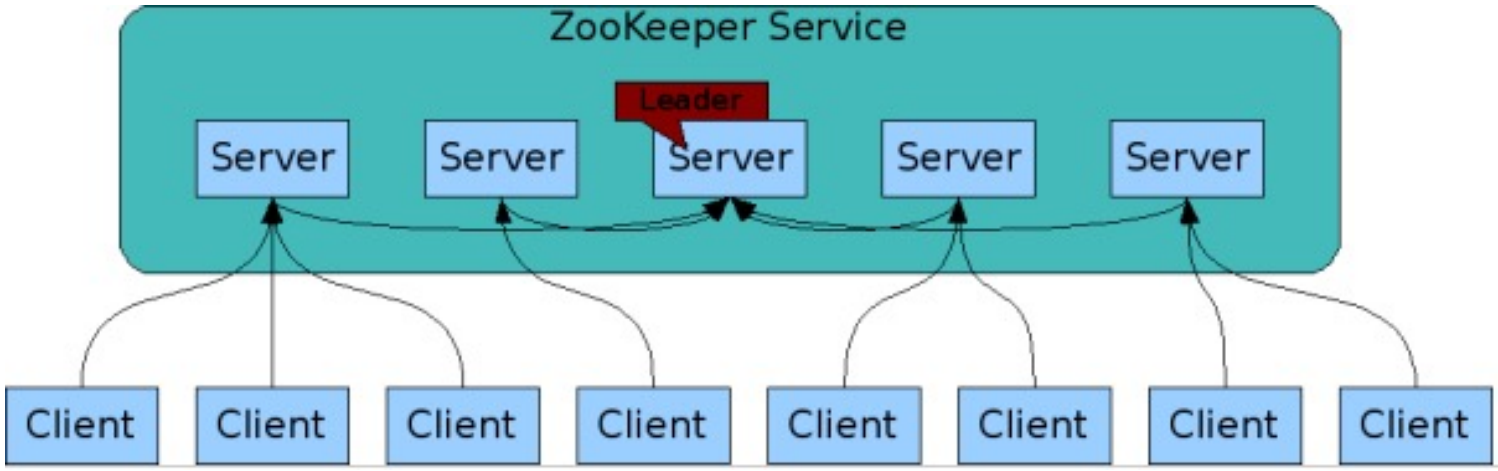
Motivación: ZooKeeper

- Servicio de coordinación de alto rendimiento para construir aplicaciones distribuidas
- Forma parte del proyecto Apache Hadoop que desarrolla SW abierto para construir aplicaciones distribuidas, escalables y fiables
- ZooKeeper se puede utilizar para resolver los siguientes problemas:
  - Implementar consenso distribuido
  - Elección de líder
  - Barreras de sincronización
  - Cerrojos distribuidos
  - Configuración dinámica
  - Servicios de gestión de grupos de procesos

Usos de ZooKeeper

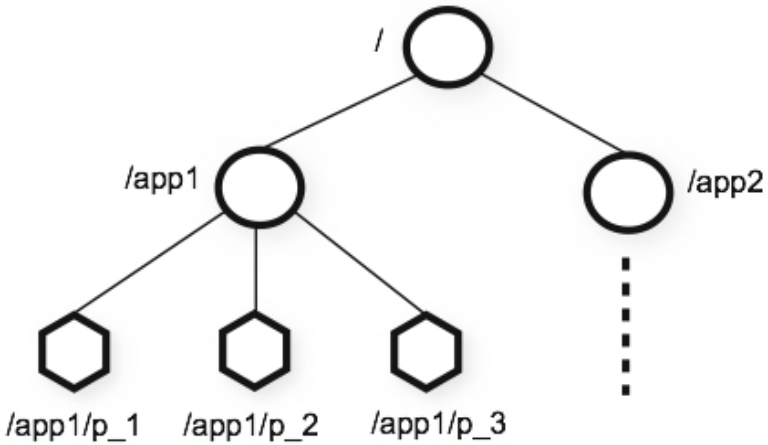
- Kafka (distributed publish/subscribe messaging System)
- Pinterest
- Twitter
- Yahoo!
- Facebook

Funcionamiento



- Datos organizados de forma jerárquica en memoria
- Los clientes establcecen una sesión cuando se conectan a ZooKeerper

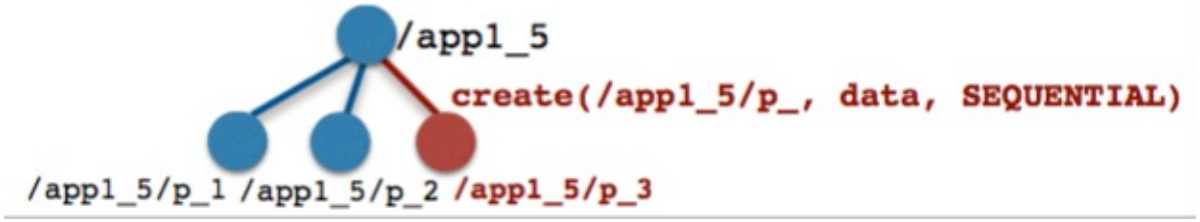
Modelo de datos



- Znode
  - Almacenado en memoria
  - Espacio de nombres jerárquicos
  - No están pensados para almacenamiento general (varios KB)
- Tipos:
  - Normales
  - Efímeros
  - Secuenciales
- Los clientes pueden manipular los znodes a través del API de ZooKeeper

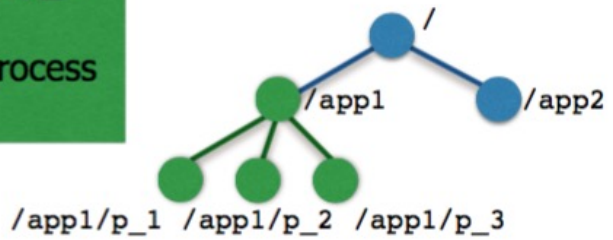
Tipos de znodes

- Efímeros: los znodes creados por un cliente son borrados al final de la sesión del cliente (o fallo del cliente)
- Secuenciales: incrementan un contador añadido al nombre del znode



Protocolo de pertenencia a un grupo de procesos

Group membership protocol:
Client process `pi` creates znode `p_i` under `/app1`.
`/app1` persists as long as the process is running.



znodes y flag *watch*

- Los clientes puede hacer operaciones de lectura sobre un znode con el flag watch activado
- El servidor notifica (callback) al cliente cuando la información del nodo ha cambiado
- Las notificaciones indican el cambio no los nuevos datos

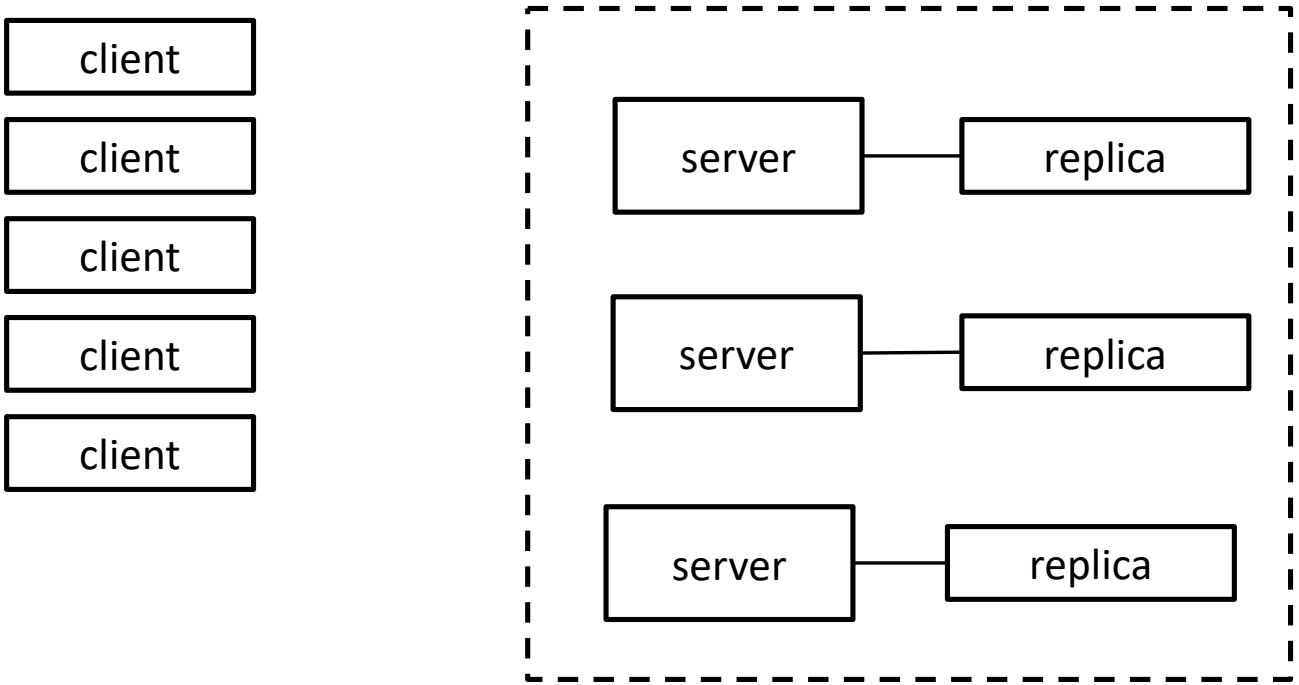
Sesiones

- Un cliente se conecta a ZooKeeper e inicia una sesión
- Las sesiones tienen asociado un timeout
- ZooKeeper considera que un cliente falla si no recibe nada una vez expirado el timeout
- La sesión finaliza cuando el cliente falla o cuando la cierra de forma explícita

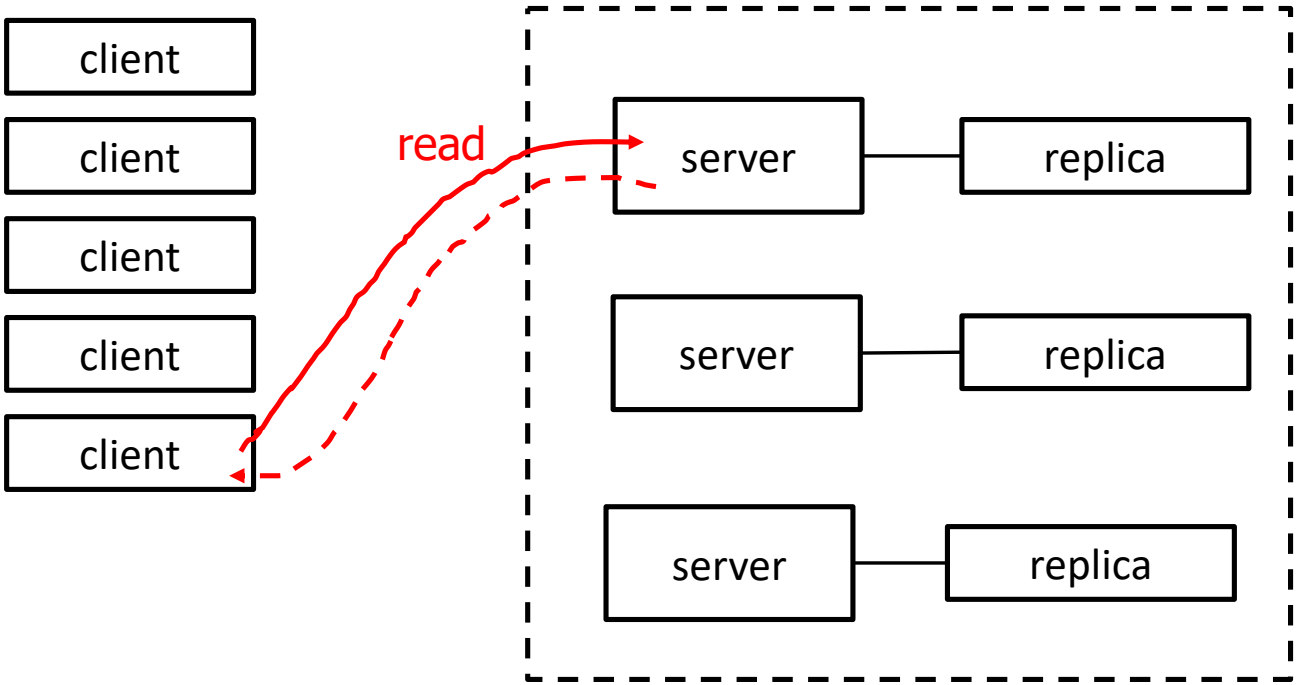
API del cliente

- Create(path, data, flags)
  - flags: ephemeral, sequential
- Delete(path, version)
- Exist(path, watch)
- getData(path, watch)
- setData(path, data, version)
- getChildren(path, watch)
- Sync(path)

