UD 9. Algoritmos distribuidos

Concurrencia y Sistemas Distribuidos

Objetivos de la Unidad Didáctica

- Conocer las **características** de los algoritmos distribuidos y ejemplos relevantes de los mismos.
- Comprender las dificultades de la **sincronización temporal**, al no existir un reloj único que todos los nodos puedan consultar.
- Comprender la **gran variedad** de problemas algorítmicos que surgen en sistemas distribuidos, identificando algunos de los más relevantes: consenso, elección de líder, exclusión mutua.
- Conocer algunos **algoritmos** a nivel básico, estudiando su funcionamiento, sus características, su complejidad y coste y el hecho de si toleran fallos o no.
- Identificar los fallos en los nodos como la mayor fuente de **complejidad** en el diseño de algoritmos distribuidos.

Algoritmos Distribuidos

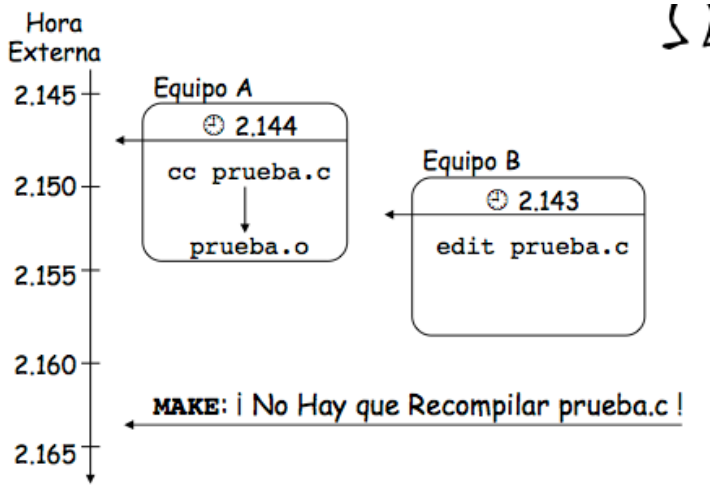
- Conceptos fundamentales algorítmicos
  - Independientes de tecnologías particulares
  - Características de los algoritmos distribuidos **descentralizados**:
    1. Ningún nodo posee toda la información completa del sistema
    2. Se ejecutan en procesadores independientes
       - El fallo de un nodo no impide que el algoritmo progrese
    3. Toman decisiones basadas en información local
    4. No hay una fuente precisa de tiempo global

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Dificultades en la sincronización temporal

- **Sincronización**: mecanismos que faciliten trabajo coordinado.
- En los sistemas distribuidos es más compleja que en los sistemas centralizados. Ejemplos:
  - Acceso a recurso compartido.
  - Ordenación de eventos



Caracterización del problema

Caracterización del problema → ***Sincronización de relojes físicos:***

- Cada nodo *i* dispone de un reloj local C_i
  - Representa un valor de tiempo universal coordinado (UTC)
- Dado cualquier instante real t
  - El objetivo es que para todos los nodos, $C_i(t)=t$
  - Es decir, que todos los relojes locales tengan la misma hora y ésta coincida con la hora "verdadera"
- Problema: los chips de reloj en que se basan los relojes C_i no son absolutamente exactos
  - Ejemplos de **tipos de relojes**:
    - Relojes basados en **cristales de cuarzo**
      - Drift (Deriva) 10^{-6} segundos por cada segundo real
    - Relojes de "**alta precisión**" → Drift 10^{-7} a 10^{-8} segundos
    - Relojes **atómicos** (los más precisos) → Drift 10^{-13}

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Algoritmos de sincronización de relojes

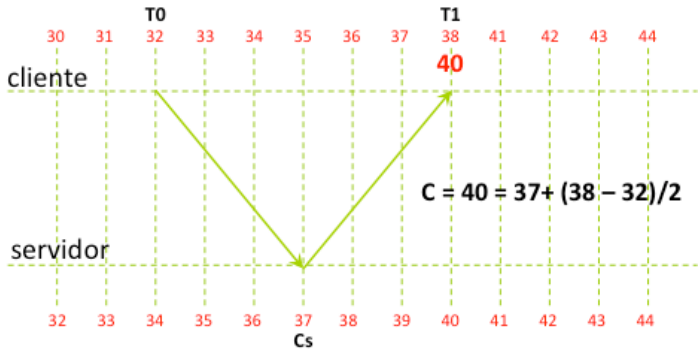
- **Algoritmos de sincronización de relojes físicos:**
  - **Algoritmo de Cristian** (F. Cristian, 1989)
  - **Algoritmo de Berkeley** (Gusella & Zatti, 1989)
  - Network Time Protocol – NTP (1995)
    - Cristian y Berkeley diseñados para redes de baja latencia (p.e. LAN)

