Tema 5
Servicios distribuidos

Sistemas Distribuidos
Grado en Ingeniería Informática
Universidad Carlos III de Madrid

Contenido

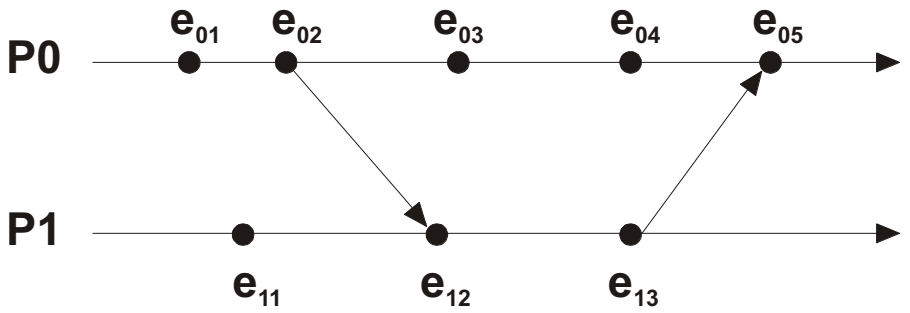
- Sincronización en sistemas distribuidos
- Relojes físicos y lógicos
- Exclusión mutua distribuida
- Algoritmos de elección
- Comunicación *multicast*
- Problemas de consenso
- Servicio de nombres

Sincronización en sistemas distribuidos

- Más compleja que en los centralizados ya que usan algoritmos distribuidos
- Los algoritmos distribuidos deben tener las siguientes propiedades:
  - La información relevante se distribuye entre varios procesos en computadores distintos
  - Los procesos toman las decisiones sólo en base a la información local
  - Debe evitarse un punto único de fallo
  - No existe un reloj común

Modelo del sistema distribuido

- Procesos secuenciales $\{P_1, P_2, ...P_n\}$ y canales de comunicación
- Eventos en P_i
  - $E_i = \{e_{i1}, e_{i2}, ...e_{in}\}$
  - Historia(P_i) = h_i = $<e_{i0}, e_{i1}, e_{i2}, ... >$ $e_{ik} \rightarrow e_{i(k+1)}$
- Tipos de eventos
  - Internos (cambios en el estado de un proceso)
  - Comunicación
    - Envío
    - Recepción
      - $e_{02} \rightarrow e_{12}$
- Diagramas espacio-tiempo



Modelos síncronos y asíncronos

- **Sistemas distribuidos asíncronos**
  - **No hay un reloj común**
  - No hacen ninguna suposición sobre las velocidades relativas de los procesos.
  - Los canales son fiables pero no existe un límite a la entrega de mensajes
  - La comunicación entre procesos es la única forma de sincronización
- **Sistemas síncronos**
  - Hay una perfecta sincronización
  - Hay límites en las latencias de comunicación
  - Los sistemas del mundo real no son síncronos

Entrega de mensajes en Internet

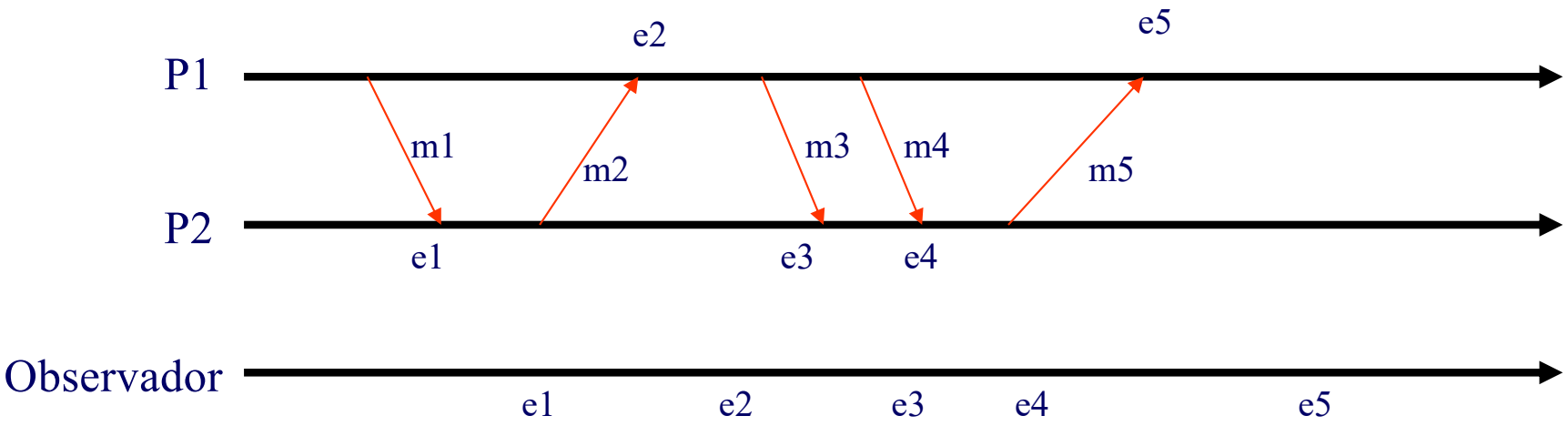
- Internet se basa en una red de conmutación de paquetes, donde los paquetes se pueden perder y las copias de mensajes, colas y retardos en la red hace que los tiempos de comunicación no sean predecibles y no estén acotados
- Internet no usa el concepto de redes de conmutación de circuitos donde sí es predecible y el ancho de banda es asignado estáticamente
- En Internet el ancho de banda se asigna dinámicamente

Tiempo en sistemas distribuidos

- Dificultades en el diseño de aplicaciones distribuidas
  - Paralelismo entre los procesadores
  - Velocidades arbitrarias de procesadores
  - No determinismo en el retardo de los mensajes. Fallos
  - Ausencia de tiempo global

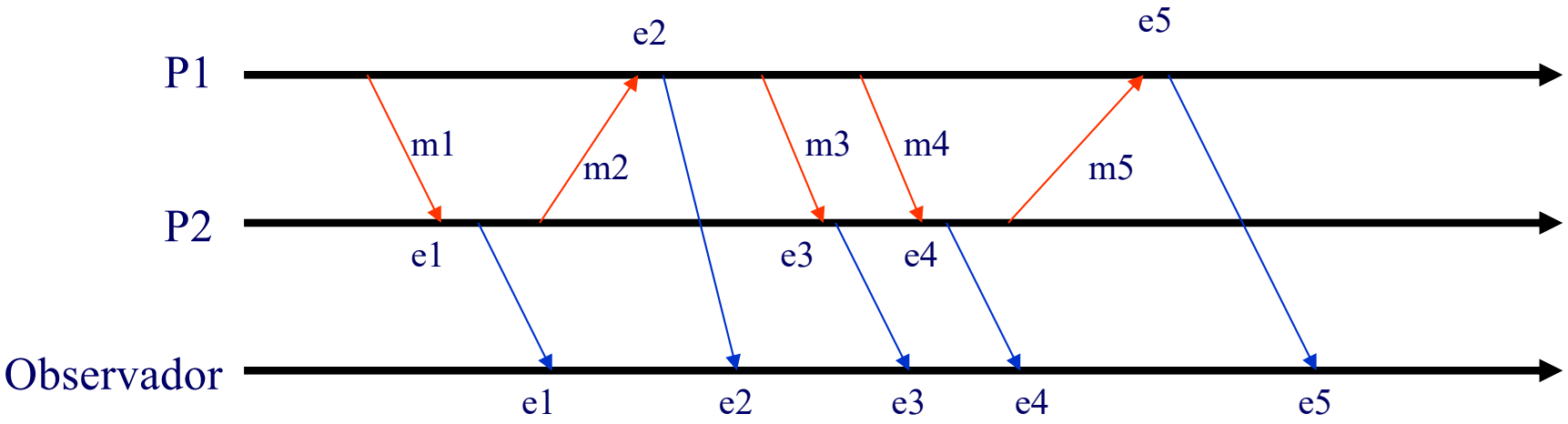
Ejemplo 1: observación de eventos

- 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



Ejemplo 1: observación de eventos

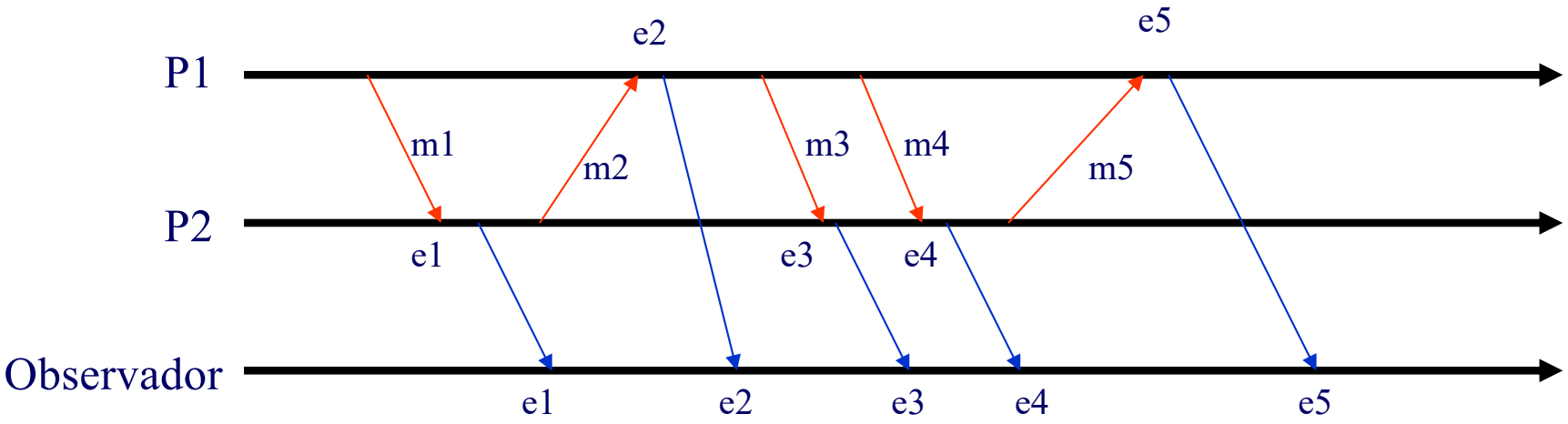
- Monitorización del comportamiento de una aplicación distribuida
  - El observador debe ordenar los eventos de recepción de mensajes en los procesos P1 y P2
    - e1, e2, e3, e4, e5



Cada proceso envía un mensaje al observador cada vez que recibe un mensaje y el observador los registra

Ejemplo 1: observación de eventos

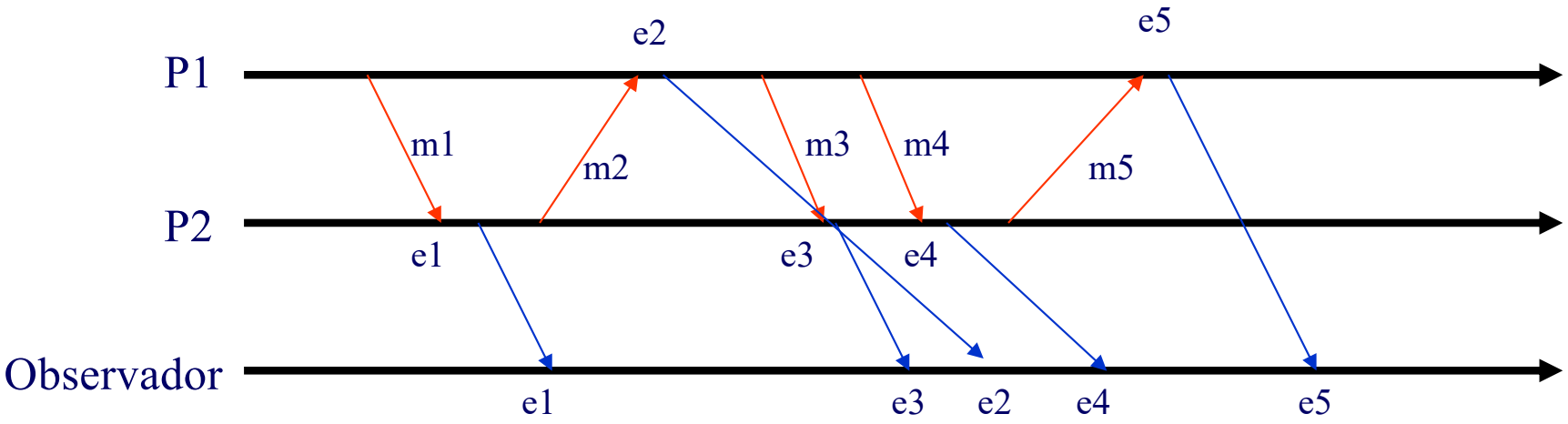
- Monitorización del comportamiento de una aplicación distribuida
  - El observador debe ordenar los eventos de recepción de mensajes en los procesos P1 y P2
    - e1, e2, e3, e4, e5



¿es correcto?

Ejemplo 1: observación de eventos

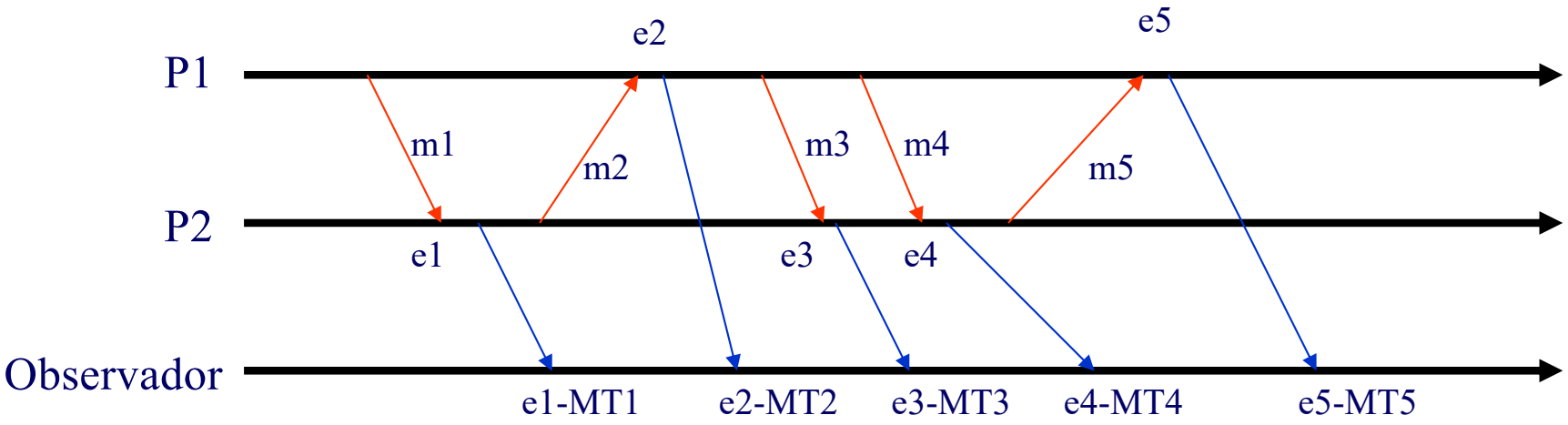
- Monitorización del comportamiento de una aplicación distribuida
  - El observador debe ordenar los eventos de recepción de mensajes en los procesos P1 y P2
    - e1, e2, e3, e4, e5



No: los mensajes pueden llegar desordenados

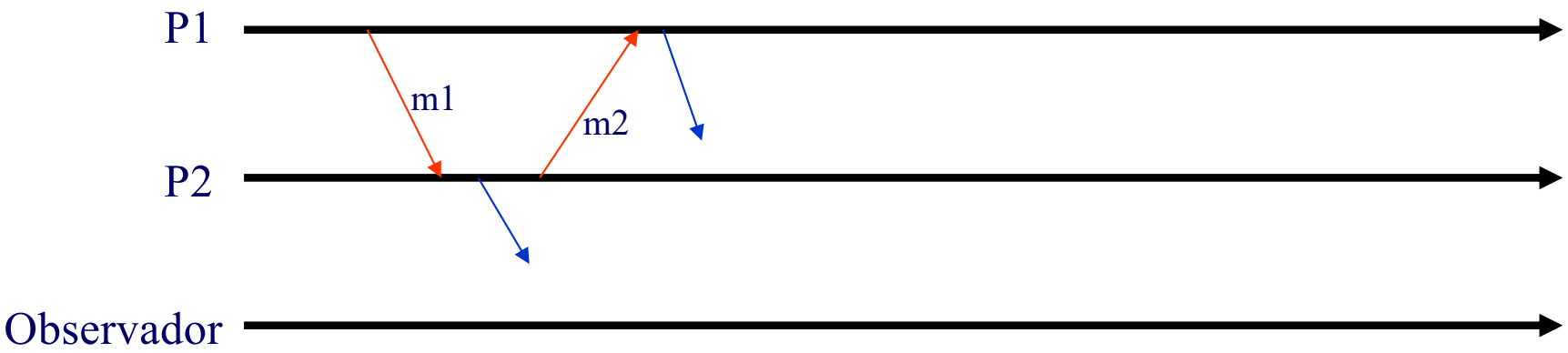
Ejemplo 1: observación de eventos

- 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



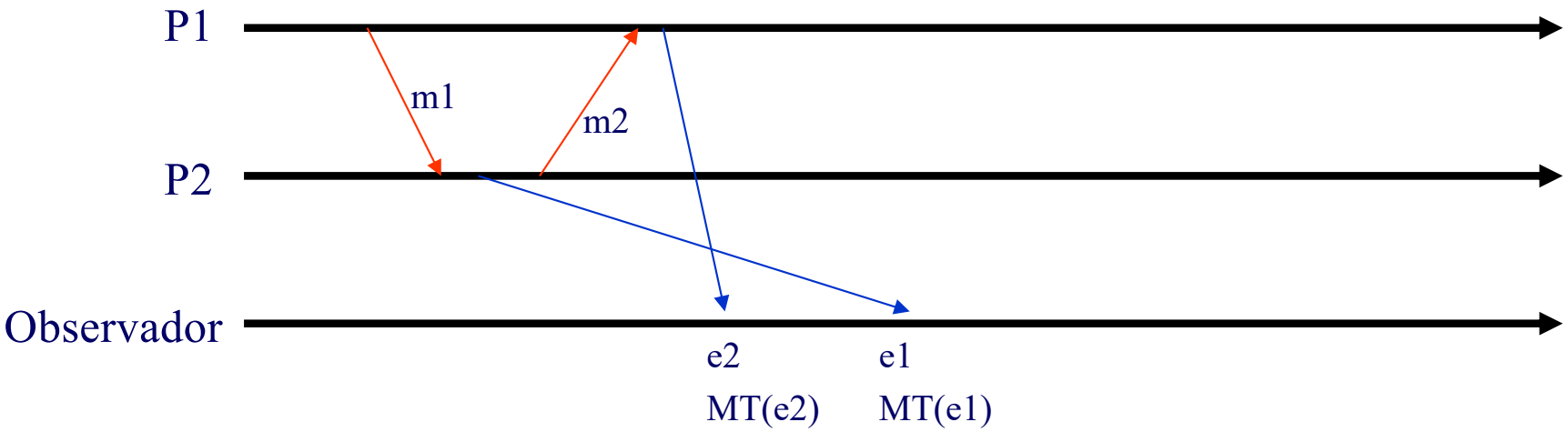
- Para ordenar eventos podemos asignarles marcas de tiempo
  - ei → ek ⇔ MT(ei) < MT (ek)

Ejemplo 1: observación de eventos ¿Marcas de tiempo en el observador?



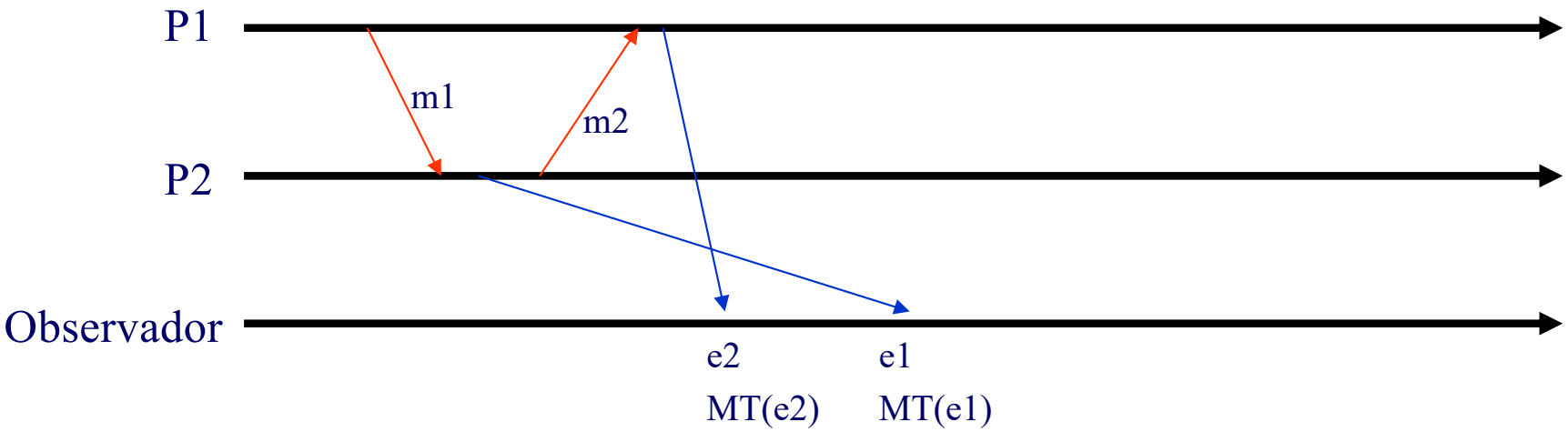
Cada proceso envía una notificación al observador y el observador le asigna una marca de tiempo

Ejemplo 1: observación de eventos ¿Marcas de tiempo en el observador?



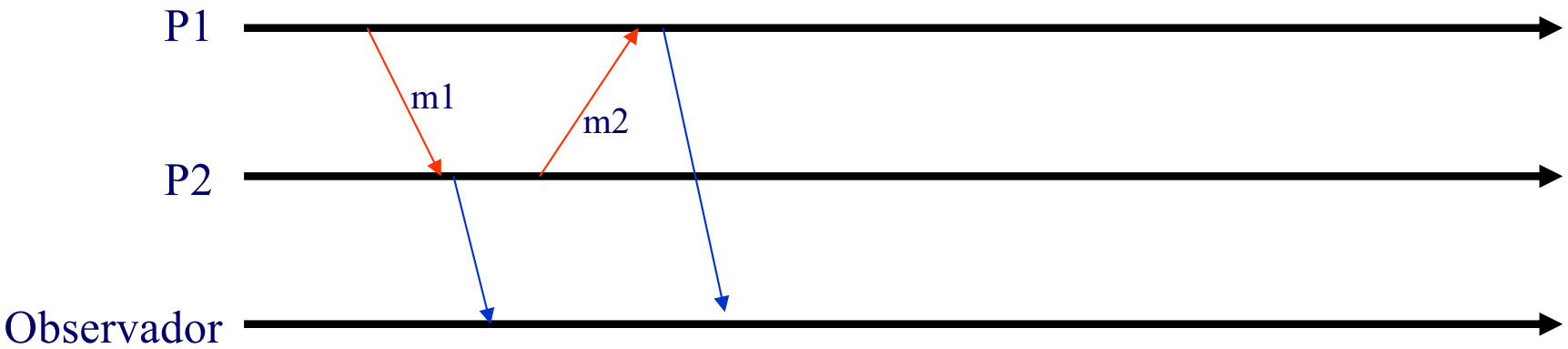
?

Ejemplo 1: observación de eventos ¿Marcas de tiempo en el observador?



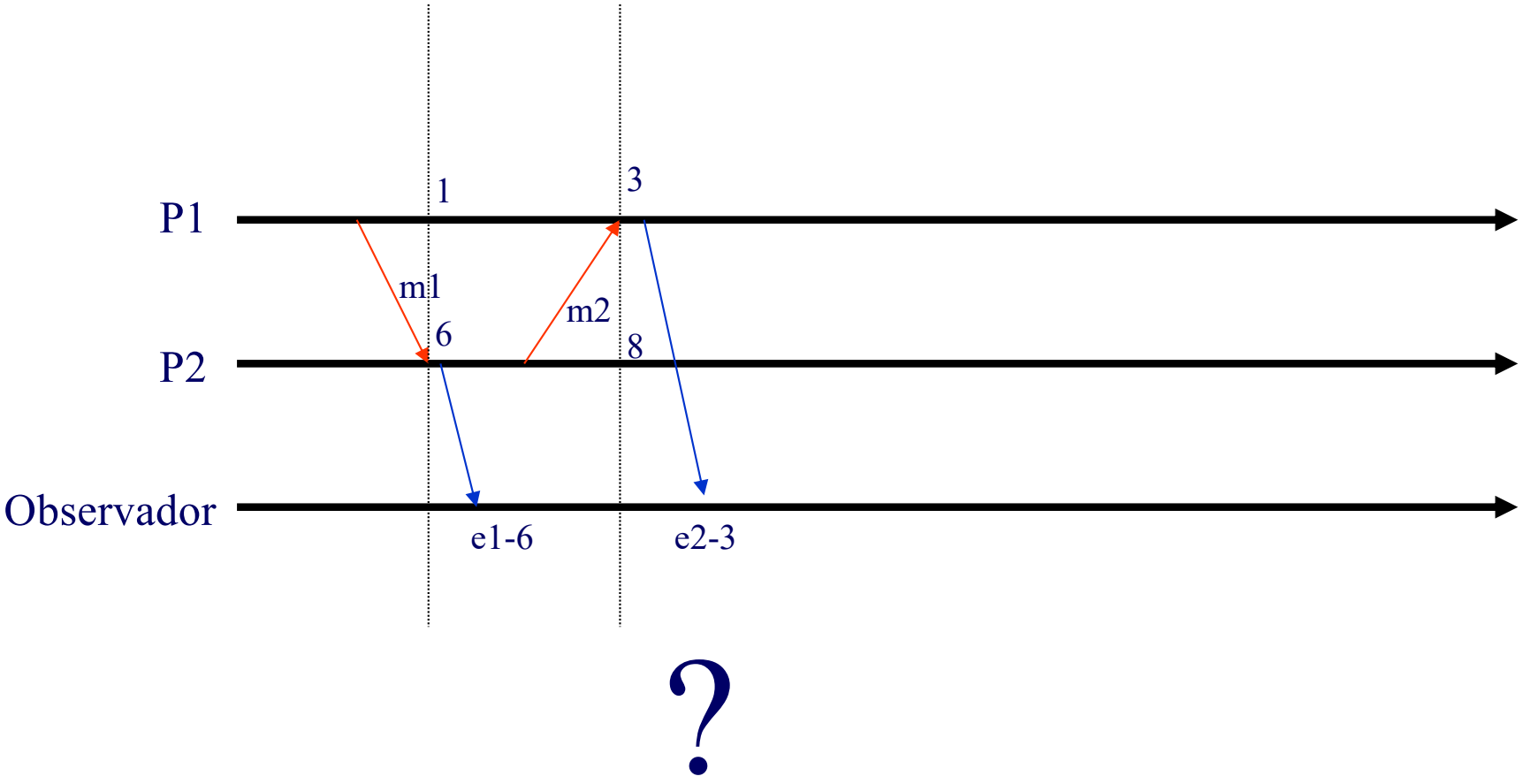
MT(e2) < MT (e1) pero e1 → e2

Ejemplo 1: observación de eventos ¿Marcas de tiempo en los procesos?

