Tema 7
Coordinación y sincronización distribuida

F. García-Carballeira, Mª. Soledad Escolar,
Luis Miguel Sánchez, Fco. Javier García

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*

Sincronización en sistemas distribuidos

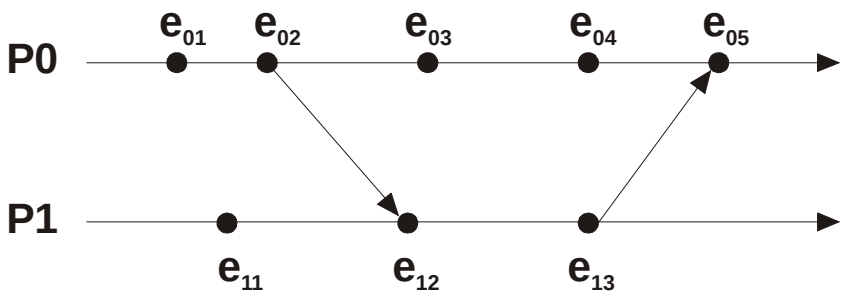
- Más compleja que en los centralizados ya que usan algoritmos distribuidos
- Los algoritmos distribuidos deben tener las siguientes propiedades:
  - La información relevante se distribuye entre varias máquinas
  - Los procesos toman las decisiones sólo en base a la información local
  - Debe evitarse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

Tiempo y distribución

- 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

Modelo del sistema

- 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
- Diagramas espacio-tiempo