Algoritmos de sincronización de relojes: **Algoritmo de Cristian**

- El algoritmo de Cristian permite sincronizar el reloj local **Cc** de un ordenador **cliente** con el reloj local **Cs** de un ordenador **servidor**
- Consideraciones:
  - El ordenador servidor dispone de un reloj muy preciso, posiblemente sincronizado con otros más precisos todavía.
    - Ej: sincronizado con una hora UTC
  - Los relojes no deben retroceder.
  - La sincronización requiere intercambio de mensajes, pero el tránsito de estos mensajes por la red consume tiempo.

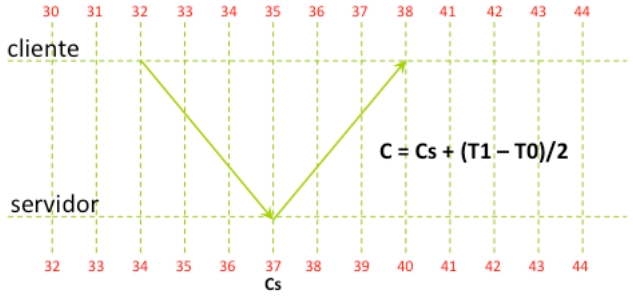
Algoritmos de sincronización de relojes: **Algoritmo de Cristian**

- **Algoritmo de Cristian:**
  - El ***cliente*** pide el valor del reloj al *servidor* en el instante **T0** (según **Cc**)
  - El **servidor** contesta con el valor de su reloj **Cs**
  - La respuesta llega al ***cliente*** en T1 (según **Cc**)
  - El cliente calcula **C = Cs + (T1 – T0)/2**
  - El **cliente** actualiza su reloj según:
    - **Si C > Cc**, se fija el valor del reloj del cliente a C, es decir **Cc=C**
    - **Si C < Cc**, se detiene Cc las siguientes **Cc-C** unidades de tiempo
      - se almacena LAG = Cc – C y se descartarán ticks del reloj del cliente hasta que haya transcurrido un tiempo LAG (se evita que retroceda el reloj: se hace que el reloj "pare").



Algoritmos de sincronización de relojes: **Algoritmo de Cristian**

- **Características:**
  - Se asume que el tiempo de envío de ambos mensajes es prácticamente igual
    - En T1 el valor de CS se habrá incrementado en (T1-T0)/2
  - Si uno de los dos mensajes tarda más en ser transmitido, el ajuste no es correcto
    - Generalmente la duración de ambos mensajes es igual
  - En todo caso, la sincronización perfecta es imposible (con este algoritmo o con cualquier otro)
  - El correcto funcionamiento de una aplicación distribuida podrá depender de los valores de los relojes locales, sólo si tolera el margen de error inherente al algoritmo de sincronización utilizado



Algoritmos de sincronización de relojes: **Algoritmo de Berkeley**

- **Algoritmo de Berkeley – Características**
  - Disponemos de un grupo de nodos formado por:
    - Un servidor, denomidado S
    - N clientes, denomidados C_i
  - Cada nodo dispone de su propio reloj local.
  - El **objetivo** es sincronizar los relojes locales de todos los nodos entre sí.
  - De forma periódica, a iniciativa del servidor, todos los nodos sincronizan sus relojes.