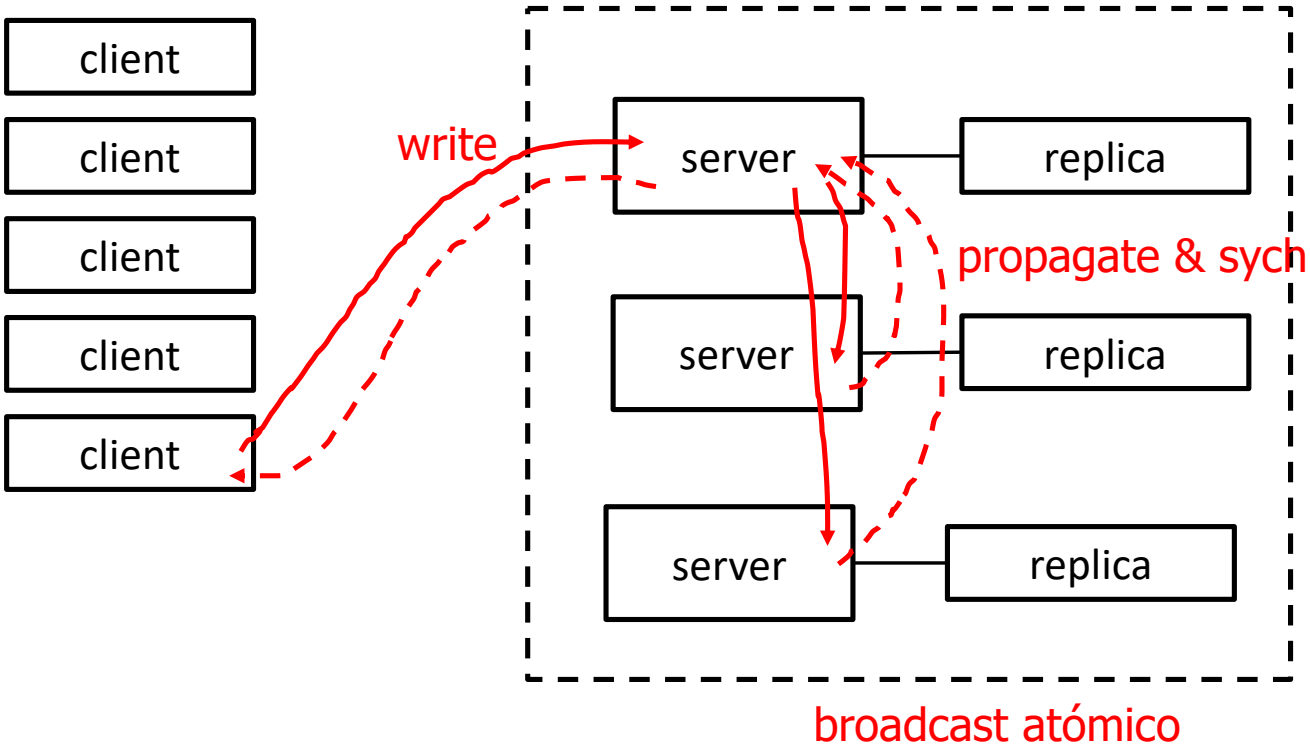
Servidores


client
client
client
client
client
server
replica
server
replica
server
replica

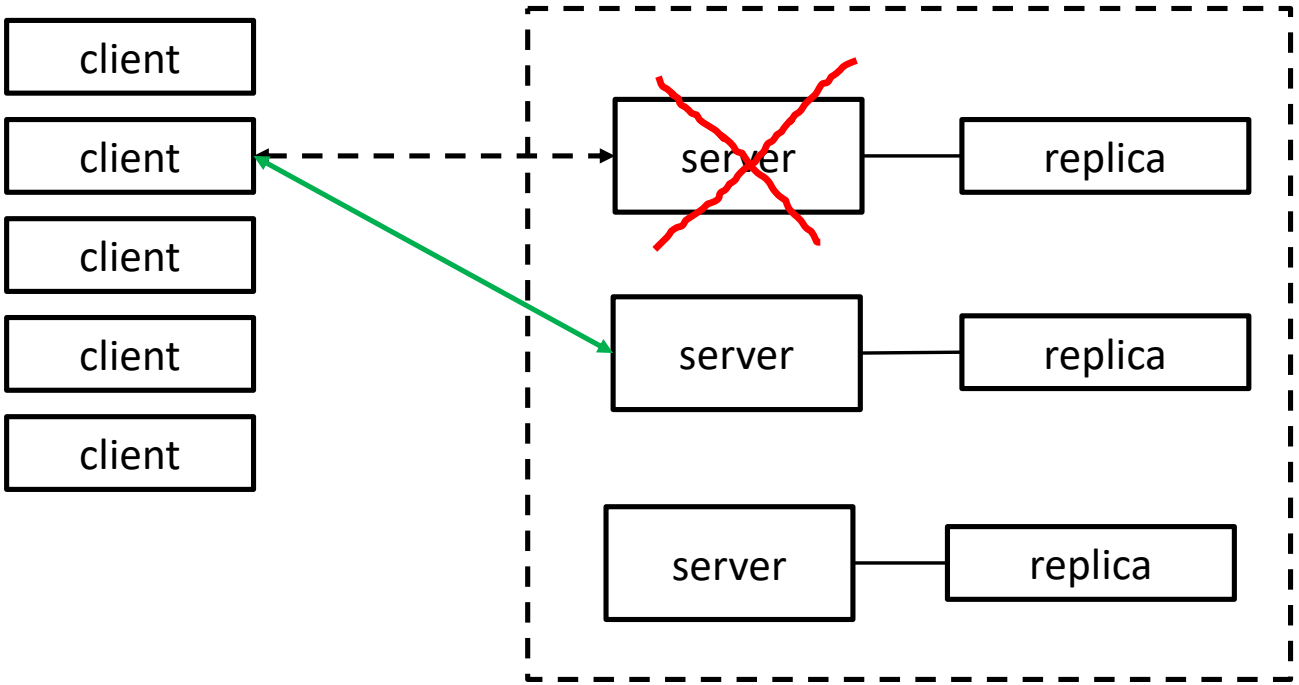
Servidores


client
client
client
client
client
read
server
replica
server
replica
server
replica

Servidores


client
client
client
client
client
write
server
server
server
replica
replica
replica
propagate & sych
broadcast atómico

Fallos


client
client
client
client
client
server
server
server
replica
replica
replica

¿Qué ofrece ZooKeeper?

- Consistencia secuencial: las actualizaciones de un cliente se aplican en el orden en el que fueron enviadas en todos los servidores
- Atomicidad: las actualizaciones se realizan de forma atómica
- Imagen única del sistema con independencia del servidor al que se conecta
- Fiabilidad: cuando una actualización se completa, perdura

¿En qué se basa ZooKeeper?

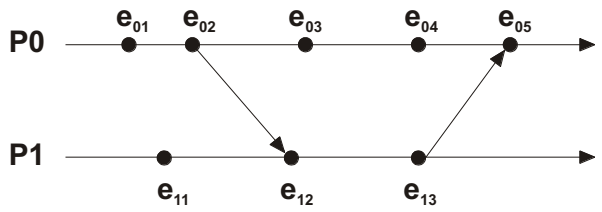
- En algoritmos distribuidos utilizados para:
  - Ordenar de eventos
  - Entrega causal de mensajes
  - Elección de un proceso coordinador (líder)
  - Consenso distribuido
  - …

Algoritmos distribuidos

- Algoritmos que trabajan en sistemas distribuidos
- Realizan tareas:
  - Comunicación
  - Gestión de datos y de recursos
  - Sincronización
  - Consenso
- Deben trabajar bajo:
  - Actividades concurrentes en múltiples localizaciones
  - Incertidumbre del tiempo, ordenación de eventos, .
  - Posibilidad de fallos (procesos, procesadores, redes)

Modelo de sistema distribuido

- Modelo de sistema:
  - Procesos secuenciales $\{P_1, P_2, ...P_n\}$ que ejecutan un algoritmo local
  - Canales de comunicación
- Eventos en P_i
  - $E_i = \{e_{i1}, e_{i2}, ...e_{in}\}$
- Tipos de eventos locales
  - Internos (cambios en el estado de un proceso)
  - Comunicación (envío, recepción)
- Diagramas espacio-tiempo



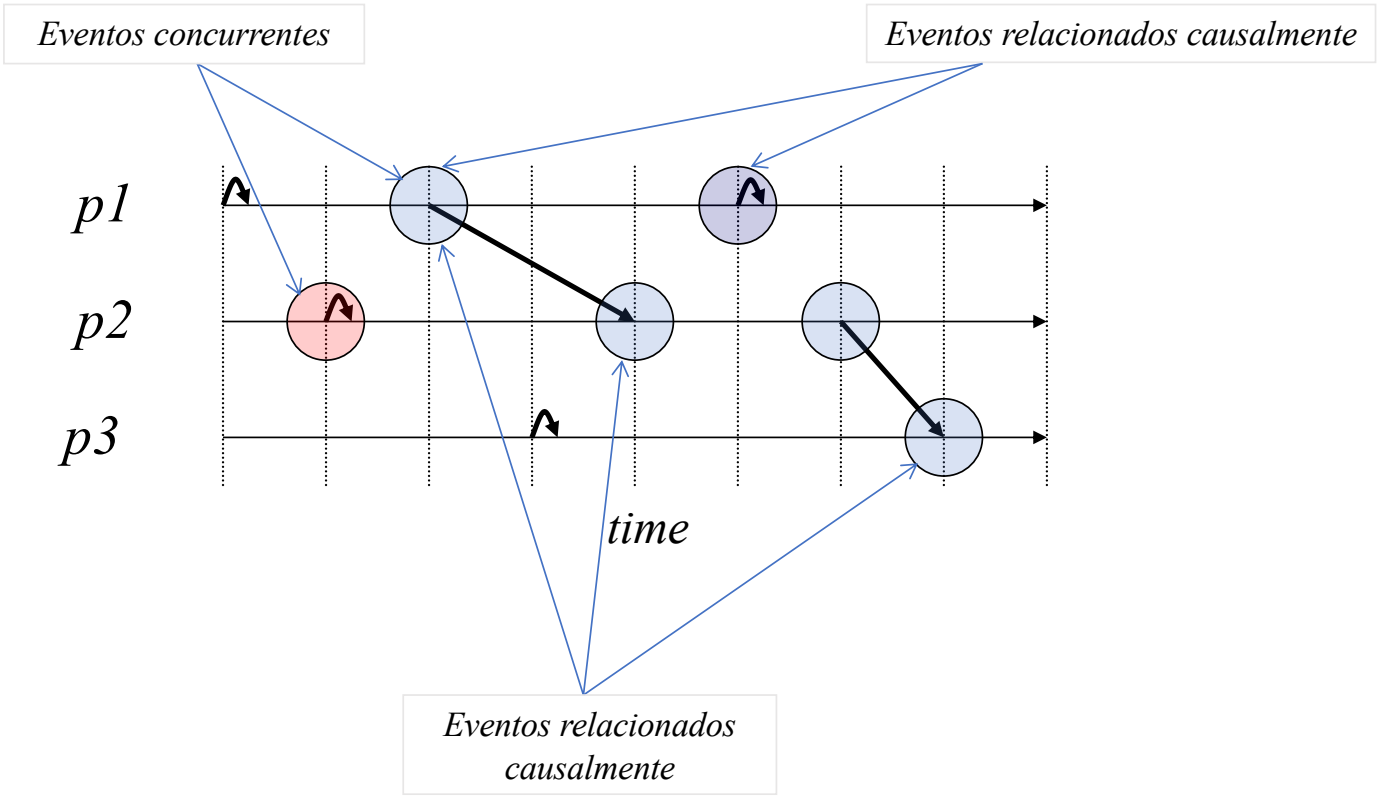
Algoritmos locales

- Algoritmo local:
  - Un proceso cambia de un estado a otro (*evento inerno*)
  - Un proceso cambia de un estado a otro y envía un mensaje a otro proceso (*evento de envío*)
  - Un proceso recibe un mensaje y cambia su estado (*evento de recepción*)
- Restricciones
  - Un proceso *p* solo puede recibir un mensaje después de haber sido enviado por otro
  - Un proceso *p* solo puede cambiar del estado *c* al estado *d* si está actualmente en el estado *c*

Orden causal

- La relación $\leq_H$ sobre eventos de una ejecución distribuida, denominada *orden causal*, se define como:
  - Si *e* ocurre antes que *f* en el mismo proceso, entonces $e \leq_H f$
  - Si *s* es un evento de envío y *r* su correspondiente evento de recepción, entonces $s \leq_H r$
  - $\leq_H$ Es transitiva
    - Si $a \leq_H b$ y $b \leq_H c$ entonces $a \leq_H c$
  - $\leq_H$ es reflexiva.
    - $a \leq_H a$ para cualquier evento
- Dos eventos , *a* y *b,* son *concurrentes* sii $a \not\leq_H b$ y $b \not\leq_H a$

Ejemplos de eventos



Propiedades de un algoritmo distribuido

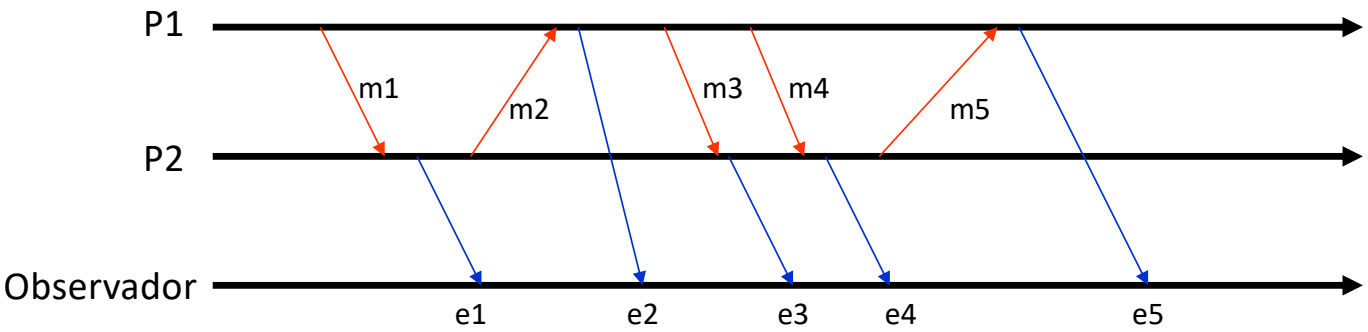
- Los algoritmos distribuidos deben tener las siguientes propiedades:
  - La información relevante se distribuye entre varias máquinas
  - Los procesos toman las decisiones sólo en base a la información local
  - Debe evitarse un punto único de fallo
  - No existe un reloj común

Ejemplos de algoritmos distribuidos

- Ordenación de eventos
- Sincronización de relojes
- Exclusión mutua distribuida
- Algoritmos de elección
- Comunicación *multicast* y ordenación de mensajes
- Problemas de consenso
- Detección de interbloqueos
- Asignación de recursos
- Planificación
- Tolerancia a fallos

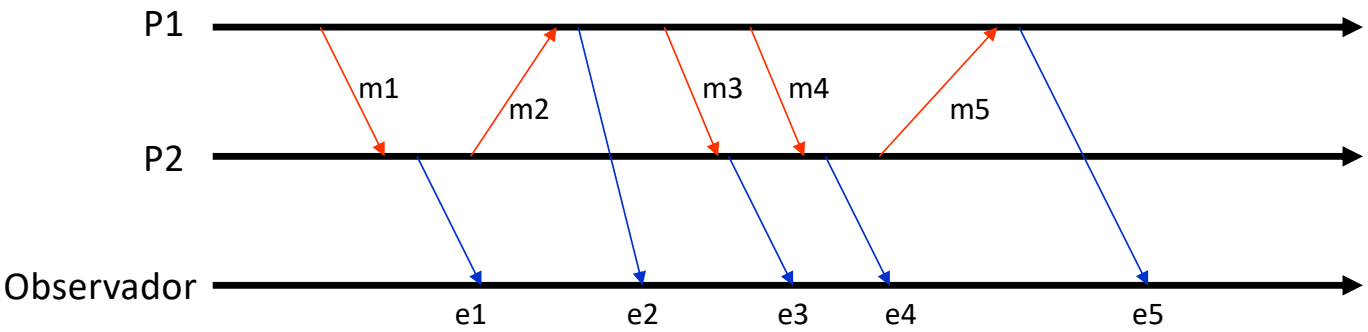
Ordenación de eventos

- Monitorización del comportamiento de una aplicación distribuida
  - Ejemplo: el observador debe ordenar los eventos de recepción de mensajes en los procesos P1 y P2
    - e1, e2, e3, e4, e5



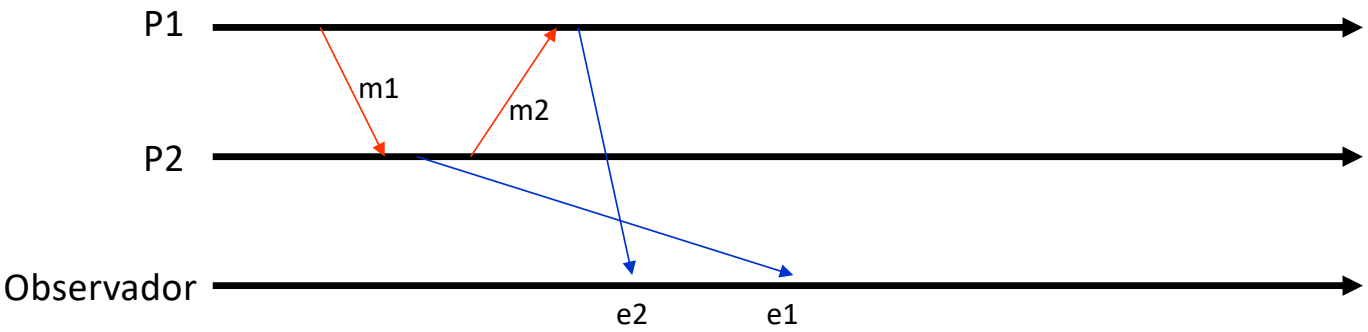
Ejemplo

- Monitorización del comportamiento de una aplicación distribuida
  - El observador debe ordenar los eventos de recepción de mensajes en los procesos P1 y P2
    - e1, e2, e3, e4, e5



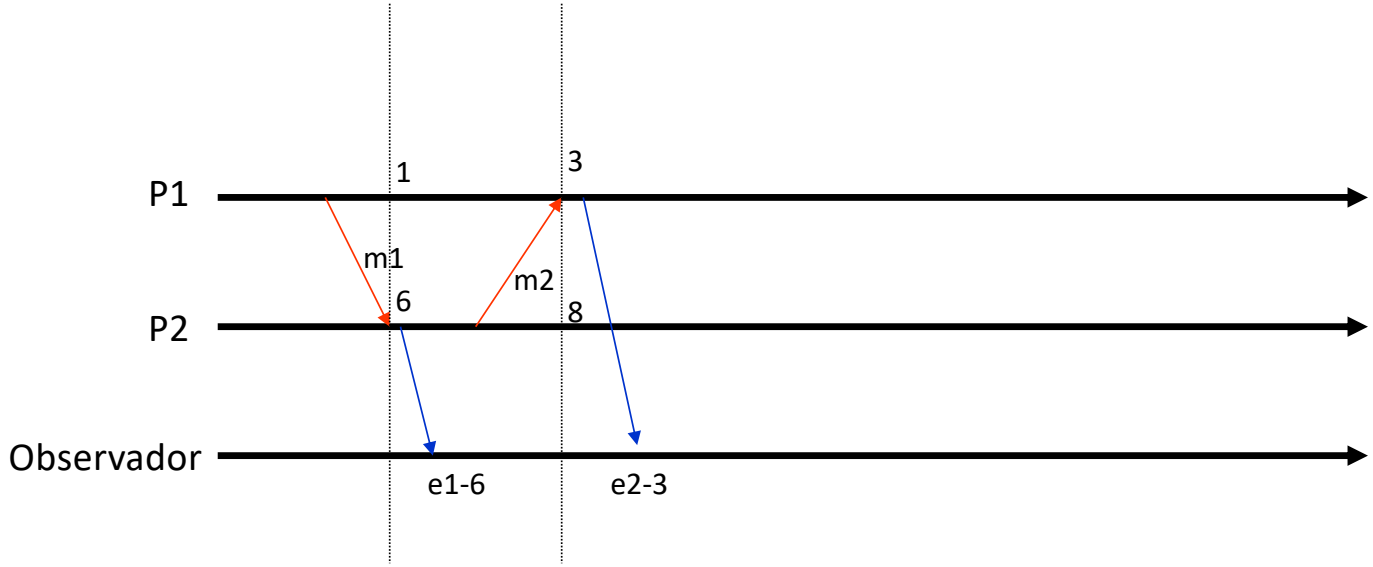
- Para ordenar eventos podemos asignarles marcas de tiempo
  - ei ← ek ⇔ MT(ei) < MT (ek)

¿Marcas de tiempo en el observador?