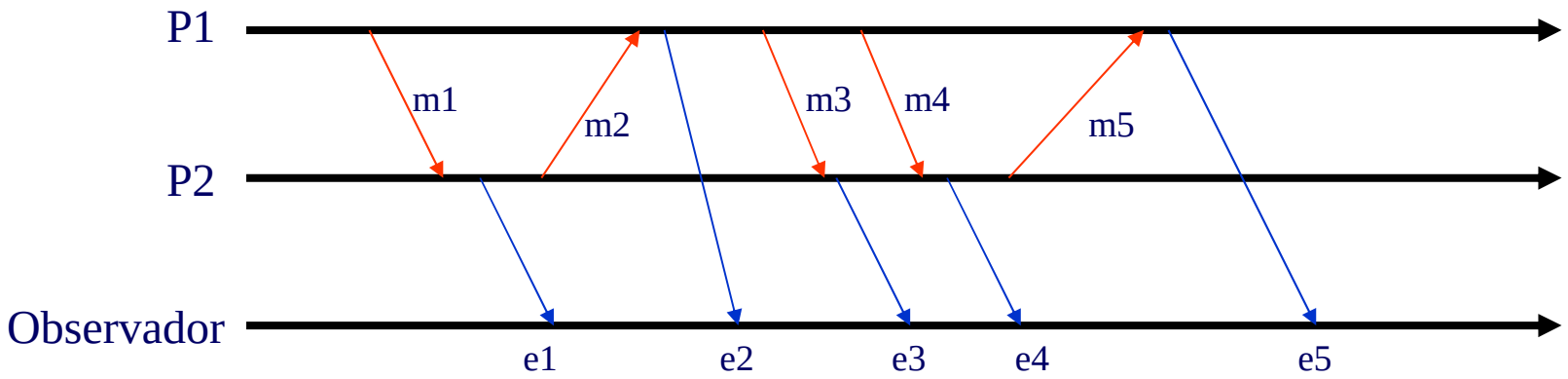


Modelos síncronos y asíncronos

- **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

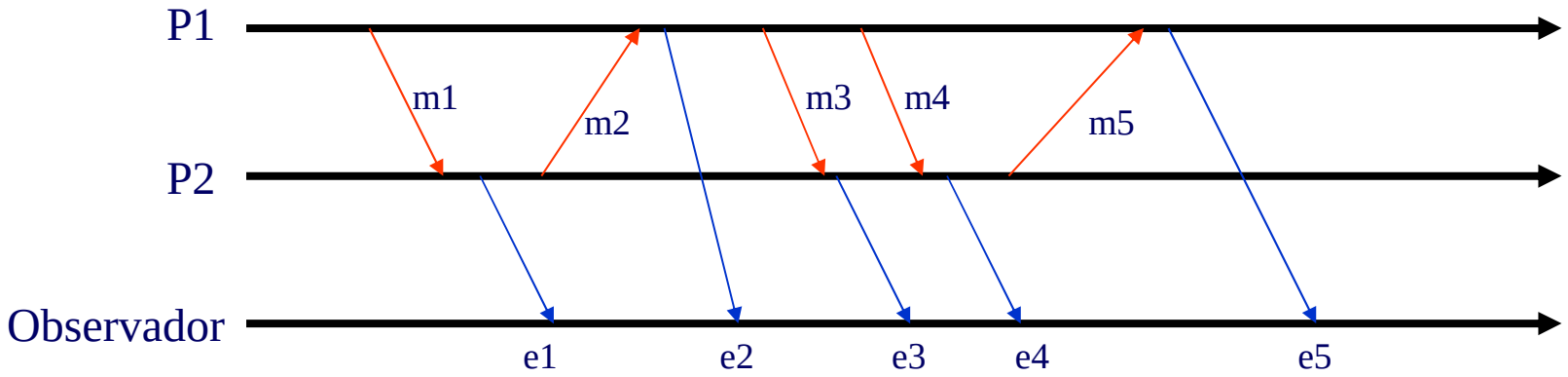
Ejemplo

- 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



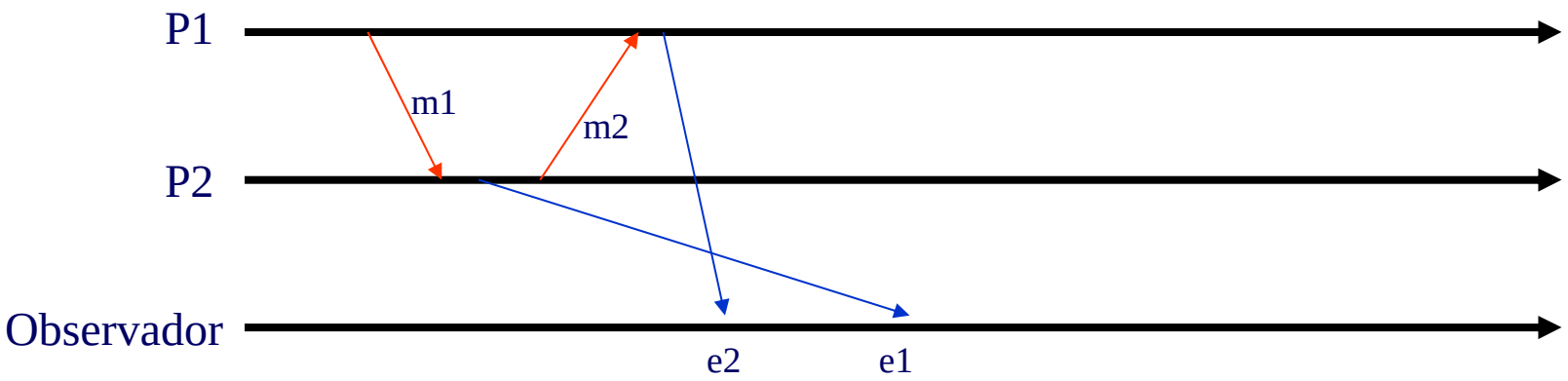
Ejemplo

- 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



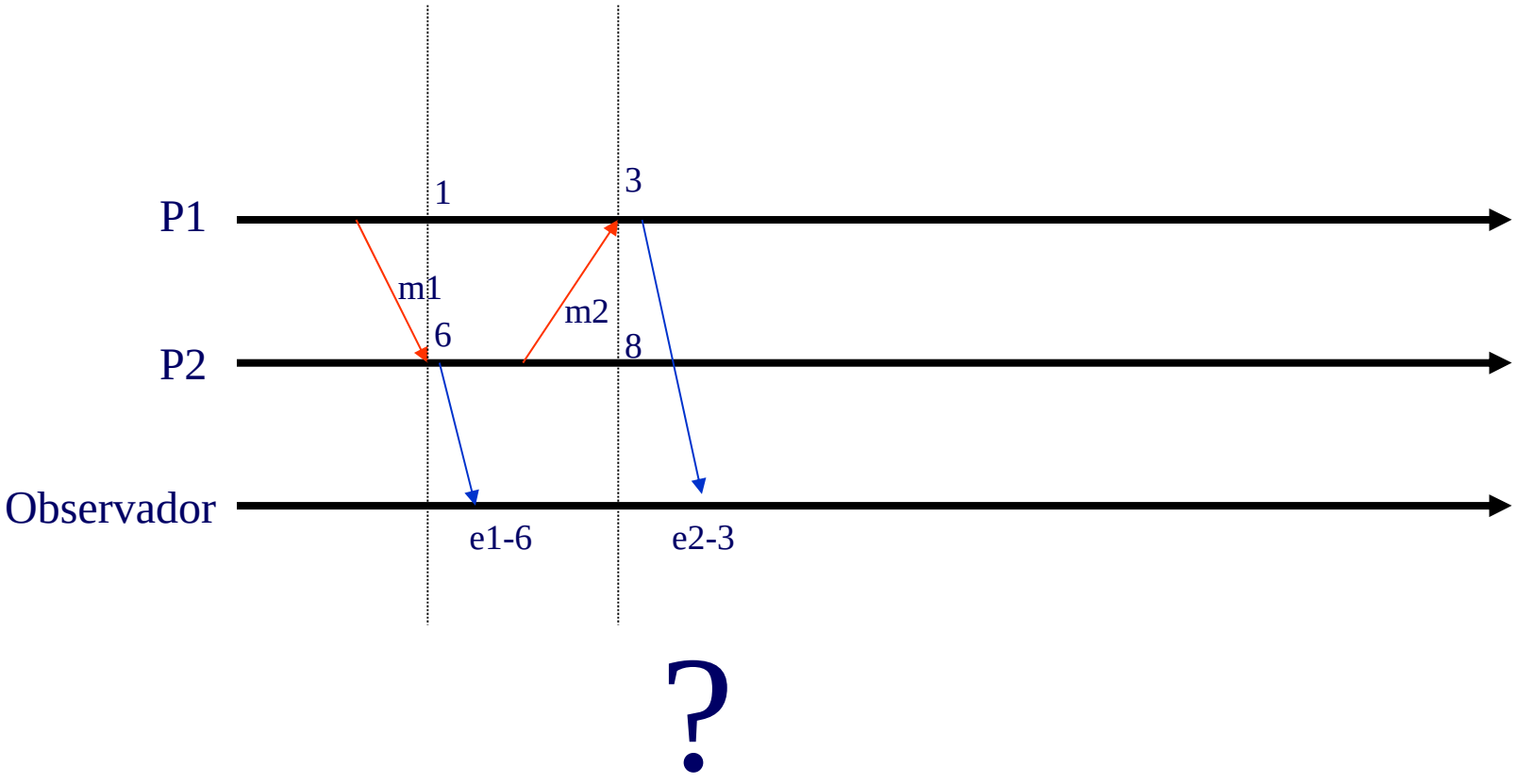
- Para ordenar eventos podemos asignarles marcas de tiempo
  - ei → ek ⇔ MT(ei) < MT (ek)

¿Marcas de tiempo en el observador?

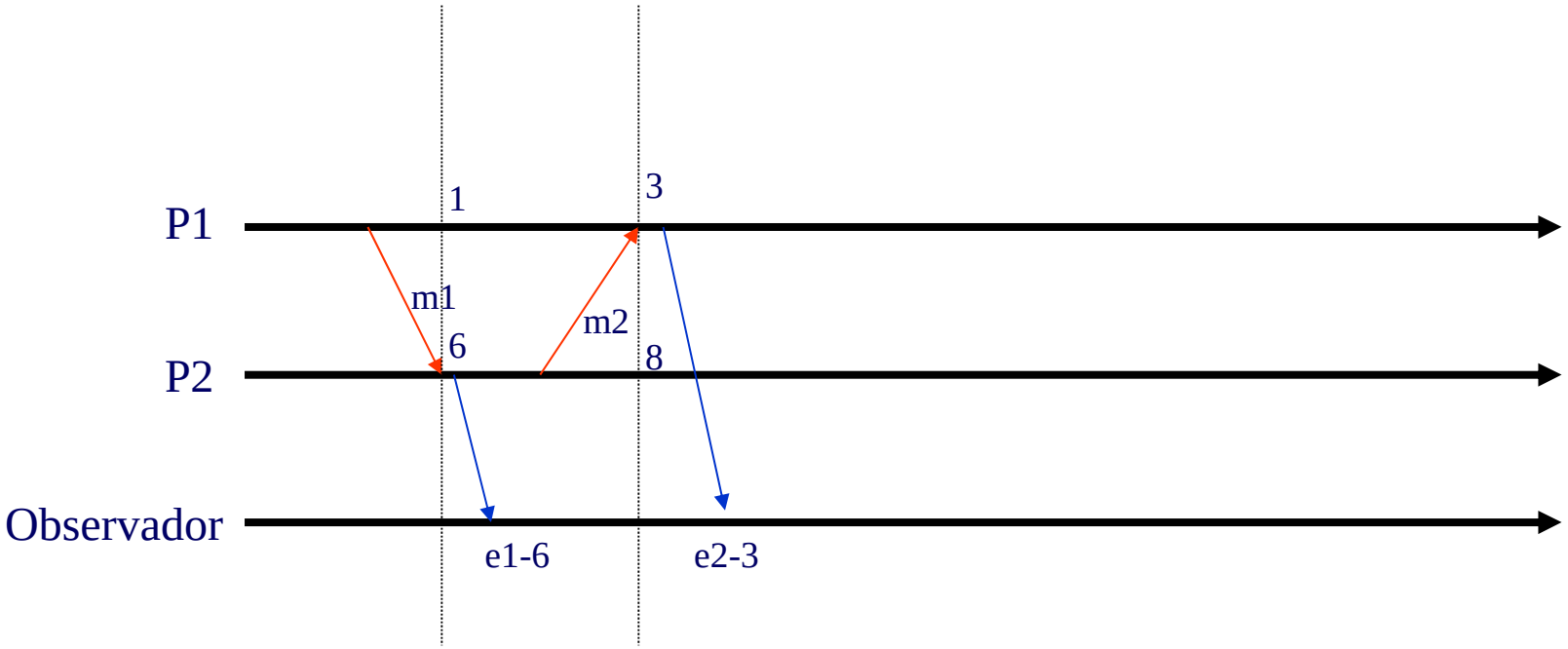


?

¿Marcas de tiempo en los procesos?