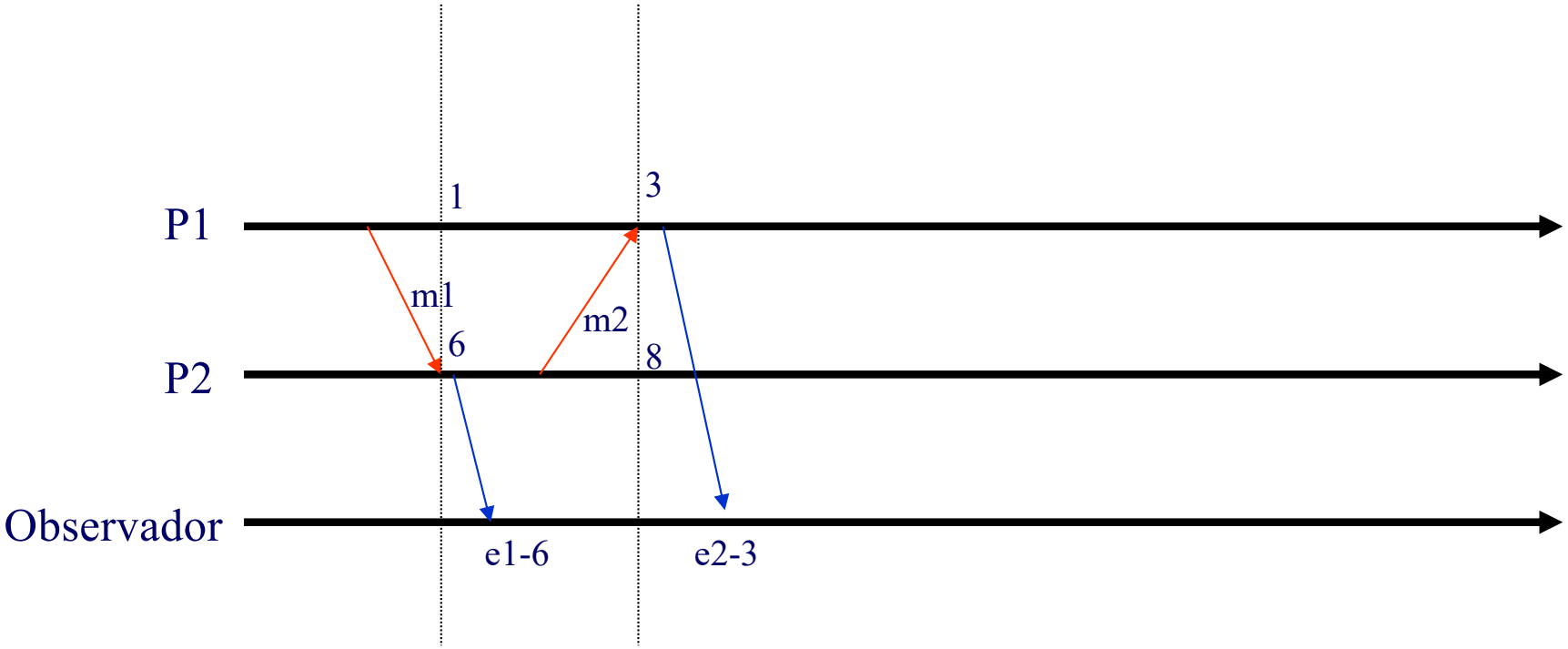


?

Ejemplo 1: observación de eventos ¿Marcas de tiempo en los procesos?

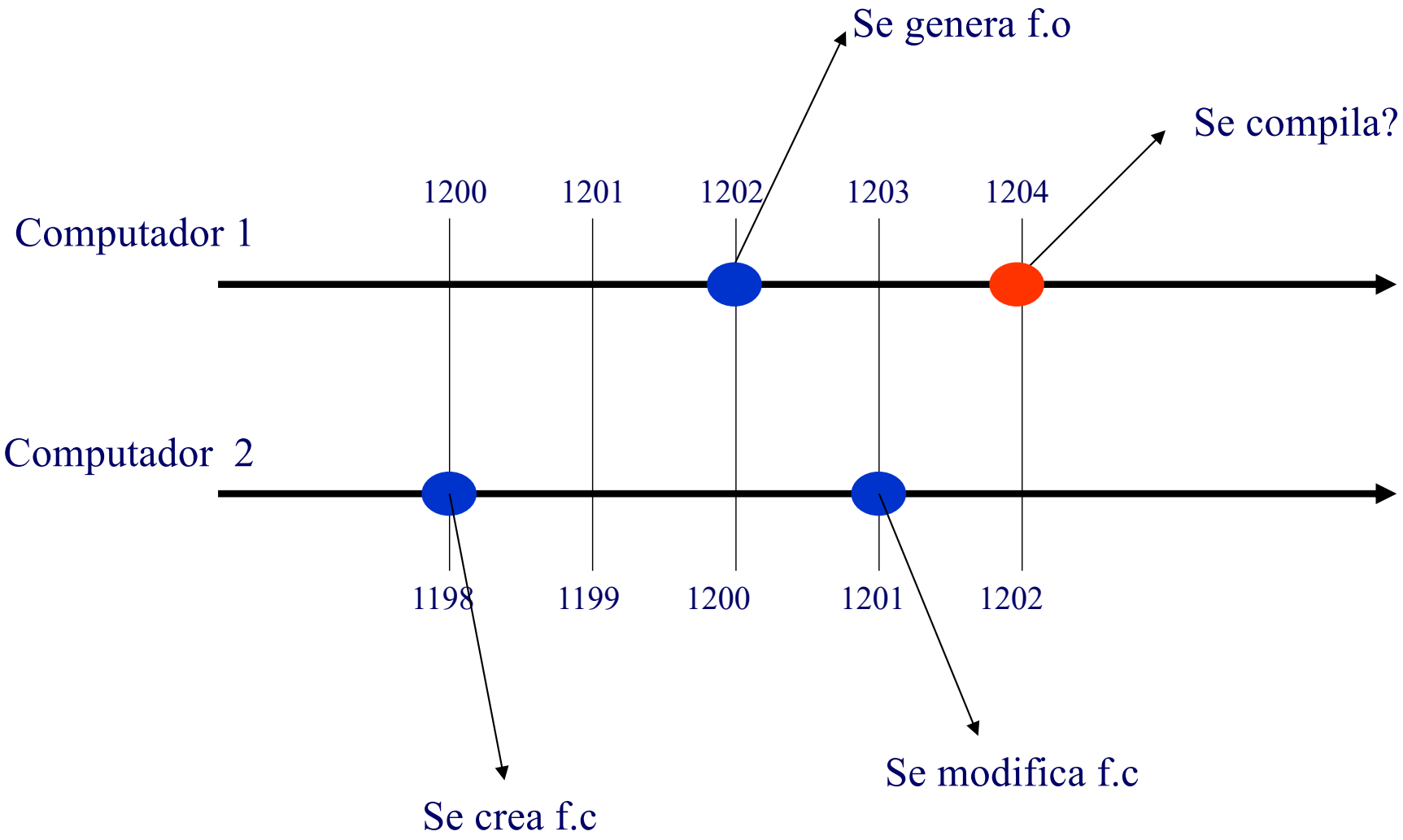


Ejemplo 1: observación de eventos ¿Marcas de tiempo en los procesos?

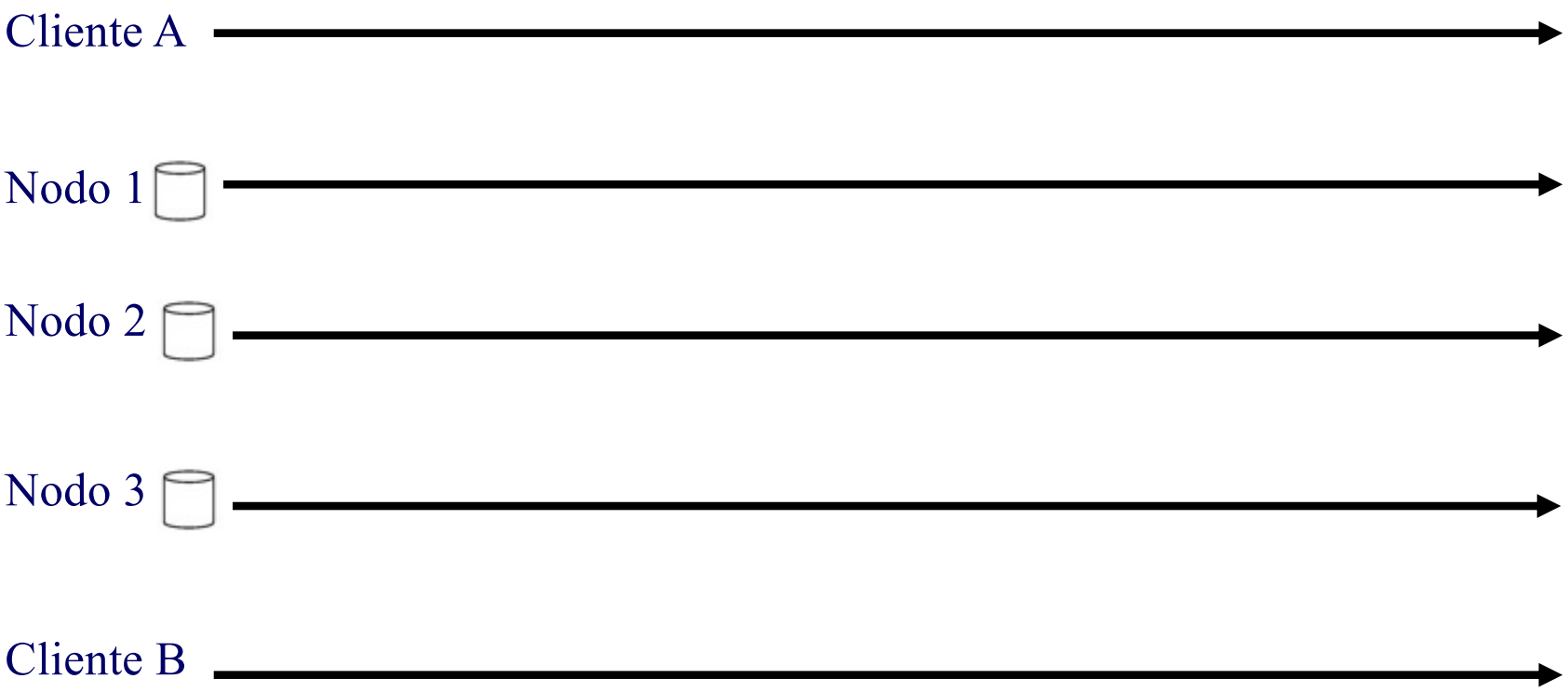


Los relojes en los procesos deben estar sincronizados

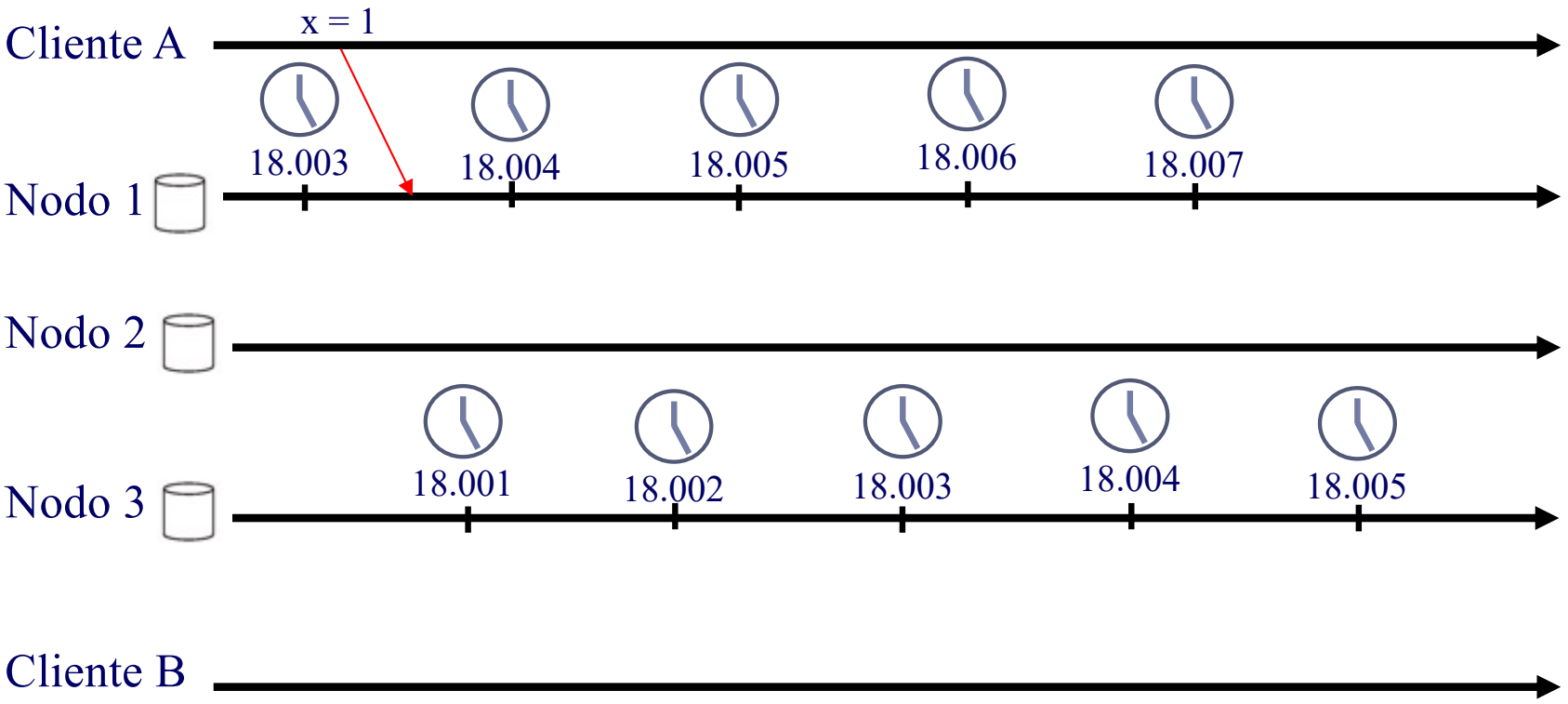
Ejemplo 2: make



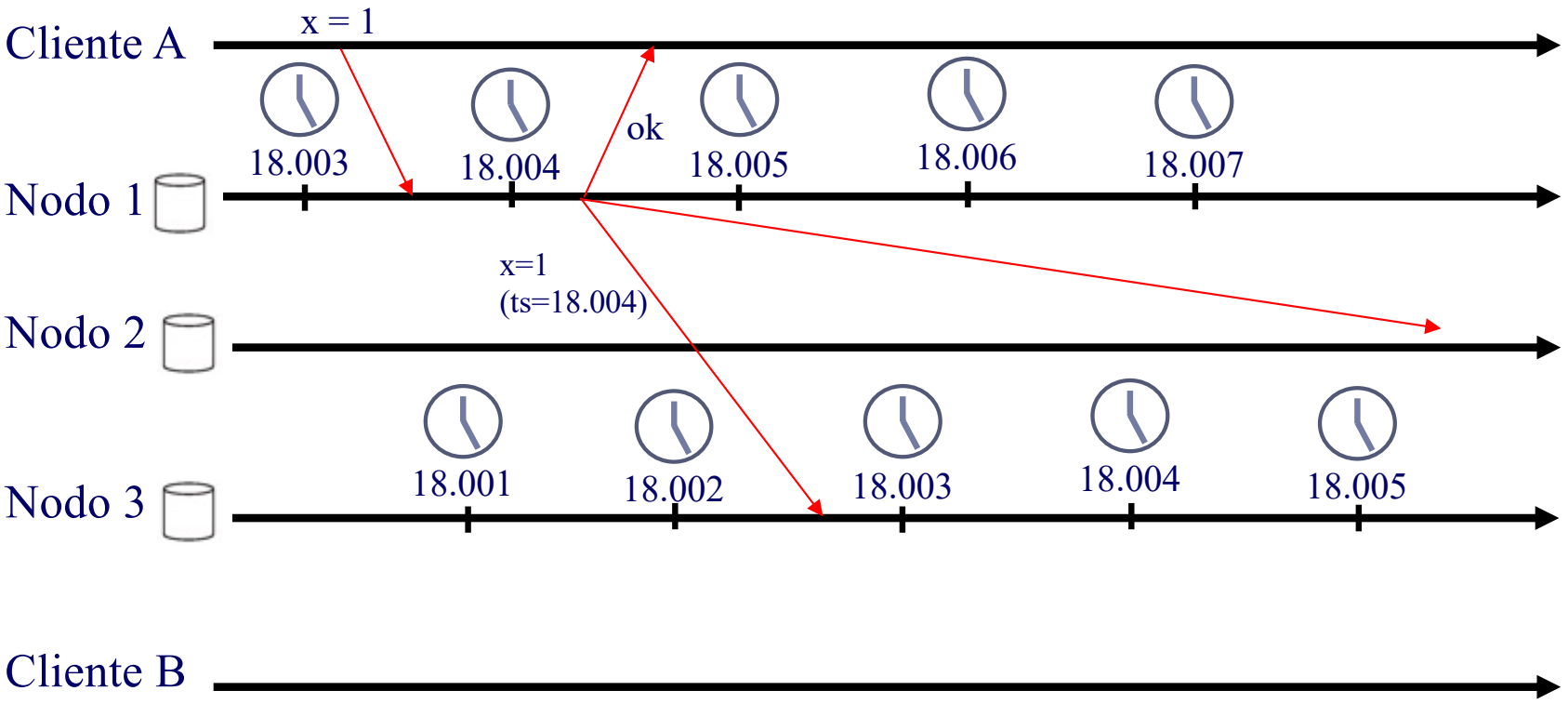
Ejemplo 3: Base de datos replicada



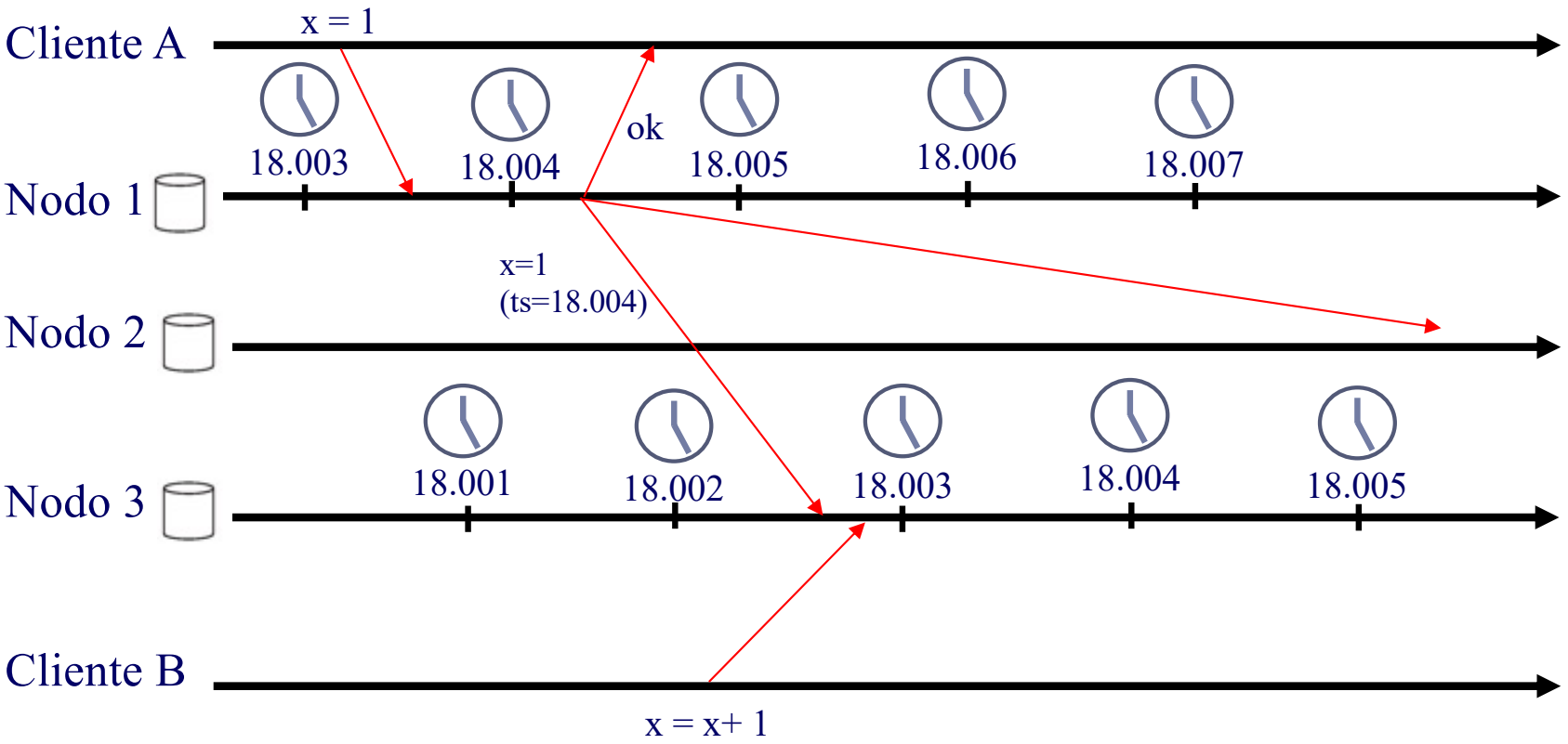
Ejemplo 3: Base de datos replicada



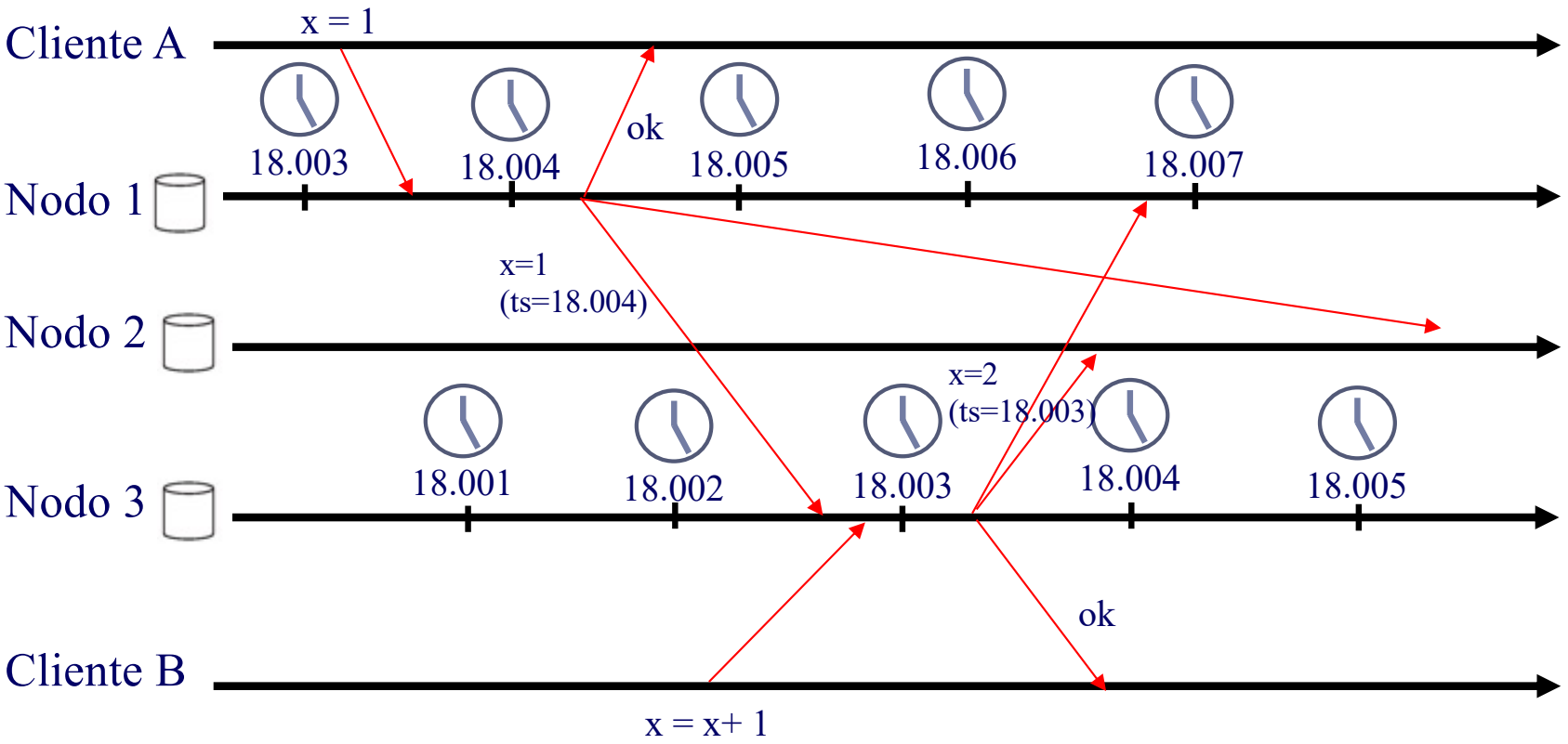
Ejemplo 3: Base de datos replicada



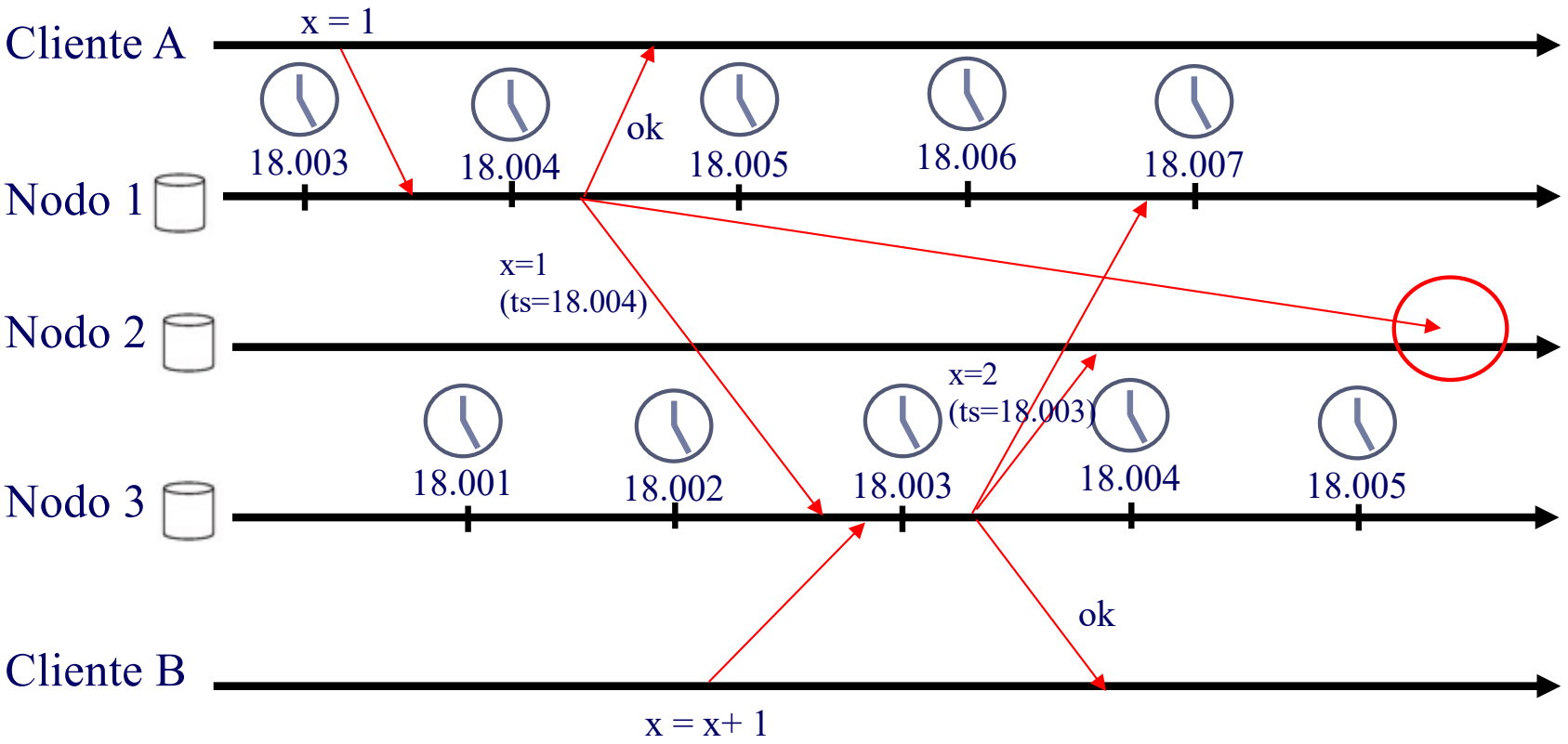
Ejemplo 3: Base de datos replicada



Ejemplo 3: Base de datos replicada



Ejemplo 3: Base de datos replicada



El nodo 2 observará como más reciente el valor de x = 1

Marcas de tiempo (*timestamps*)