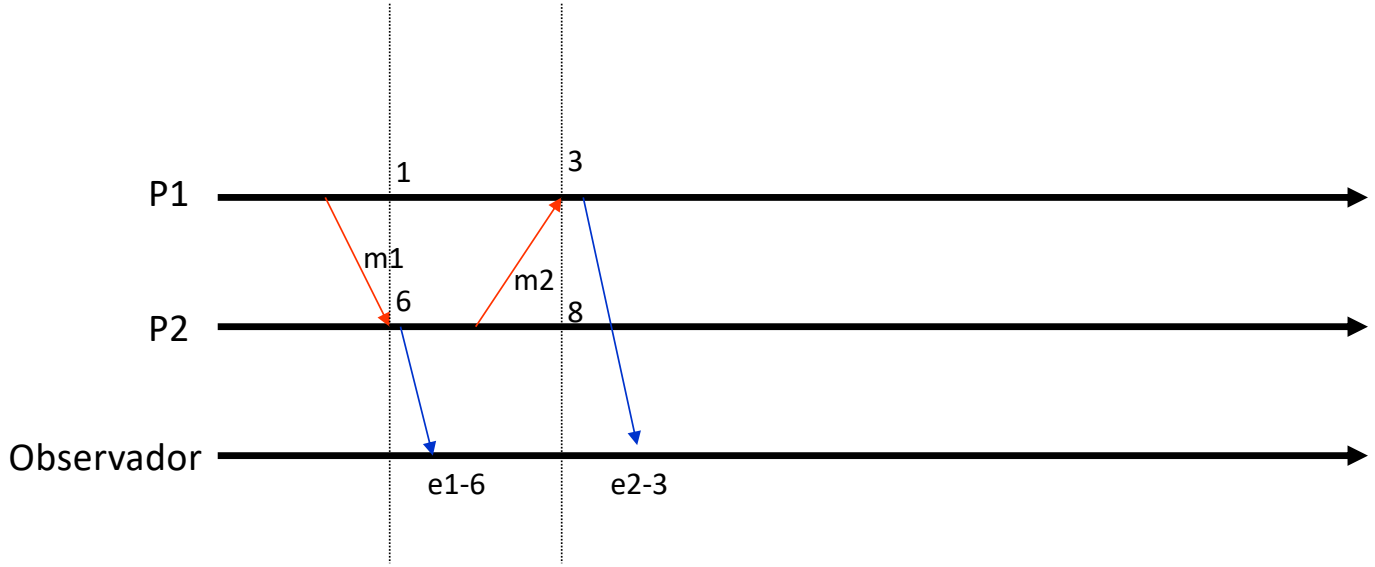
?

¿Marcas de tiempo en de los procesos?



?

¿Marcas de tiempo en de los procesos?

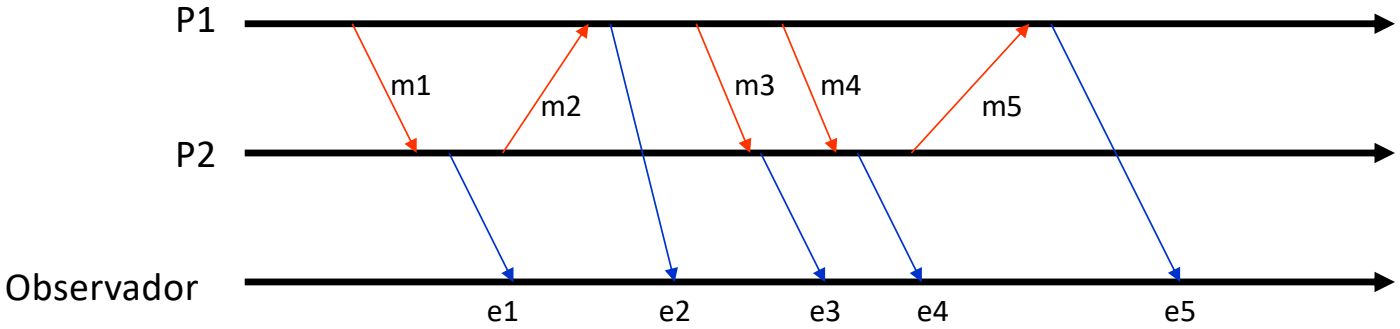


Los relojes deben estar sincronizados

Relojes lógicos

- Dado que no se pueden sincronizar perfectamente los relojes físicos en un sistema distribuido, no se pueden utilizar relojes físicos para ordenar eventos

- ¿Podemos ordenar los eventos de otra forma?



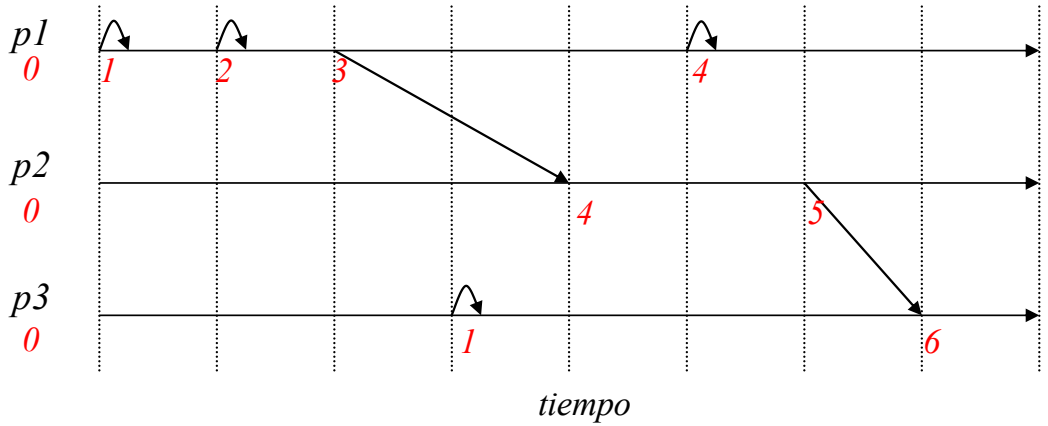
Aplicación de los relojes lógicos

- Sincronización de relojes lógicos
- Depuración distribuida
- Registro de estados globales
- Monitorización
- Entrega causal
- Actualización de réplicas

Relojes lógicos (algoritmo de Lamport)

- Algoritmo de Lamport (1978)
- Cada proceso *P* mantiene una variable entera RL_p (reloj lógico)
- Cuando un proceso *P* genera un evento, $RL_p=RL_p+1$
- Cuando un proceso envía un mensaje *m* a otro le añade el valor de su reloj
- Cuando un proceso Q recibe un mensaje *m* con un valor de tiempo *t*, el proceso actualiza su reloj, $RL_q=max(RL_q,t)+1$
- El algoritmo asegura que si $a \leq_H b$ entonces RL(a) < RL(b)

Ejemplo



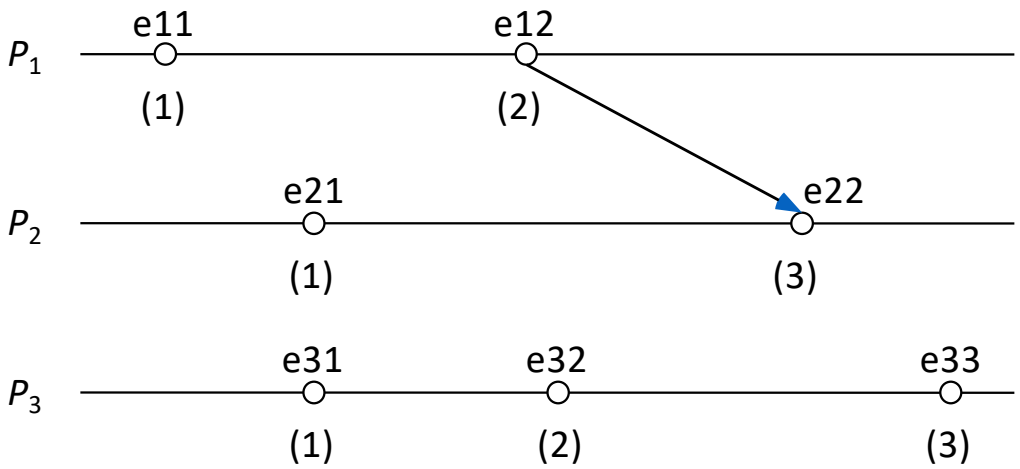
Relojes lógicos totalmente ordenados

- Los relojes lógicos de Lamport imponen sólo una relación de orden parcial:
  - Eventos de distintos procesos pueden tener asociado una misma marca de tiempo.
- Se puede extender la relación de orden para conseguir una relación de orden total añadiendo el nº de proceso
  - (T_a, P_a): marca de tiempo del evento *a* del proceso *P*
- $(T_a, P_a) < (T_b, P_b)$ sí y solo si
  - $T_a < T_b$ o
  - $T_a = T_b$ y $P_a < P_b$

Problemas de los relojes lógicos

- No bastan para caracterizar la causalidad
  - Dados *RL(a)* y *RL(b)* no podemos saber:
    - si *a* precede a *b*
    - si *b* precede a *a*
    - si *a* y *b* son concurrentes
- Se necesita una relación (*F(e),* <) tal que:
  - $a \leq_H b$ si y sólo si *F(a) < F(b)*
  - Los relojes vectoriales permiten representar de forma precisa la relación de causalidad potencial

Problemas de los relojes lógicos



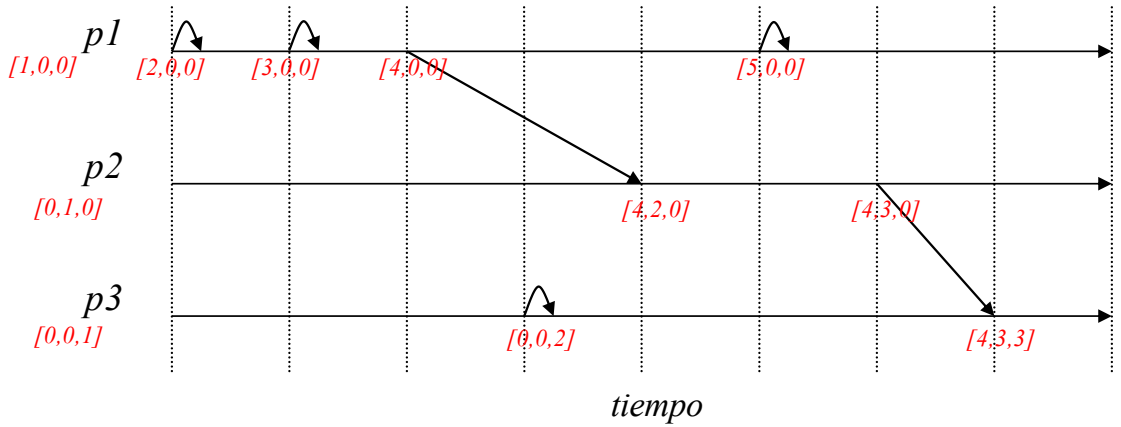
C(e11) < C(e22), y e11←e22 es cierto

C(e11) < C(e32), pero e11 ← e32 es falso

Relojes vectoriales

- Desarrollado independientemente por Fidge, Mattern y Schmuck
- Todo proceso lleva asociado un vector de enteros RV
- $RV_i[a]$ es el valor del reloj vectorial del proceso i cuando ejecuta el evento a.
- Mantenimiento de los relojes vectoriales
  - Inicialmente $RV_i = 0 \ \forall i$
  - Cuando un proceso *i* genera un evento
    - $RV_i[i\ ] = RV_i[i\ ] + 1$
  - Todos los mensajes llevan el RV del envío
  - Cuando un proceso *j* recibe un mensaje con RV
    - $RV_j = max(RV_j , RV\ )$ (componente a componente)
    - $RV_j[j\ ] = RV_j[j\ ] + 1$ (evento de recepción)

Ejemplo



Propiedades de los relojes vectoriales

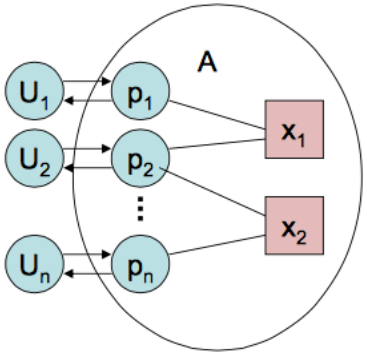
- RV < RV´ si y solo si
  - RV $\neq$ RV´ y
  - RV[i] $\leq$ RV´[i], $\forall$ i
- Dados dos eventos a y b
  - a $\leq_H$ b si y solo si RV(a) < RV(b)
  - a y b son concurrentes cuando
    - Ni RV(a) $\leq$ RV(b)[i] ni RV(b) $\leq$ RV(b)

Exclusión mutua distribuida

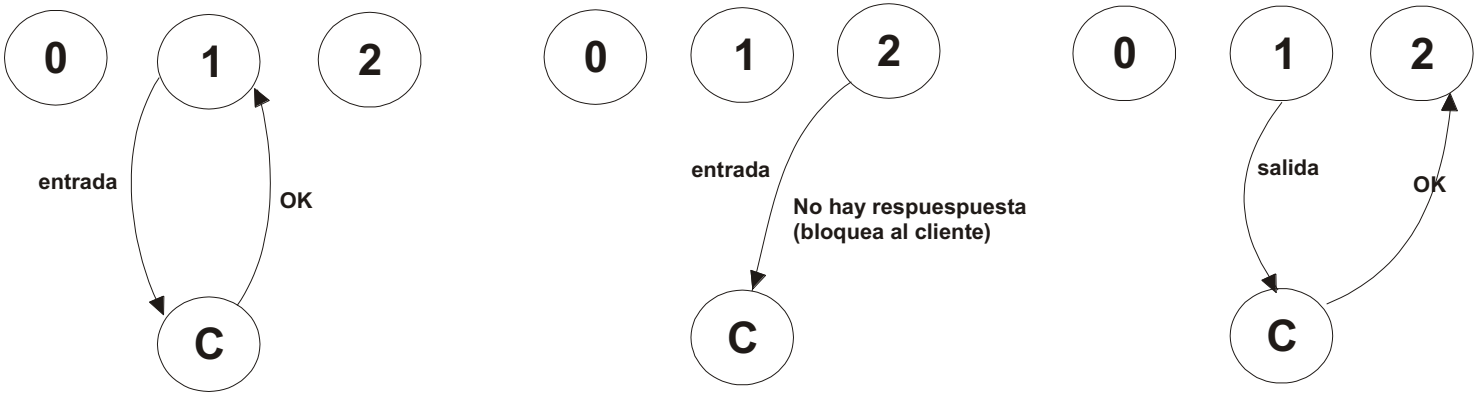
- Los procesos ejecutan el siguiente fragmento de código

  entrada()
  SECCIÓN CRÍTICA
  salida()

- *Requisitos* para resolver el problema de la sección crítica
  - Exclusión mutua
  - Progreso
  - Espera acotada
- *Algoritmos*
  - Algoritmo centralizado
  - Algoritmo distribuido
  - Anillo con testigo
  - …..