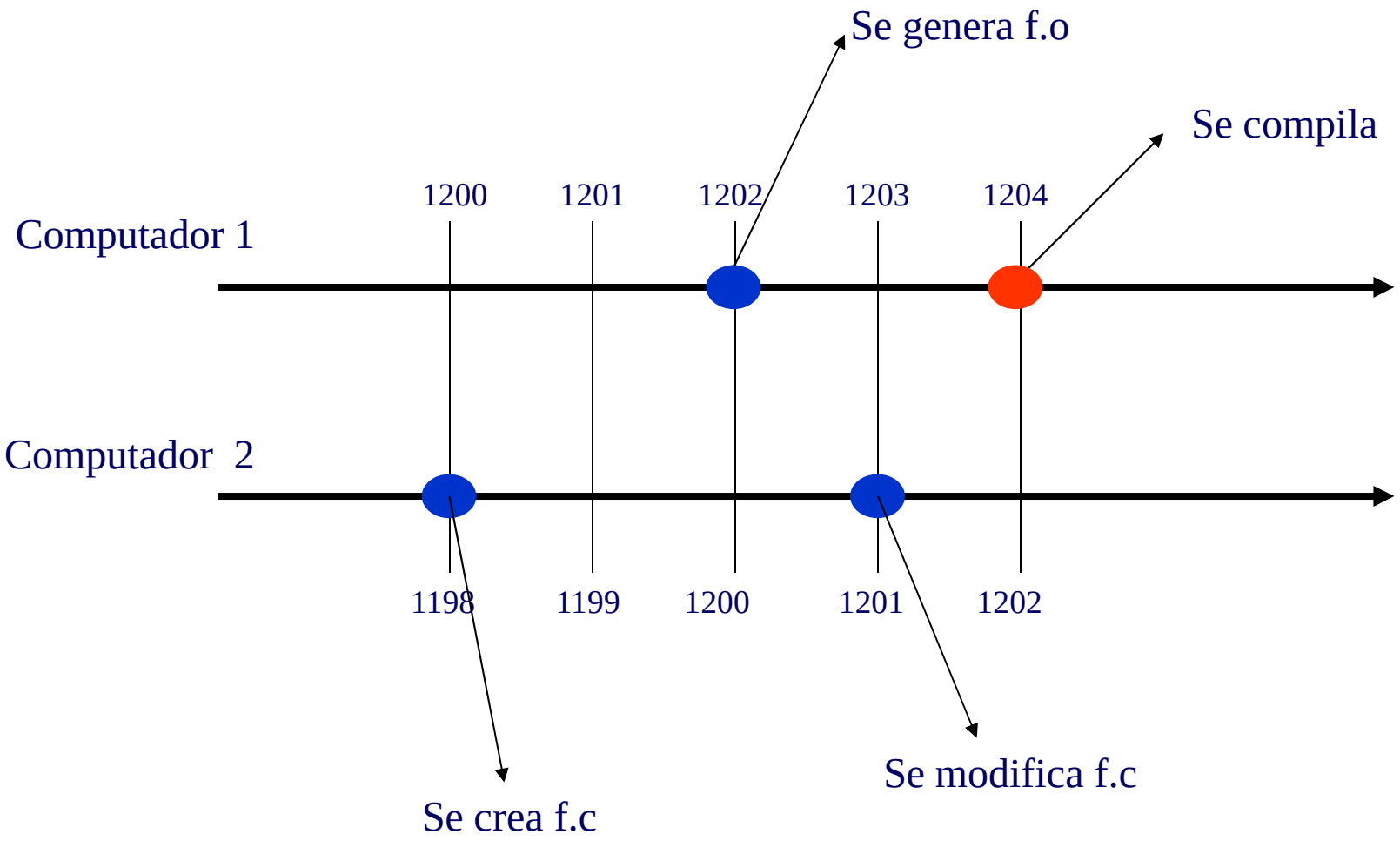


¿Marcas de tiempo en los procesos?



Los relojes deben estar sincronizados

Ejemplo 2: make



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

Ejemplo

- `int gettimeofday (struct timeval *tp,`
  `struct timezone *tzp)`
  - Devuelve el número de segundos y microsegundos transcurridos desde 1 de Enero de 1970

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

- **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
  $\Rightarrow$

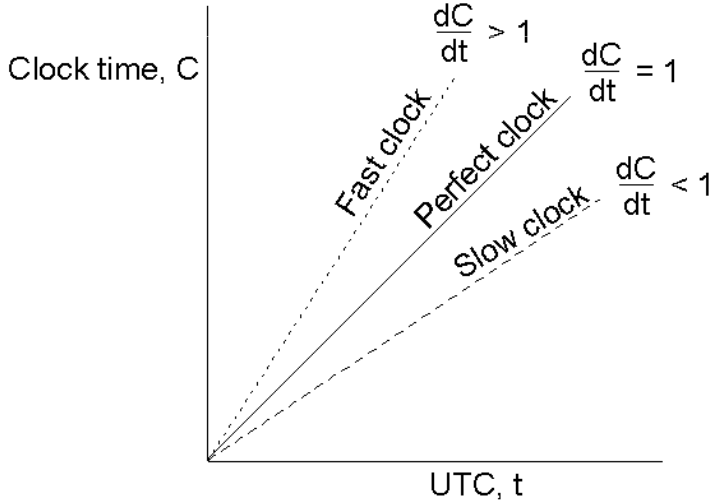
Corrección de relojes

- Corrección HW:
  - Un reloj HW es correcto si su deriva p está acotada
    - Ej: 10^{-6} segundos/segundo
  - $(1 - p)(t' - t) \leq H(t') - H(t) \leq (1 + p)(t' - t)$

- Monotónico
  - $t' > t \Longrightarrow C(t') > C(t)$
  - Un reloj nunca se puede retrasar
  - Solo se puede ralentizar las actualizaciones por SW:
    - $C_i(t) = a\, H_i(t) + b$ (cambiando a y b)

Métodos de sincronización de relojes

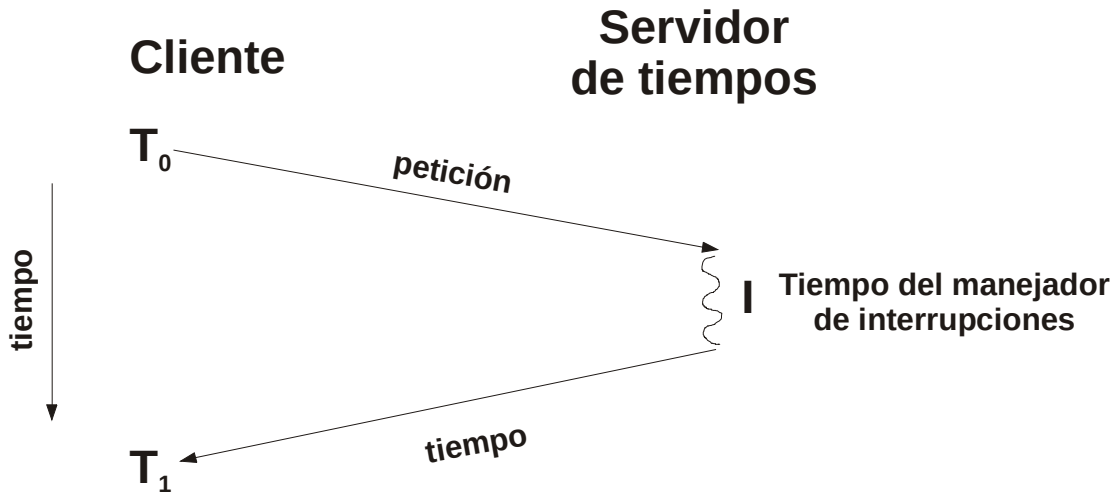
- Sincronización en un sistema síncrono
- **Algoritmo** de Cristian
- **Algoritmo** de Berkeley
- *Network time protocol*



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 variar o 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 u/2 ≤
- El problema es que en un sistema asíncrono $T_{transmit}$ no está acotado