- Relojes físicos
- Relojes lógicos

Relojes físicos

- Para ordenar dos eventos de un proceso basta con asignarles una **marca de tiempo**
- Para un instante físico **t**
  - $H_i(t)$: valor del reloj HW (oscilador)
  - $C_i(t)$: valor del reloj SW (generado por el SO)
    - $C_i(t) = a\ H_i(t) + b$
      - Ej: # ms o ns transcurridos desde una fecha de referencia
    - Resolución del reloj: periodo entre actualizaciones de $C_i(t)$
      - Determina la ordenación de eventos
- Dos relojes en dos computadores diferentes dan medidas distintas
  - Necesidad de sincronizar relojes físicos de un sistema distribuido

Tiempo del sistema (Linux)

```
int clock_gettime(clockid_t clk_id,
                  struct timespec *tp);

      struct timespec {
             time_t tv_sec; /* seconds */
             long tv_nsec; /* nanoseconds */
      };
```

- Devuelve el número de segundos transcurridos desde 1 de Enero de 1970 y el número de nanosegundos dentro del actual segundo
- Valores de clk_id:
  - CLOCK_REALTIME: Reloj del sistema. Este reloj puede sufrir ajustes para corregir la fecha.
  - CLOCK_MONOTONIC: Igual que CLOCK_REALTIME pero no se realizan ajustes, por tanto su cuenta es creciente sin saltos bruscos. Útil para medir duraciones de eventos

Medición de tiempos

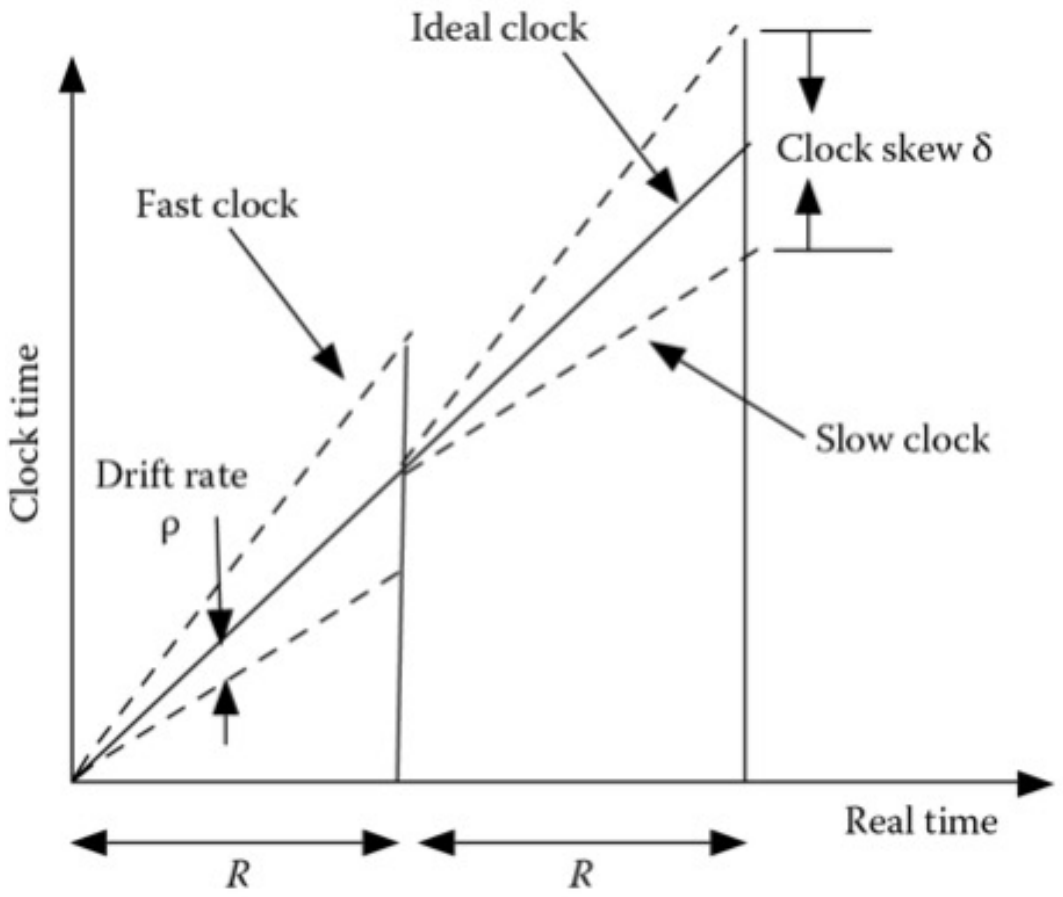
```
clock_gettime(CLOCK_MONOTONIC , &T_inicio);
Acción a medir
clock_gettime(CLOCK_MONOTONIC , &T_fin);

Acu1 = T_fin.tv_sec – T_inicio.tv_sec;
Acu2 = (T_fin.tv_nsec – T_inicio.tv_nsec) / (double)1000000000;
Tiempo = Acu1 + Acu2;
Printf("Tiempo en segndos = %lf\n, Tiempo);
```

Sincronización de relojes físicos

- Los computadores de un sistema distribuido poseen relojes que **no** están sincronizados (**derivas**)
- Importante asegurar una correcta sincronización
  - En aplicaciones de tiempo real
  - Ordenación natural de eventos distribuidos (fechas de ficheros)
  - Análisis de rendimiento
- Tradicionalmente se han empleado protocolos de sincronización que intercambian mensajes
- Actualmente se puede mejorar mediante GPS
  - Los computadores de un sistema poseen todos un GPS
  - Uno o dos computadores utilizan un GPS y el resto se sincroniza mediante protocolos clásicos

Sincronización de relojes físicos



Sincronización de relojes físicos

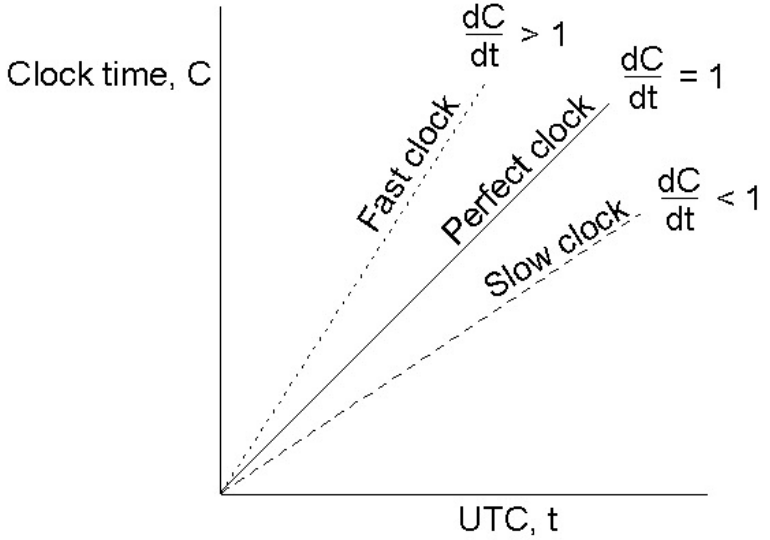
- **D**: Cota máxima de sincronización
- **S**: fuente del tiempo UTC, t
- **Sincronización externa**:
  - Los relojes están sincronizados si $|S(t) - C_i(t)| < D$
  - Los relojes se consideran sincronizados dentro de D
- **Sincronización interna** entre los relojes de los computadores de un sistema distribuido
  - Los relojes están sincronizados si $|C_i(t) - C_j(t)| < D$
  - Dados dos eventos de dos computadores se puede establecer su orden en función de sus relojes si están sincronizados
- Sincronización externa $\not\Leftarrow$ sincronización interna
  $\Rightarrow$

Tiempo universal coordinado (UTC)

- Estándar de tiempo que regula los relojes y el tiempo en el mundo

Métodos de sincronización de relojes

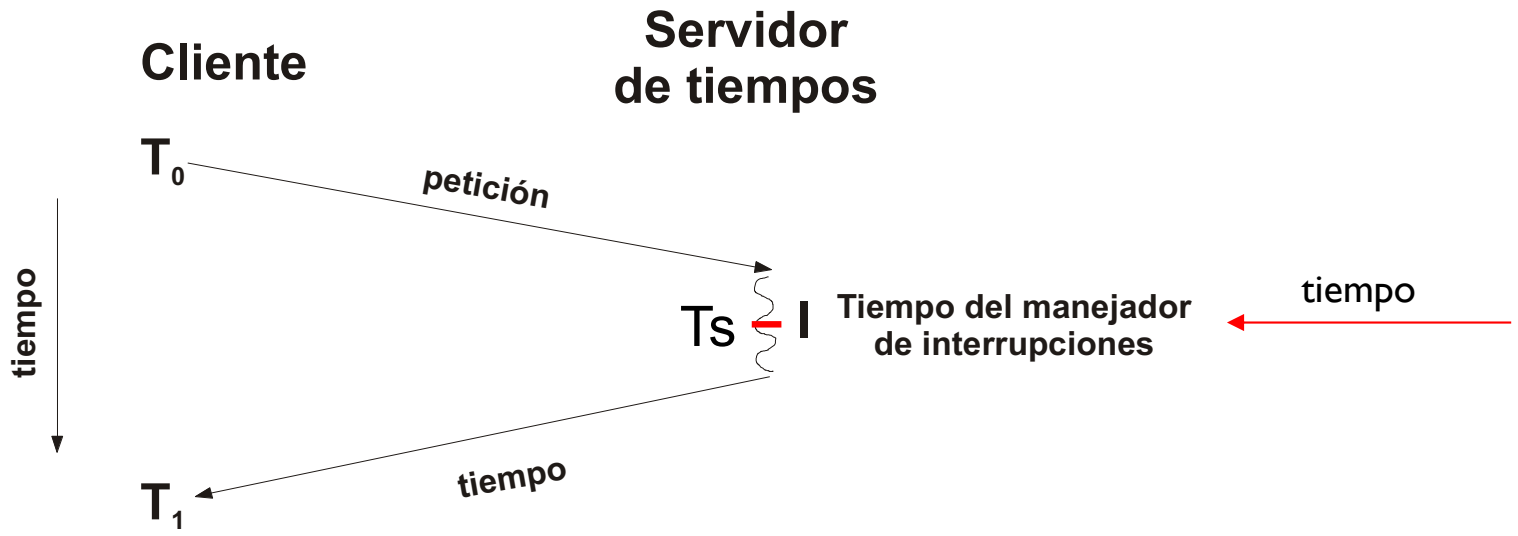
- Sincronización en un sistema síncrono
- **Algoritmo** de Cristian
- **Algoritmo** de Berkeley
- *Network time protocol*



Sincronización en un sistema síncrono

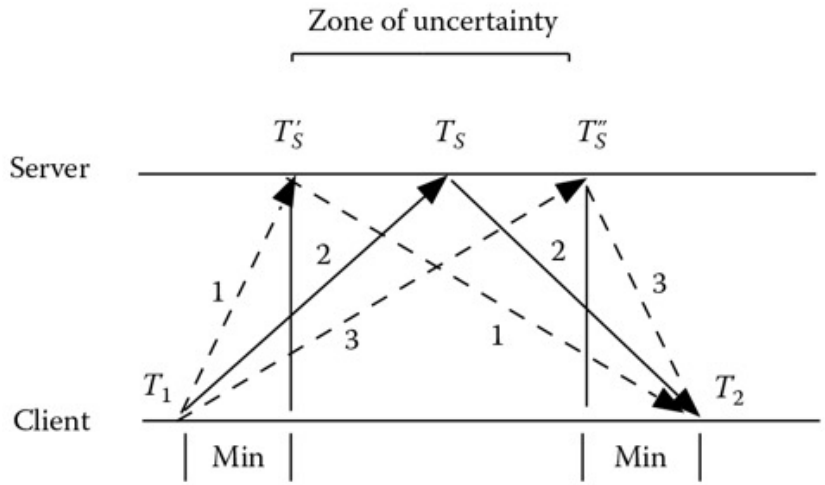
- **P1** envía el valor de su reloj local **t** a **P2**
  - P2 puede actualizar su reloj al valor **t + $T_{transmit}$** si $T_{transmit}$ es el tiempo que lleva enviar un mensaje
  - Sin embargo, $T_{transmit}$ puede desconocerse
    - Se compite por el uso de la red
    - Congestión de la red
- En un **sistema síncrono** se conoce el tiempo mínimo y máximo de transmisión de un mensaje
- u = (max - min)
  - Si P2 fija su reloj al valor t + (max+min)/2, entonces la deriva máxima es $\leq$ u/2
- El problema es que en un sistema asíncrono $T_{transmit}$ no está acotado

Algoritmo de Cristian



- El cliente realiza una petición para obtener el tiempo
- El servidor responde con el tiempo de su reloj (Ts)
- El cliente actualiza su reloj a tiempo Ts + $(T_1-T_0)/2$
- Para mejorar la precisión se pueden hacer varias mediciones y descartar cualquiera en la que T_1-T_0 exceda de un límite
- Precisión del resultado = $\pm\frac{T_1-T_0}{2}$

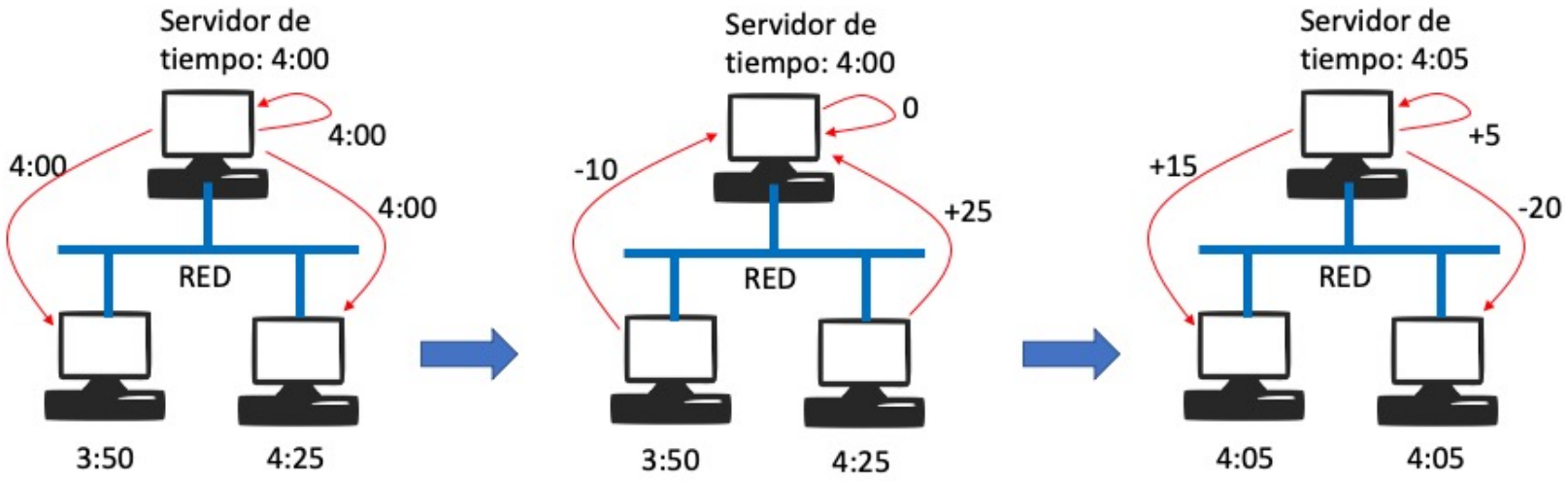
Algoritmo de Cristian. Mejorando la precisión



- Min: tiempo mínimo de transmisión de un mensaje
- El valor que obtiene el servidor Ts se encuentra en el intervalo [T's, T''s] = [T1+Min, T2-Min]
- La precisión del resultado en este caso es $\pm \frac{T_1 - T_0}{2} - T_{min}$

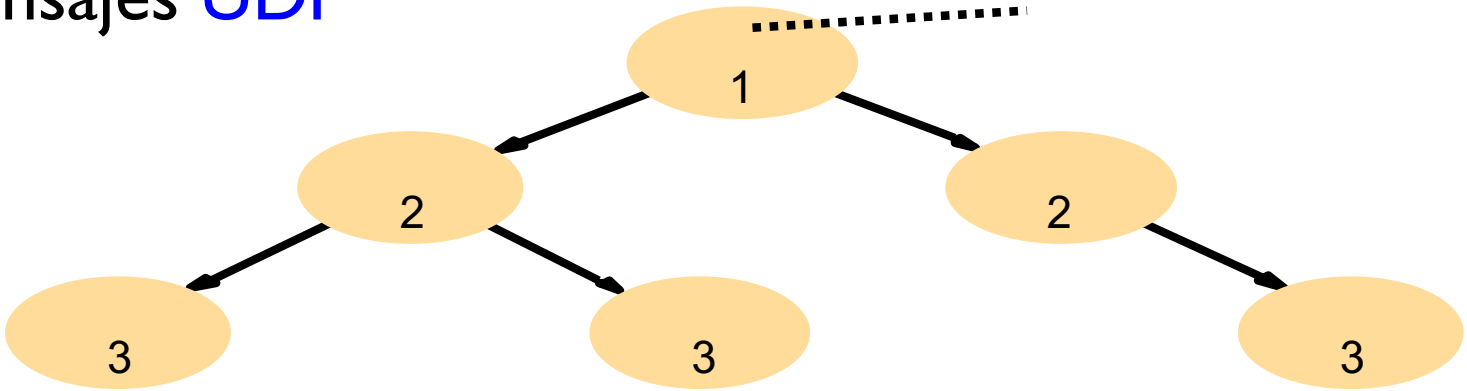
Algoritmo de Berkeley

- El servidor de tiempo realiza un muestreo periódico de todas las máquinas para pedirles el tiempo
- Calcula el tiempo promedio y le indica a todas las máquinas que avancen su reloj a la nueva hora o que disminuyan la velocidad de actualización



Network time protocol (NTP)

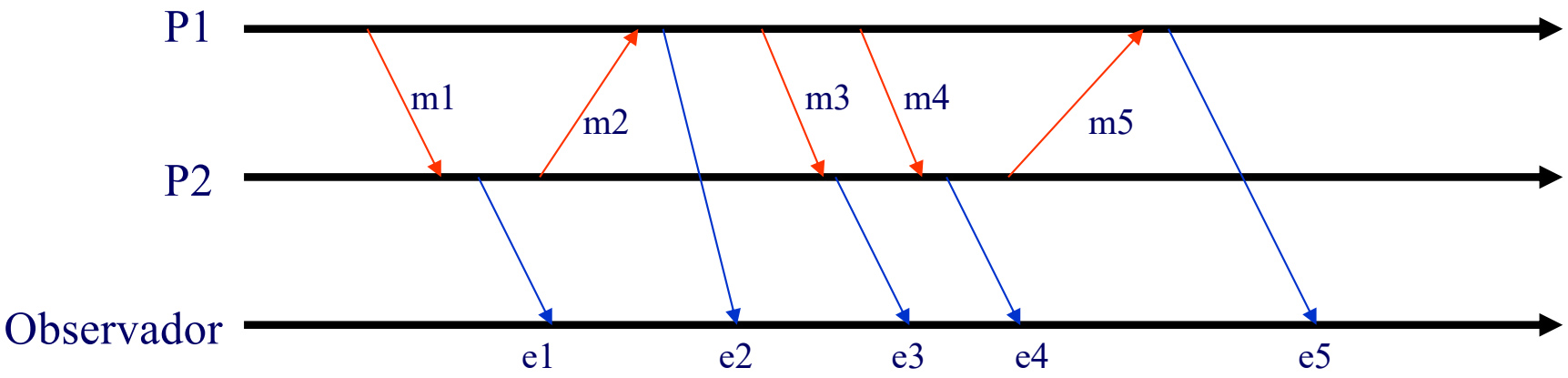
- Servicio para sincronizar a máquinas en Internet con el UTC
- 3 modos de sincronización
  - multicast: para redes LAN de alta velocidad
  - RPC: similar al algoritmo de Cristian
  - simétrico: entre pares de procesos
- Se utilizan servidores localizados a través de Internet con mensajes UDP



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

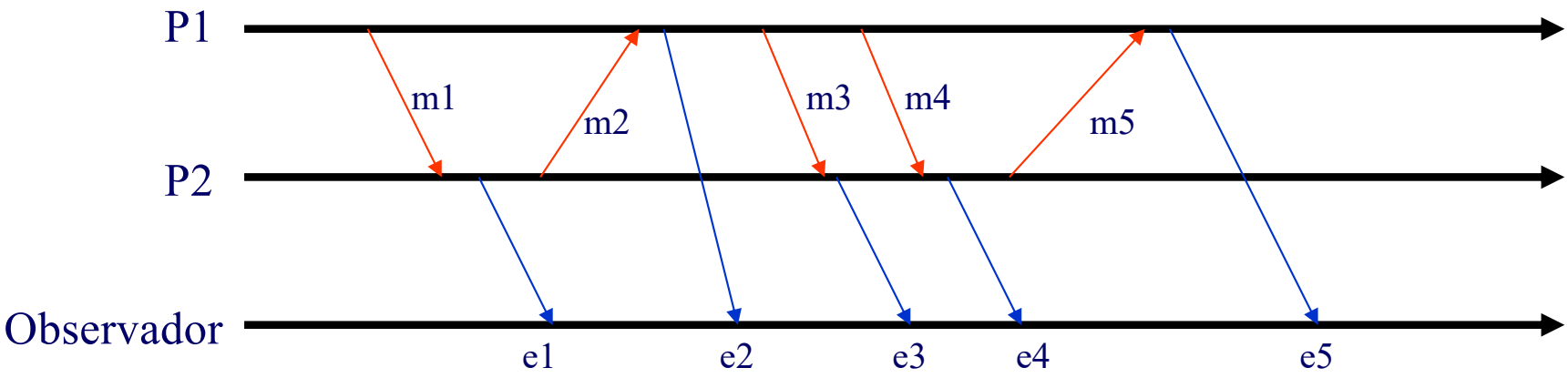
- ¿Podemos ordenar los eventos de otra forma?



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?



Empleando el concepto de reloj lógico

Causalidad potencial

- En ausencia de un reloj global la **relación causa-efecto** es la única posibilidad de ordenar eventos
- Relación de causalidad potencial (**Lamport,1978**) se basa en dos observaciones:
  1. Si dos eventos ocurren en el mismo proceso (pi(i=1..N)), entonces ocurrieron en el mismo orden en que se observaron
  2. Si un proceso hace send(m) y otro receive(m), entonces send se produjo antes que el evento receive
- Entonces, Lamport define la relación de causalidad potencial
  - **Precede a** (→)entre cualquier par de eventos del SD
    - Ej: a→b
- **Orden parcial**: reflexiva, anti-simétrica y transitiva
  - Dos eventos son concurrentes (a || b) si no se puede deducir entre ellos una relación de causalidad potencial

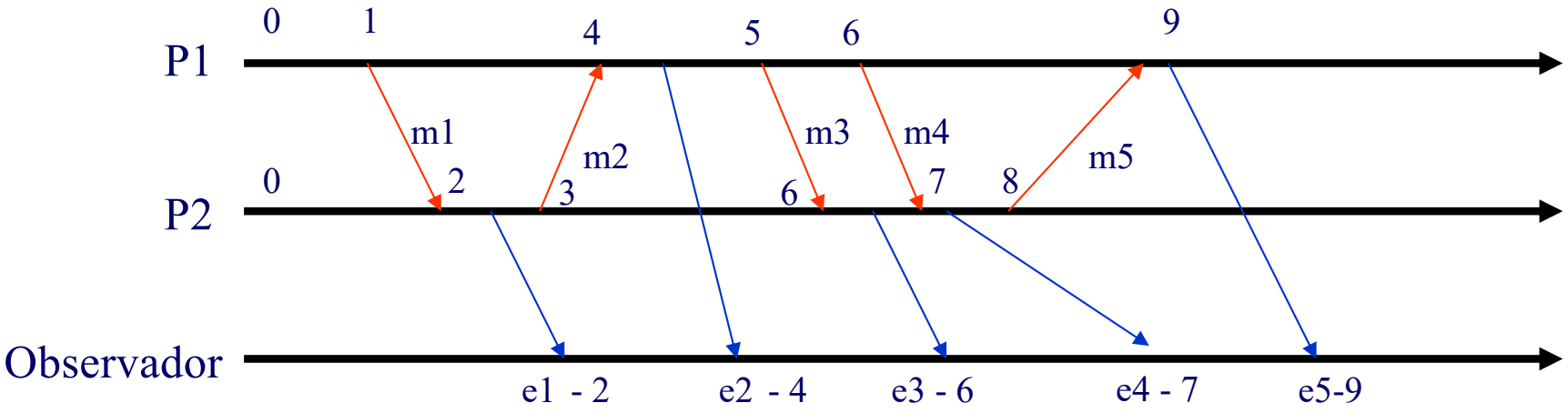
Importancia de la causalidad potencial

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

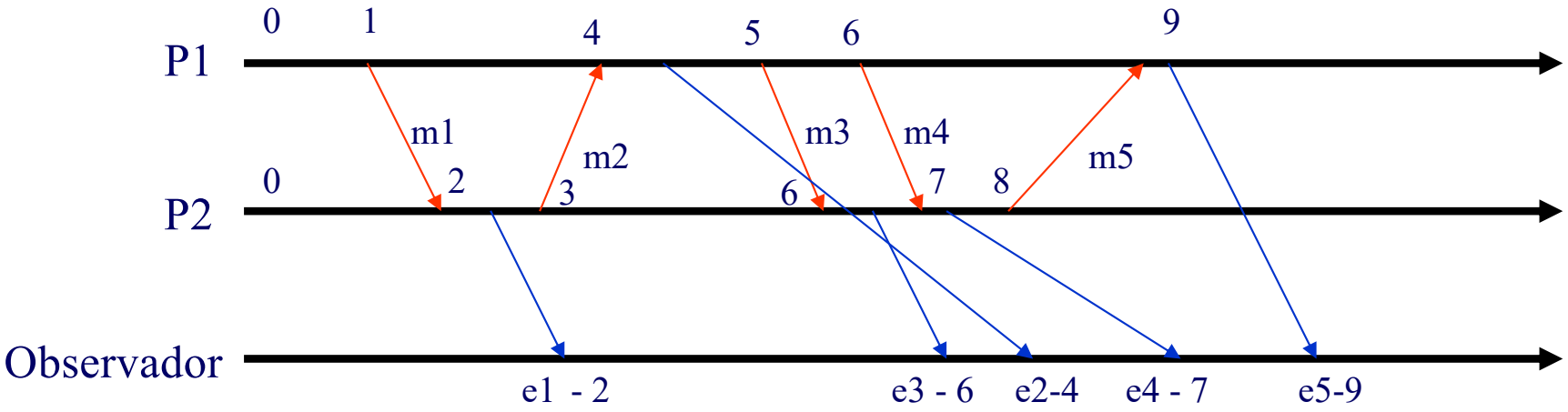
Relojes lógicos (algoritmo de Lamport)

- Útiles para **ordenar eventos** en ausencia de un reloj común
- **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 → b entonces RL(a) < RL(b)**
  - ***Lo contrario no se puede demostrar***

Ejemplo


P1
P2
Observador
0
1
4
5
6
9
0
2
3
6
7
8
m1
m2
m3
m4
m5
e1 - 2
e2 - 4
e3 - 6
e4 - 7
e5-9

Ejemplo



Aunque los mensajes lleguen desordenados al observador, la marca lógica de tiempo permite ordenar los eventos.