Algoritmo centralizado

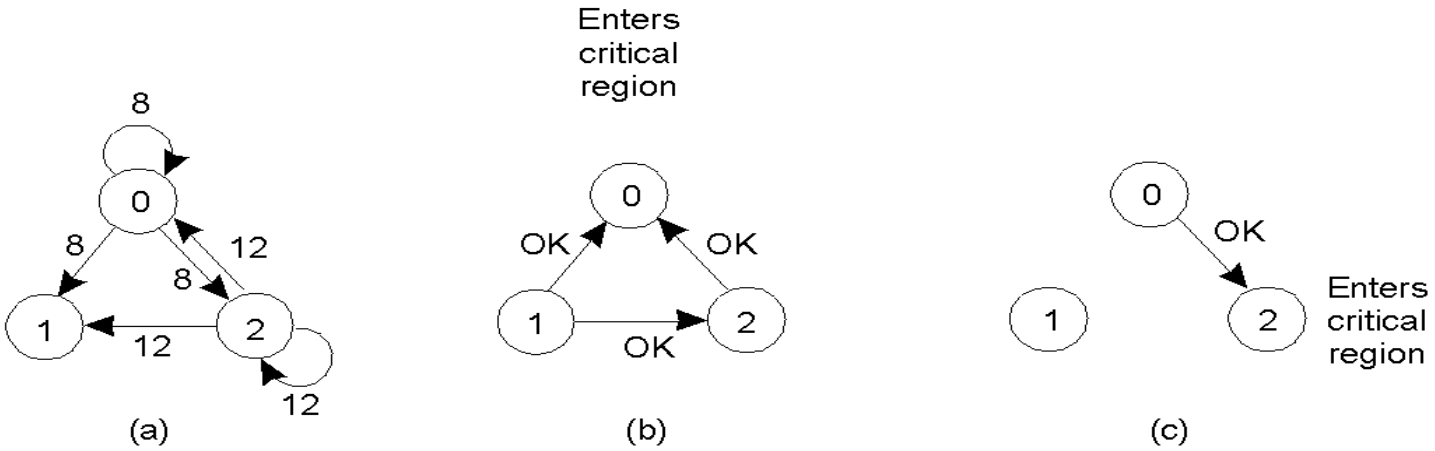


Ejemplo

a) Dos procesos (P0, P2) quieren entrar en la región al mismo tiempo

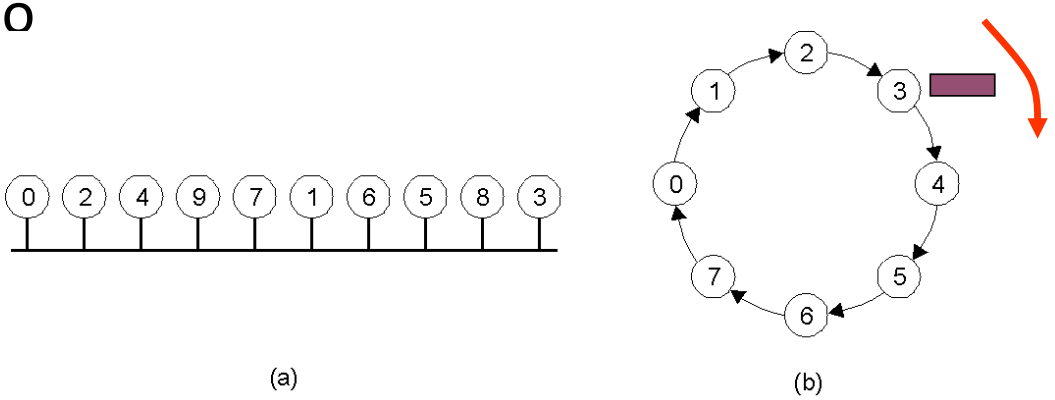
b) El proceso 0 tiene la marca de tiempo más baja, entra él.

c) Cuando el proceso 0 acaba, envía un OK, de esa forma el proceso 2 entra



Anillo con testigo

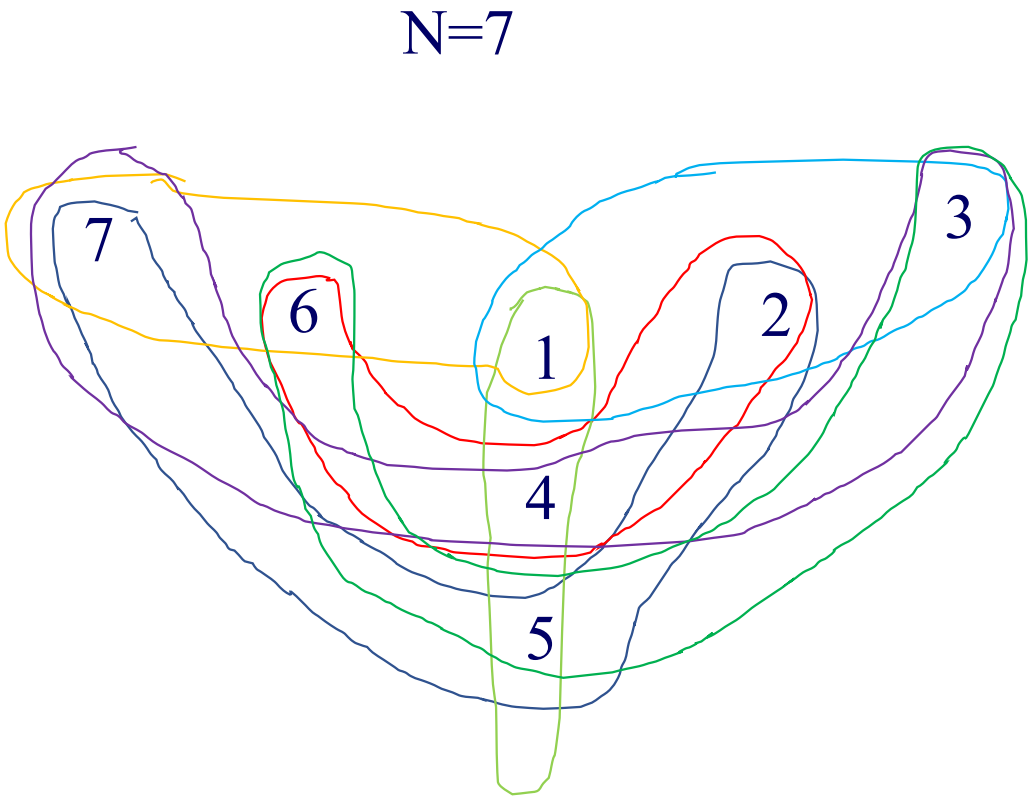
- Los procesos se ordenan conceptualmente como un **anillo**
- Por el anillo circula un **testigo**
- Cuando un proceso quiere entrar en la SC debe esperar a recoger el testigo
- Cuando sale de la SC envía el testigo al nuevo proceso del anillo



Ejemplo. Algoritmo de votación de Maekawa

- Idea: no solicitar permiso de todos los procesos, solo de un subconjunto
  - Los subconjuntos deben solaparse
- A cada proceso Pi se le asocia un conjunto de votantes
  - { Vi | i = 1, …, N }
    - Pi está en Vi
    - Vi $\cap$ Vj $\neq \varnothing$
    - Todos subconjuntos de igual tamaño
    - Cada proceso Pj está contenido en M subconjutos votantes
- Solución optima
  - K ~
  - M = K$\sqrt{N}$

Ejemplo para 7 procesos



Conjuntos

7, 6, 1

1, 4, 5

1, 2, 3

6, 4, 2

7, 4, 3

6, 5, 3

7, 5, 2

Algoritmo de votación de Maekawa

```
enter():
   state := WANTED;
   Multicast request to processes in Vi - { Pi };
   Wait until (K - 1) responses are received;
   state := HELD;

When Pj receives a request from Pi, i ≠ j:
   if(state == HELD or voted == TRUE) {
      Queue request without responding;
   } else {
      Reply to Pi;
      voted := TRUE;
   }
```

Algoritmo de votación de Maekawa

```
release():
   state :=RELEASED;
   Multicast release to processes in Vi - { Pi };

When Pj receives a release msg from Pi i ≠ j:
   if(request queue == EMPTY) {
      voted := FALSE;
   } else {
      Remove head of queue, P(k);
      Reply to process P(k);
      voted := TRUE;
   }
```

Cerrojos en ZooKeeper

Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```

Algoritmos de elección

- Útil en aplicaciones donde es necesario la existencia de un coordinador
  - Varios procesos necesitan elegir un líder/coordinador
  - Por ejemplo, en una red *token ring*, cuando el testigo se pierde
  - En ZooKeeper cuando el coordinador falla
- El algoritmo debe ejecutarse cuando **falla el coordinador**
- Algoritmos **de elección**
  - Algoritmo del matón
  - Algoritmo de anillo
- El objetivo de los algoritmos es que la elección sea única aunque el algoritmo se inicie de forma concurrente en varios procesos

Algoritmo del máton

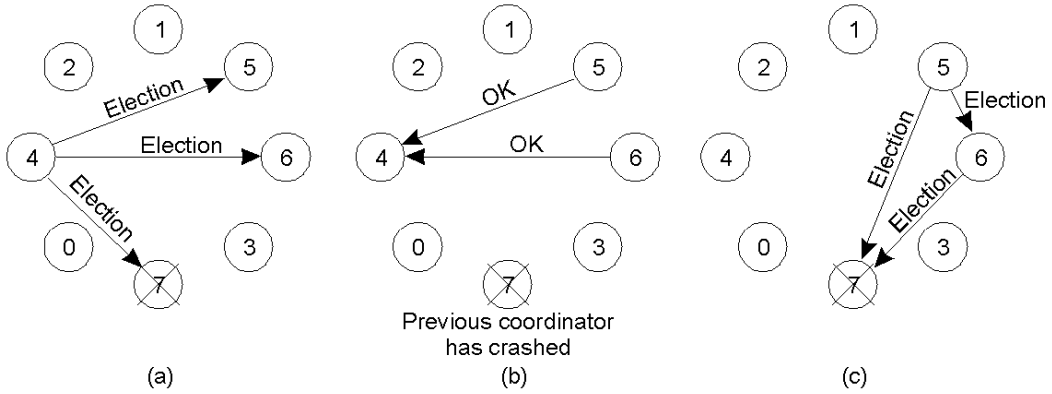
- Suposiciones:
  - Comunicación síncrona
  - Procesos completamente conectados
  - Entrega de mensajes fiable

Algoritmo del matón. Ejemplo

- Utiliza **timeouts** (T) para detectar fallos
- Asume que cada proceso conoce qué procesos tiene ID mayores
- 3 tipos de mensajes:
  - **coordinador:** anuncio a todos los procesos con IDs menores
  - **elección**: enviado a procesos con IDs mayores
  - **OK**: respuesta a elección
    - Si no se recibe dentro de T, el emisor de elección envía coordinador
    - En caso contrario, el proceso espera durante T a recibir un mensaje coordinador. Si no llega comienza una nueva elección

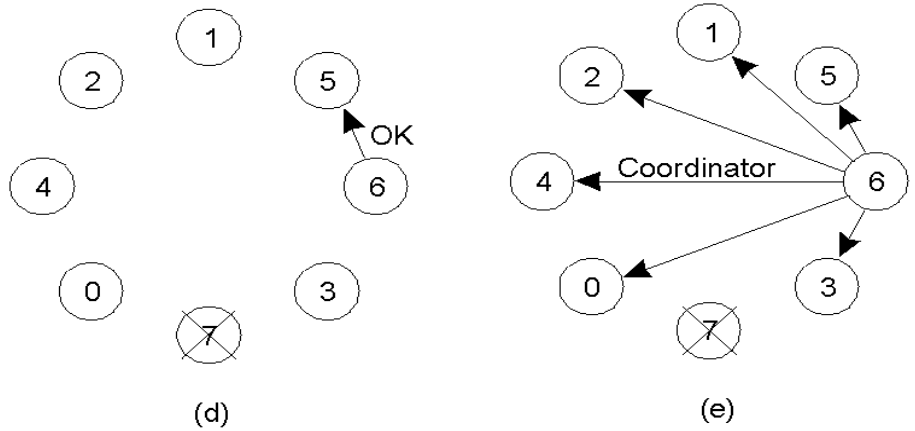
Algoritmo del matón. Ejemplo

- Cuando un proceso P observa que el coordinador no responde inicia una elección:



a) Proceso 4 envía elección
b) Proceso 5 y 6 responden, diciéndole que pare
c) Ahora 5 y 6 comienzan la elección …

Algoritmo del matón. Ejemplo



d) Proceso 6 dice a 5 que pare

e) Proceso 6 indica a todos que es el coordinador

Algoritmo LCR (basado en anillo)

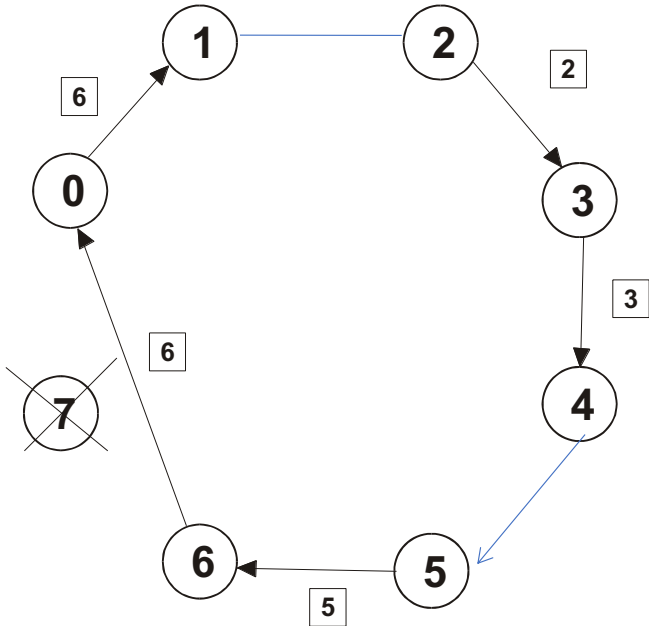
- Algoritmo LeLarnn, Chang y Roberts
- Suposiciones:
  - Comunicación síncrona
  - Procesos en anillo
  - Identificador único para cada proceso
  - Tamaño de la red desconocida