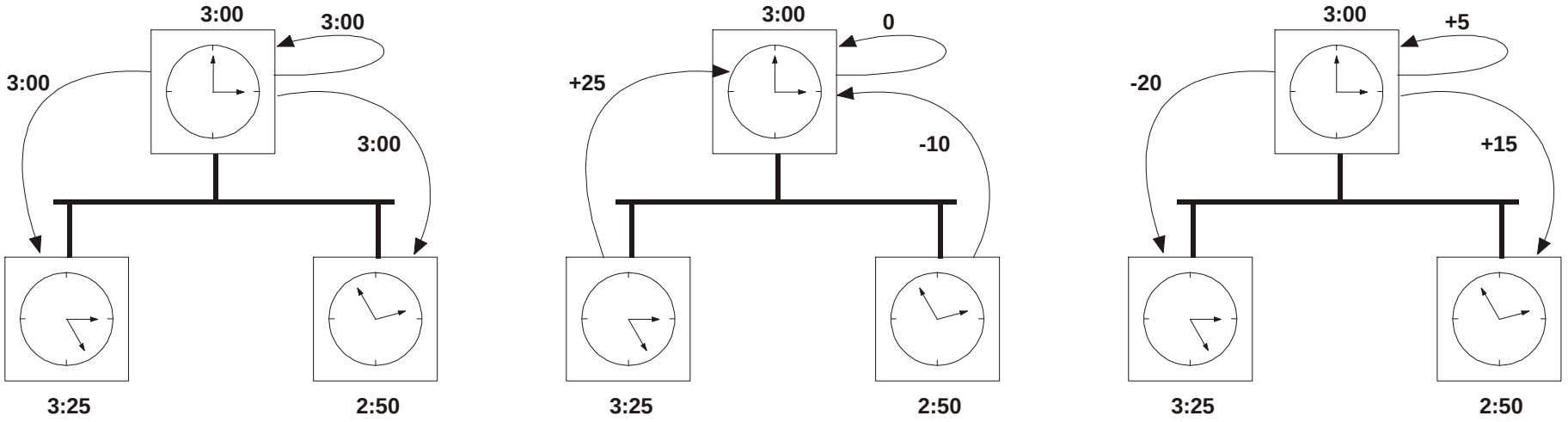
Algoritmo de Cristian



- **Tiempo de transmisión** del mensaje: $(T_1-T_0)/2$
- **Tiempo en propagar** el mensaje: $(T_1-T_0-I)/2$
- El valor t que devuelve el servidor se puede incrementar en el $(T_1-T_0-I)/2$. El valor en el cliente será $t + (T_1-T_0-I)/2$
- Para mejorar la precisión se pueden hacer varias mediciones y descartar cualquiera en la que T_1-T_0 exceda de un límite

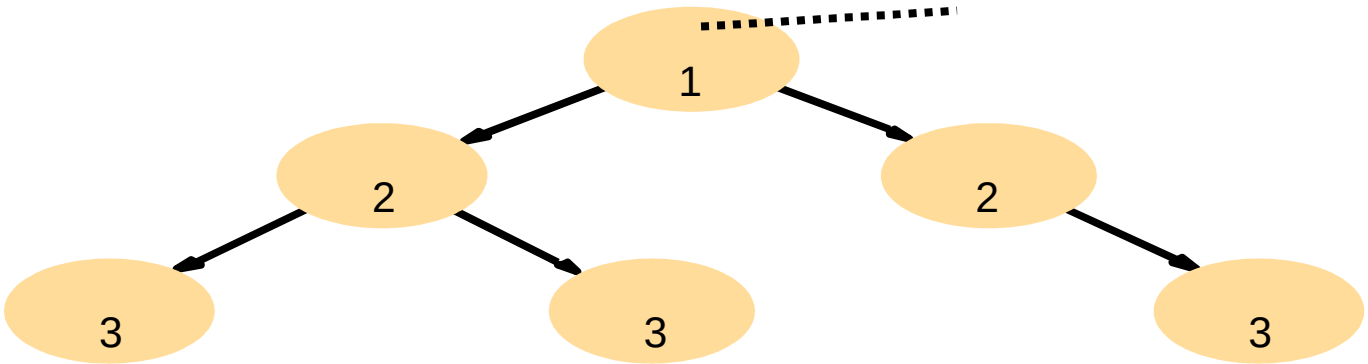
Algoritmo de Berkeley

- 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)

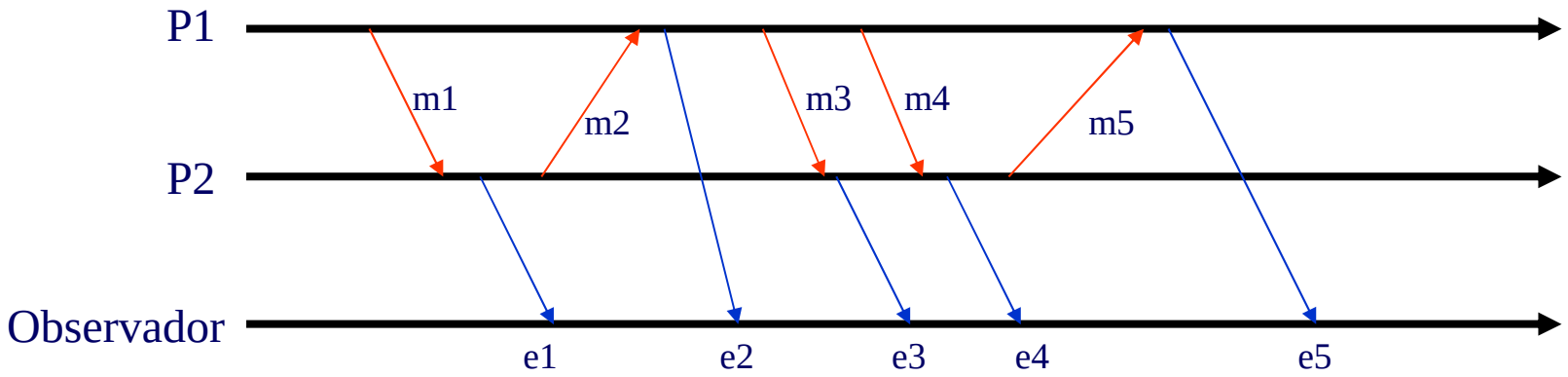
- Servicio para sincronizar a máquinas en Internet con el UTC
- 3 modos de sincronización
  - multicast: para redes LAN de alta velocidad
  - RPC: similar al algoritmo de Cristian
  - simétrico: entre pares de procesos
- Se utilizan mensajes UDP



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?



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

- Lamport demuestra que es posible sincronizar todos los relojes lógicos para obtener un estándar de tiempo único sin ambigüedades