e1 - 2 e2 - 4 e3 - 6 e4 - 7 e5-9

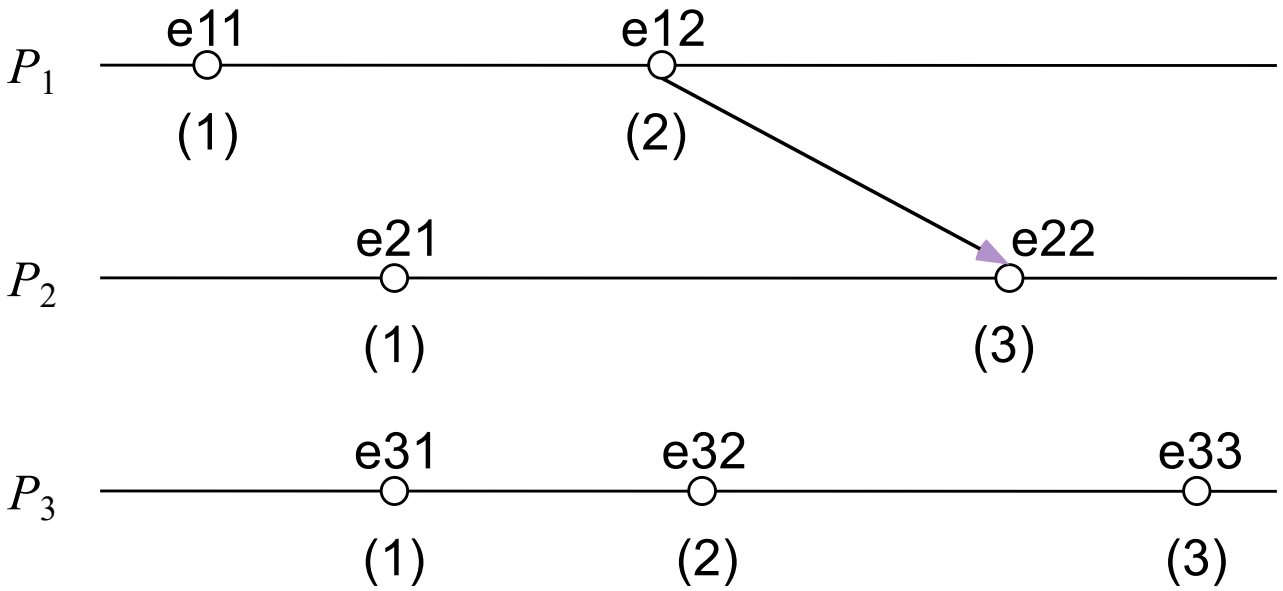
Relojes lógicos totalmente ordenados

- 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 **identificador 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:
  - $a \rightarrow b$ 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



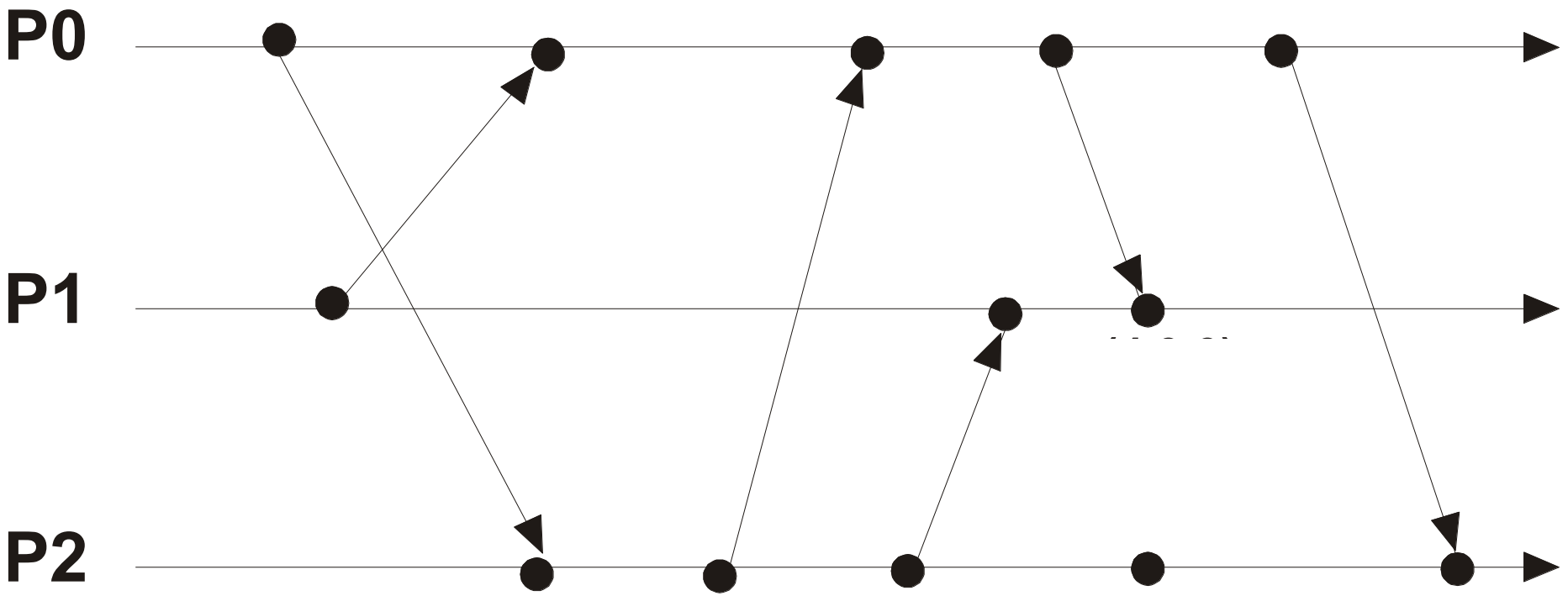
C(e11) < C(e22), y e11→e22 es cierto

C(e11) < C(e32), pero e11→ e32 es falso (son concurrentes)

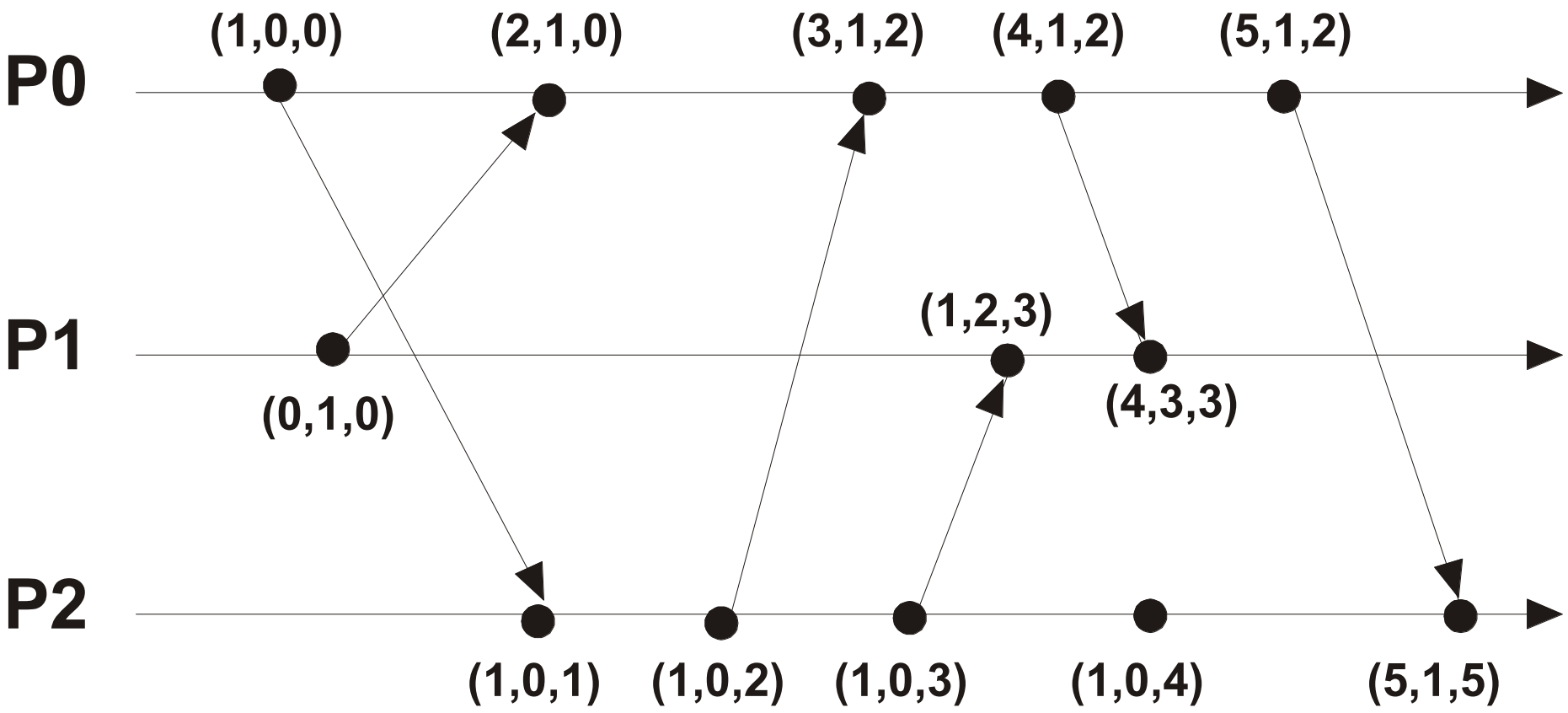
Relojes vectoriales

- 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 RVi
    - $RV_j = max(RV_j, RVi)$ (componente a componente)
    - $RV_j[j] = RV_j[j] + 1$ (evento de recepción)

Relojes vectoriales. Ejemplo



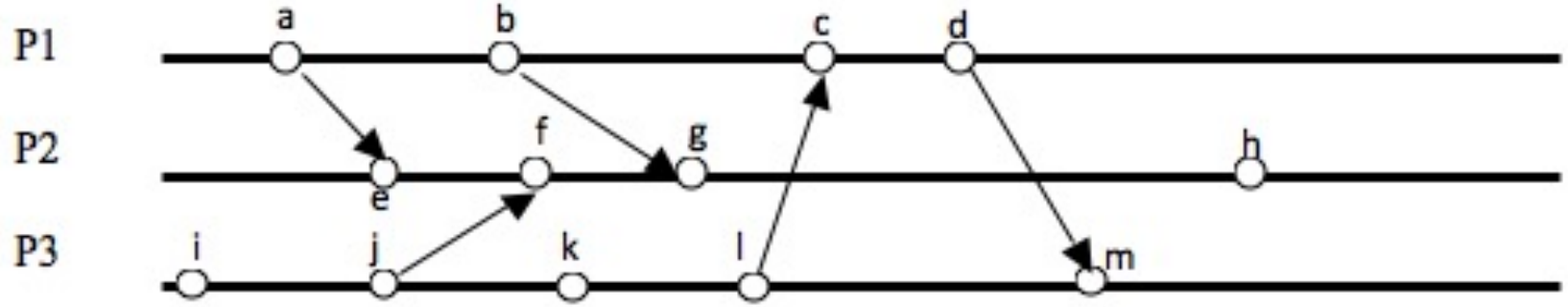
Relojes vectoriales. Ejemplo



Propiedades de los relojes vectoriales

- RV < RV´ si y solo si
  - RV ≠ RV´ y
  - RV[i] ≤ RV´[i], ∀ i
- Dados dos eventos a y b
  - a→ b si y solo si RV(a) < RV(b)
  - a y b son concurrentes cuando
    - Ni RV(a) ≤ RV(b) ni RV(b) ≤ RV(a)

Ejemplo



Exclusión mutua distribuida

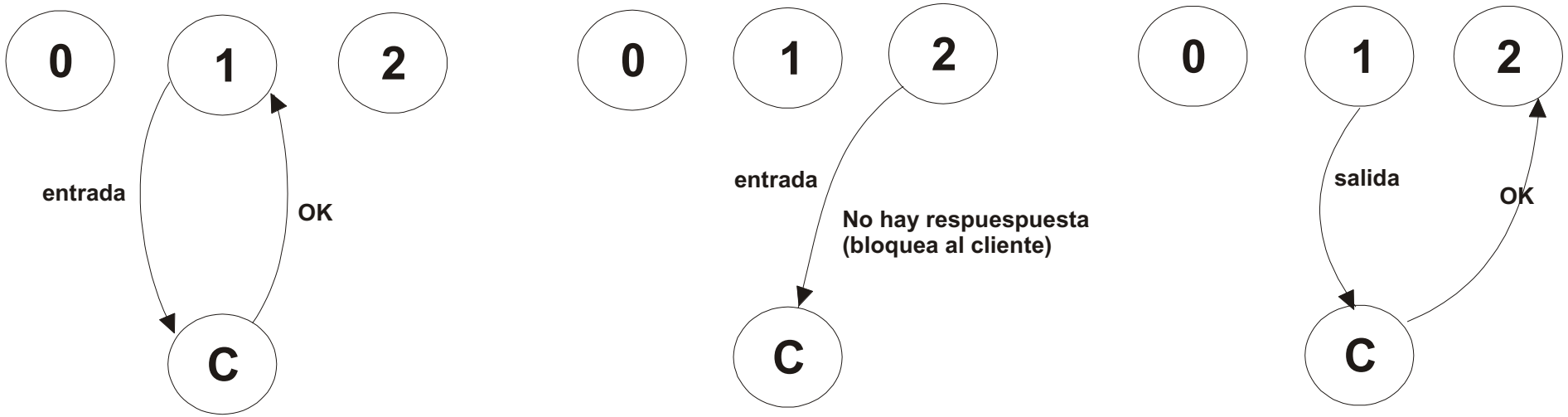
- Los procesos ejecutan el siguiente fragmento de código

```
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 basado en quorum

Algoritmo centralizado

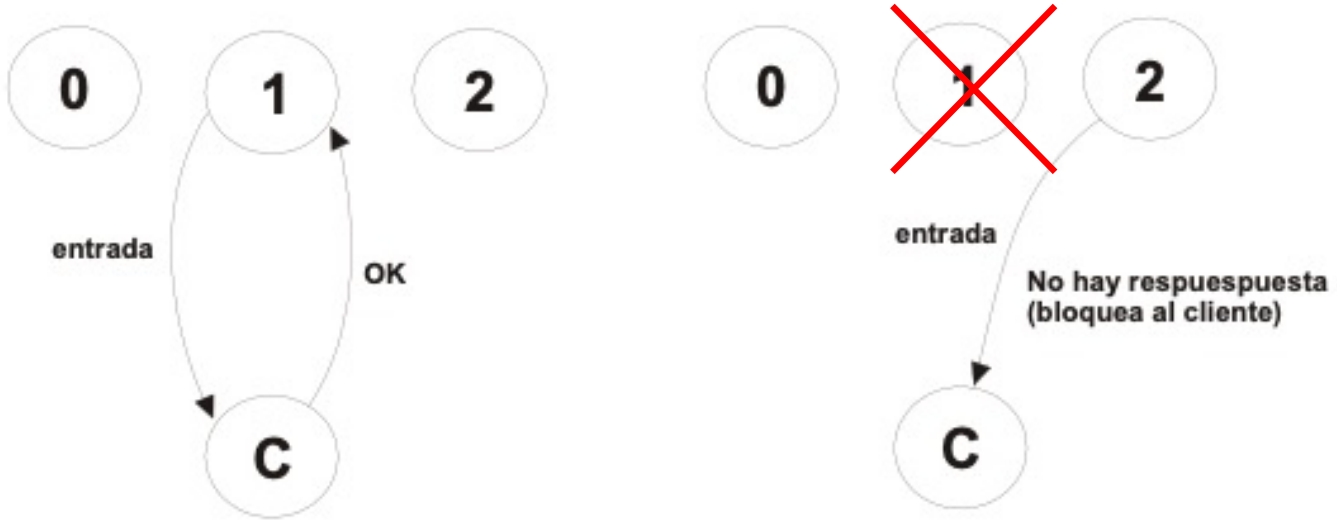
- Existe un proceso coordinador



@Fuente: *Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez.* **Mc Graw Hill**

Problemas del modelo centralizado

- ¿qué ocurre si el proceso que bloqueo el cerrojo falla?



Posible solución

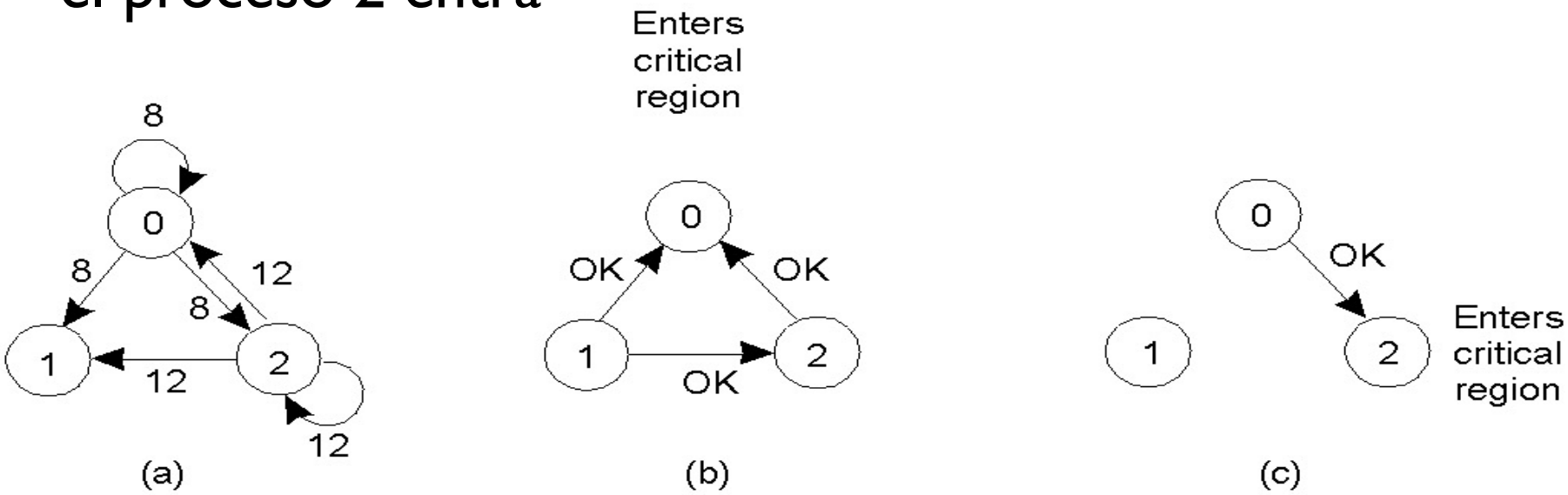
- Uso de temporizadores
- El cerrojo se libera transcurrido un cierto tiempo

Algoritmo distribuido de Ricart y Agrawala

- Algoritmo de **Ricart** y **Agrawala** requiere la existencia de un orden total de todos los mensajes en el sistema
- Un proceso k que quiere entrar en una sección crítica (**SC**) envía un mensaje a todos los procesos (y a él mismo) con una marca de tiempo lógica (MT_K, p_k)
- Cuando un proceso recibe un mensaje
  - Si el receptor no está en la SC ni quiere entrar envía OK al emisor
  - Si el receptor ya está en la SC no responde
  - Si el receptor desea entrar, compara la marca de tiempo del mensaje. Si el mensaje tiene una marca menor envía OK.
  - Un proceso entra en la SC cuando recibe N-1 OK
  - Cuando un proceso sale de la SC envía OK al que no haya contestado previamente

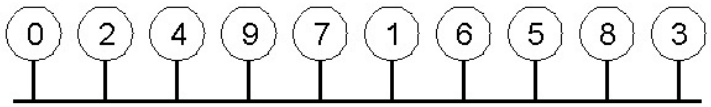
Ejemplo de algoritmo distribuido

a) Dos procesos (P0, P2) quieren entrar en la región al mismo tiempo
b) El proceso 0 tiene la marca de tiempo más baja, espera 2 OK
c) Cuando el proceso 0 acaba, envía un OK, de esa forma el proceso 2 entra

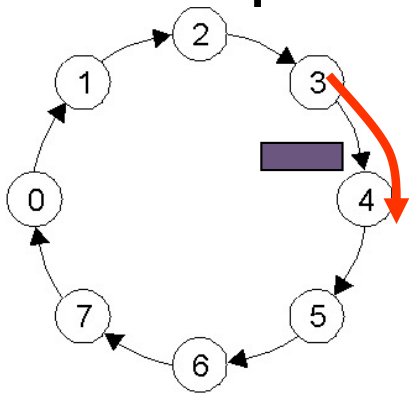


Anillo con testigo

- 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



(a)

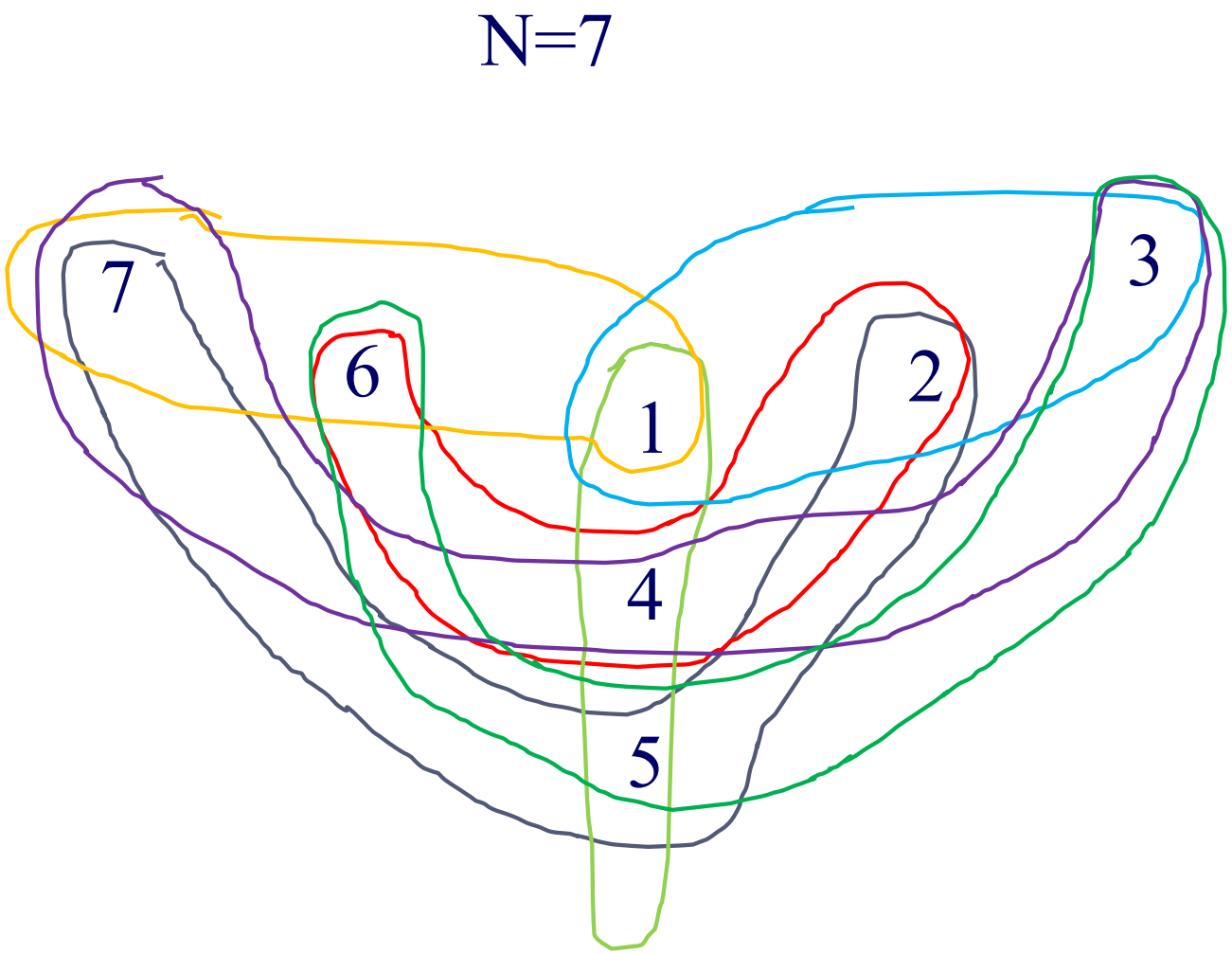


(b)

Ejemplo. Algoritmo de quorum 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 ∩ Vj ≠ ∅
    - Todos subconjuntos de igual tamaño
    - Cada proceso Pj está contenido en M subconjutos votantes
- Solución optima
  - K ~ $\sqrt{N}$
  - M = K

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 quorum 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 quorum 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;
   }
```

Algoritmos de elección

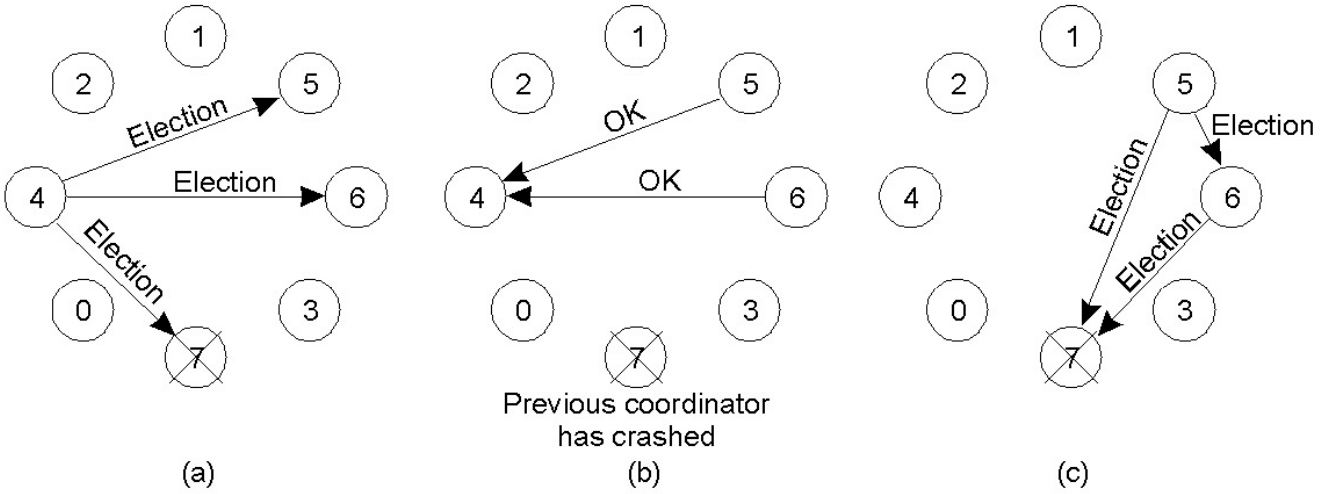
- Útil en aplicaciones donde es necesario la existencia de un coordinador
- El algoritmo debe ejecutarse cuando **falla el coordinador**
- Algoritmos **de elección**
  - Algoritmo del matón
  - Algoritmo basado en anillo
- El objetivo de los algoritmos es que la elección sea única aunque el algoritmo se inicie de forma concurrente en varios procesos