Algoritmo del anillo

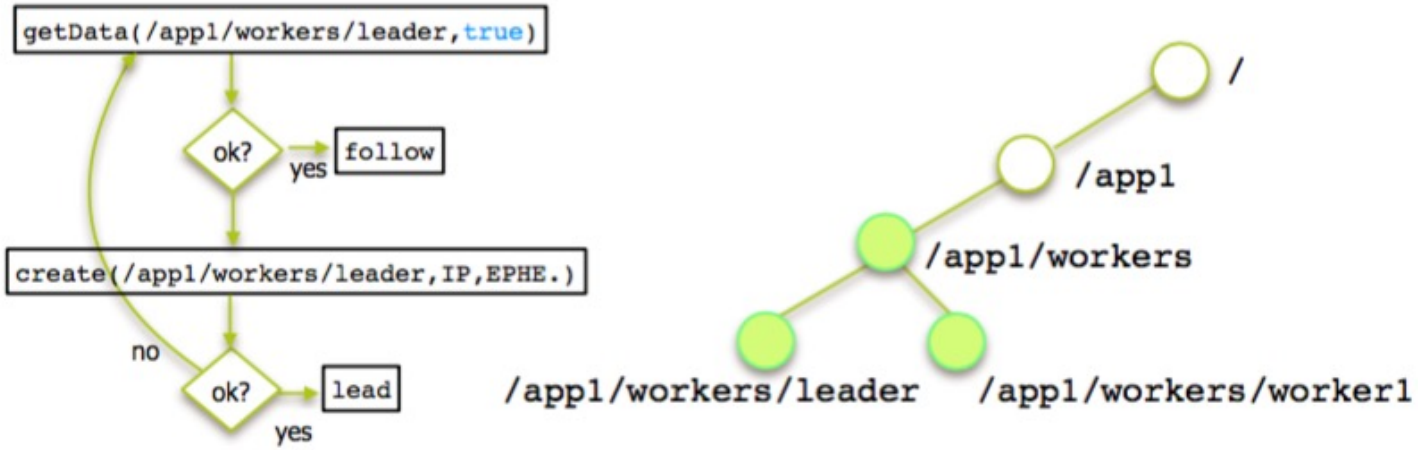
- Cualquier proceso puede comenzar la elección y envía un mensaje de **elección** a su vecino con su identificador y se marca como participante
- Cuando un proceso recibe un mensaje de **elección** compara el identificador del mensaje con el suyo:
  - Si es mayor reenvía el mensaje al siguiente
  - Si es menor y no es un participante sustituye el identificador del mensaje por el suyo y lo reenvía.
  - Si es menor y es un participante no lo reenvía
  - Cuando se reenvía un mensaje el proceso se marca como participante
- Cuando un proceso recibe un identificador con su número y es el mayor se elige como coordinador
- El coordinador notifica al resto

Algoritmo del anillo

- El 2 y el 5 generan mensajes de elección y lo envían al siguiente
- Se elige como coordinador el proceso que recibe un mensaje con su nº y es el mayor
- Este proceso a continuación envía mensajes a todos informando que es el coordinador



Elección de líder con ZooKeeper



Comunicación multicast

- Las primitivas de comunicación básicas soportan la comunicación uno a uno.
- *Broadcast*: el emisor envía un mensaje a todos los nodos del sistema.
- *Multicast*: el emisor envía un mensaje a un subconjunto de todos los nodos.
- Estas operaciones se emplean normalmente mediante operaciones punto a punto.

Utilidad

- **Servidores replicados**: un servicio replicado consta de un grupo de servidores. Las peticiones de los clientes se envían a todos los miembros del grupo. Aunque algún miembro del grupo falle la operación se realizará.
- Mejor rendimiento:
  - Replicando datos.
  - Cuando se cambia un dato, el nuevo valor se envía a todos los procesos que gestionan las réplicas.

Tipos de multicast

- ***Multicast no fiable***: no hay garantía de que el mensaje se entregue a todos los nodos.
- ***Multicast fiable***: el mensaje es recibido por todos los nodos en funcionamiento.
- ***Multicast atómico***: el protocolo asegura que todos los miembros del grupo recibirán los mensajes de diferentes nodos en el mismo orden.
- ***Multicast causal***: asegura que los mensaje se entregan de acuerdo con las relaciones de causalidad.

Justificación del multicast atómico

- Sea un banco con una base de datos replicada para almacenar las cuentas de los clientes.
- Considere la cuenta X con un saldo de 1000 euros.
  - Un usuario ingresa 200 eurosenviando un multicast a las dos bases de datos.
  - Al mismo tiempo se paga un 10% de intereses, enviando un multicast a las dos bases de datos.
  - ¿Qué ocurre si los mensajes llegan en diferente a orden a las dos bases de datos?

Implementación del multicast

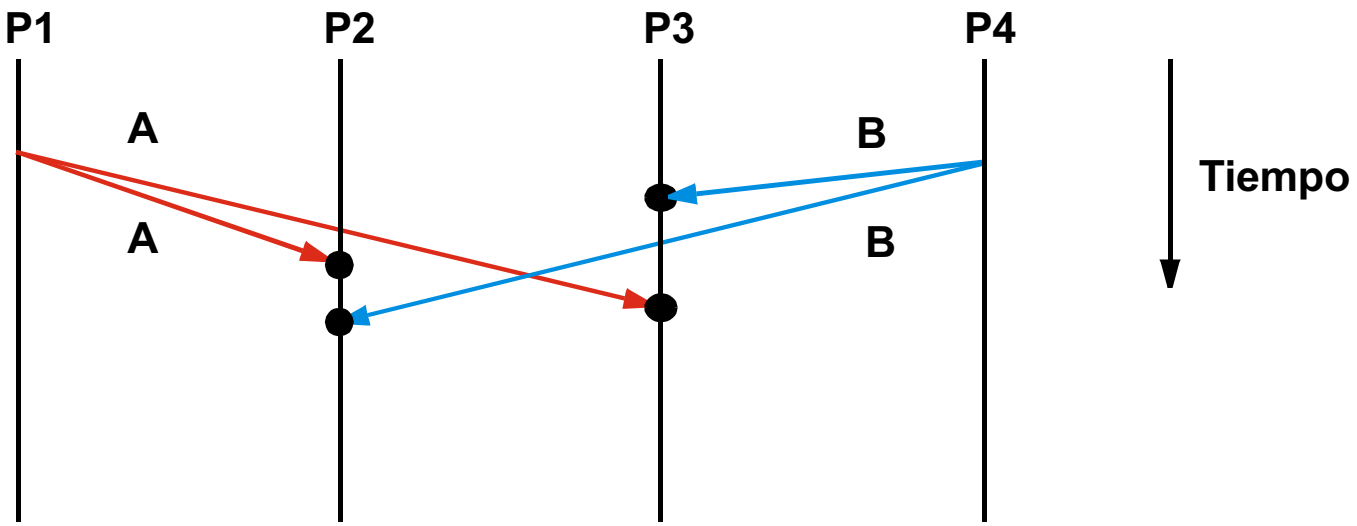
- Implementación de operaciones multicast:
  - Mediante operaciones punto a punto.
  - Mecanismo poco fiable.

- Problemas de fiabilidad:
  - Alguno de los mensajes se puede perder.
  - El proceso emisor puede fallar después de realizados algunos envíos. En este caso algunos procesos no recibirán el mensaje.

Multicast fiable

- Se envía un mensaje a todos los procesos y se espera confirmación de todos.
  - Si todos confirman el multicast se ha completado.
  - Si alguno no confirma se retransmite. Si no envía confirmación se puede asumir que el proceso ha fallado y se elimina del grupo.
- Si el emisor falla durante el proceso la operación no será atómica.
  - Para que la operación sea atómica, si el emisor falla alguno de los receptores debe completar el envío a todos los demás.
    - Cuando un proceso recibe un mensaje lo envía al resto
    - Cuando un proceso recibe un mensaje envía una confirmación y monitoriza al emisor para ver si falla. Si falla completa el multicast.
    - En cualquier caso hay que descartar los mensajes duplicados

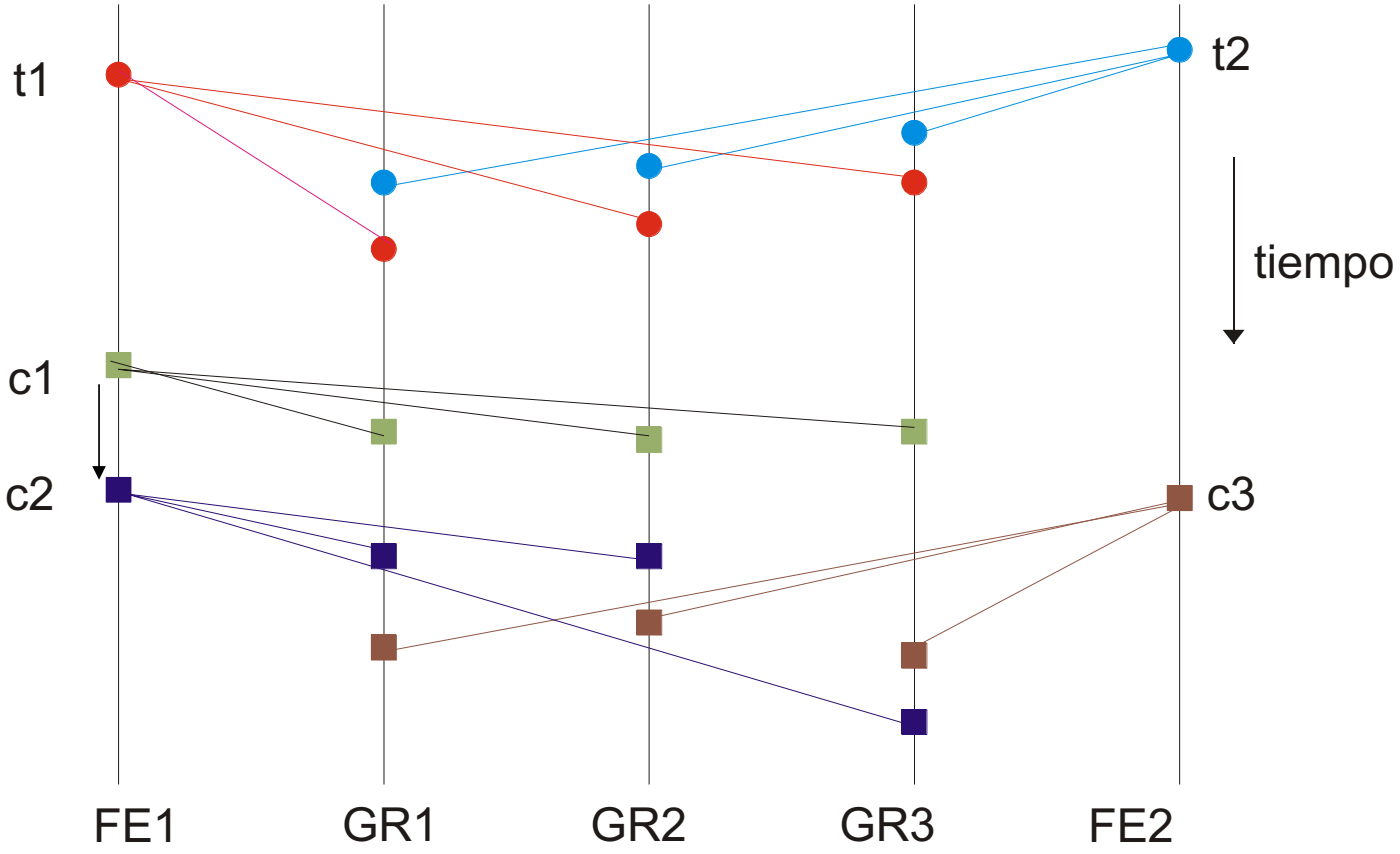
Ejemplo de multicast sin ordenación



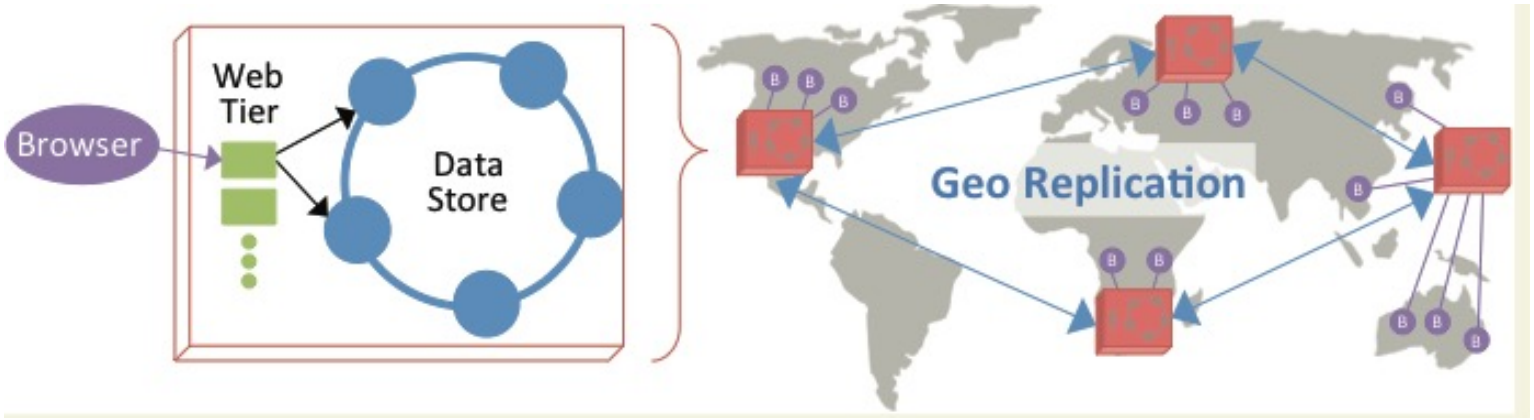
Ordenación de las actualizaciones

- Es importante el orden en el que se realizan las peticiones
  ¿Qué ocurre en un sistema asíncrono cuando un cliente modifica un dato y más tarde otro cliente quiere consultar ese dato?
- Algunas aplicaciones requieren un orden en la realización de las peticiones.
- *Orden total:* dadas dos peticiones r_1 y r_2 entonces o r_1 es procesada en todos los procesos antes que r_2 o r_2 lo es antes que r_1.
- *Ordenación causal*: se basa en la relación de precedencia que recoge las relaciones de causalidad potencial. Si r_1 precede a r_2 entonces r_1 se procesa antes que r_2 en todos los procesos