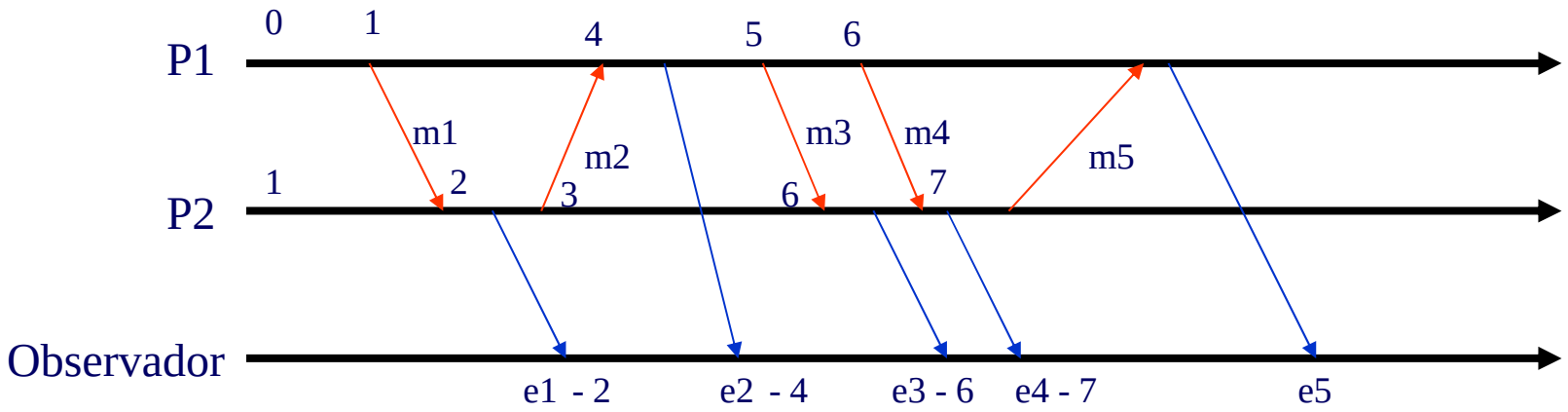
Importancia de la causalidad potencial

- 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



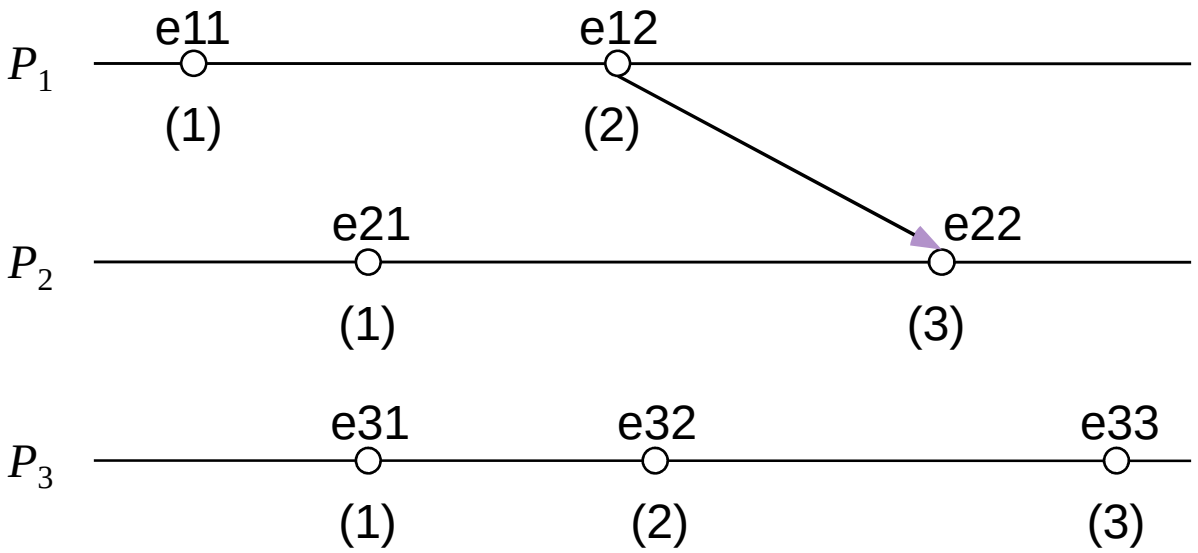
Relojes lógicos totalmente ordenados

- 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



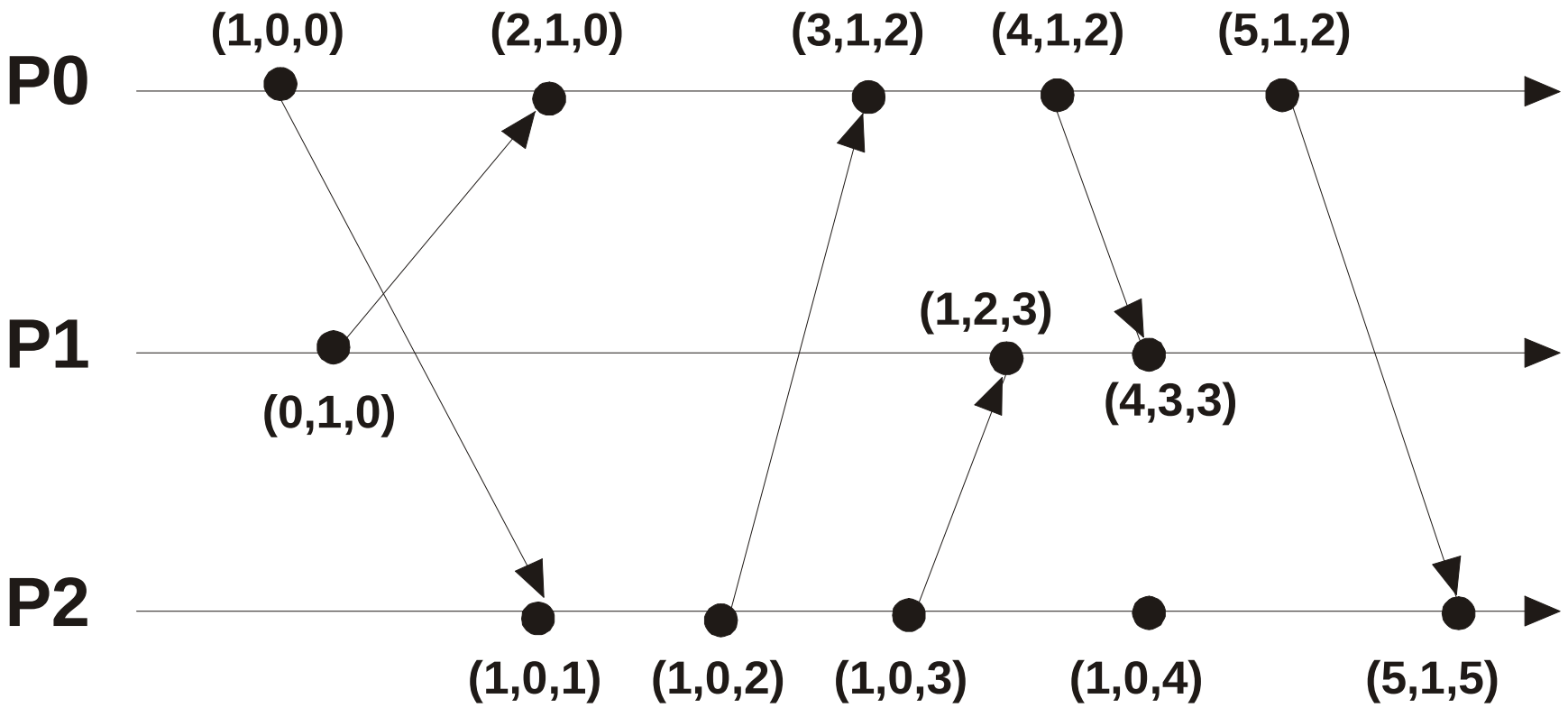
$C(e11) < C(e22)$, y $e11 \rightarrow e22$ es cierto

$C(e11) < C(e32)$, pero $e11 \rightarrow e32$ 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 RVi
    - $RV_j = max(RV_j, RVi)$ (componente a componente)
    - $RV_j[j] = RV_j[j] + 1$ (evento de recepción)

Relojes vectoriales



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)

Coordinación y consenso

- Exclusión mutua distribuida
- Algoritmos de elección
- Comunicación multicast

Exclusión mutua distribuida

- Los procesos ejecutan el siguiente fragmento de código

  entrada()