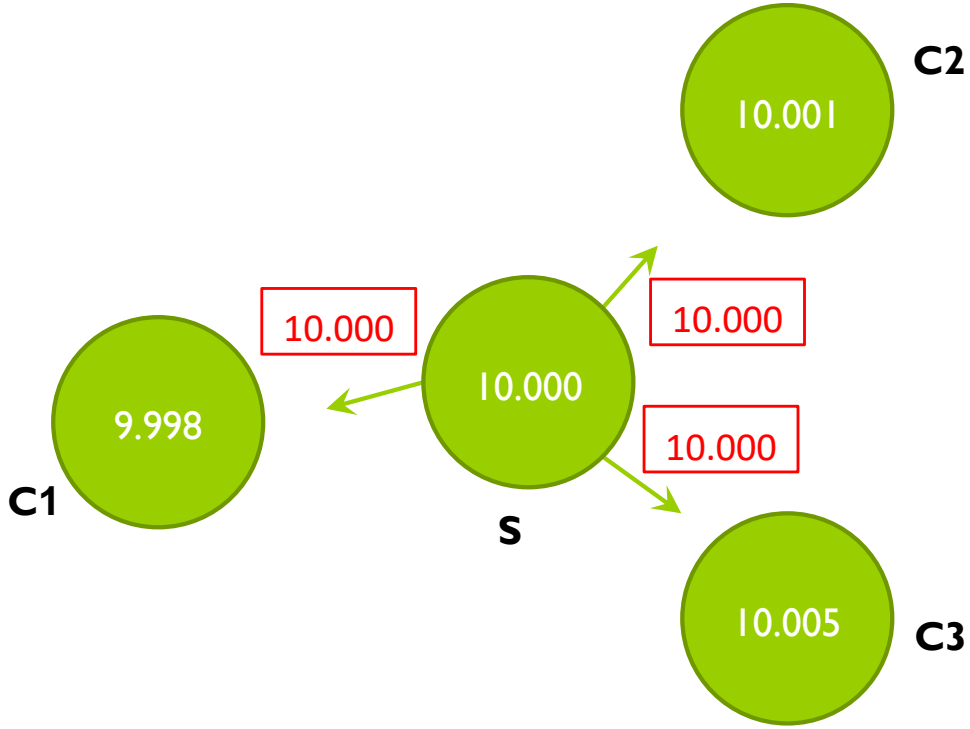
Algoritmos de sincronización de relojes: **Algoritmo de Berkeley**

- **Algoritmo de Berkeley**
  - El servidor difunde el valor de su reloj de manera periódica.
  - Cada cliente calcula la diferencia $\mathbf{D_i}$ entre su reloj local y el valor de reloj que ha notificado el servidor en el mensaje que ha enviado.
  - Cada cliente notifica dicha diferencia D_i al servidor
  - Dadas las respuestas, el servidor:
    1. calcula la diferencia media (incluyendo al servidor)
    2. actualiza su propio reloj,
    3. contesta a todos los clientes con la diferencia que debe aplicar cada uno para actualizar su reloj local.