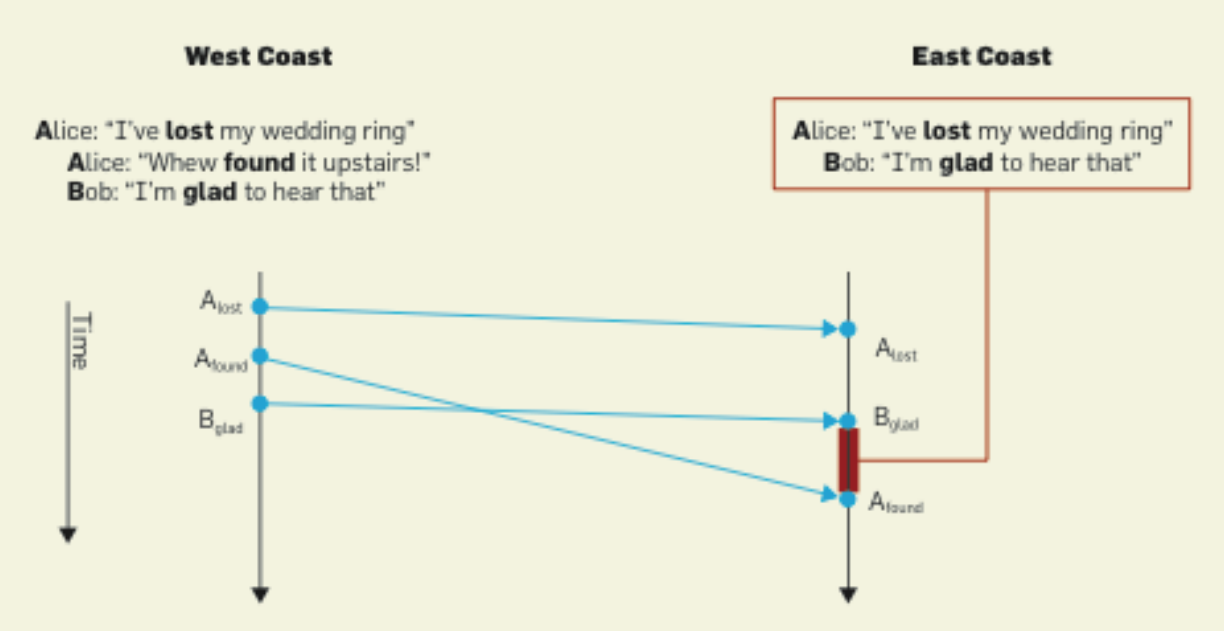
Ordenación total y causal



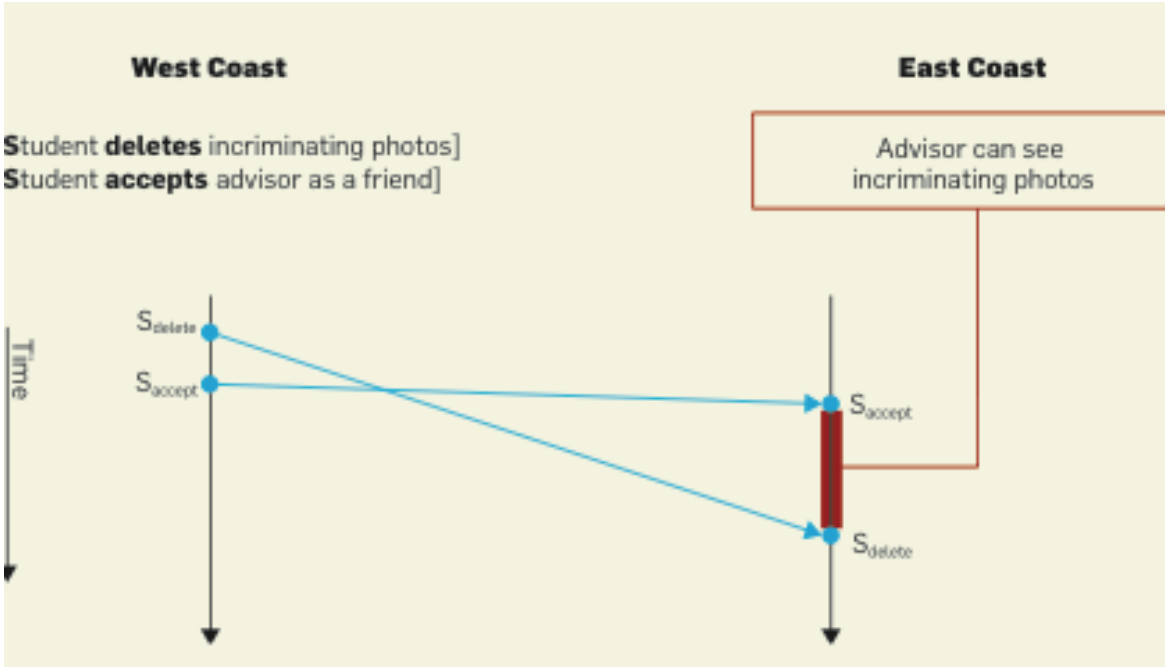
Problemas en servidores georeplicados



Tipos de anomalías que no siguen orden causal (1)

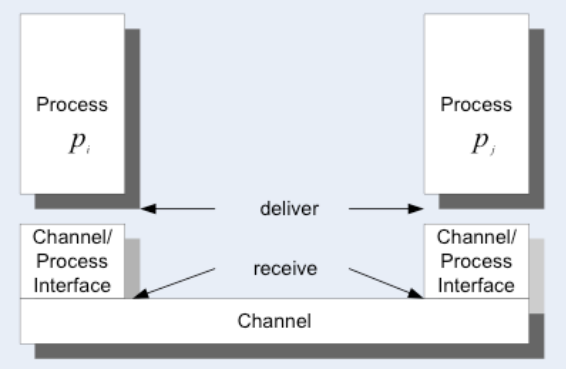


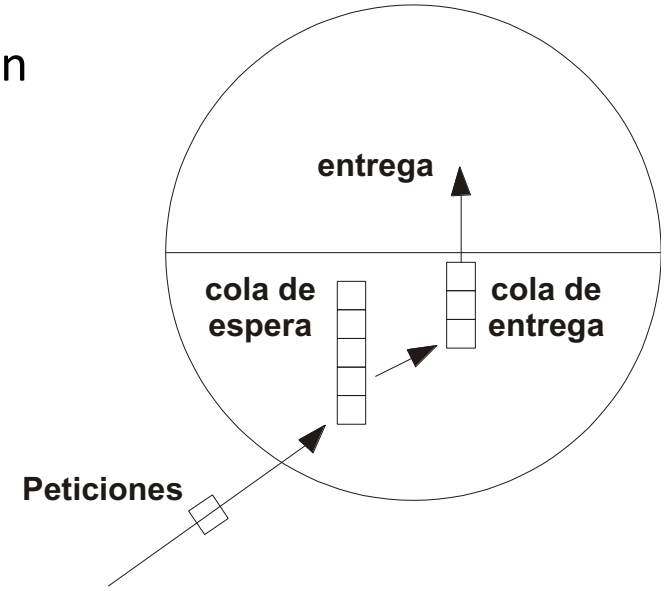
Tipos de anomalías que no siguen orden causal (2)



Implementación

- Una petición recibida no se entrega hasta que las restricciones de ordenación se puedan cumplir.
- Un mensaje estable pasa a la cola de entrega
- Debe asegurarse
  - Seguridad: ningún mensaje fuera de orden
  - Progreso: todos los mensajes se entregan





Implementación de la ordenación total

- Se asigna a cada petición un **identificador** de orden total (IOT)
- Este identificador se utiliza para entregar los mensajes en el mismo orden a todos los procesos
- **Método centralizado**:
  - Se utiliza un proceso secuenciador encargado de asignar IOT a los mensajes
  - Cada mensaje se envían al secuenciador
    - El secuenciador incrementa IOT
    - El secuenciador le asigna un IOT y lo envía a los procesos
  - Cuando un proceso recibe un mensaje con un IOT mayor del esperado, pide al secuenciador que le envíe de nuevo el mensaje
  - Posible cuello de botella y punto de fallo crítico

Método distribuido

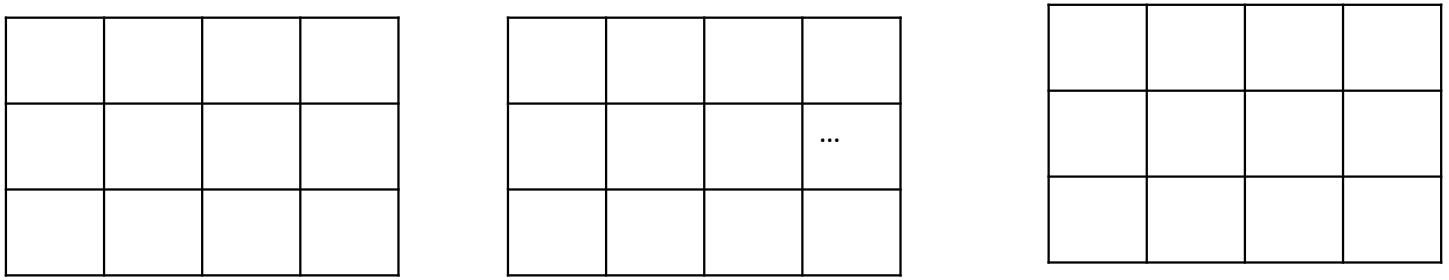
- Birman y Joseph 1987
- Cada proceso q en el grupo almacena:
  - A_q: mayor número de secuencia acordado que se ha observado hasta el momento
  - P_q: su mayor número de secuencia propuesto
  - Los identificadores deben incluir el número de proceso para asegurar un orden total
- Cuando un proceso p realiza un BCAST envía el mensaje al resto
- Cada proceso q que recibe el mensaje de p
  - Propone P_q=Max(A_q, P_q) + 1
  - Almacena (m, P_q) en su cola y lo marca como no entregable
  - Envía P_q al emisor del mensaje (p)
- El proceso q recibe todos los números de secuencia propuesto y selecciona el mayor A como el siguiente número de secuencia acordado y lo envía a todos
  - En q se fija A_q = Max(A_q, A) y se marca el mensaje como entregable
  - Se reordena la cola y si está el primero se entrega

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

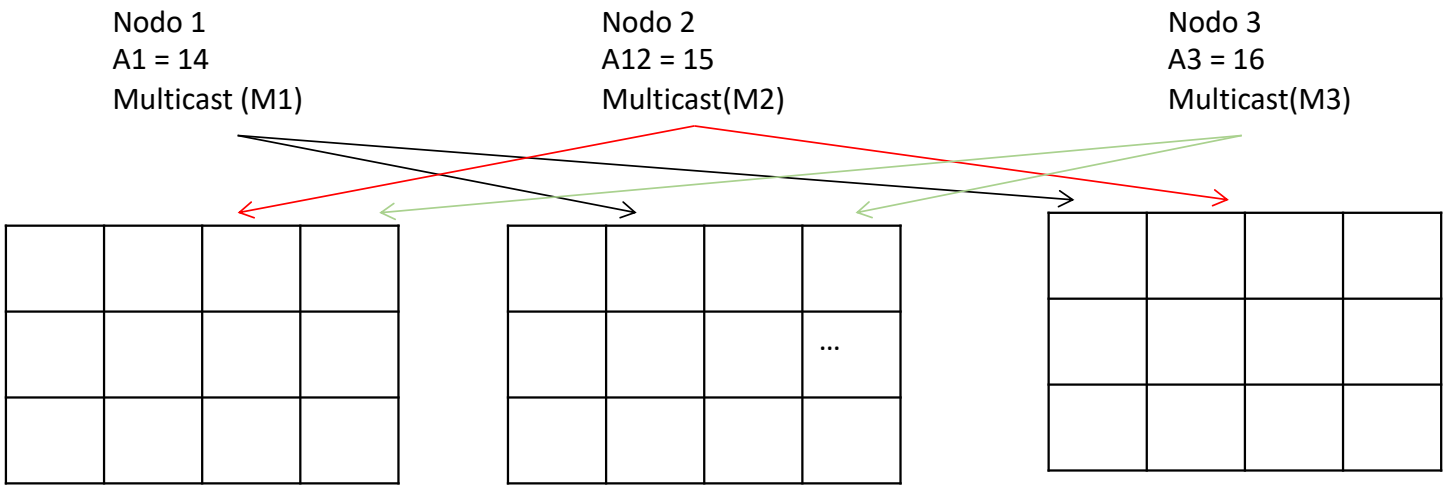
Nodo 2
A12 = 15
Multicast(M2)

Nodo 3
A3 = 16
Multicast(M3)



Inicialmente:
Los tres nodos realizan un multicast simultáneo

Ejemplo



Inicialmente:
Los tres nodos realizan un multicast simultáneo

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	16.1	17.1	…
U	U	U	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.2	18.2	…
U	U	U	

Nodo 3
A3 = 16
Multicast(M3)

M1	M3	M2	
17.3	18.3	19.3	…
U	U	U	

Etapa 1:
Los mensajes llegan a los receptores en orden distinto
Se les propone un número de secuencia, P_q=Max(A_q, P_q) + 1
(se añade identificador de proceso)
Se insertan en las colas y se marcan como no entregables (U)

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	17.1	…
U	U	U	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	U	U	

Nodo 3
A3 = 16
Multicast(M3)

M1	M3	M2	
17.3	18.3	19.3	…
U	U	U	

Etapa 2:
El Nodo 1 recibe las marcas asociada a M1 envidas por el nodo 2 (17.2) y 3 (17.3)
y calcula el máximo de las tres, y se las envía al resto (17.3)

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

Nodo 3
A3 = 16
Multicast(M3)

M1	M3	M2	
17.3	18.3	19.3	…
D	U	U	

Etapa 2:
Se marca M1 como entregable y se reordenan las colas
M1 se puede entregar en el nodo 3 porque ser el primero de la cola

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	U	

Etapa 2:
M1 se entrega en el nodo 3

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
16.2	17.3	18.2	…
U	D	U	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	U	

Etapa 3:
El Nodo 2 recibe las marcas asociada a M2 envidas por el nodo 1 (17.1) y 3 (19.3) ,
Calcula el máximo (19.3)

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	19.3	17.3	…
U	U	D	

Nodo 2
A12 = 15
Multicast(M2)

M2	M1	M3	
19.3	17.3	18.2	…
U	D	U	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	U	

Etapa 3:
El Nodo 2 recibe las marcas asociada a M2 envidas por el nodo 1 (17.1) y 3 (19.3) ,
Calcula el máximo (19.3)
Se la envía al resto

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	19.3	…
U	D	D	

Nodo 2
A12 = 15
Multicast(M2)

M1	M3	M2	
17.3	18.2	19.3	…
D	U	D	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	D	

Etapa 3:
M2 se marca como entregable y se reordenan las colas

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	19.2	…
U	D	D	

Nodo 2
A12 = 15
Multicast(M2)

M3	M2	
18.2	19.3	…
U	D	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	D	

Etapa 3:
M2 se marca como entregable y se reordenan las colas
M1 se entrega en el nodo 2

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	19.2	…
U	D	D	

Nodo 2
A12 = 15
Multicast(M2)

M3	M2	
18.2	19.3	…
U	D	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	D	

Etapa 4:
El Nodo 3 recibe las marcas asociada a M3 envidas por el nodo 1 (15.1) y 3 (18.2)
Calcula el máximo de todas (18.3)

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
18.3	17.3	19.2	…
U	D	D	

Nodo 2
A12 = 15
Multicast(M2)

M3	M2	
18.3	19.3	…
U	D	

Nodo 3
A3 = 16
Multicast(M3)

M3	M2	
18.3	19.3	…
U	D	

Etapa 4:
El Nodo 3 recibe las marcas asociada a M3 envidas por el nodo 1 (15.1) y 3 (18.2)
Calcula el máximo de todas (18.3)
Se las envía al resto

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M1	M3	M2	
17.3	18.3	19.2	…
D	D	D	

Nodo 2
A12 = 15
Multicast(M2)

M3	M2	
18.3	19.3	…
D	D	

Nodo 3
A3 = 16
Multicast(M3)