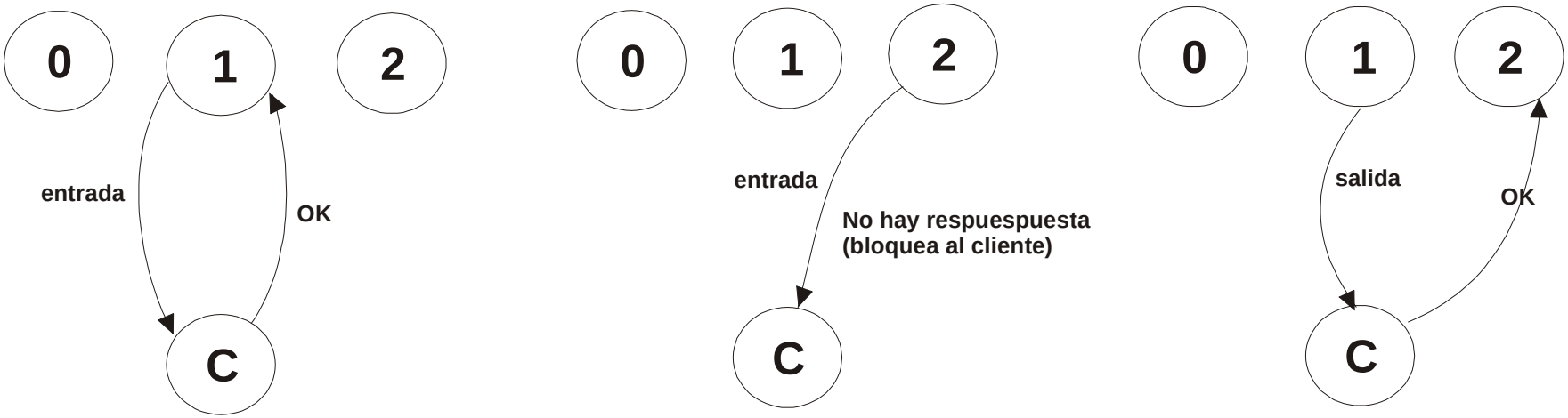
  SECCIÓN CRÍTICA

  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

- Existe un proceso coordinador



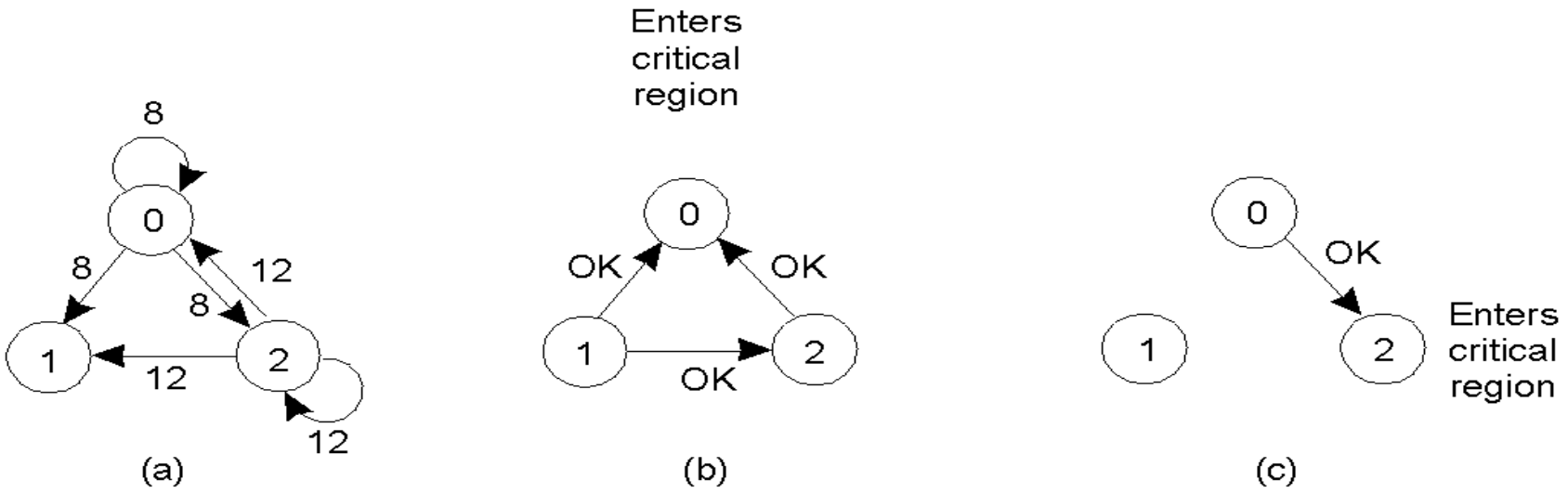
@Fuente: *Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez.* **Mc Graw Hill**

Algoritmo distribuido de Ricart y Agrawala

- Algoritmo de **Ricart** y **Agrawala** requiere la existencia un orden total de todos los mensajes en el sistema
- Un proceso 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
- 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. En caso contrario entra y no envía nada.
- Cuando un proceso recibe todos los mensajes puede entrar

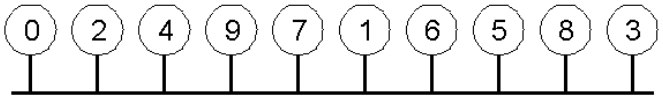
Ejemplo de algoritmo distribuido

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

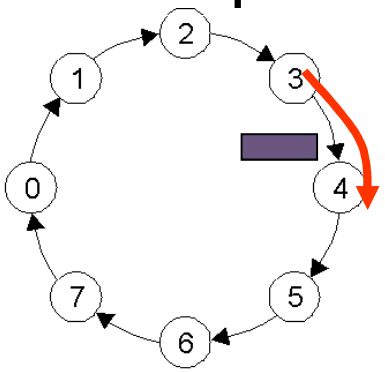


Anillo con testigo

- 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



(a)



(b)

Comparación de los algoritmos

- Centralizado
  - Mensajes: 3
  - Retardo: 2
  - Problemas: fallo del coordinador
- Distribuido
  - Mensajes: 2(n-1)
  - Retardo: 2(n-1)
  - Problemas: fallo de cualquier proceso
- Anillo con testigo
  - Mensajes: 1 a n-1
  - Retardo: 1 a n-1
  - Problemas: pérdida del testigo, fallo en un proceso

Algoritmos de elección

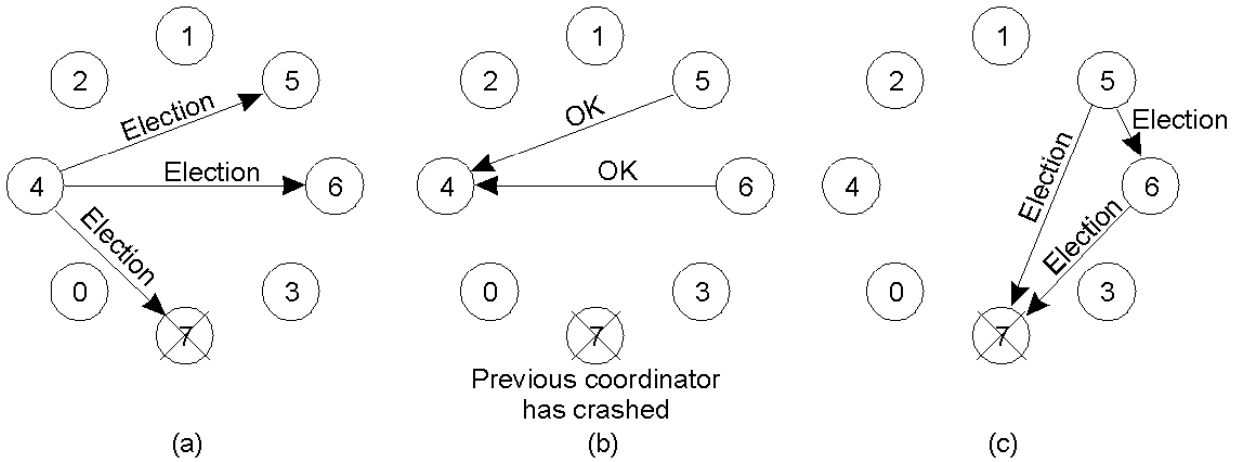
- Útil en aplicaciones donde es necesario la existencia de un coordinador
- El algoritmo debe ejecutarse cuando **falla el coordinador**
- Algoritmos **de elección**
  - Algoritmo del matón
  - Algoritmo de anillo
- El objetivo de los algoritmos es que la elección sea única aunque el algoritmo se inicie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