Ejemplo de Algoritmo de Berkeley

- El servidor difunde su reloj en **T0**

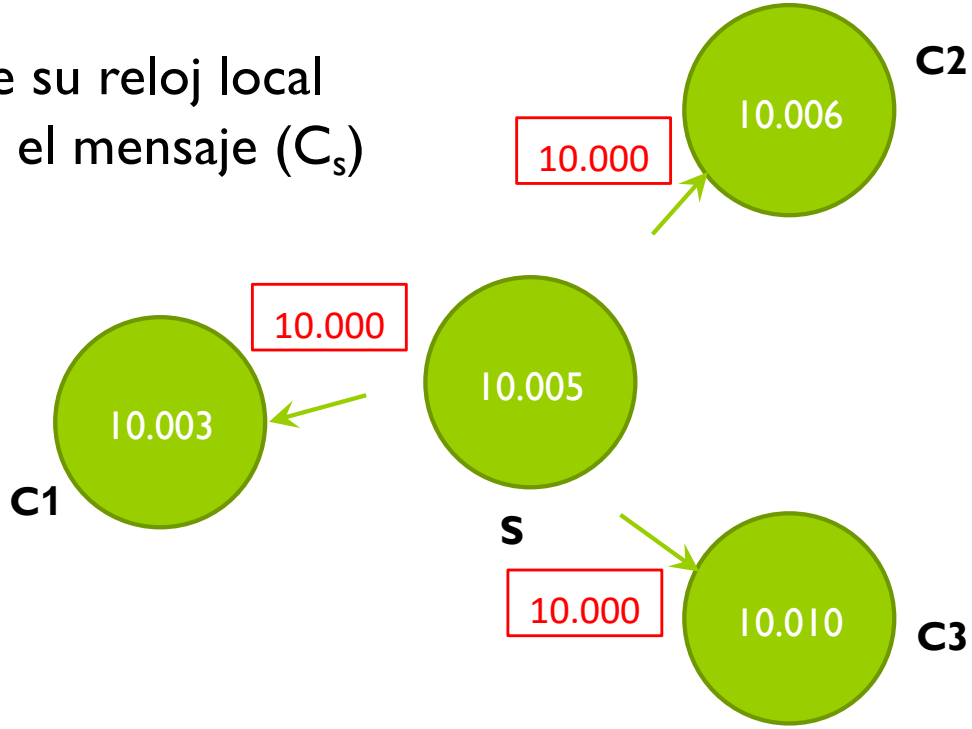
T0	10.000



Ejemplo de Algoritmo de Berkeley

- Los mensajes llegan a los clientes
  - Ejemplo: los mensajes llegan tras 5 unidades de tiempo. *Para simplificar, asumimos que llegan a la vez*
- Cada cliente calcula la diferencia $\mathbf{D_i}$ entre su reloj local (C_c) y el que ha notificado el servidor en el mensaje (C_s)
  - $D_i = C_c - C_s$

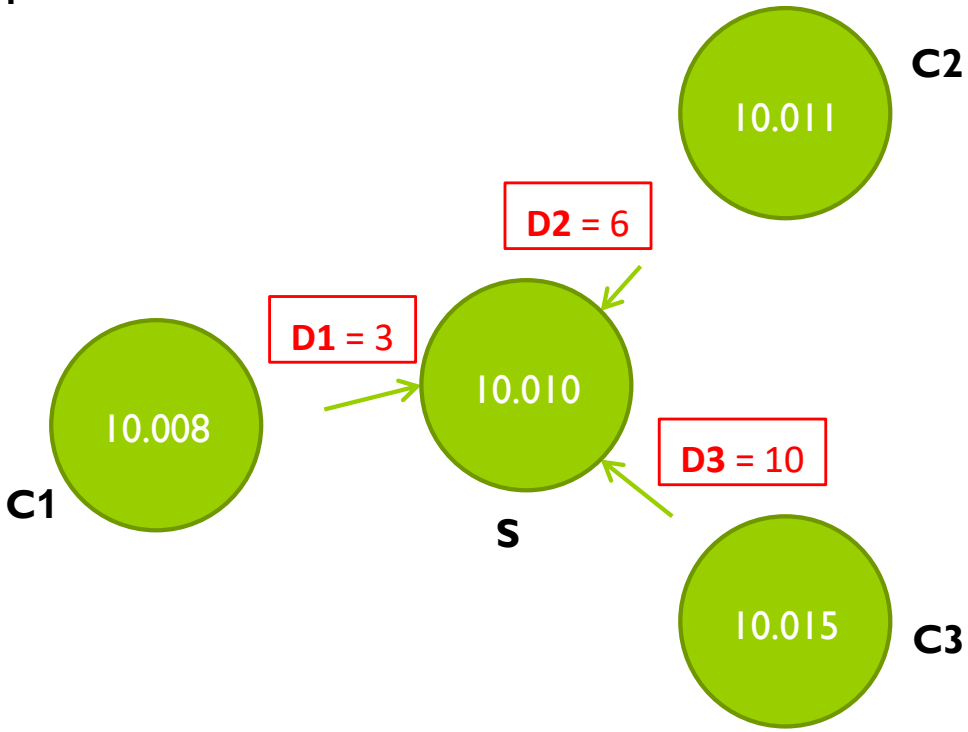
T0	10.000
D1	10.003 - 10.000 = 3
D2	10.006 - 10.000 = 6
D3	10.010 - 10.000 = 10



Ejemplo de Algoritmo de Berkeley

- Y cada cliente notifica dicha diferencia ($\mathbf{D_i}$) al servidor
- La respuesta de cada cliente llega en $\mathbf{T1_i}$
  - *Los mensajes llegan tras 5 unidades de tiempo*
  - *Para simplificar, asumimos que llegan a la vez*