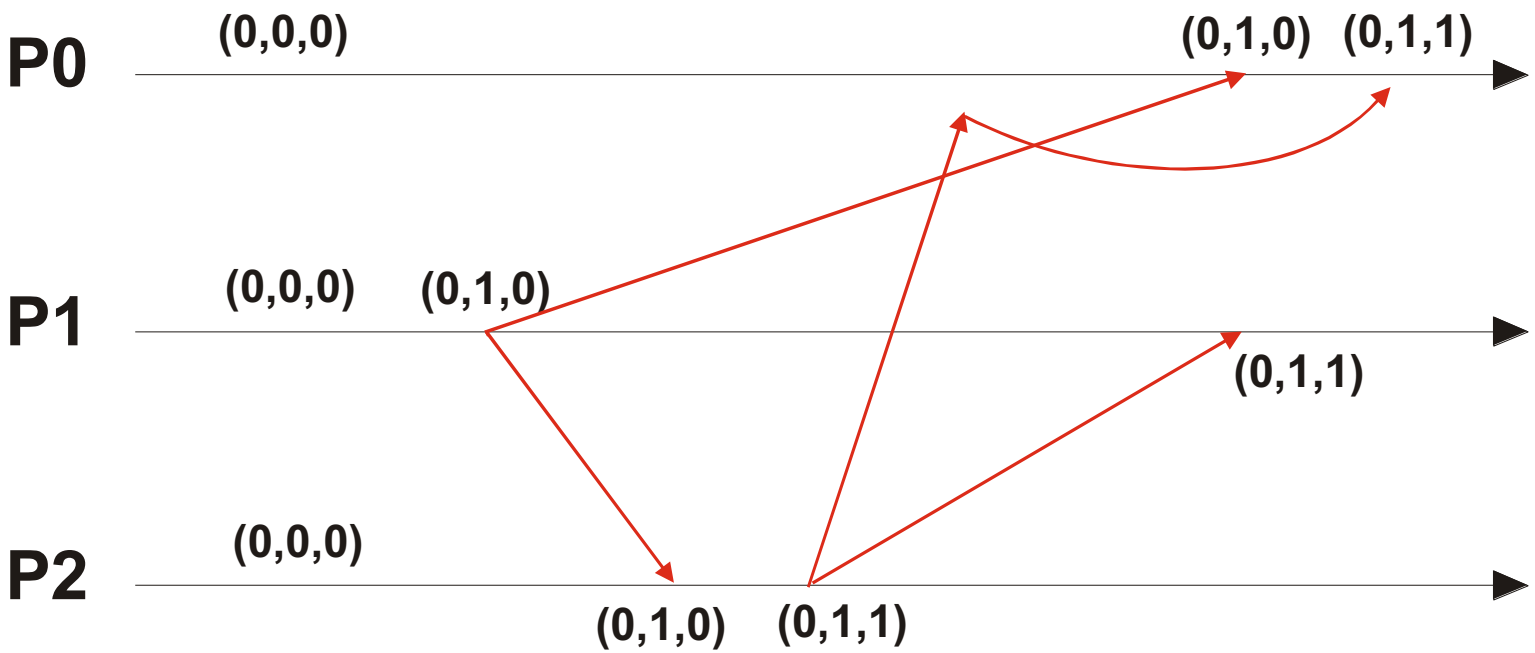
M3	M2	
18.3	19.3	…
D	D	

Etapa 4:
Se marca M3 como entregable y se reordenan las colas
Se pueden entregar todos los mensajes en todos los nodos
El orden de entrega: M1, M3 y M2 (se asegura el orden de entrega en todos los nodos)

Implementación de la ordenación causal

- Cada proceso **p_i**, almacena un **vector VT** con n componentes
- En el proceso **p_j**, la componente **i** indica el último mensaje recibido de *i*
- Algoritmo para actualizar el vector
  - Todos los procesos inicializan el vector a 0
  - Cuando **p_i** envía un nuevo mensaje incrementa **$VT_i(i)$** en 1 y añade VT al mensaje
- Cuando a **p_j** le llega un mensaje de **p_i** con VT se entrega si:
  - $VT(i) = VT_j(i) + 1$ (siguiente en la secuencia de pi)
  - $VT(k) \leq VT_j(k)$ para todo $k \neq i$ (todos los mensajes anteriores se han entregado a i)
- Cuando un mensaje con VT se entrega a p_j se actualiza su vector:
  - $VT_j = max(VT_j, VT)$, para k=1, 2, ... , n

Ejemplo



Ejemplo

- Vector enviado por el proceso 0: (4, 6, 8, 2, 1, 5)
- Vector en el proceso 1: (3, 7, 8, 2, 1, 5)
- Vector en el proceso 2: (3, 5, 8, 2, 1, 5)
- ¿Se puede entregar el mensaje enviado por el 0?

Ejemplo

- Vector enviado por el proceso 0: (4, 6, 8, 2, 1, 5)
- Vector en el proceso 1: (3, 7, 8, 2, 1, 5)
- Vector en el proceso 2: (3, 5, 8, 2, 1, 5)
- ¿Se puede entregar el mensaje enviado por el 0?

Cuando a p_j le llega un mensaje de p_i con VT se entrega si:

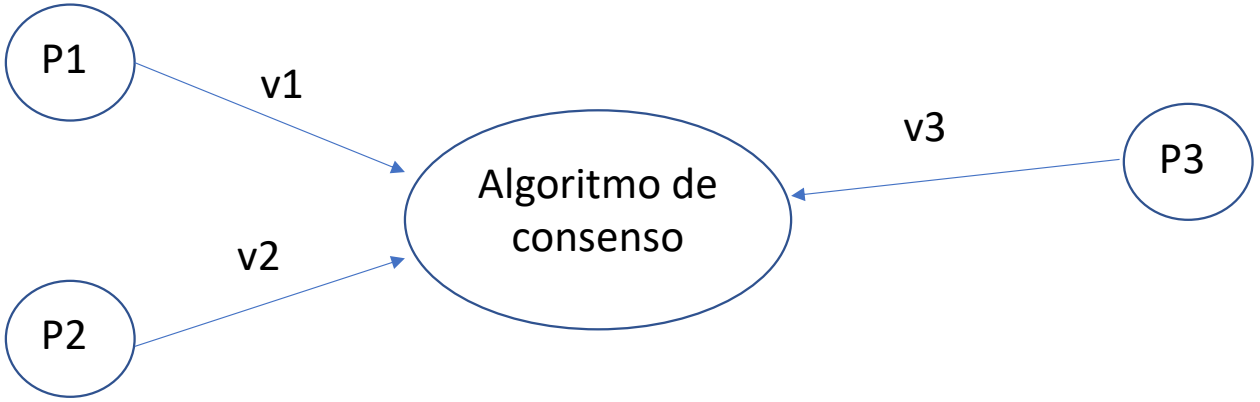
- $VT(i) = VT_j(i) + 1$ (siguiente en la secuencia de pi)
- $VT(k) \leq VT_j(k)$ para todo $k \neq i$ (todos los mensajes anteriores se han entregado a i)

Ejemplo

- Vector enviado por el proceso 0: (4, 6, 8, 2, 1, 5)
- Vector en el proceso 1: (3, 7, 8, 2, 1, 5)
- Vector en el proceso 2: (3, 5, 8, 2, 1, 5)
- ¿Se puede entregar el mensaje enviado por el 0?
  - Al 1 si:
    - Es el siguiente en la secuencia de mensajes recibidos del 0 y no se han perdido mensajes.
  - Al 2 no:
    - Es el siguiente en la secuencia de mensajes recibidos del 0.
    - Le falta un mensaje del proceso 1

Protocolos de consenso

- Dado un conjunto de proceso $P_1 \ldots P_n$ que se comunican mediante paso de mensajes, el objetivo es alcanzar un acuerdo sobre un determinado valor aun en presencia de fallos



Aplicaciones

- Transacciones distribuidas
- En grupos de procesos
- Elección de líder
- Sincronización de relojes
- Servidores replicados

Consenso basado en quorums

- Se definen dos operaciones READ y WRITE
- Hay un conjunto de N nodos, que sirven peticiones
  - Un READ debe realizarse sobre R copias
  - Un WRITE debe realizarse sobre W copias
  - Cada réplica tiene un número de versión V
  - Debe cumplirse que:
    - $R + W > N$
    - $W + W > N$
    - $R, W < N$

Método de votación

- READ
  - Se lee de R réplicas, se queda con la copia con la última versión
- WRITE
  - Hay que asegurarse que todas las réplicas se comportan como una sola (seriabilidad)
  - Se realiza en primer lugar una operación READ para determinar el número de versión actual.
  - Se calcula el nuevo número de versión.
  - Se inicia un 2PC para actualizar el valor y el número de versión en W o más réplicas.

Two-phase-commit

- Two-phase-commit (2PC), Jim Gray (1970)
- Denominamos coordinador al proceso que realiza la operación

Coordinador:

multicast: *ok to commit*?

recoger las respuestas

todos ok => *send(commit)*

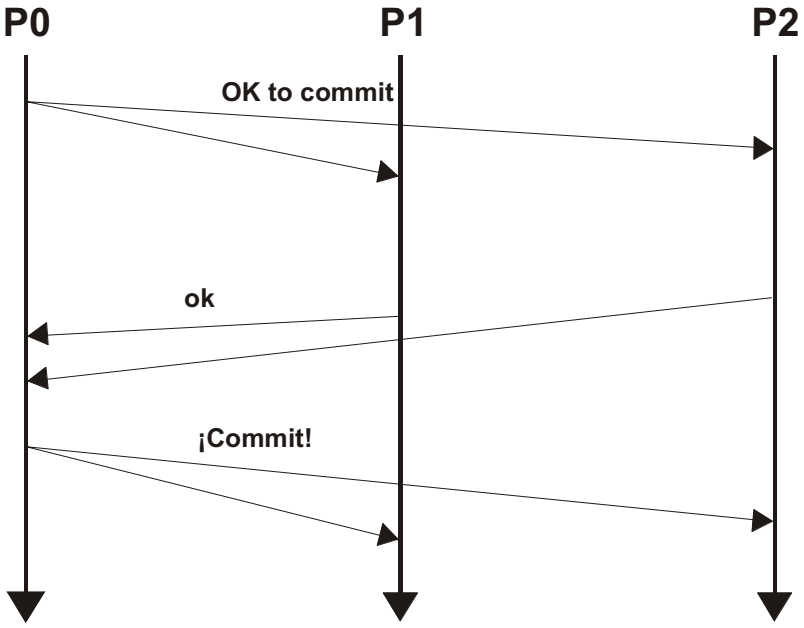
else => *send(abort*)

Procesos:

ok to commit=> guardar la petición en un área temporal y responder *ok*

commit => hacer los cambios permanentes

abort => borrar los datos temporales



Paxos

- Leslie Lamport (1989)
- Protocolo para alcanzar consenso basado en máquinas de estado finito (asegura seguridad y terminación)
- Todos los nodos alcanzan consenso a pesar de fallos en los nodos, particiones de red y retrasos en la entrega de mensajes

- Paxos made simple (Leslie Lamport)

Aplicación de Paxos

- Google: Chubby (servicio de cerrojos distribuido basado en Paxos)
  - Bigtable y otros servicios de Google utilizan Chubby
- Yahho: ZooKeeper (servicio de cerrojos distribuido basado en Paxos)

Acuerdo bizantino

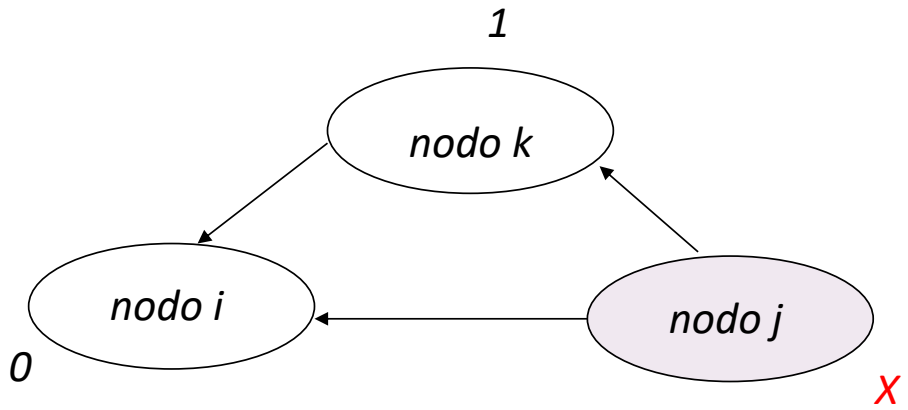
- En la mayoría de las ocasiones cuando un componente o sistema falla, su funcionamiento es arbitrario.
  - Pueden enviar información diferente a diferentes componentes con los que se comunica.
  - Alcanzar un acuerdo entre las observaciones que hacen diferentes componentes puede ser complicado en presencia de fallos.
- El objetivo con acuerdo bizantino es alcanzar un **acuerdo** sobre un determinado valor en un sistema donde los componentes pueden fallar de forma arbitraria.
- Importancia:
  - Permite enmascarar fallos arbitrarios.
  - Permite construir procesadores con fallos de tipo fallo-parada

Definición del problema

- Sistema distribuido compuesto por una serie de nodos (generales) que intercambian información entre ellos.
- Los componentes pueden exhibir fallos bizantinos (generales traidores)
  - Un nodo con fallo puede enviar información diferente a diferentes nodos (para un mismo dato).

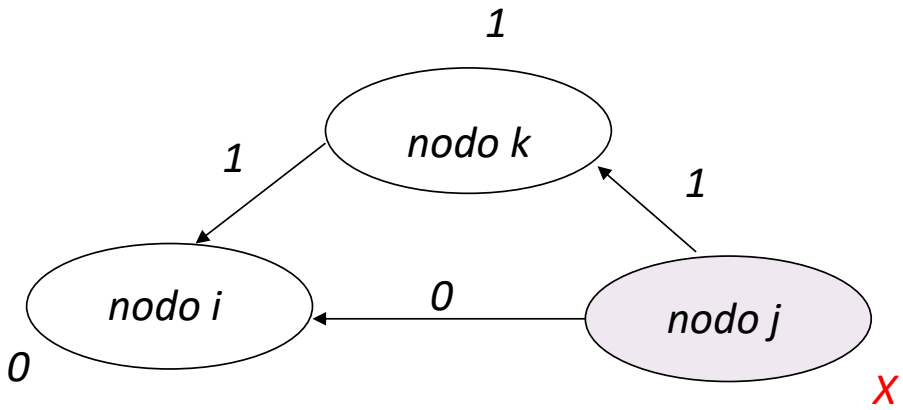
- **Objetivo**: que los nodos sin fallo alcancen un acuerdo o consenso sobre un determinado valor (ataque, retirada, espera). Es decir que *vean* el mismo valor para un dato.

Problema con tres nodos

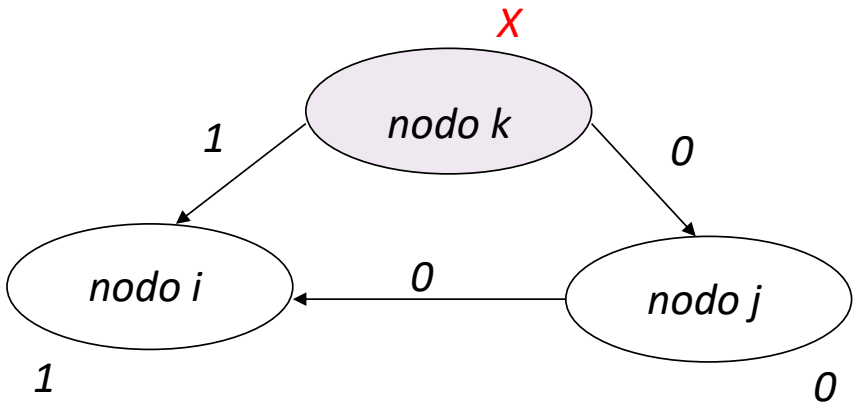


¿puede el nodo i y nodo k llegar a un consenso en el valor?