Algoritmo del 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

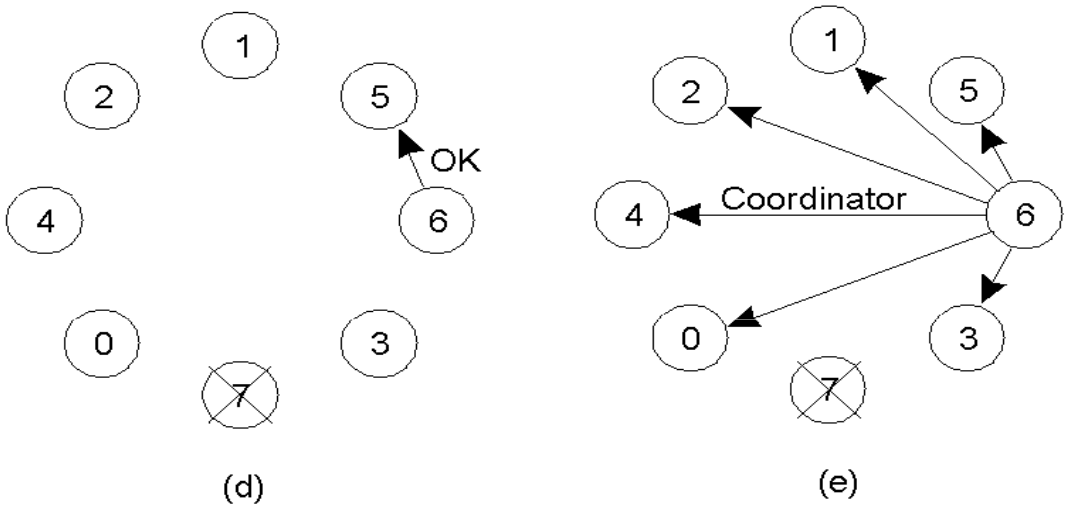
Algoritmo del matón. Ejemplo

- 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



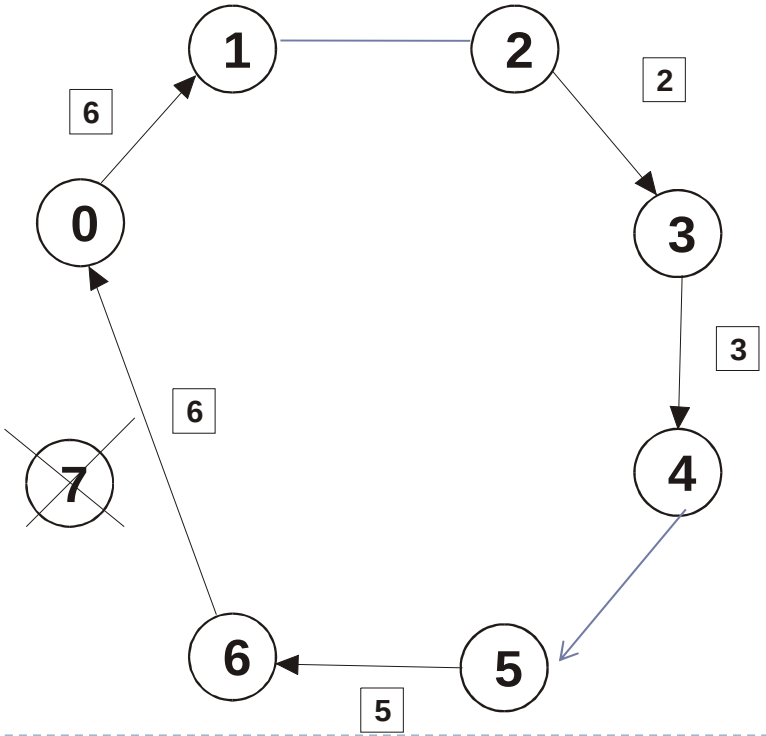
d) Proceso 6 dice a 5 que pare
e) Proceso 6 indica a todos que es el coordinador

Algoritmo 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

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



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

Trabajo práctico

- Implementar un semáforo distribuido utilizando semáforos POSIX. Emplear un algoritmo centralizado que se encarga de gestionar los semáforos

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

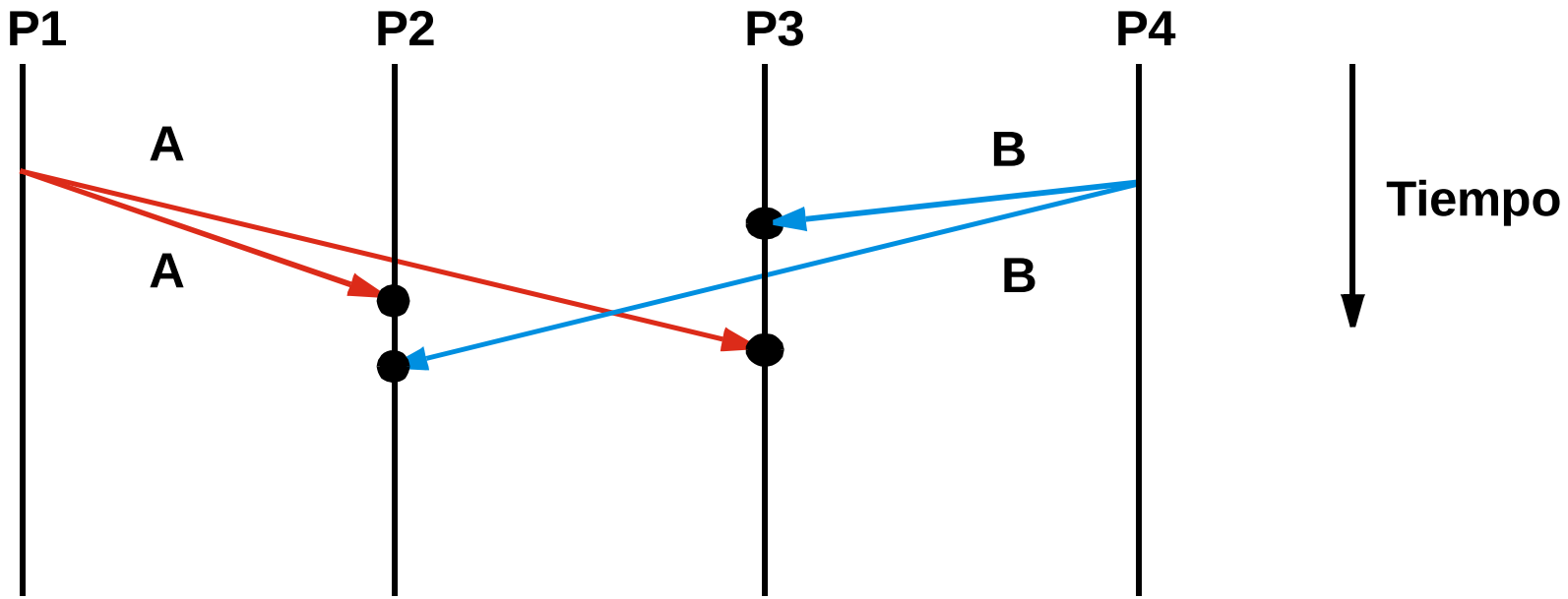
- Implementación de operaciones multicast:
  - Mediante operaciones **punto a punto**
  - Mecanismo poco fiable