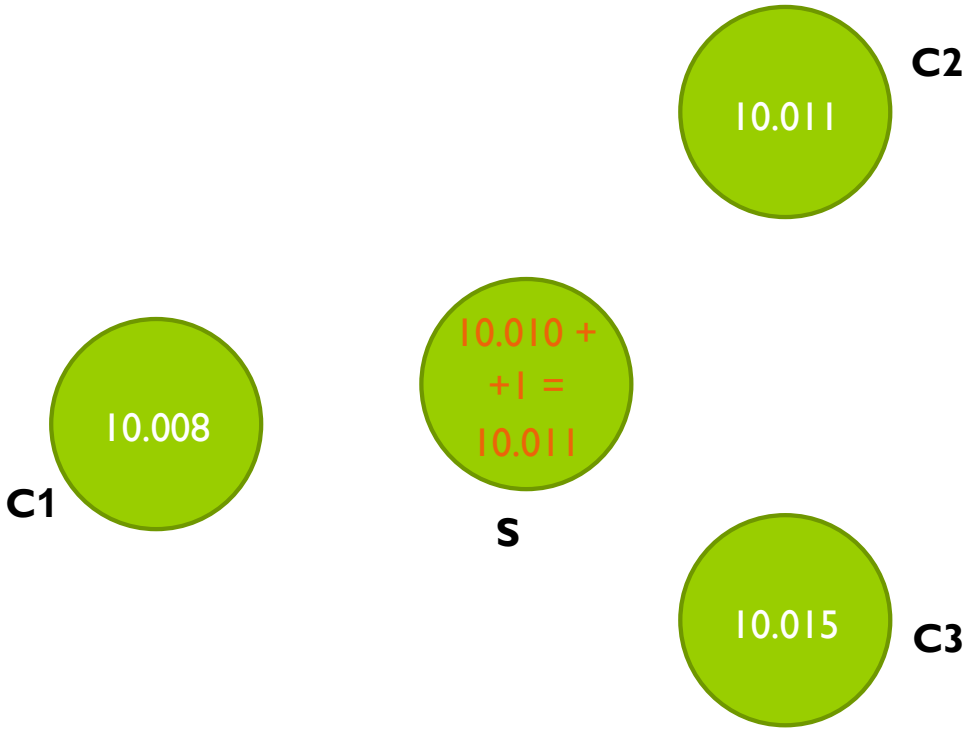
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**



Ejemplo de Algoritmo de Berkeley

- El servidor ajusta la diferencia notificada $D_i' = D_i - (T1_i - T0)/2$
- Y calcula la diferencia media, incluyendo al servidor
  $D = \sum D_i' / (N+1)$
- El servidor ajusta su reloj incrementándolo en **D**

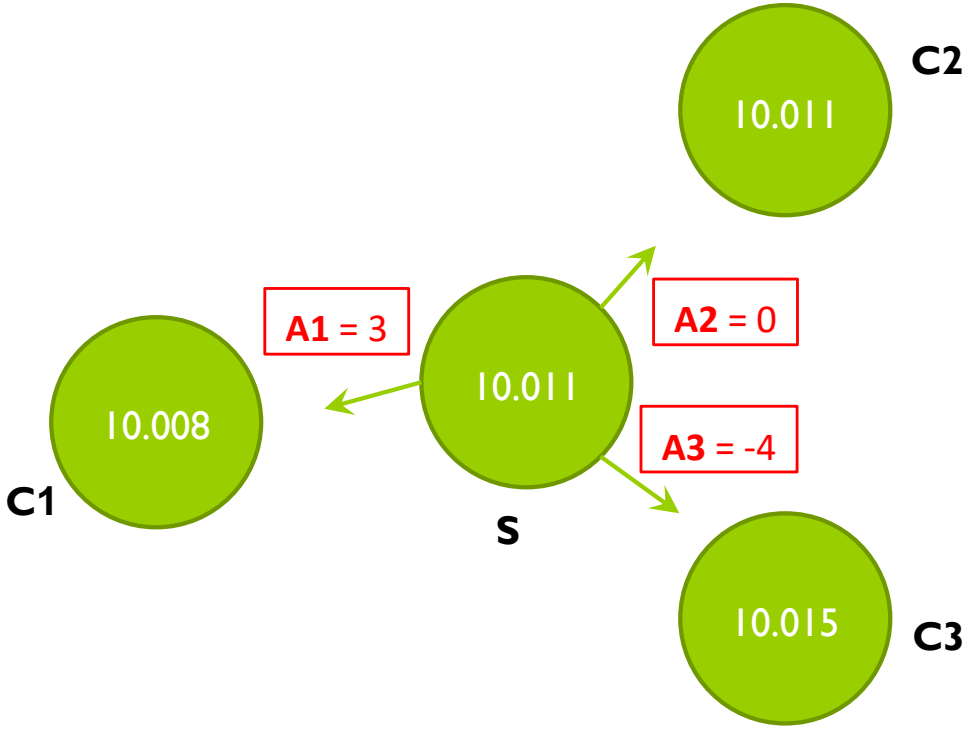
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**