Algoritmo del 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

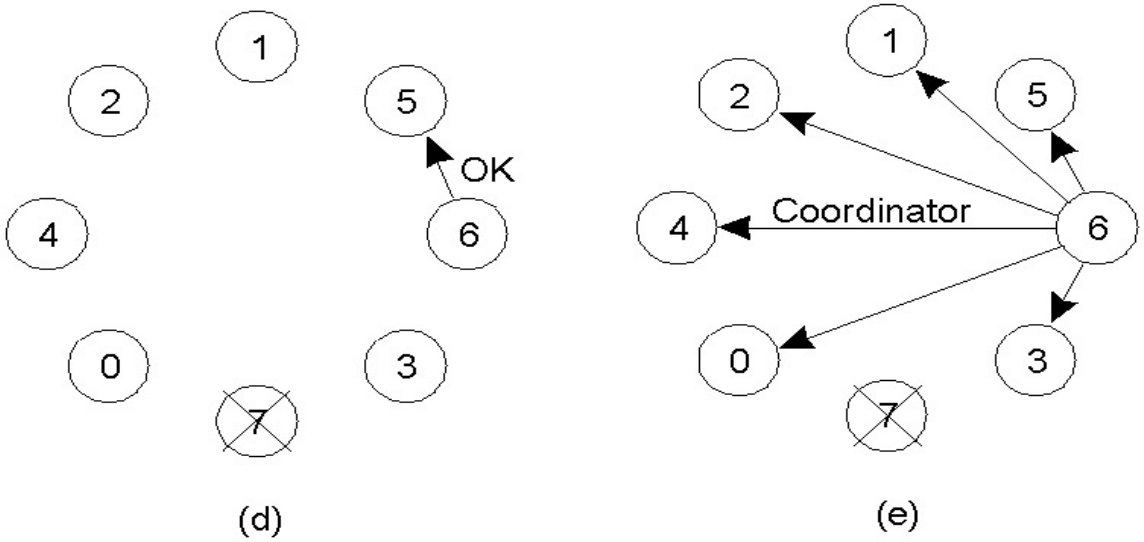
Algoritmo del matón. Ejemplo

- 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



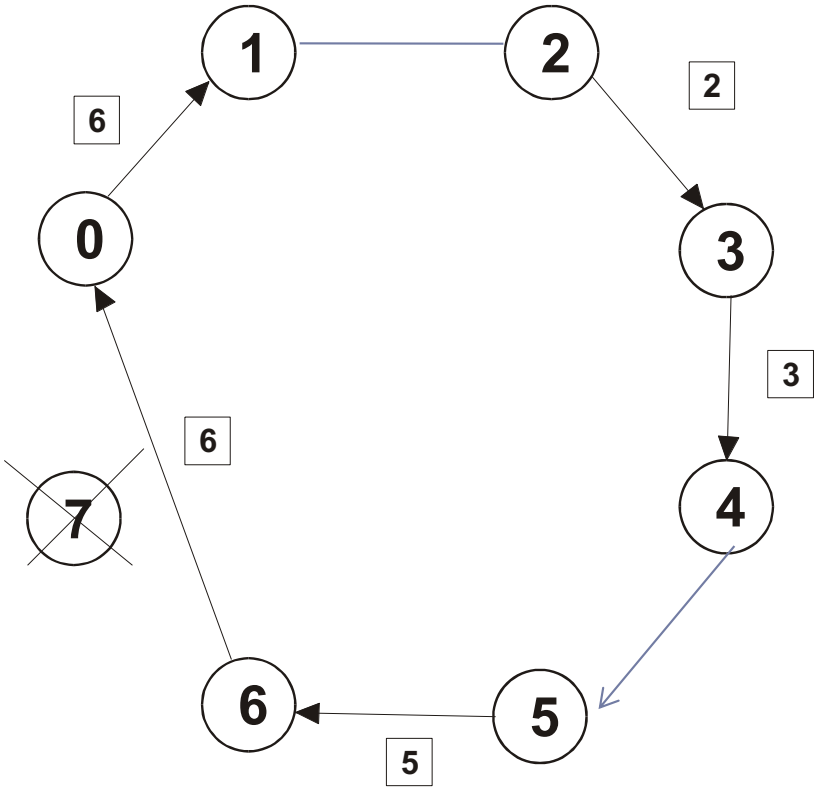
d) Proceso 6 dice a 5 que pare
e) Proceso 6 indica a todos que es el coordinador

Algoritmo basado en 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

Algoritmo basado en 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



Interbloqueo distribuido

- **Interbloqueos** en la asignación de recursos. Existe interbloqueo cuando se cumplen las siguientes condiciones
  - Exclusión mutua
  - Retención y espera
  - No expulsión
  - Condición de espera circular
- **Interbloqueos** en el mal uso de operaciones de sincronización
- **Interbloqueos** en las comunicaciones
  - Todos los procesos están esperando un mensaje de otro miembro del grupo y no hay mensajes de camino

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 euros enviando 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

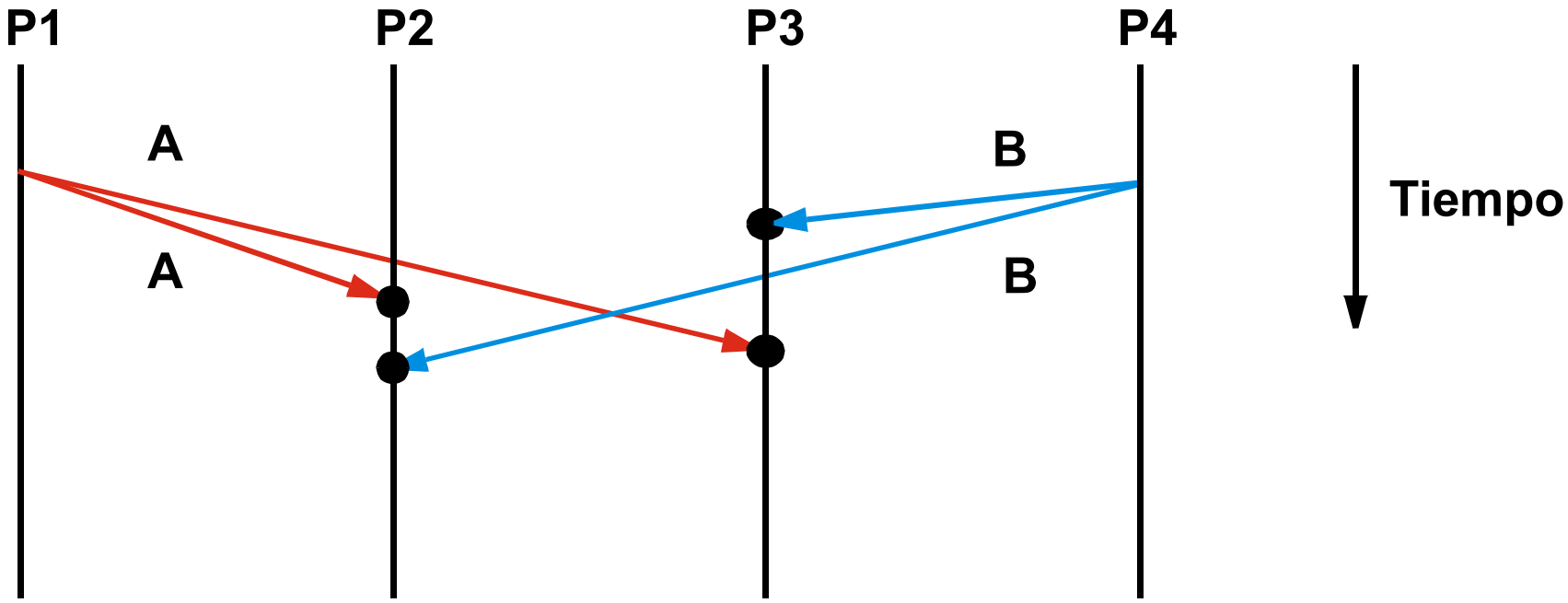
- Implementación de operaciones multicast:
  - Mediante operaciones **punto a punto**
  - Mecanismo poco fiable

- 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 envía una confirmación y monitoriza al emisor para ver si falla. Si falla completa el multicast

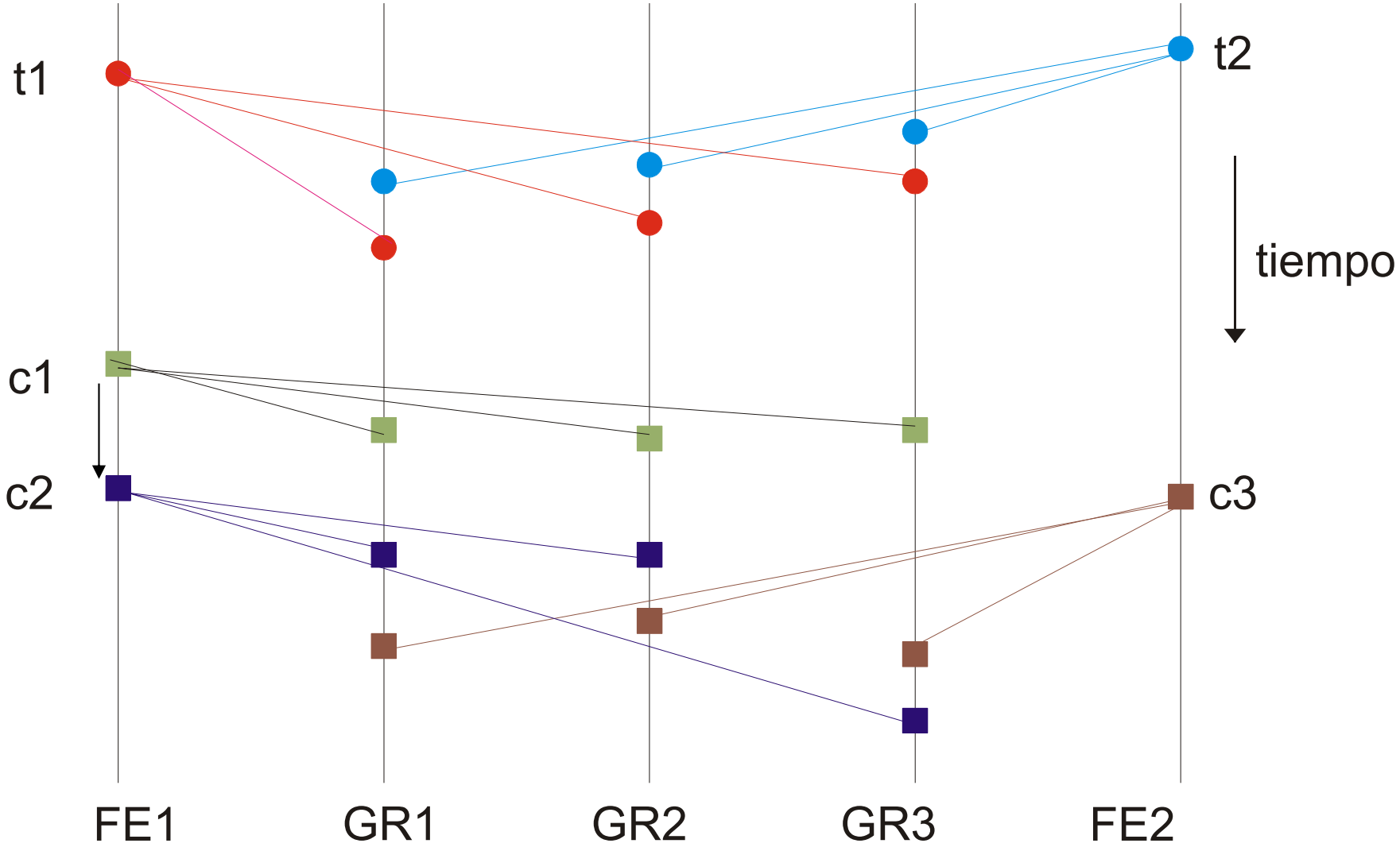
Ejemplo de multicast sin ordenación



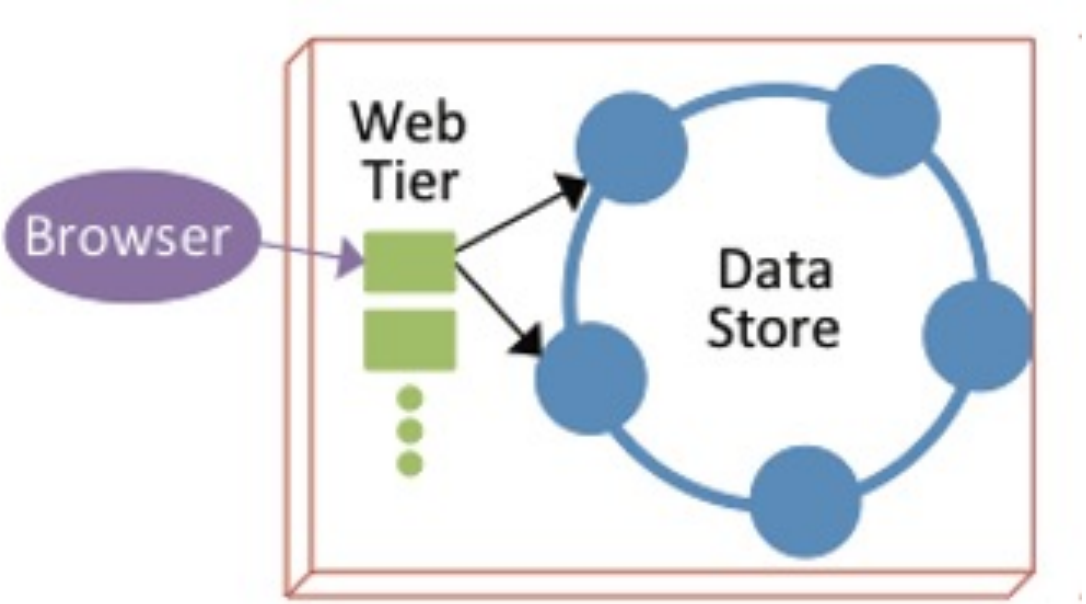
Ordenación de las actualizaciones

- 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

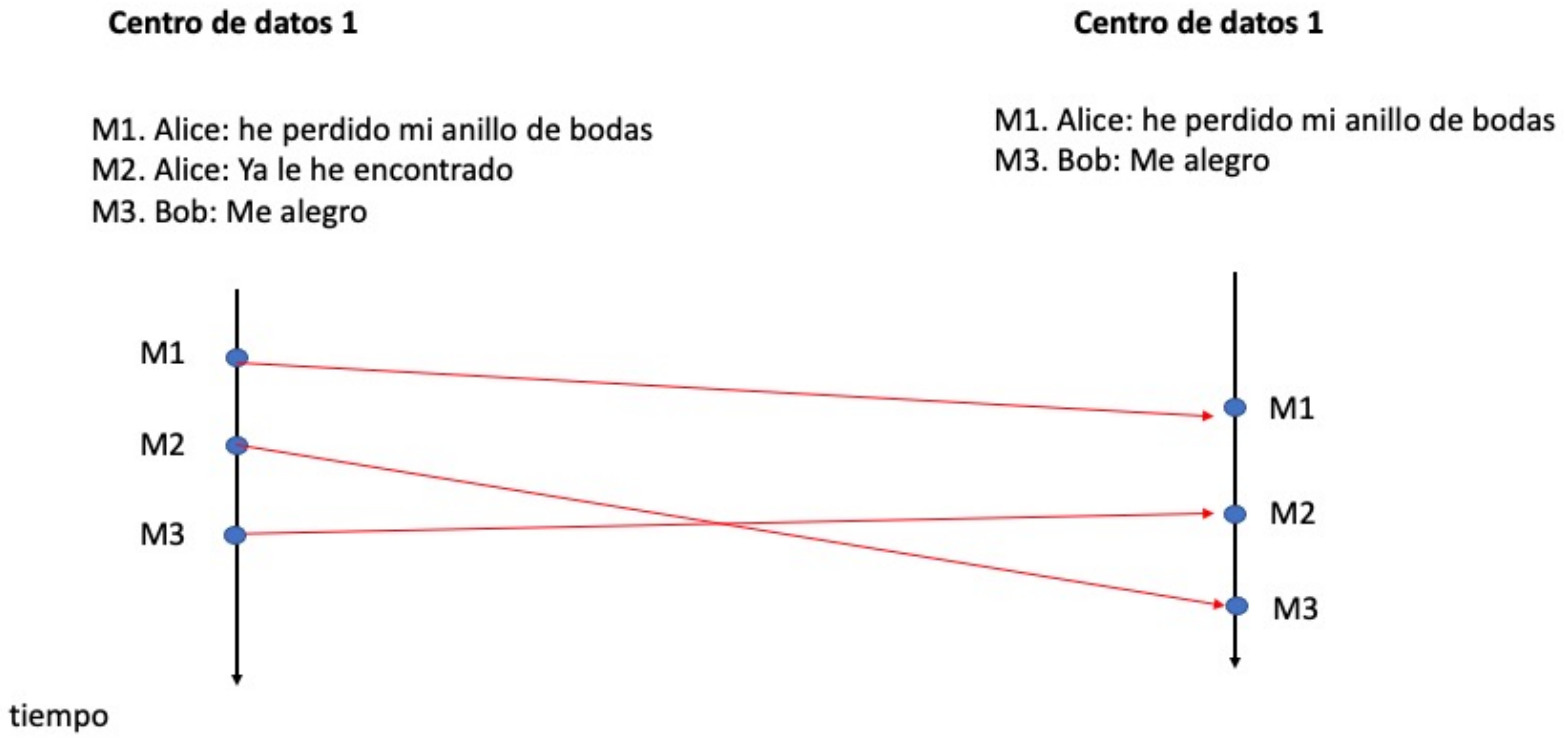
Ordenación total y causal



Problemas en servidores georeplicados

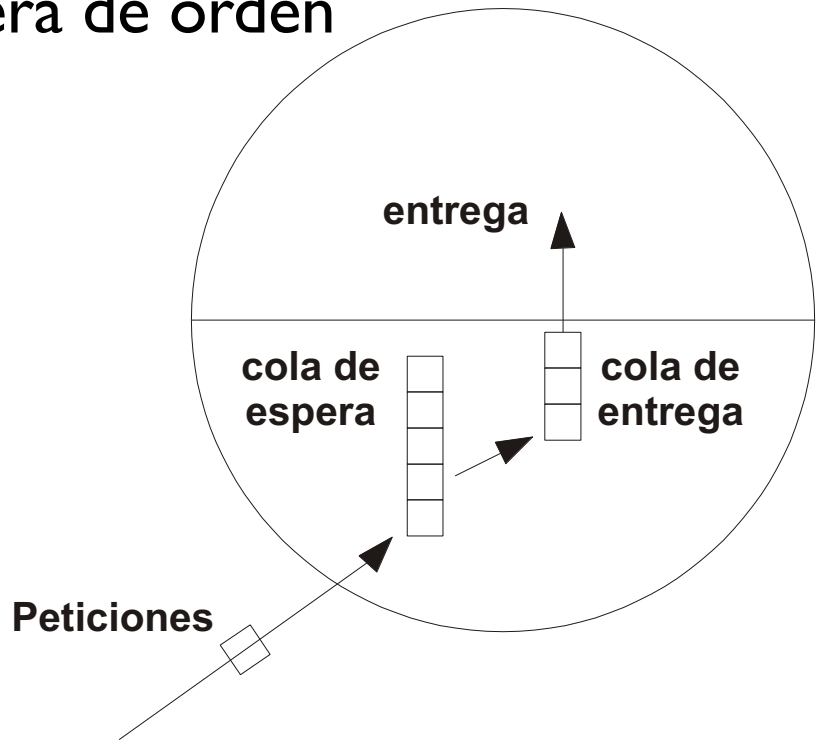


Ejemplo anomalía que no siguen orden causal



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 entrega



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

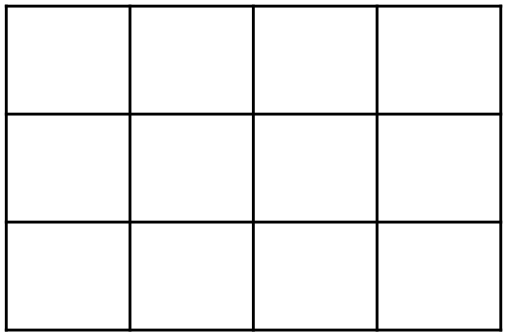
- Birman y Joseph 1987
- Cada proceso q en el grupo almacena:
  - A_q: mayor número de secuencia acordado que se ha observado hasta el momento
  - P_q: su mayor número de secuencia propuesto
  - Los identificadores deben incluir el número de proceso para asegurar un orden total
- Cuando un proceso p 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 de espera y lo marca como no entregable
  - Envía P_q al emisor del mensaje (p)
- El proceso p recibe todos los números de secuencia propuestos 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

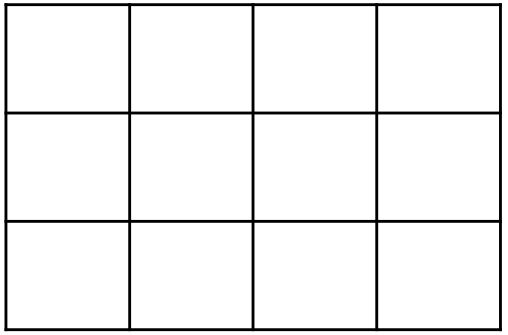
Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

Nodo 2
A2 = 15
Multicast(M2)

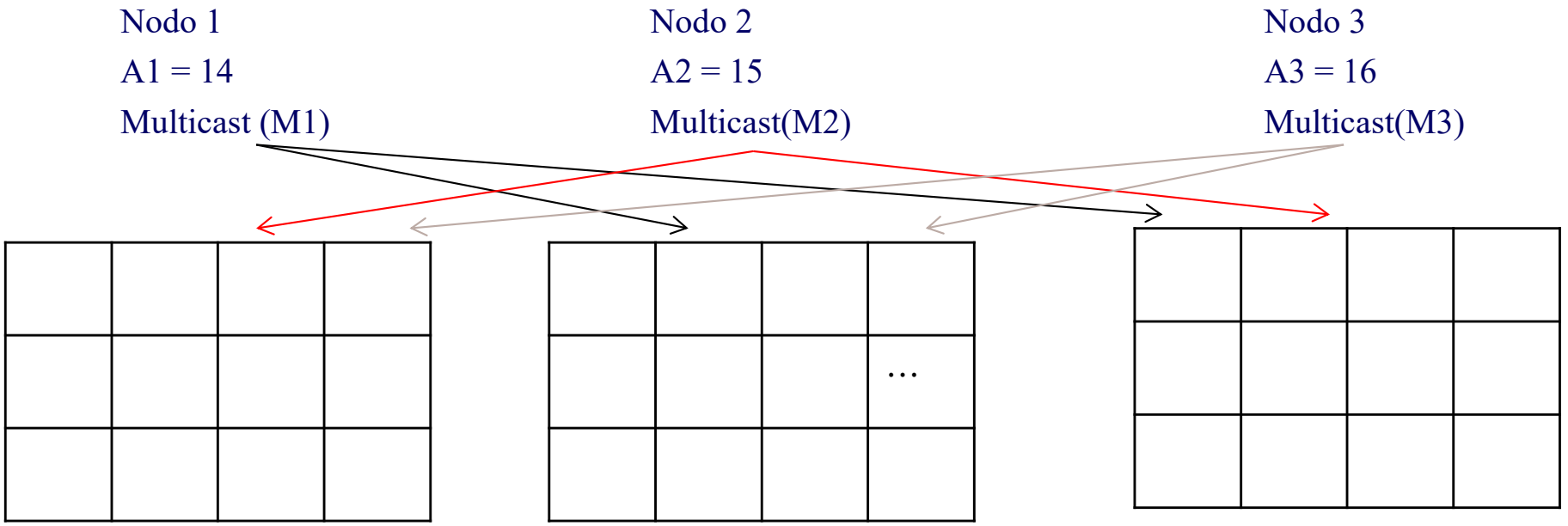
Nodo 3
A3 = 16
Multicast(M3)





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
A2 = 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 (inicialmente Pq=Aq)
- (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	16.1	17.1	…
U	U	U	

Nodo 2
A2 = 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 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	M1	M2	
15.1	17.3	17.1	…
U	U	U	

Nodo 2
A2 = 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:

- Todos actualizan la marca asociada a M1

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	17.1	…
U	D	U	

Nodo 2
A2 = 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:

- Todos actualizan la marca asociada a M1y marcan los mensajes como entregables

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Nodo 2
A2 = 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 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
A2 = 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 y desaparece de la cola de espera

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M2	M1	
15.1	17.1	17.3	…
U	U	D	

Nodo 2
A2 = 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
A2 = 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
A2 = 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	19.3	…
U	D	D	

Nodo 2
A2 = 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:

- M1 se se puede entregar en el nodo 2

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	19.3	…
U	D	D	

Nodo 2
A2 = 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:

- M1 se entrega en el nodo 2 y desaparece de la cola de espera

Ejemplo

Nodo 1
A1 = 14
Multicast (M1)

M3	M1	M2	
15.1	17.3	19.3	…
U	D	D	

Nodo 2
A2 = 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	19.3	…
U	D	D	

Nodo 2
A2 = 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
A2 = 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	

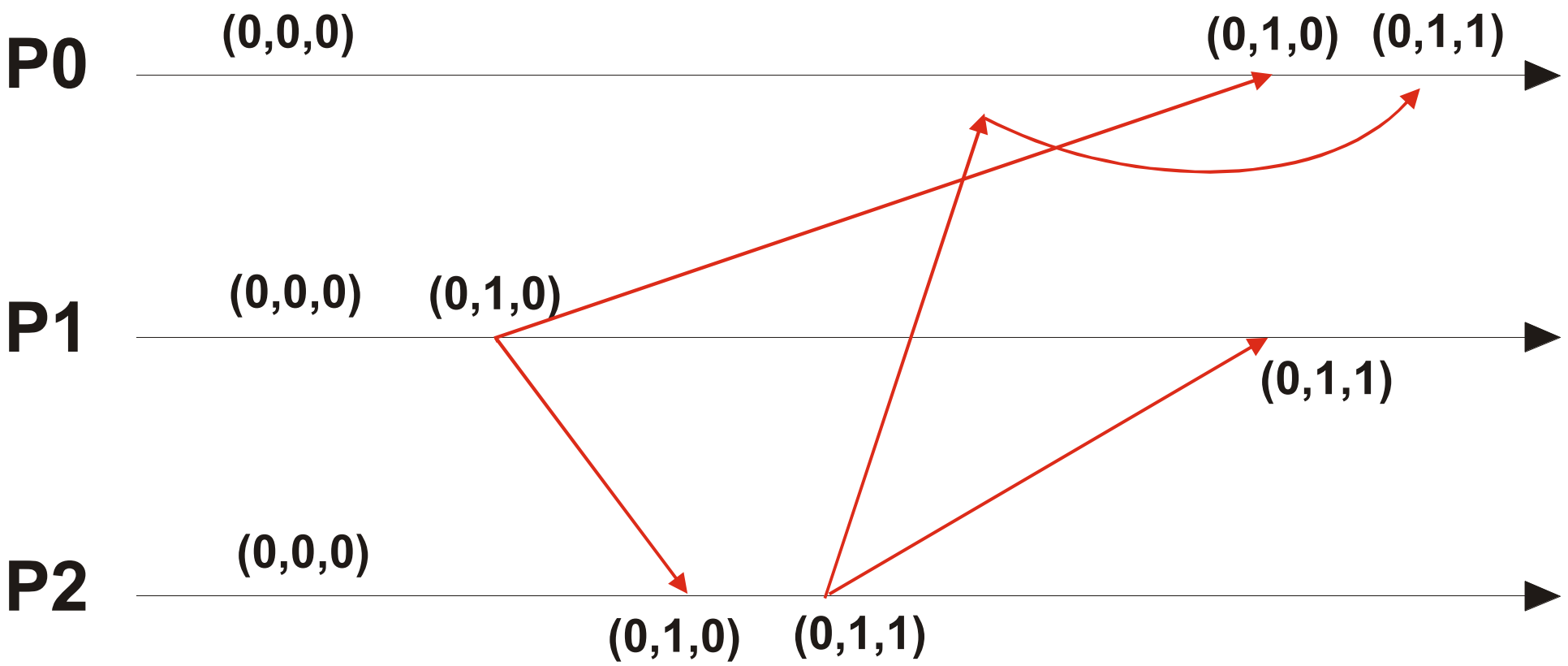
Etapa 4:

- 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

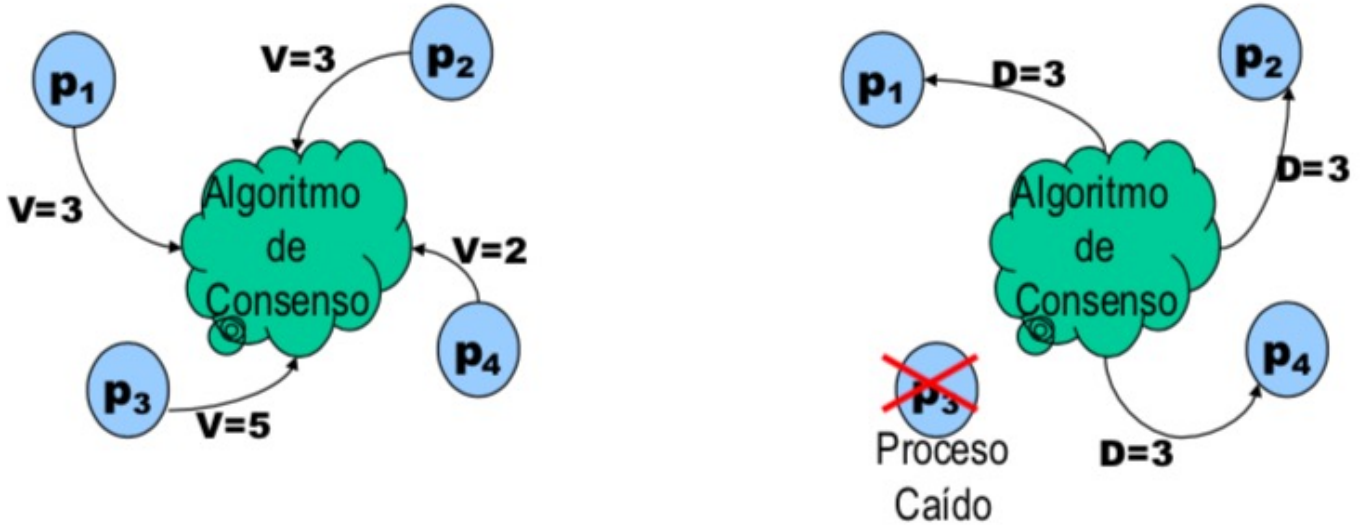


Ejemplo

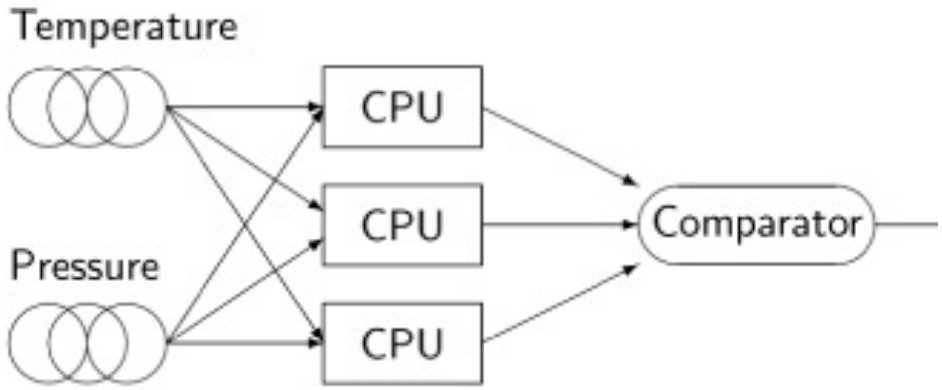
- Vector enviado por el proceso 0: (4, 6, 8, 2, 1, 5)
- Vector en el proceso 1: (3, 6, 9, 2, 1, 5)
- Vector en el proceso 2: (3, 5, 8, 2, 1, 5)
- ¿Se puede entregar el mensaje enviado por el 0?
  - Al 1 si:
    - Es el siguiente en la secuencia de mensajes recibidos del 0 y no se han perdido mensajes.
  - Al 2 no:
    - Es el siguiente en la secuencia de mensajes recibidos del 0.
    - Le falta un mensaje del proceso 1
      - El proceso 0 ha recibido un mensaje del 1 (6) que no ha sido recibido por el 2 (5)

Problemas 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



Ejemplo en un sistema replicado



- Los sensores pueden obtener valores distintos
- Un sensor puede fallar y generar fallos arbitrarios
- Una CPU puede fallar

Tipos de fallo

- Fallo parada: el sistema que falla deja de funcionar
- Fallo con recuperación: el sistema que falla vuelve a funcionar en algún momento
- Fallos bizantinos: el componente que falla genera fallos arbitrarios

Acuerdo bizantino

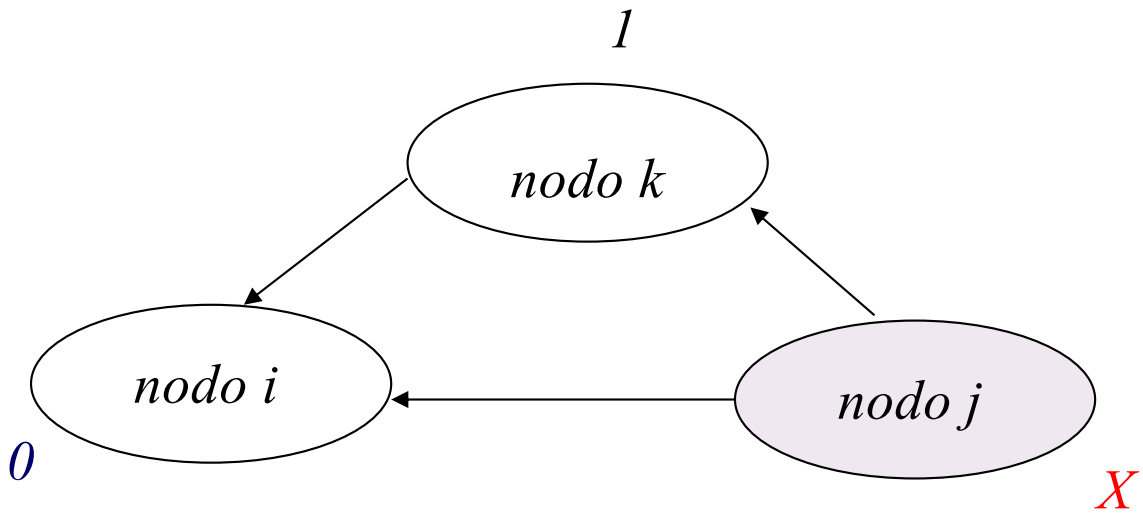
- En la mayoría de las ocasiones cuando un componente o sistema falla, su funcionamiento es arbitrario.
  - Pueden enviar información diferente a diferentes componentes con los que se comunica.
  - Alcanzar un acuerdo entre las observaciones que hacen diferentes componentes puede ser complicado en presencia 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
Problema de los Generales Bizantinos

- 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).

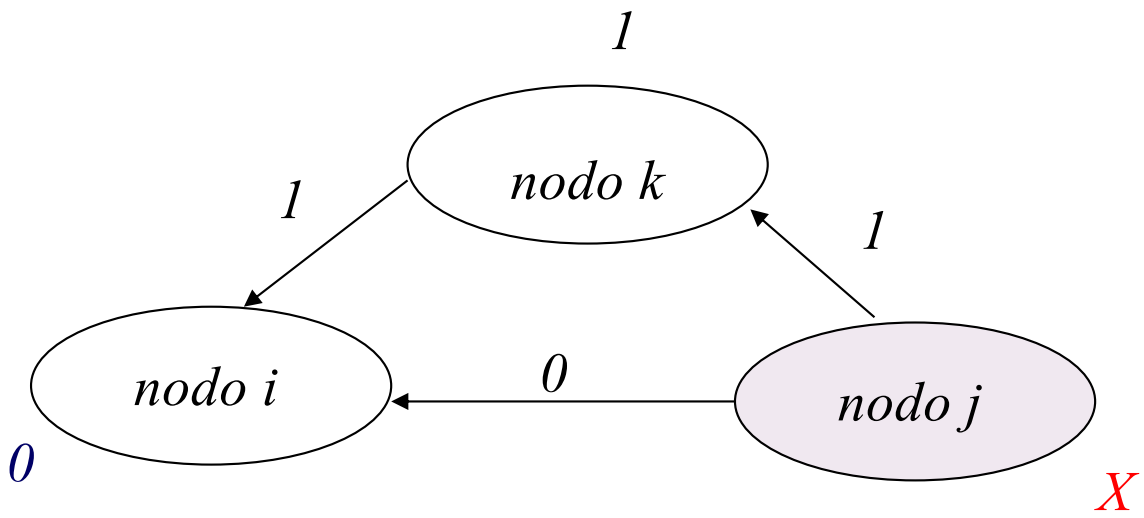
- **Objetivo**: que los nodos sin fallo alcancen un acuerdo o consenso sobre un determinado valor (ataque, retirada, espera). Es decir que *vean* el mismo valor para un dato.
- ¿Cuántos nodos hacen falta para hacer frente a m fallos?

Problema con tres nodos



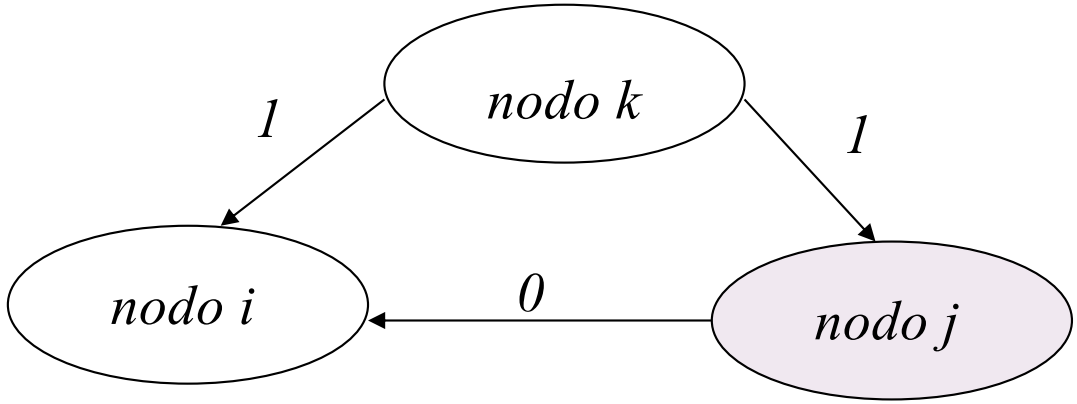
¿puede el nodo i y nodo k llegar a un consenso en el valor?

Problema con tres nodos



Problema con tres nodos

- 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

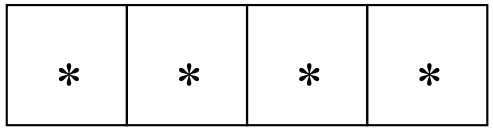
- Suposiciones:
  - Los mensajes enviados se entregan correctamente.
  - El receptor conoce al emisor de un mensaje
  - Se puede detectar la ausencia de un mensaje
- Algoritmo (*Lamport, Pease y Shostack*) para cuatro nodos:
  - Se basa en el intercambio de mensajes entre los nodos
  - Cada nodo N_i 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 $V_i(j) = \text{null}$ ($\forall \ i \neq j$)
  - 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

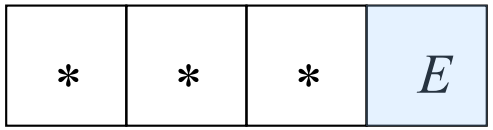
- Después del intercambio de mensajes, cada nodo construye un vector con los valores mayoritarios
- Si no existe mayoría entonces se asume que no hay un observación coherente.

Ejemplo. Etapa inicial

Nodo 1



Nodo 4

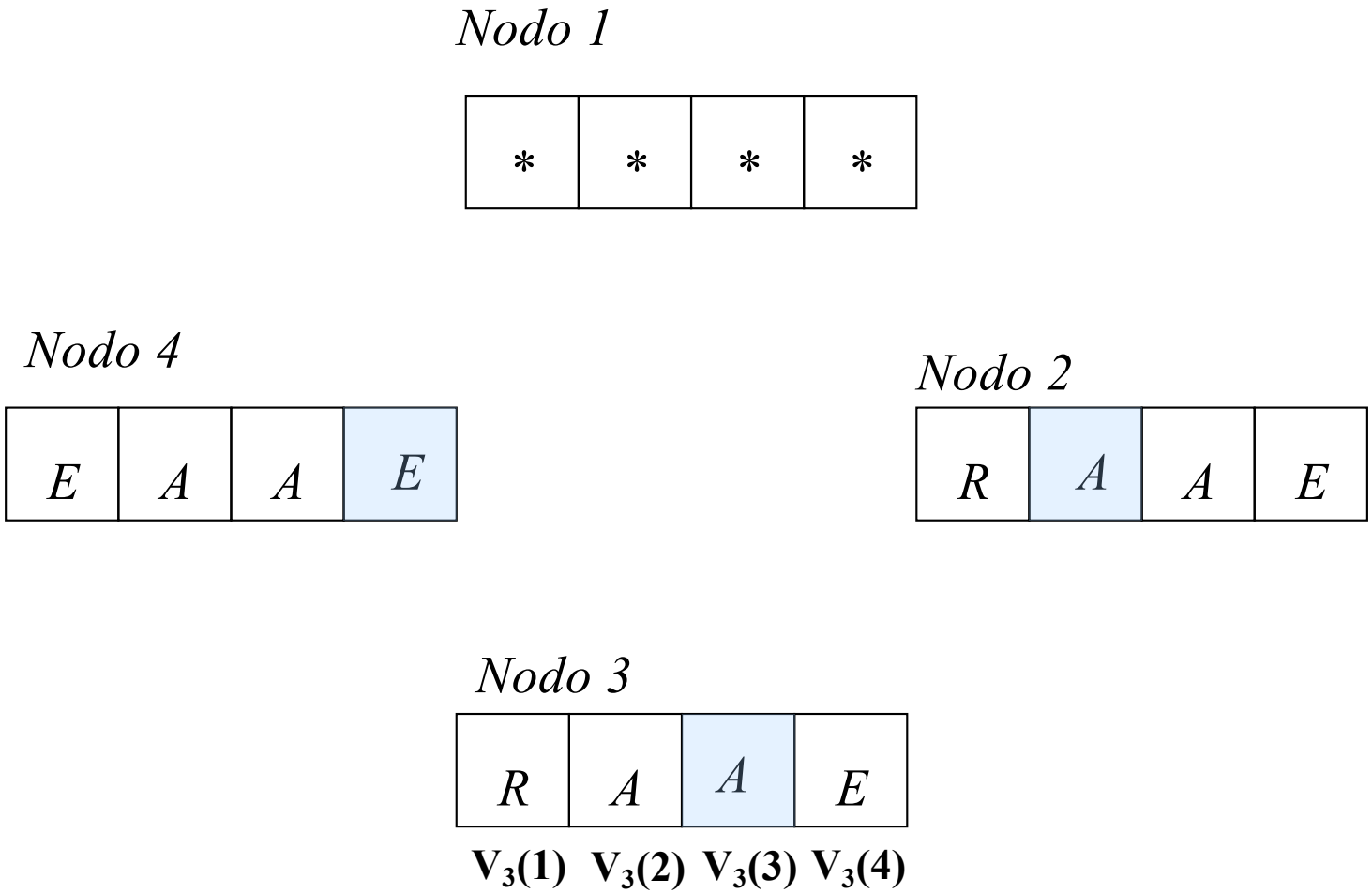


Nodo 2

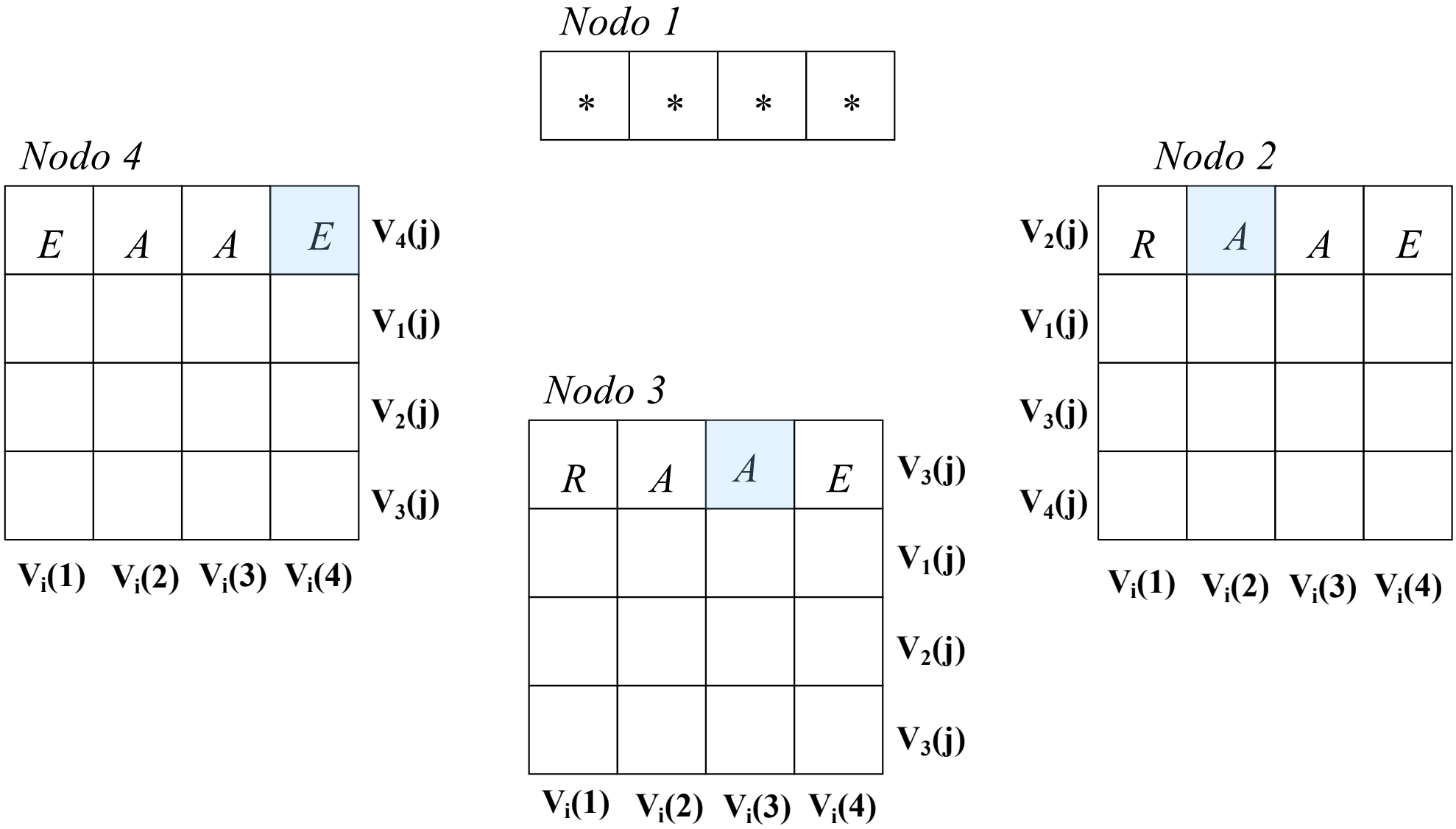
Nodo 3

Observaciones:
E: esperar
A: atacar
R: retirarse

Ejemplo. Primer intercambio de mensajes

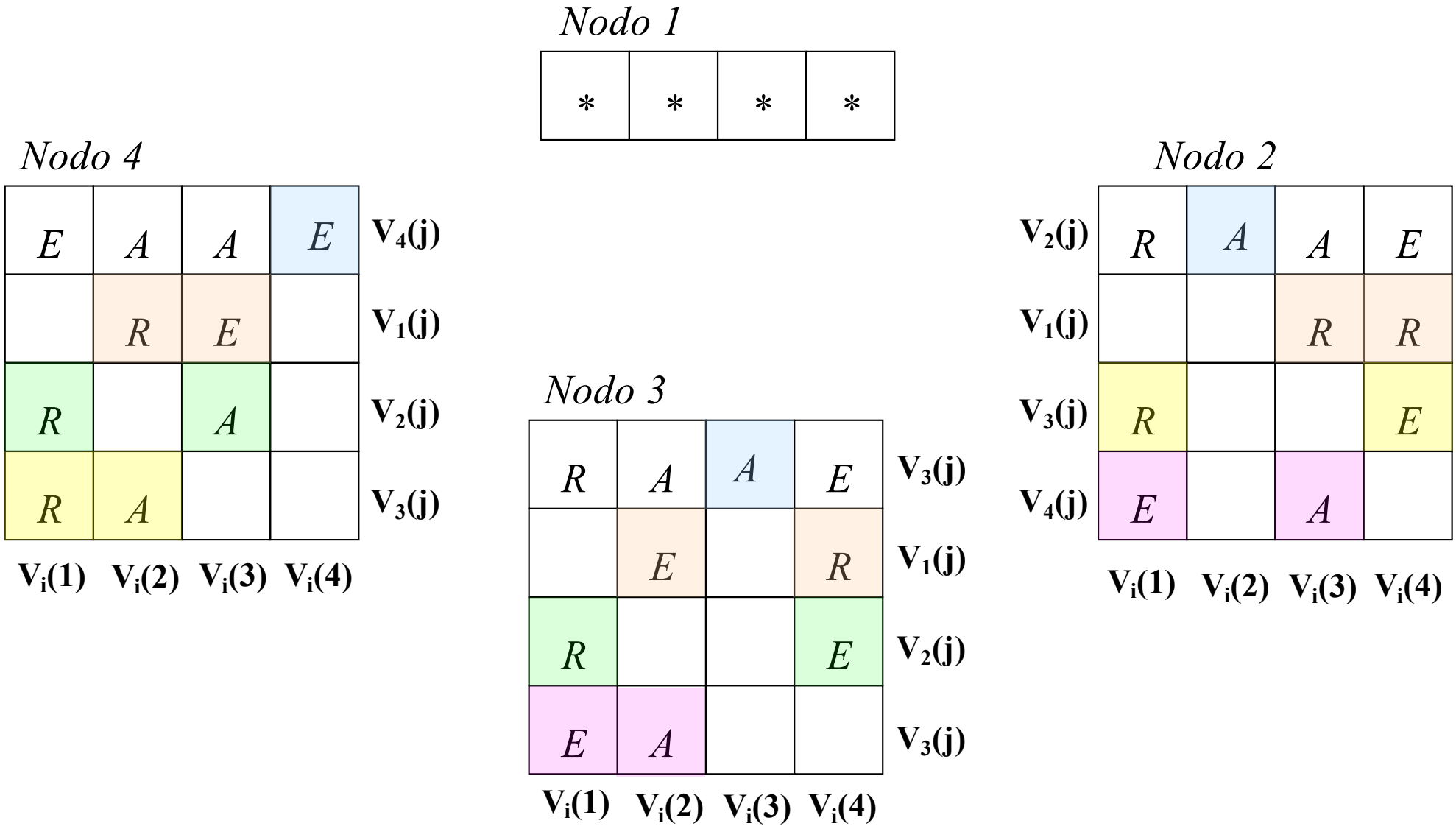


Ejemplo. Segundo intercambio de mensajes



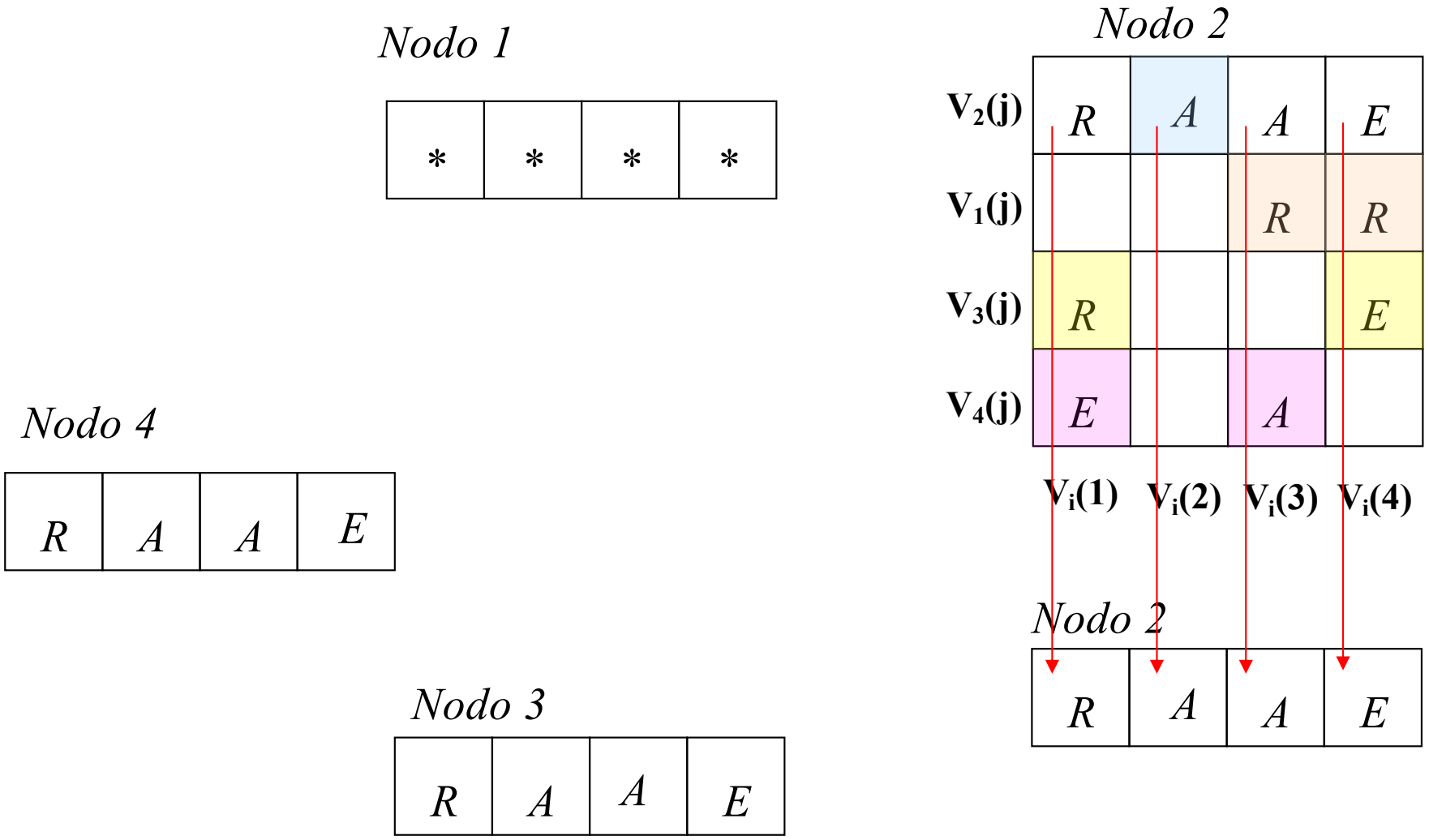
$V_i(j)$ -> La observación que de **j** dice **i**

Ejemplo. Segundo intercambio de mensajes



$V_i(j)$ -> La observación que de **j** dice **i**

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

Servicio de nombres

- Objetivo: descubrir recursos en sistemas distribuido
- Tipos de recursos:
  - Ficheros, usuarios, grupos, procesos, dispositivos, máquinas, …
- El nombre del recurso permite referirse a una entidad única en un sistema distribuido (aunque pueda estar replicada y haya varios nombres para la misma entidad)
  - Ejemplo: en sockets se identifica con IP + puerto

Nombres y servicios

- Los nombres utilizados en sistemas distribuidos son específicos de algún servicio concreto
- Ejemplos:
  - Nombre de fichero
  - Nombres de usuarios
  - Nombres de hosts
  - Nombres de objetos remotos o servicios remotos en caso de servicios de llamadas a procedimientos remotos o invocaciónón de métodos remotos