Problema con tres nodos

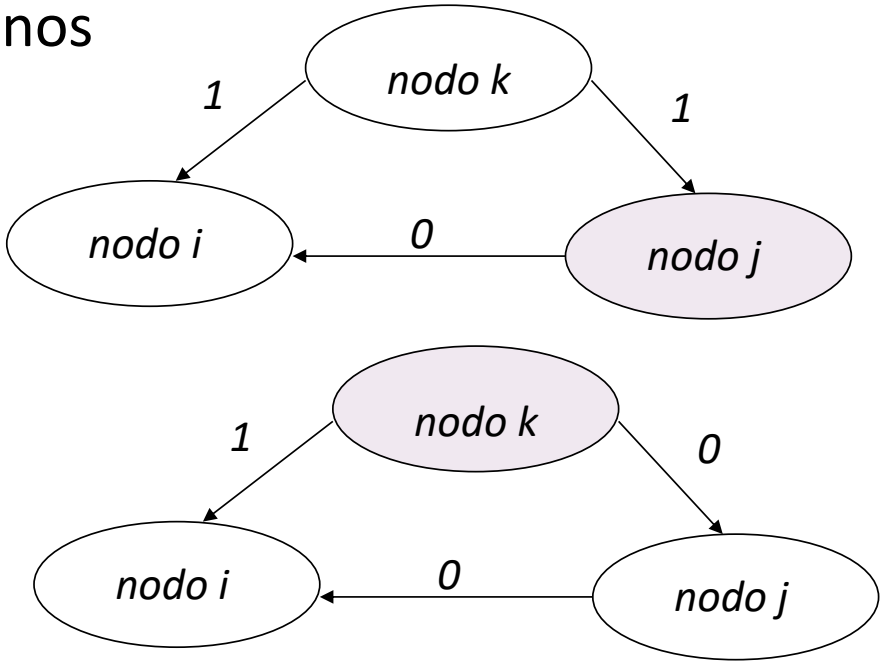


Problema con tres nodos



Problema con tres nodos

- Utilizando tres nodos, si uno falla el problema no puede resolverse.
- Se necesitan *3m+1* nodos para hacer frente a *m* fallos bizantinos



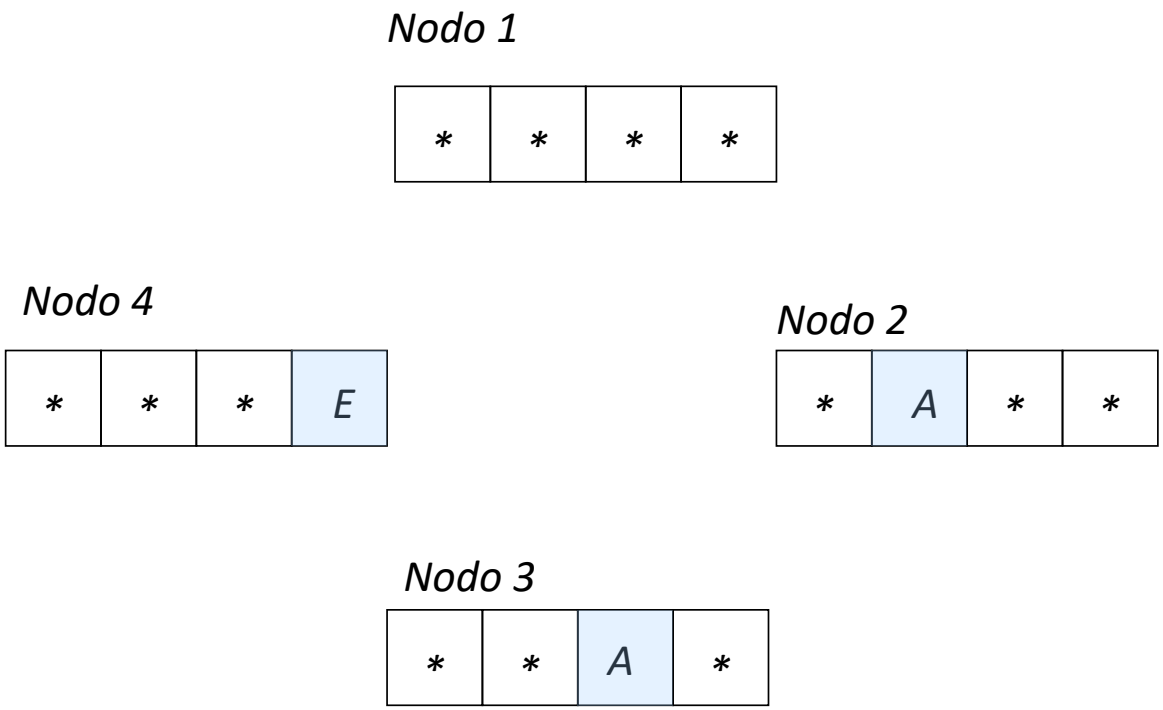
Solución para cuatro nodos

- Suposiciones:
  - Los mensajes enviados se entregan correctamente.
  - El receptor conoce al emisor de un mensaje
  - Se puede detectar la ausencia de un mensaje
- Algoritmo (*Lamport, Pease y Shostack*) para cuatro nodos:
  - Se basa en el intercambio de mensajes entre los nodos
  - Cada nodo N_i realiza la observación O_i
  - Cada nodo mantiene un vector V con información recibida de los otros nodos. $V_i(j)$ almacena el valor recibido de N_j
  - Inicialmente $V_i(i) = O_i$ y $V_i(j) = null$ ($\forall\ i \neq j$)
  - Cada nodo envía un mensaje al resto indicando su observación.
  - Cuando un nodo recibe una observación actualiza su vector y envía a los otros dos nodos la observación recibida.

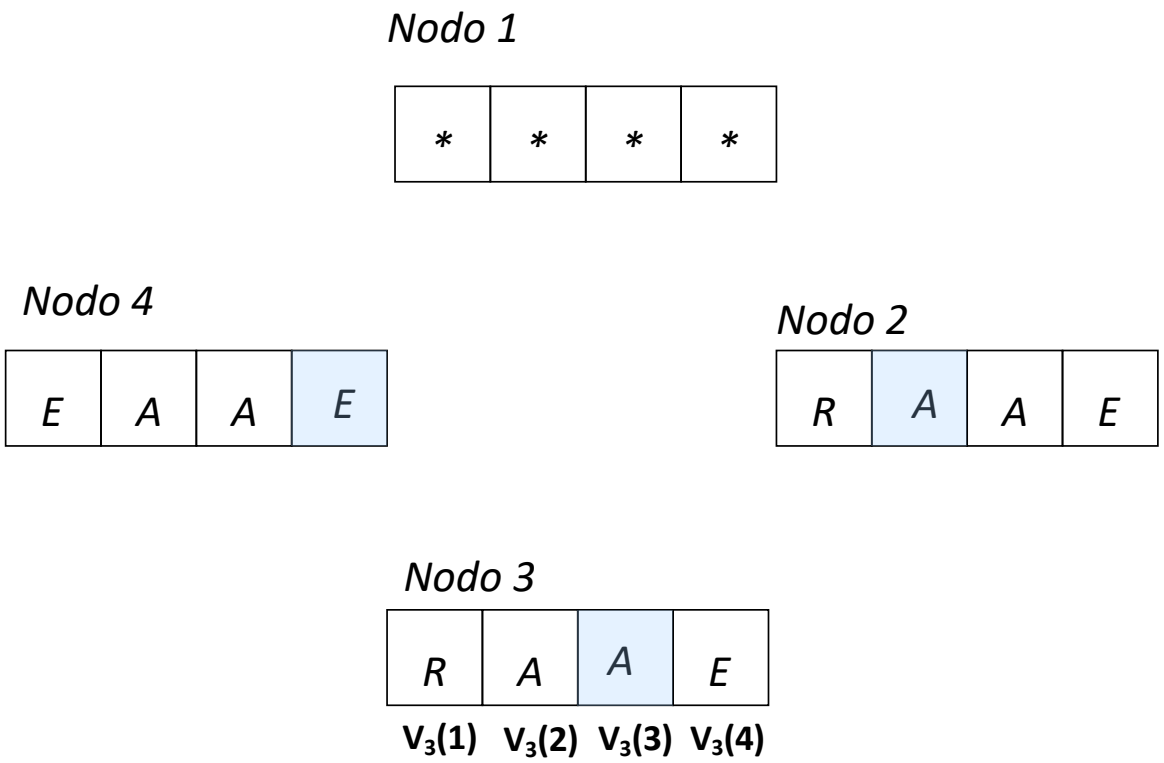
Solución

- Después del intercambio de mensajes, cada nodo construye un vector con los valores mayoritarios
- Si no existe mayoría entonces se asume que no hay un observación coherente.

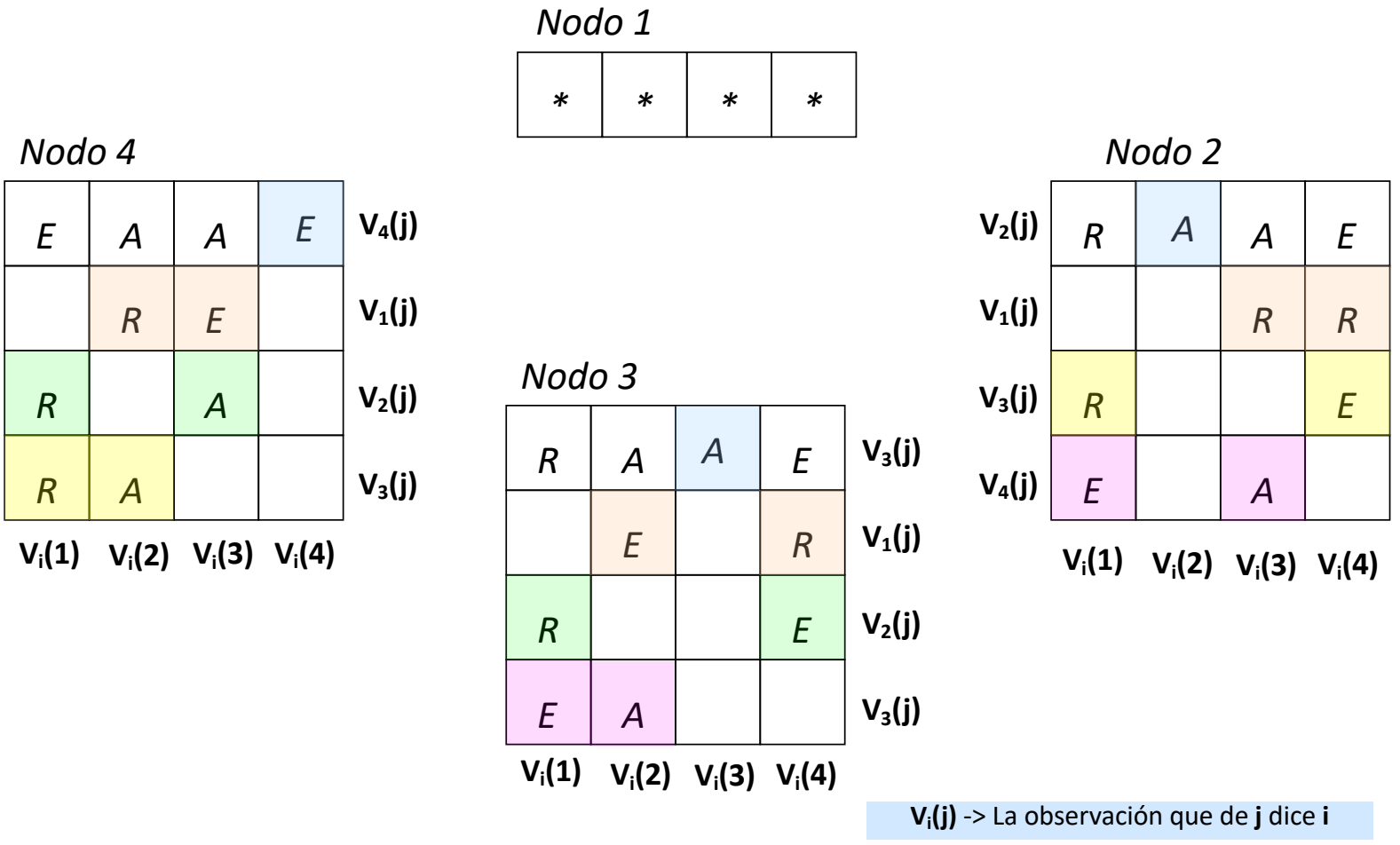
Ejemplo. Etapa inicial



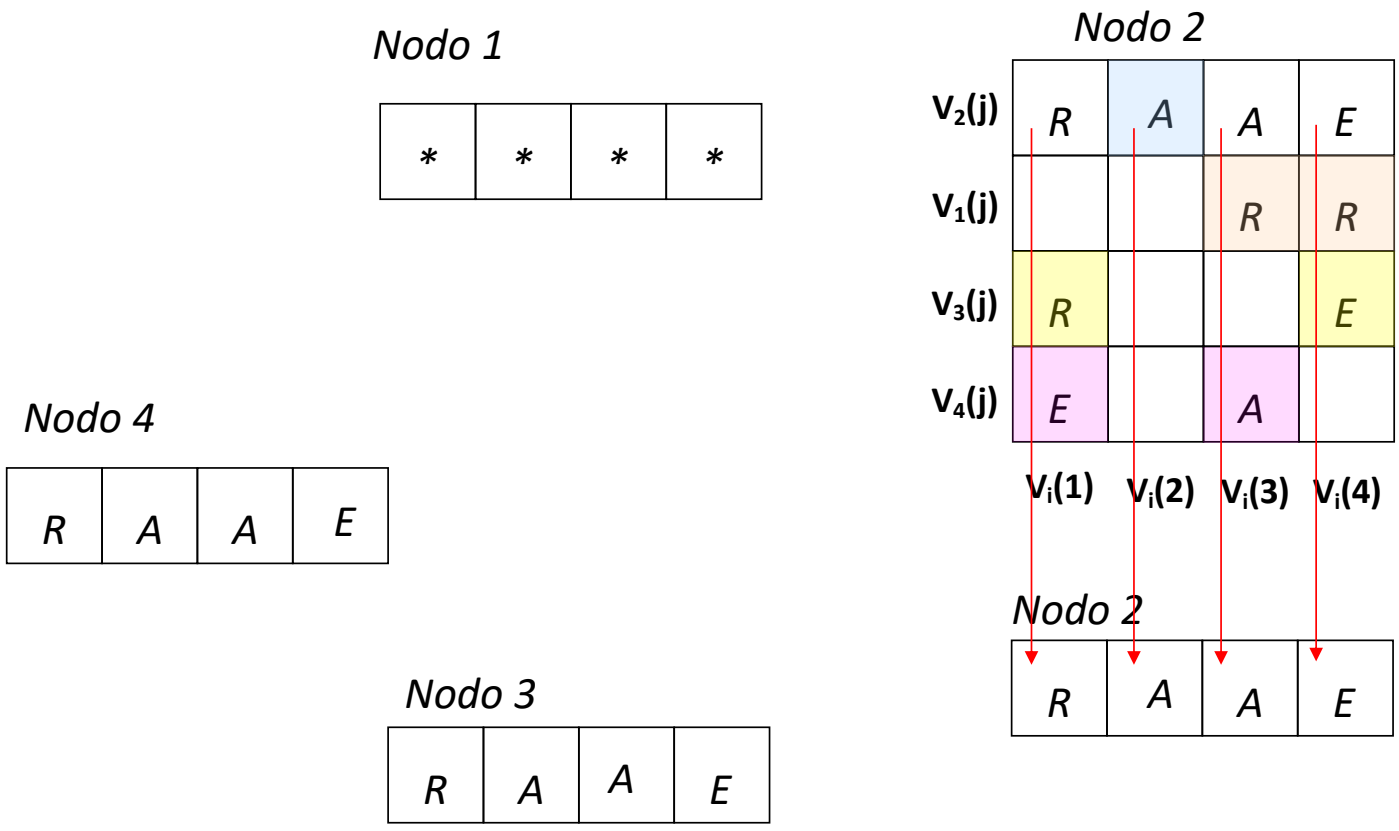
Ejemplo. Primer intercambio de mensajes



Ejemplo. Segundo intercambio de mensajes



Etapa final


Nodo 1
* * * *
Nodo 2
V2(j)
V1(j)
V3(j)
V4(j)
R A A E
R R
R E
E A
Vi(1) Vi(2) Vi(3) Vi(4)
Nodo 2
R A A E
Nodo 4
R A A E
Nodo 3
R A A E