Ejemplo de Algoritmo de Berkeley

- Se notifica el ajuste $A_i = D - D_i'$ a cada cliente

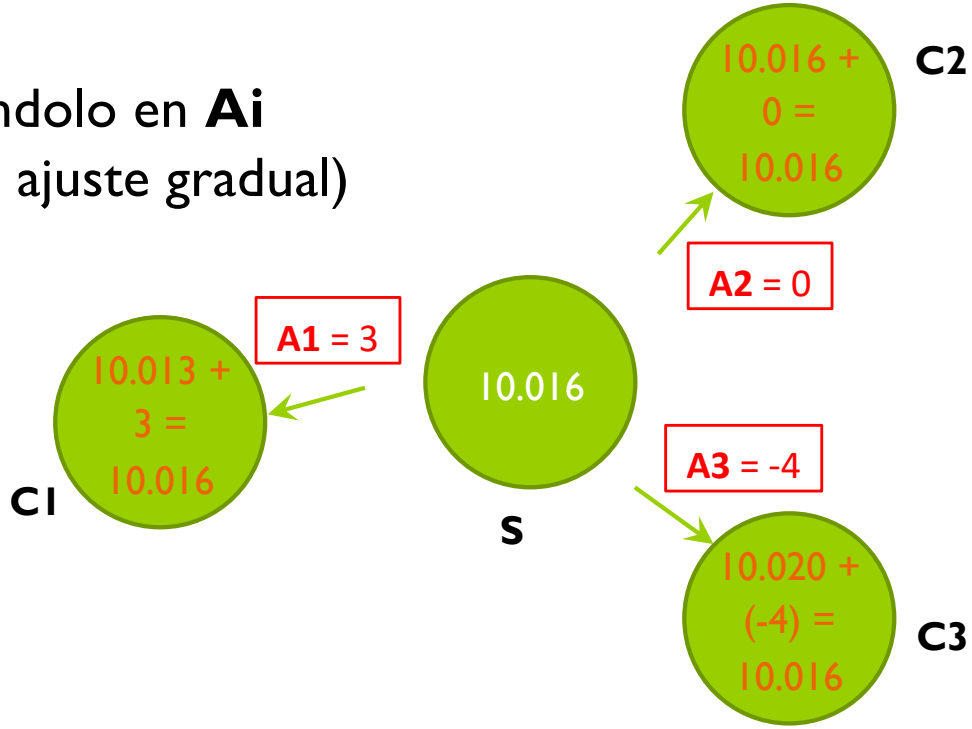
T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**
A1	1 - (-2) = **3**
A2	1 - 1 = **0**
A3	1 - 5 = **-4**



Ejemplo de Algoritmo de Berkeley

- Los mensajes llegan a los clientes
  - *Los mensajes llegan tras 5 unidades de tiempo. Para simplificar, asumimos que llegan a la vez*
- Cada cliente ajusta su reloj incrementándolo en **Ai**
  (si el ajuste es negativo, se programa un ajuste gradual)

T0	**10.000**
D1	10.003 - 10.000 = **3**
D2	10.006 - 10.000 = **6**
D3	10.010 - 10.000 = **10**
T1i	**10.010**
D1'	3 - (10010 - 10000)/2 = **-2**
D2'	6 - (10010 - 10000)/2 = **1**
D3'	10 - (10010 - 10000)/2 = **5**
D	(-2 + 1 + 5 + 0)/4 = **1**
A1	1 - (-2) = **3**
A2	1 - 1 = **0**
A3	1 - 5 = **-4**



Algoritmos de sincronización de relojes: **Algoritmo de Berkeley**

- Consideraciones adicionales:
  - No persigue sincronizar todos los relojes con el instante "real", sino llegar a un acuerdo entre los nodos
  - Si alguna diferencia $\mathbf{D_i}$ es muy distinta a las demás, no se tiene en cuenta
  - Si el servidor falla, se inicia un algoritmo de elección de líder para escoger otro servidor
  - La sincronización exacta es imposible debido a la variabilidad del tiempo en la transmisión de los mensajes

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Relojes lógicos (Lamport, 1978)

- Indican el orden en que suceden ciertos eventos, no el instante real en que suceden
- Resultan útiles para muchos tipos de aplicaciones distribuidas que sólo requieren saber si un evento ha sucedido antes que otro
- A diferencia de los relojes físicos, su sincronización es perfecta pero tienen ciertas limitaciones
- Ideas clave:
  - Si dos nodos no interactúan (si no intercambian mensajes) no es necesario que tengan el mismo valor de reloj.
  - En general es importante el orden global en que ocurren los eventos y no el tiempo real en que suceden.