Identificadores de recursos

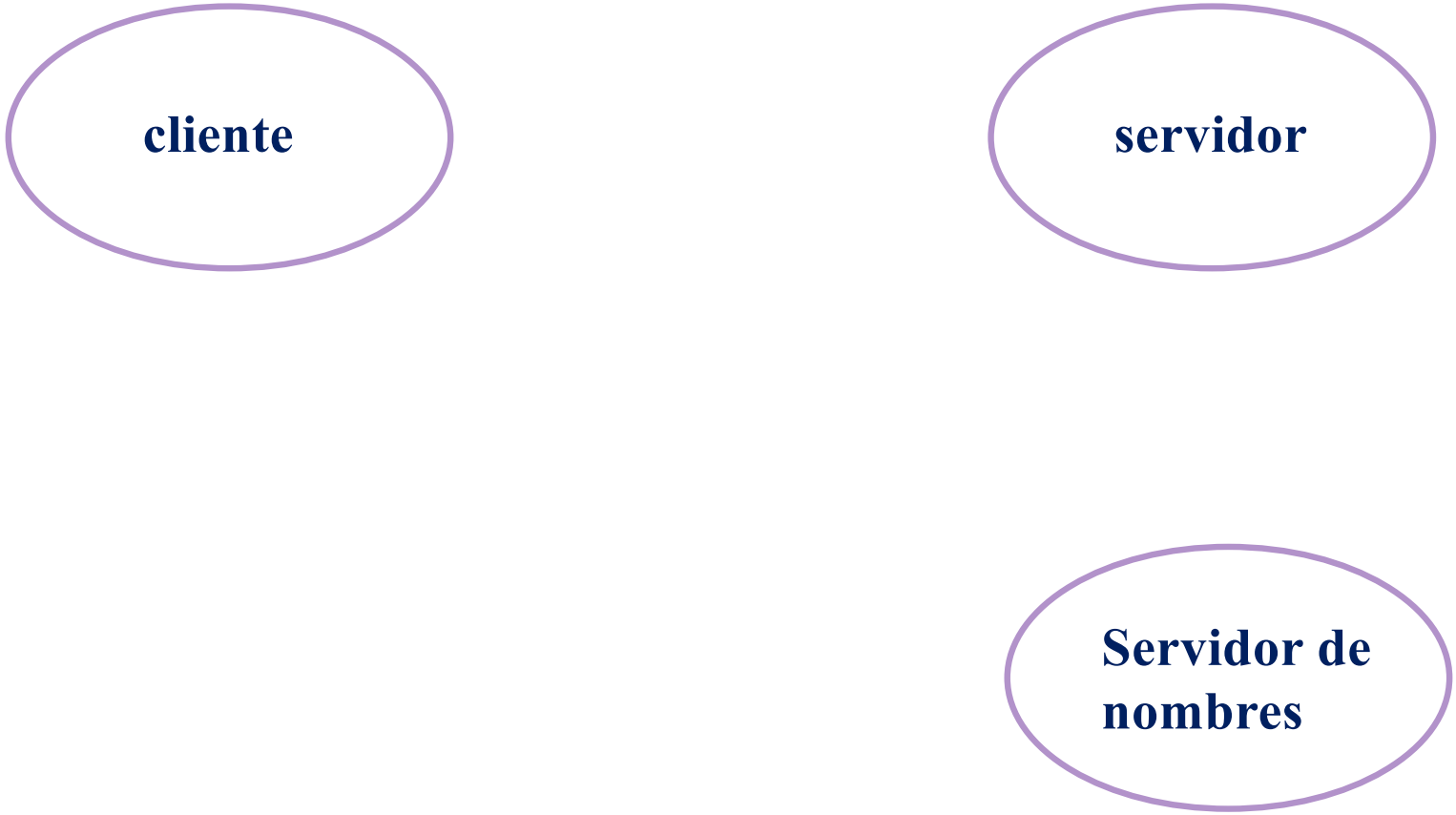
- URI (*Uniform Resource Identifier*): identifica recursos en Internet
  - URL (*Uniform Resource locators*): URI que proporcionan información para localizar recursos
    - Puede verse afectado si el recurso se mueve
  - URN (*Uniform resource names*)
    - URI que solo utilizan nombres sin incluir información de localización
    - Requiere un proceso de traducción
  - Ejemplos:
    - urn:ietf:rfc:3187
    - http://tools.ietf.org/html/rfc3187.html

Servicio de nombres

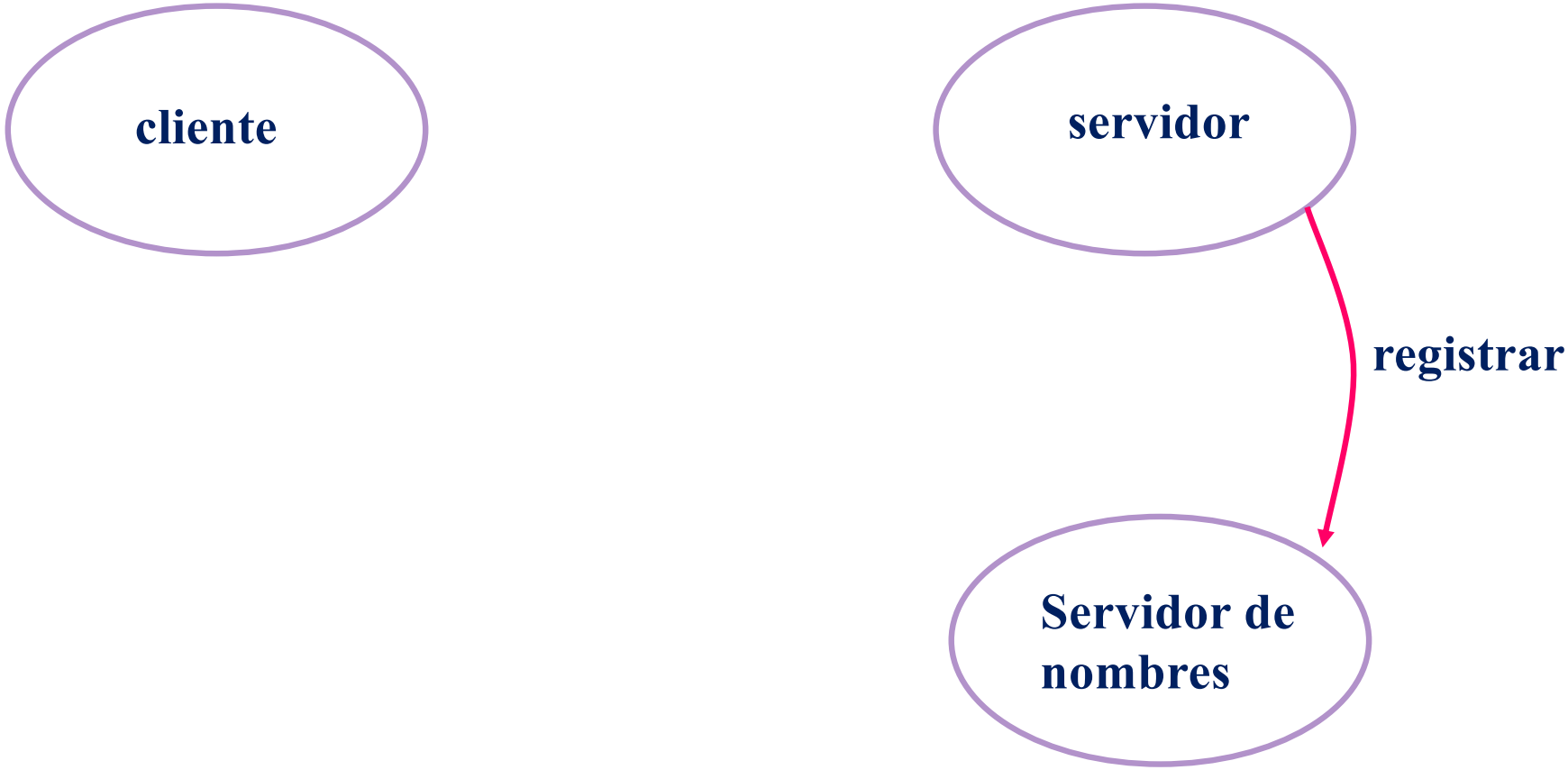
- Un servicio de nombres almacena de forma general pares <nombre, atributos>

Ejemplo: ¿Cómo obtener la dirección de un servidor de aplicaciones?

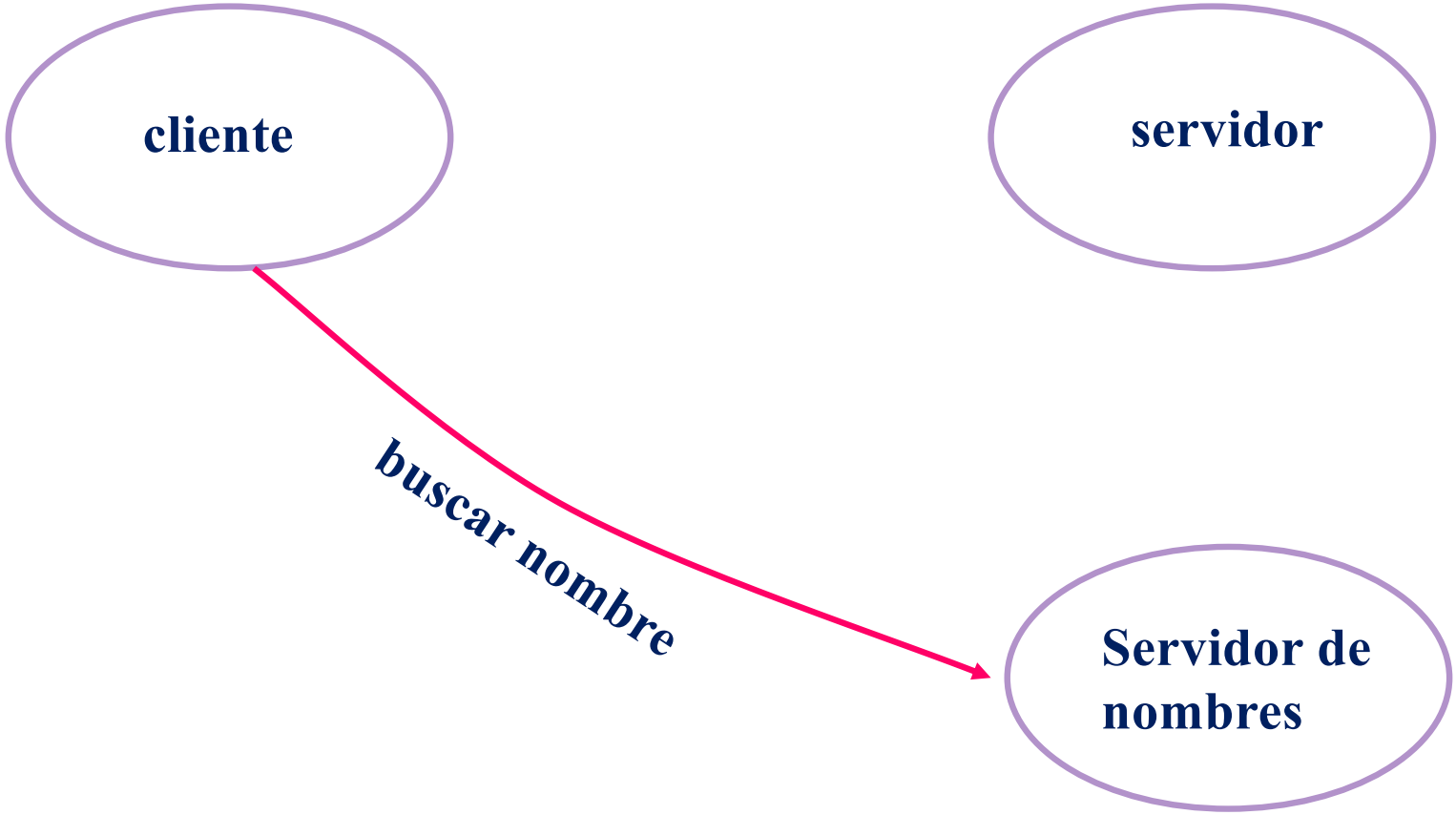
Estructura general de un servicio de nombres



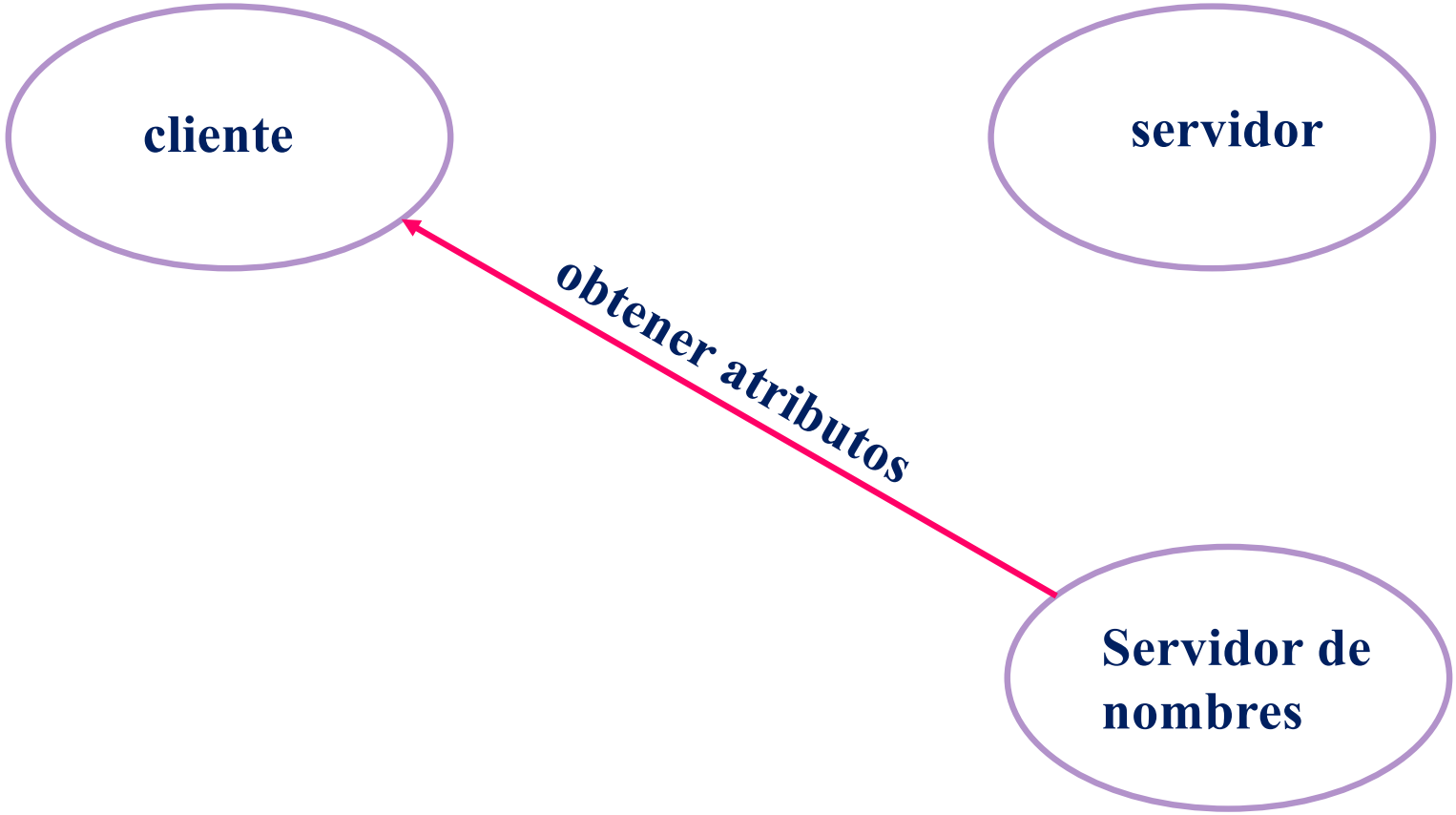
Estructura general de un servicio de nombres



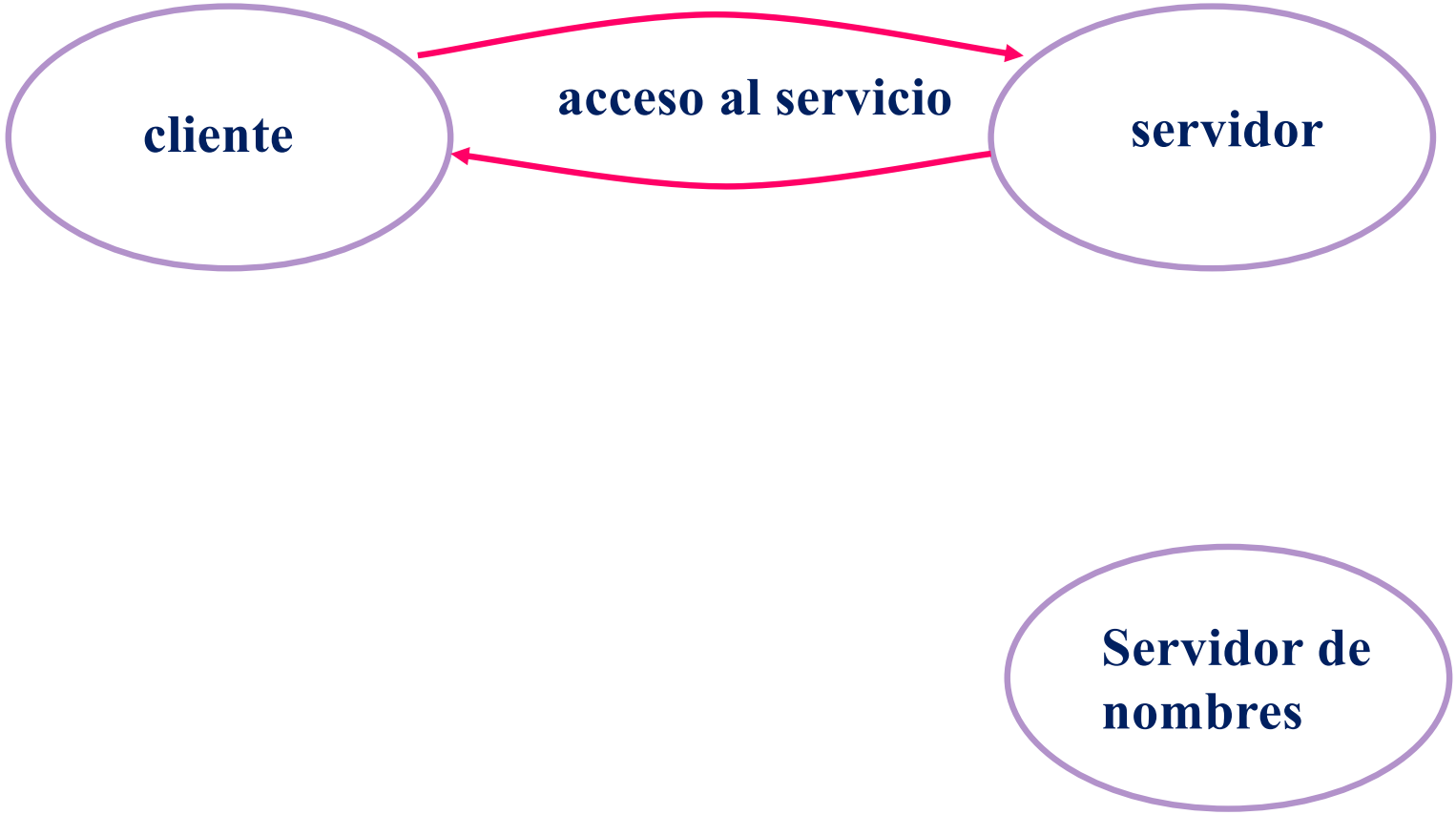
Estructura general de un servicio de nombres



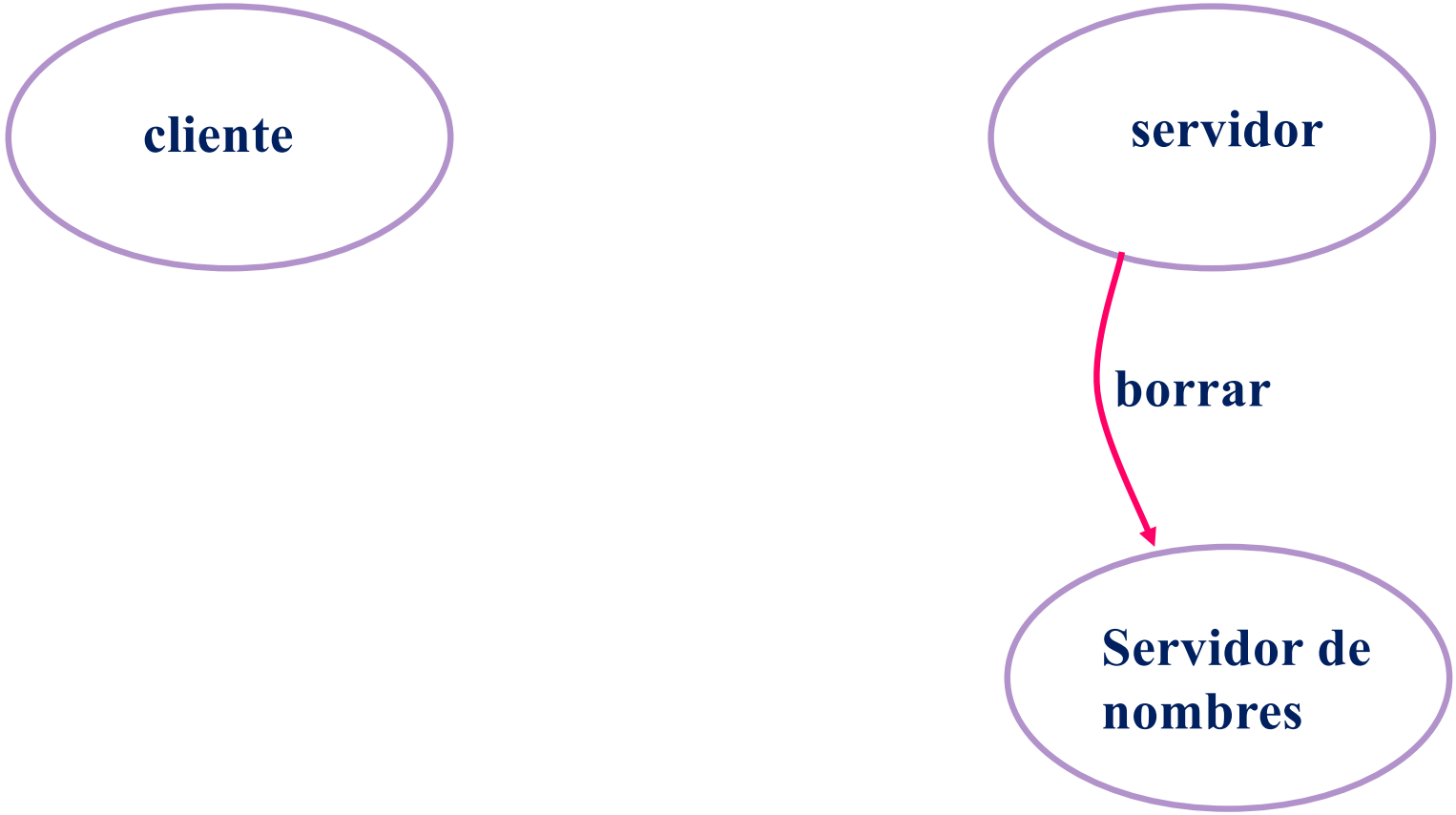
Estructura general de un servicio de nombres



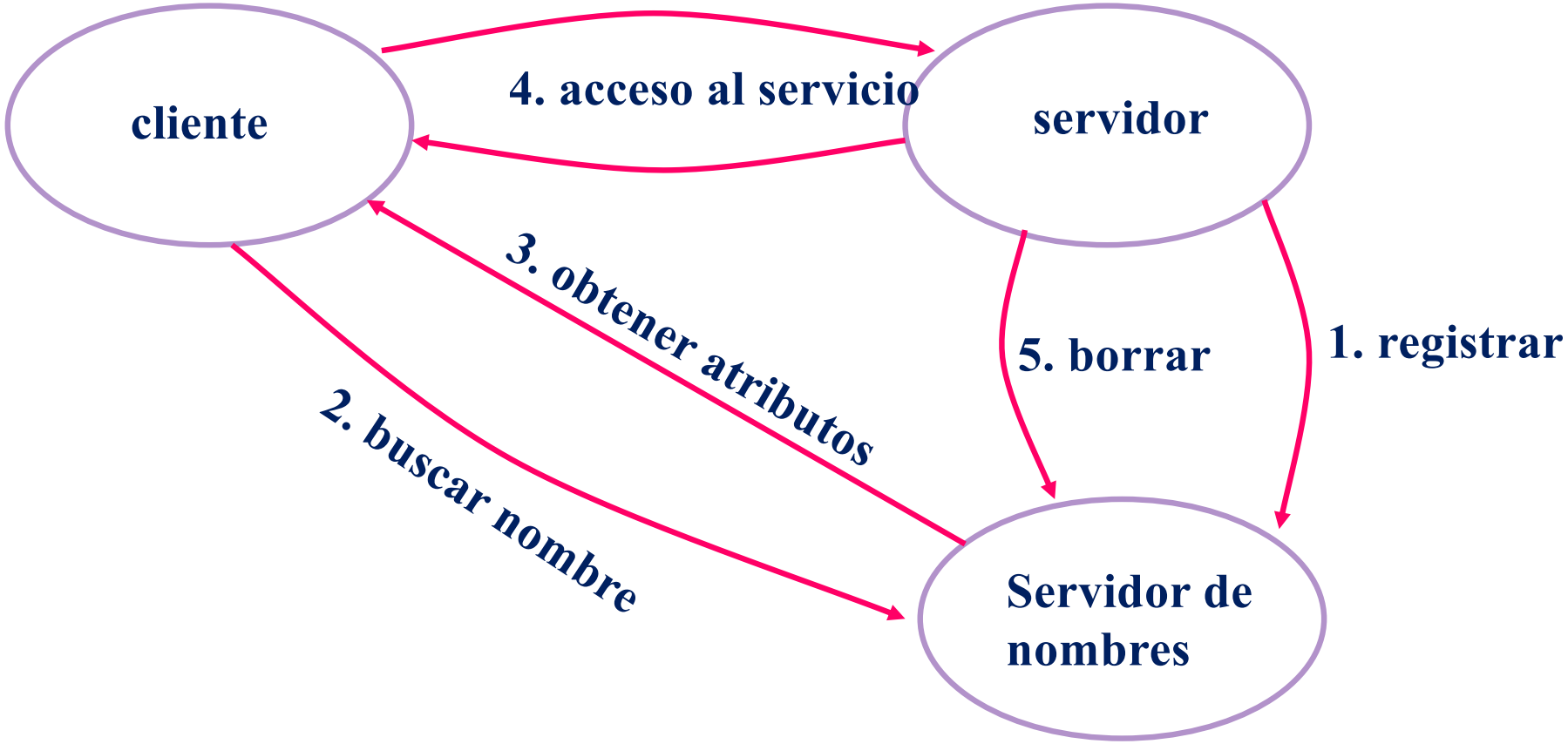
Estructura general de un servicio de nombres