- 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

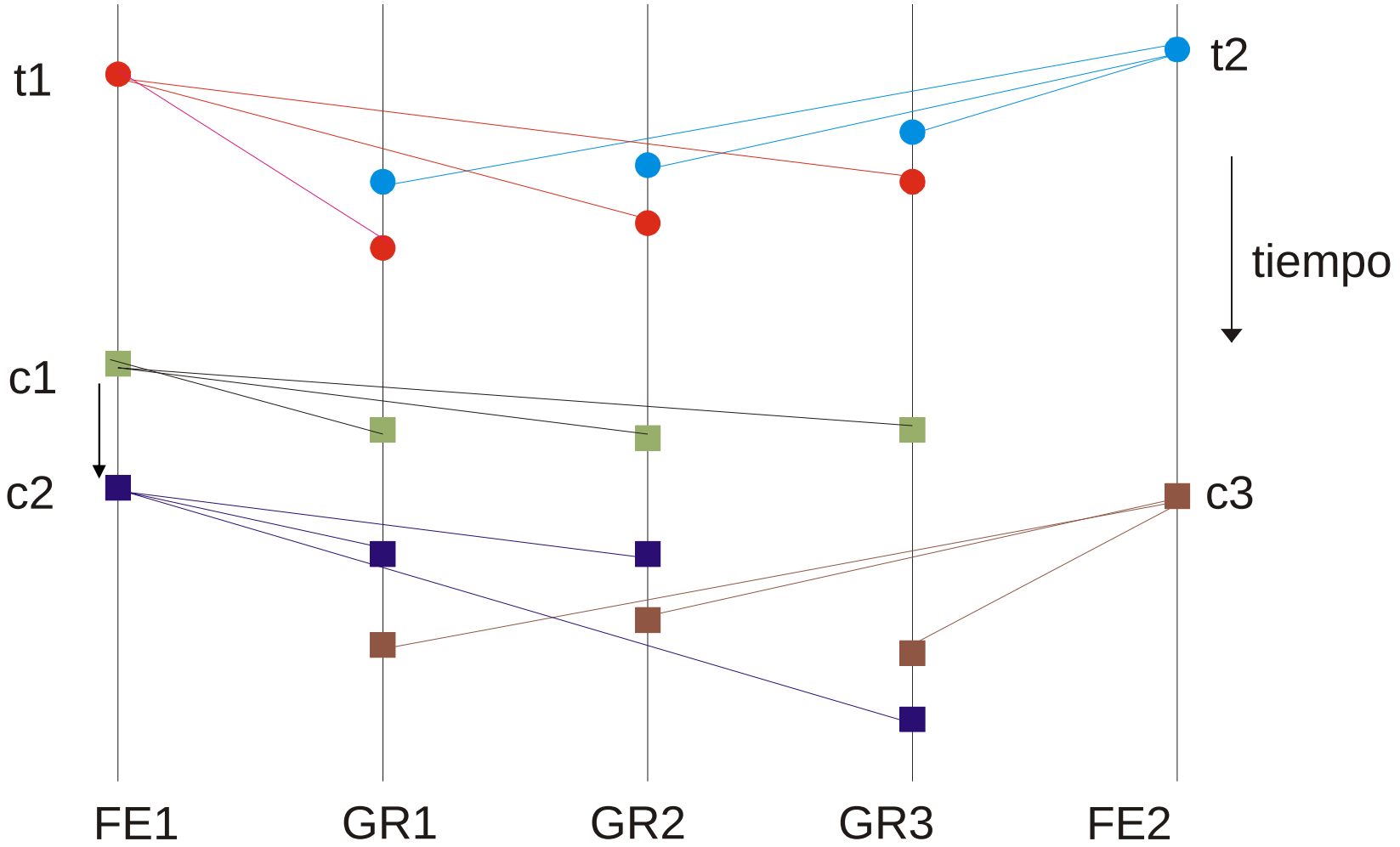


Ordenación de las actualizaciones

- Es importante el orden en el que se realizan las peticiones

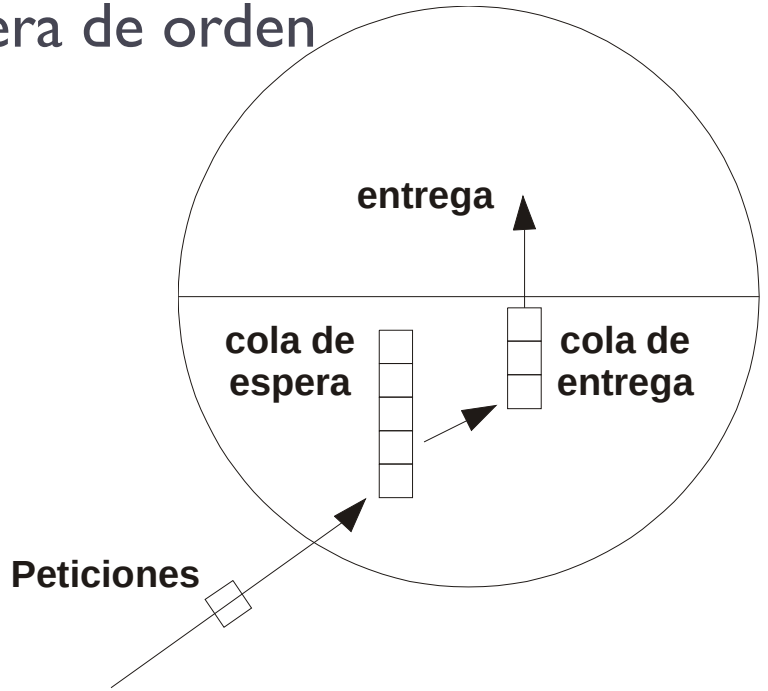
  ¿Qué ocurre en un sistema asíncrono cuando un cliente modifica un dato y más tarde otro cliente quiere consultar ese dato?
- Algunas aplicaciones requieren un orden en la realización de las peticiones
- **Orden total:** dadas dos peticiones r_1 y r_2 entonces o r_1 es procesada en todos los procesos 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



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

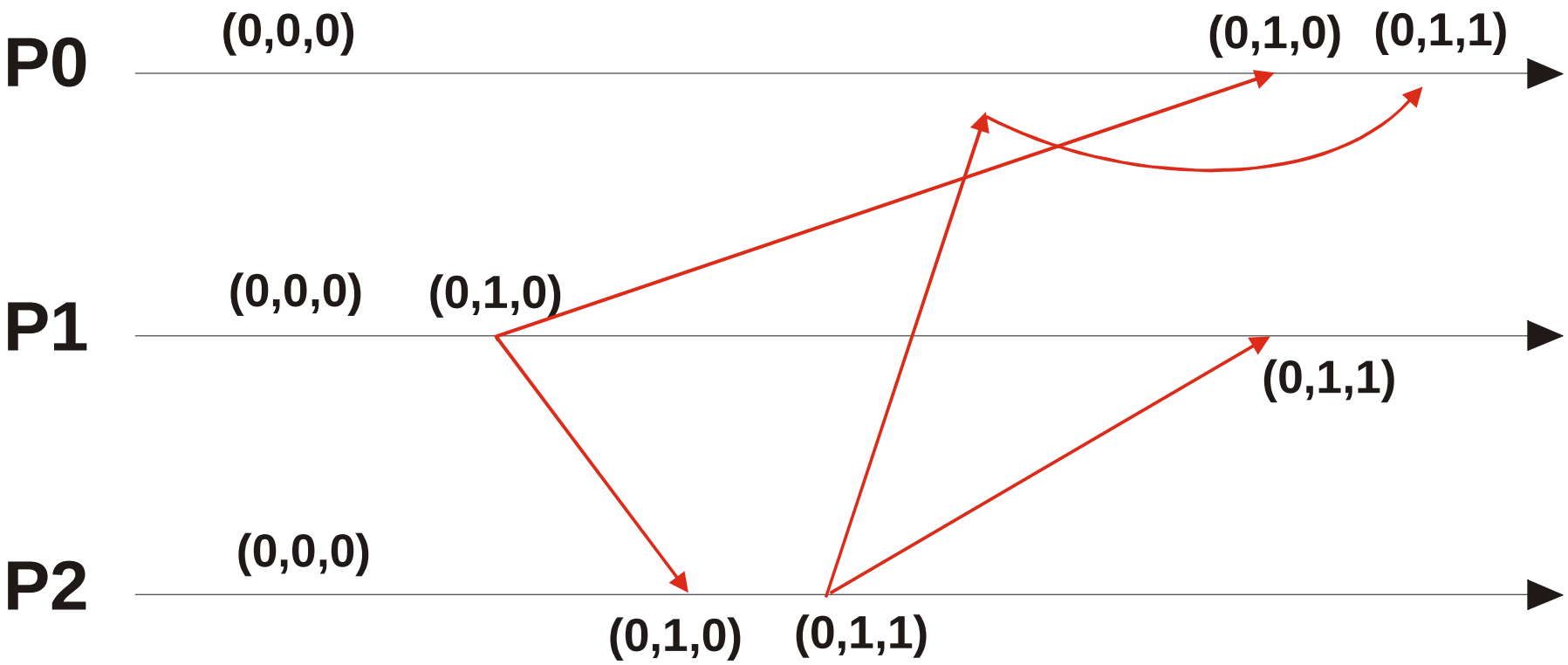
Método distribuido

- Un emisor A asigna un **número secuencial al mensaje** y lo envía a todos los procesos del grupo (un número mayor que el anterior)
- Cada miembro del grupo elige su propia marca:
  - Mayor que cualquier otra marca que hubiese enviado o recibido, le incrementa i y le envía de regreso a A.
- Cuando A recibe todas las marcas elige la mayor y envía un mensaje a todos los miembros
- Cada miembro entregará los mensajes según el orden de las marcas de tiempo

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?
  - 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