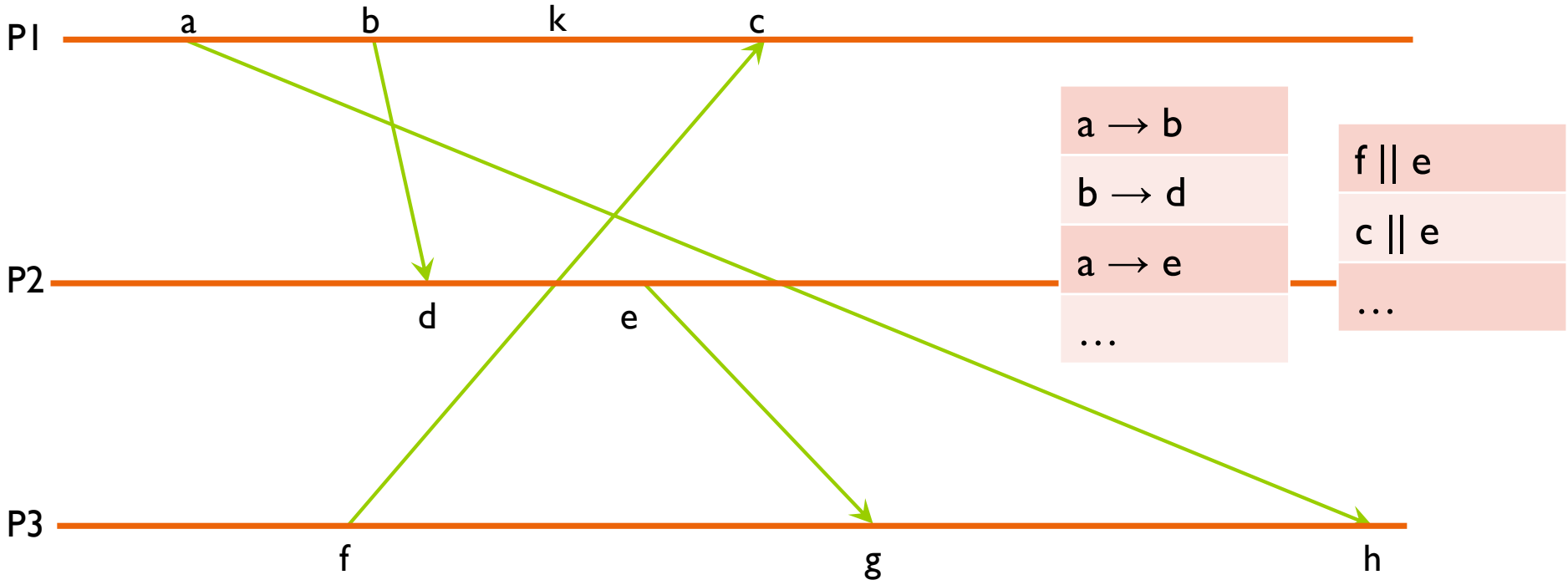
Relojes lógicos: Relación "ocurre antes"

- Para sincronizar los relojes lógicos, se define la relación "ocurre-antes" (*happens-before*)
- **a** → **b**
  - "**a** ocurre antes que **b**"
  - Todos los nodos están de acuerdo en que primero ven el evento **a** y después ven el evento **b**
- Se puede observar de forma directa:
  - Si **a** y **b** son eventos del mismo nodo y **a** ocurre antes que **b**, **a** → **b**
  - Si **a** es el evento de envío de un mensaje por parte de un nodo, y **b** el evento de recepción de ese mensaje por parte de otro nodo, **a** → **b**

Relojes lógicos: Relación "ocurre antes"

- Es una relación **transitiva**
  - si **a** → **b** y **b** → **c**, entonces **a** → **c**
- Dos eventos **x** e **y** son **concurrentes** si no se puede decir cuál de ellos ocurre antes: **x||y**
- La relación "ocurre antes" establece un **orden parcial** entre los eventos de un sistema, puesto que no todos los eventos están relacionados entre sí
  - pues puede haber eventos concurrentes

Relojes lógicos: Relación "ocurre antes"