Estructura general de un servicio de nombres

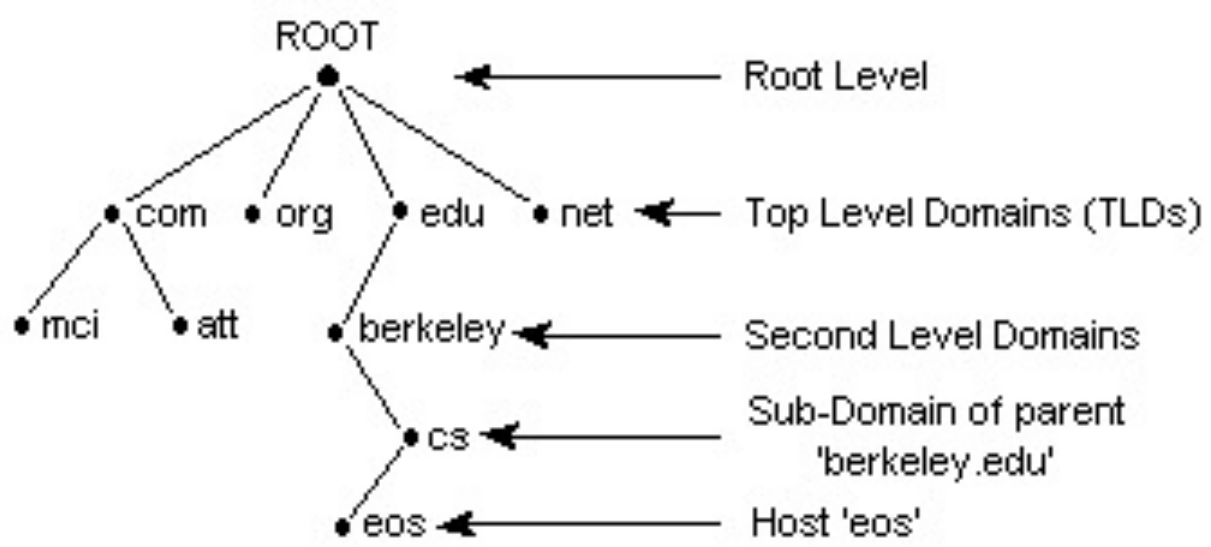


Estructura general de un servicio de nombres



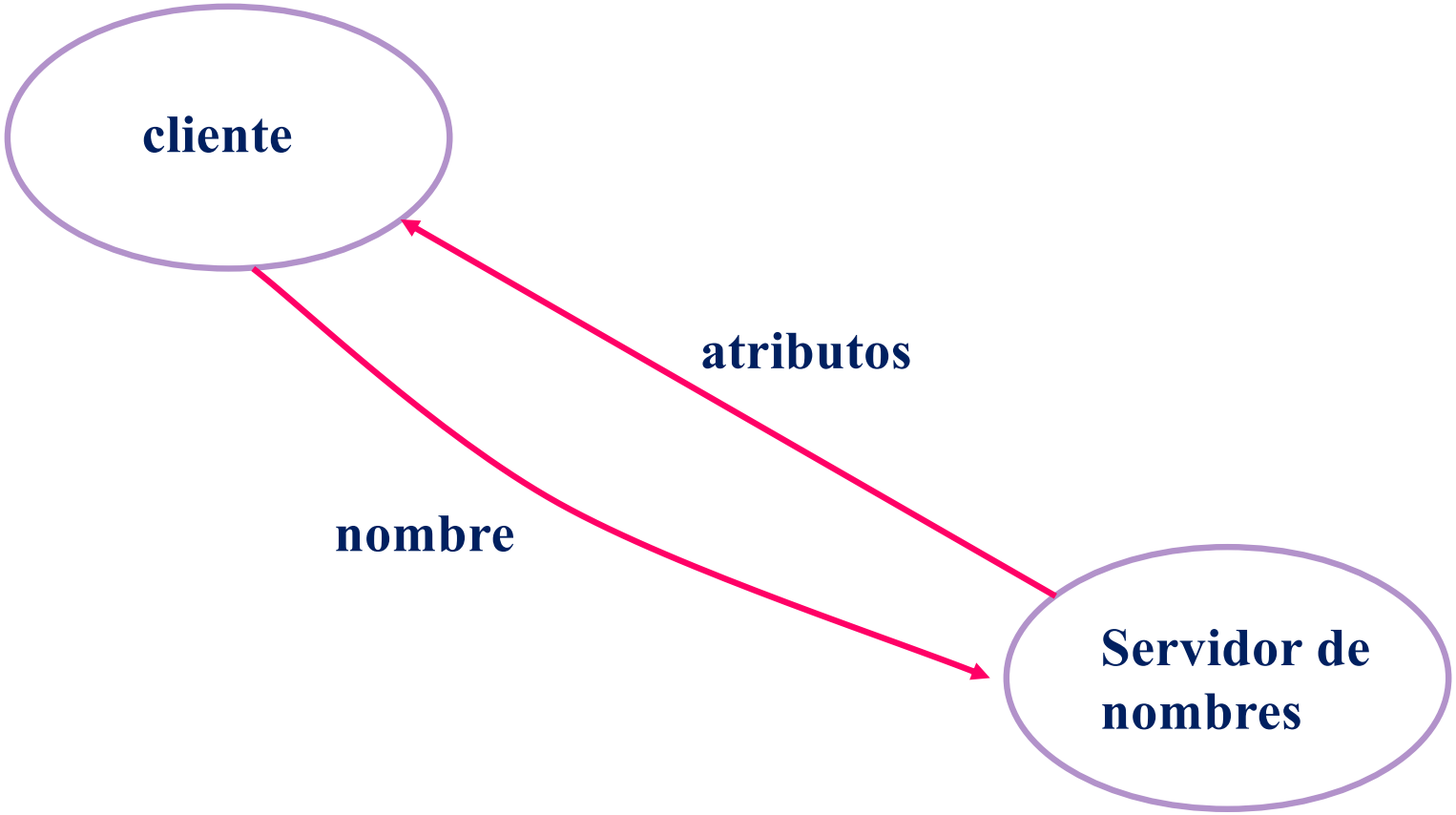
Espacio de nombres

- El espacio de nombres es el conjunto de nombres válidos reconocidos por un servicio particular
  - Plano
  - Jerárquico

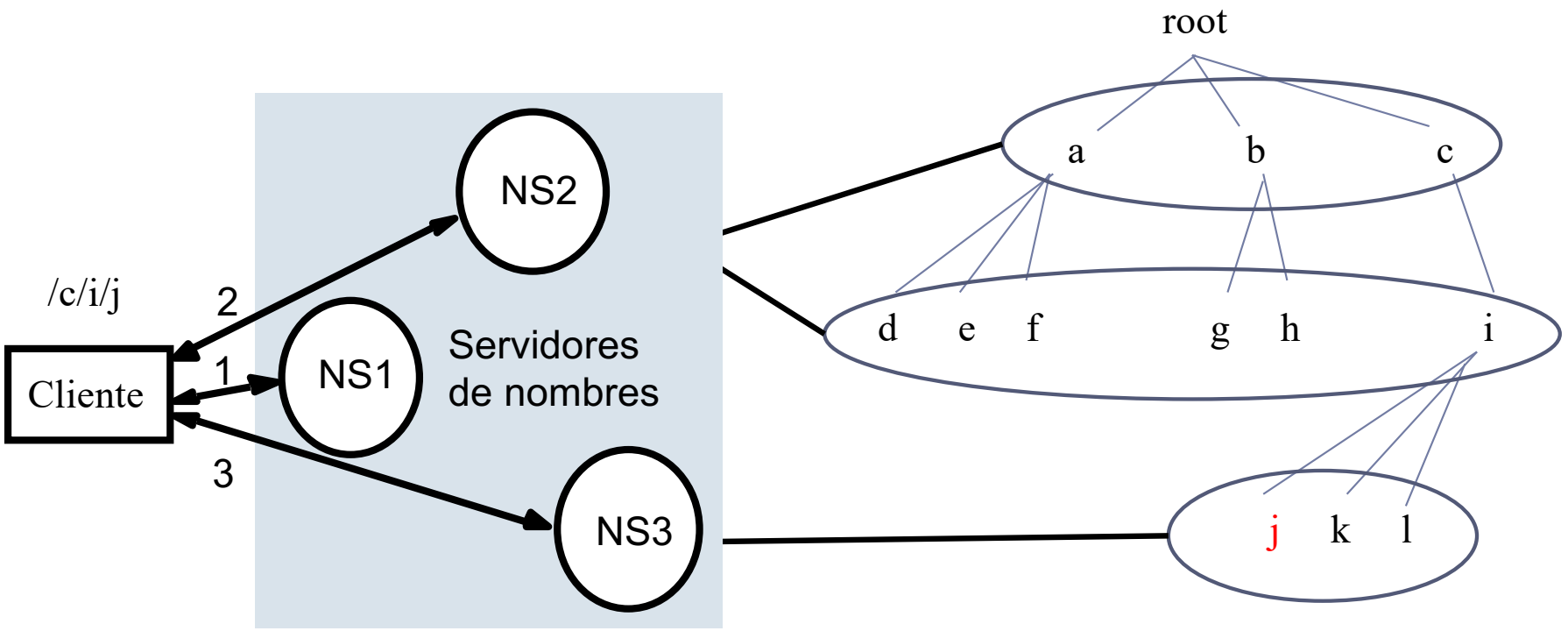


Resolución de nombres

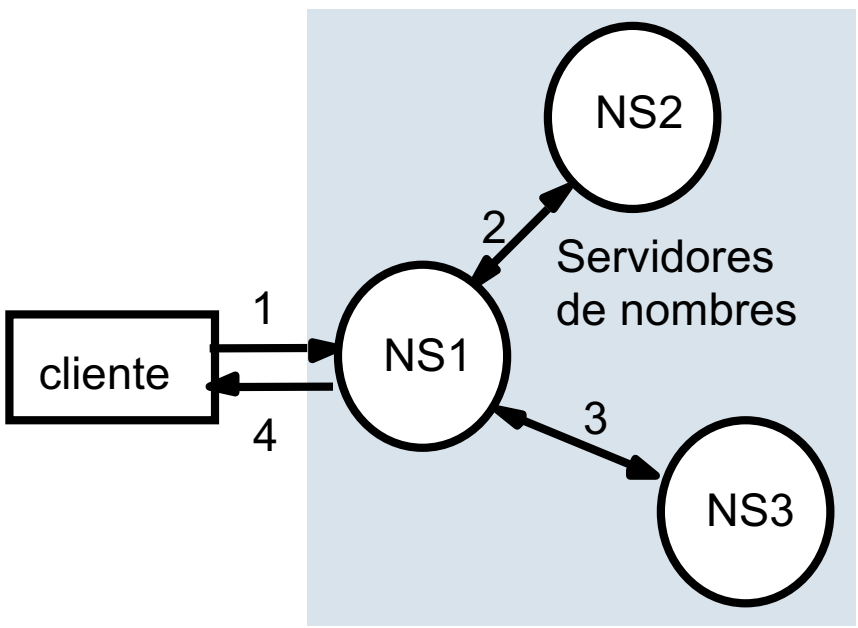
- Proceso iterativo que permite a un cliente obtener los atributos de interés a partir de un determinado nombre



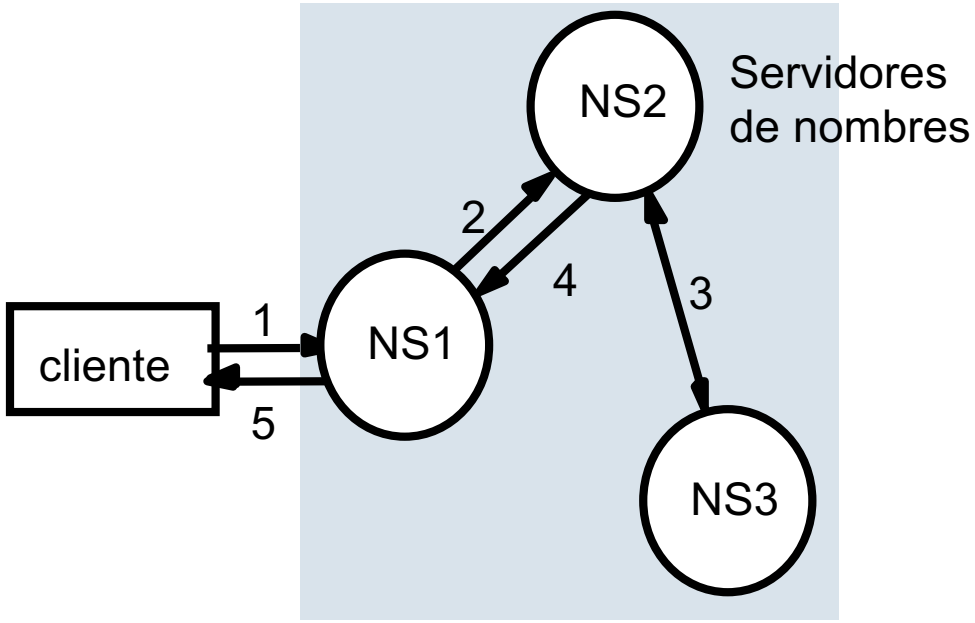
Resolución iterativa guiada por el cliente



Resolución guiada por el servidor

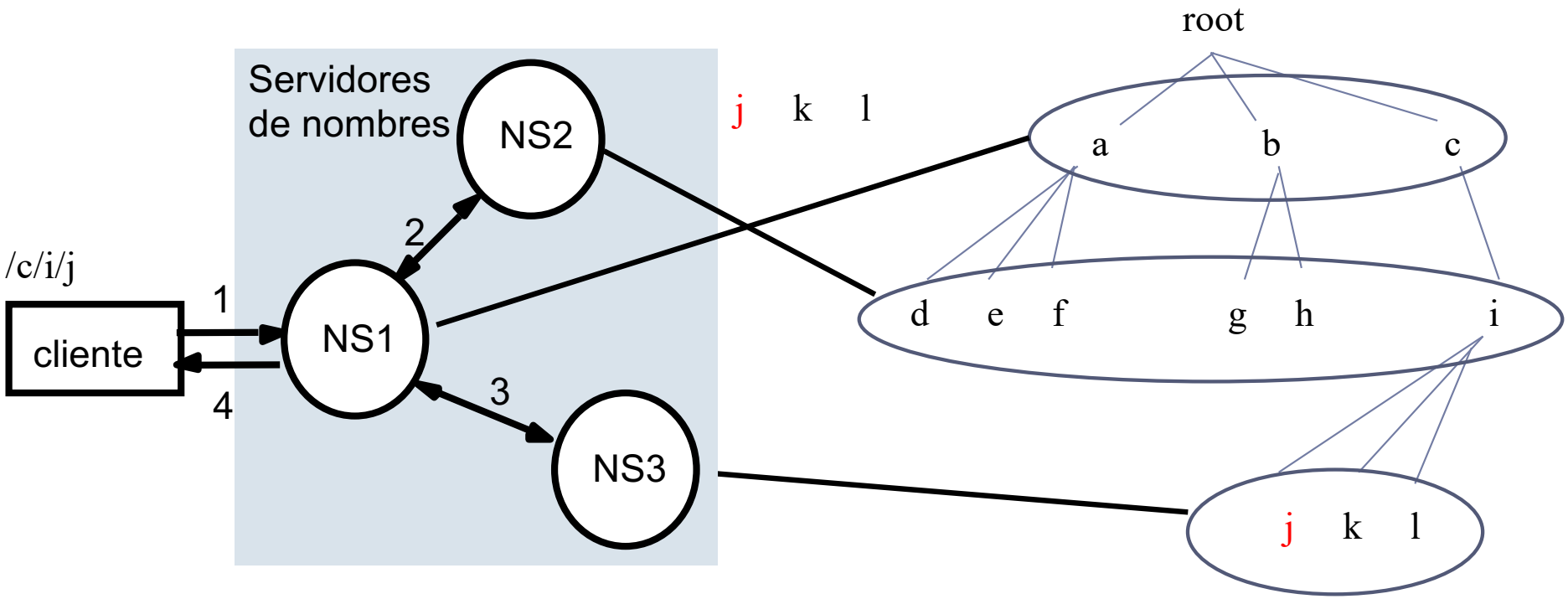


No recursiva



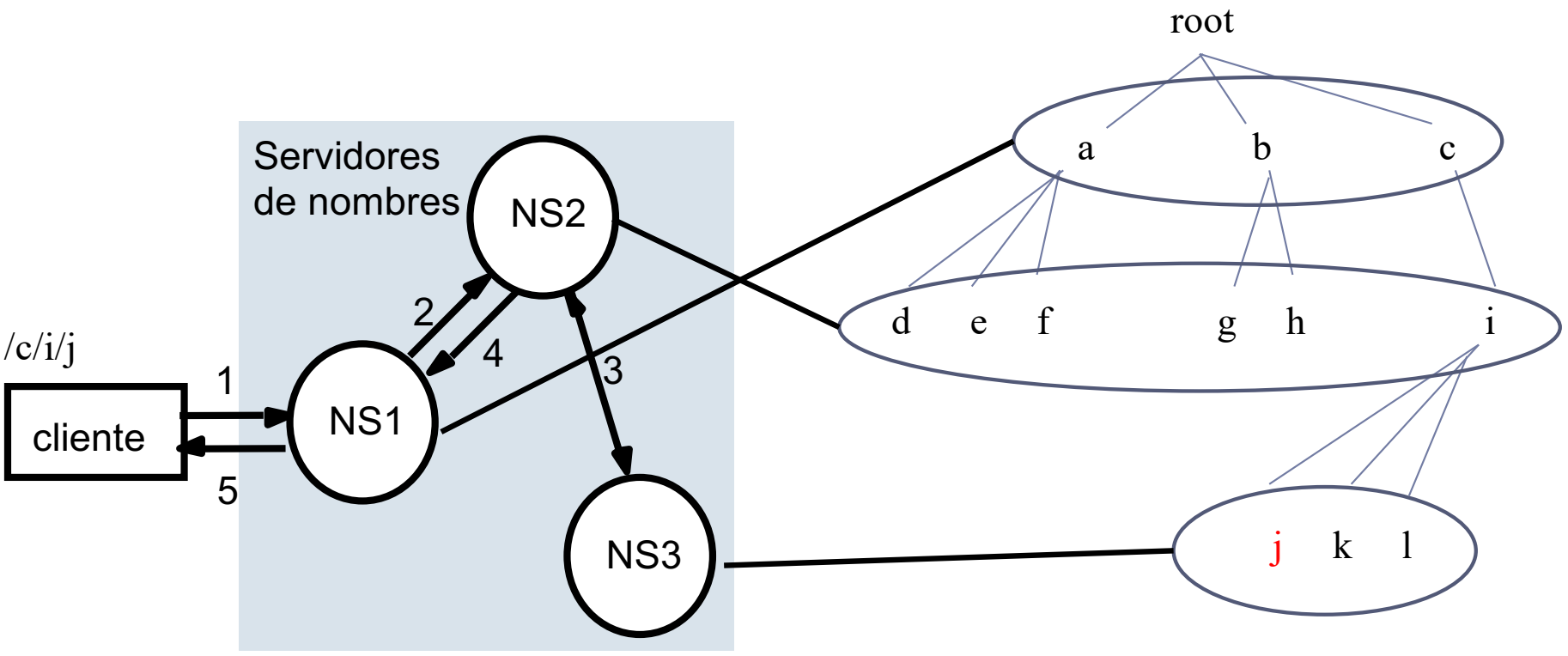
Recursiva

Resolución guiada por el servidor



No recursiva

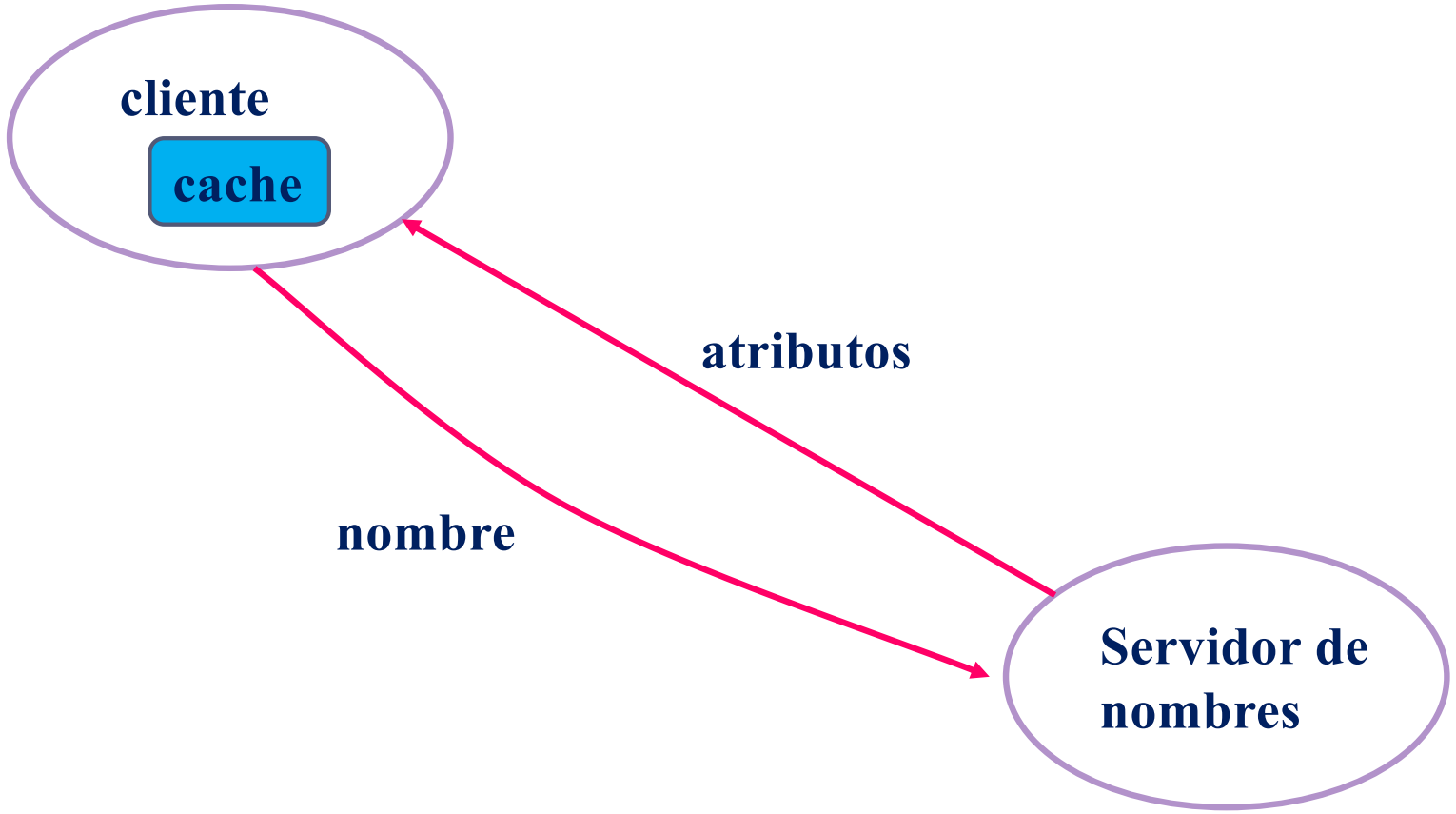
Resolución guiada por el servidor



Recursiva

Resolución de nombres
Empleo de cachés en el cliente

- Proceso iterativo que permite a un cliente obtener los atributos de interés a partir de un determinado nombre

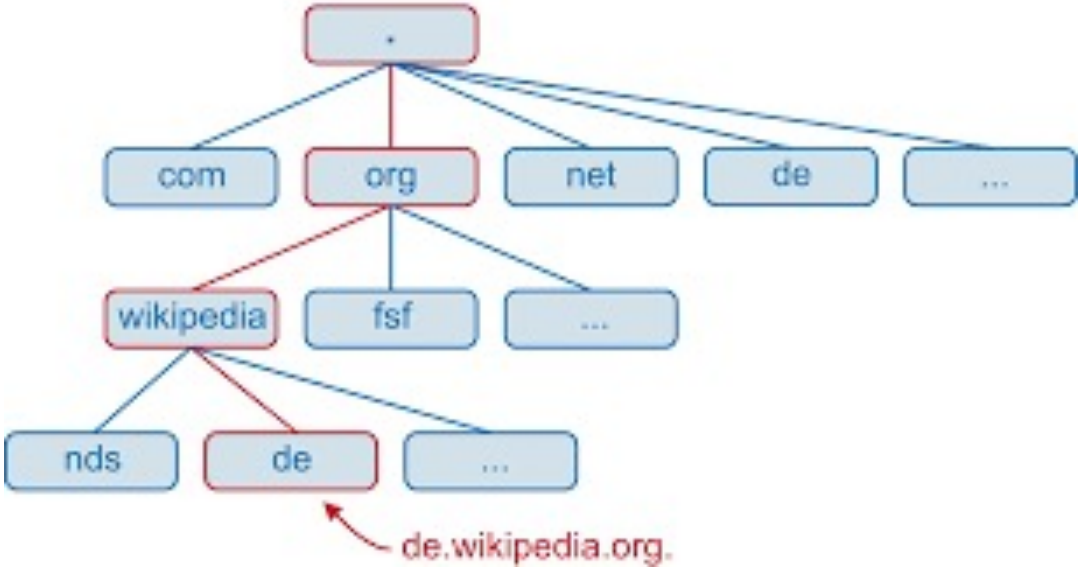


Ejemplos de servicios de nombres

- DNS: traduce nombres de dominio a direcciones IP
- LDAP (*Lighweight Directory Access Protocol*): servicio de directorio que consta de una base de datos con información sobre nombres de personas. Habitualmente almacena información de autenticación (usuario y contraseña)
- *portmapper*: servidor de nombres utilizado en RPC. Obtiene el puerto asociado a servicios RPC registrados
- *rmiregistry*: servicio de registro de objetos remotos en Java

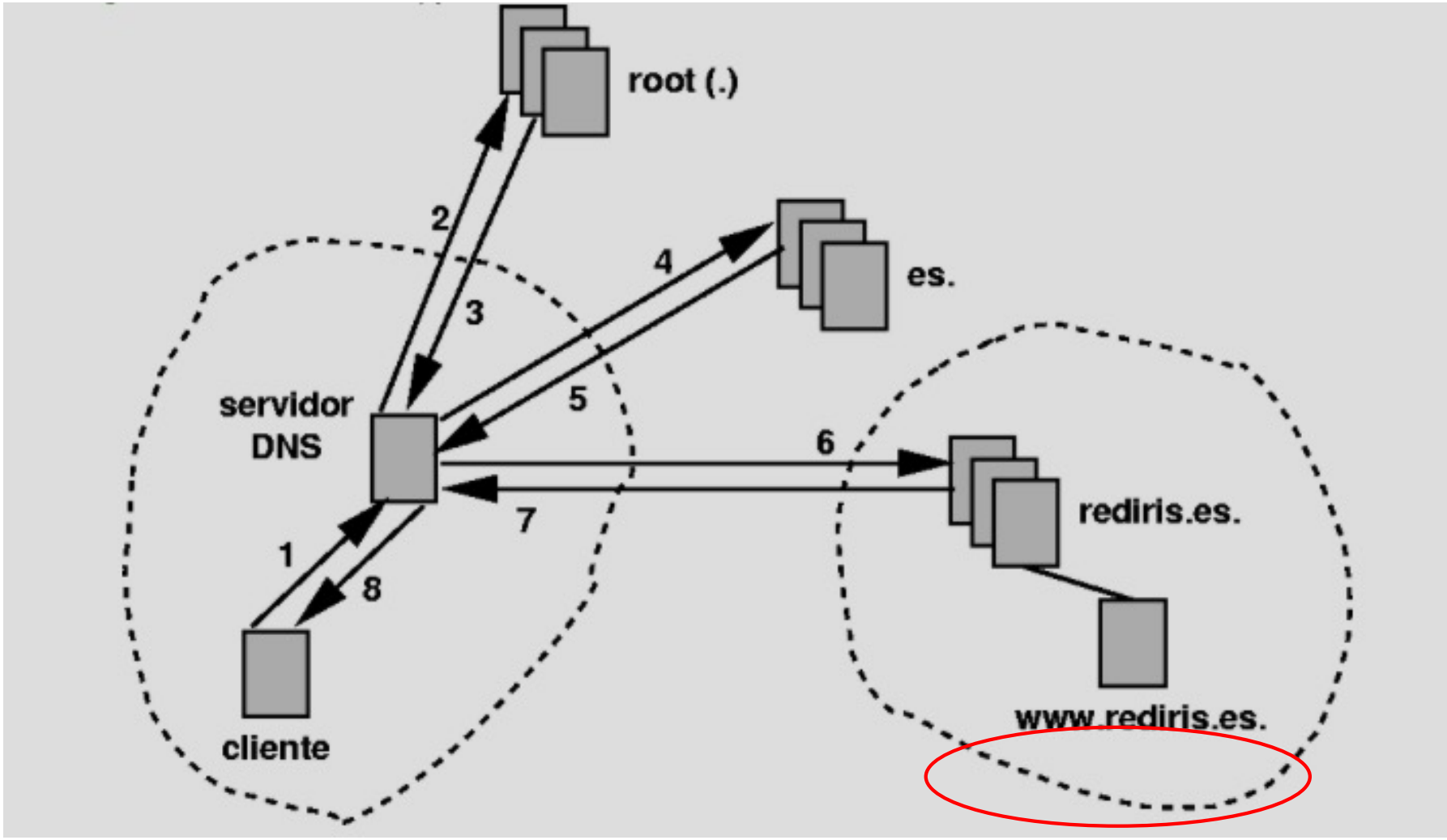
DNS

- Base de datos jerárquica que almacena información sobre nombres de dominio



Fuente: Dns-raum.svg by Wikimedia Commons, CC BY-NC-SA 2.0

Resolución DNS



Aspectos de implementación

- Almacenamiento de la información: servicio de directorios, base de datos
- Migración del servicio de nombres
- Replicación del servicio
- Cache de la información en los clientes
- Localización del servicio de nombres