Relojes lógicos: Relojes lógicos de Lamport

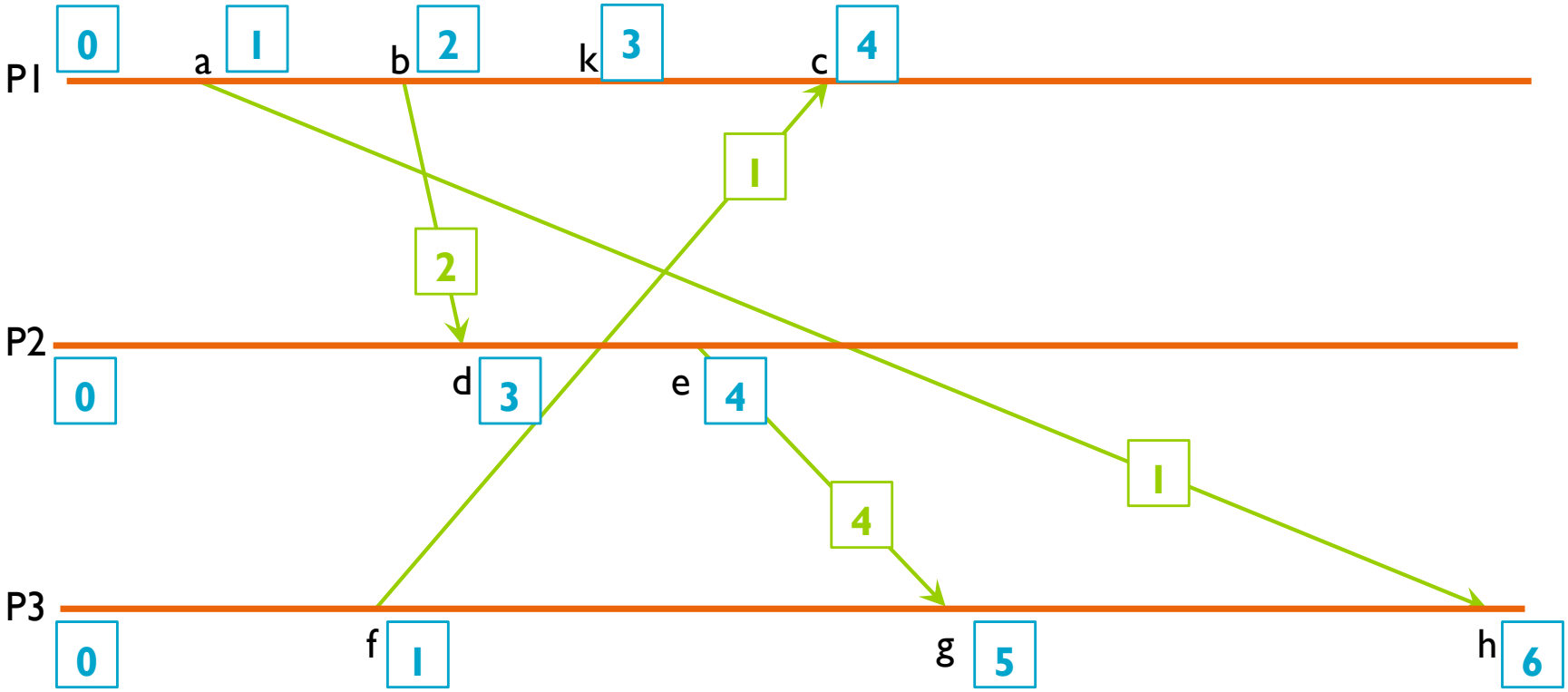
- Es necesaria una forma de medir la relación *ocurre-antes*: usando el *reloj lógico de Lamport*.
- Los relojes lógicos deben marcar el instante en que ocurren los eventos, de forma que asocien un valor a cada evento.
- Llamamos **C(a)** al valor del reloj lógico del evento **a**.
- Los relojes lógicos deben satisfacer que:
  - Si **a** → **b** , entonces **C(a) < C(b)**.
- Si reescribimos las condiciones de la relación *ocurre-antes*:
  - Para dos eventos de un mismo nodo a y b, si a ocurre antes que b, entonces C(a) < C(b)
  - Para eventos correspondientes de envío y recepción a y b, se cumple que C(a) < C(b).
- Adicionalmente el reloj nunca debe decrecer.

Algoritmo de Lamport:

- Cada nodo dispone de un contador (reloj lógico) C_p inicializado a 0

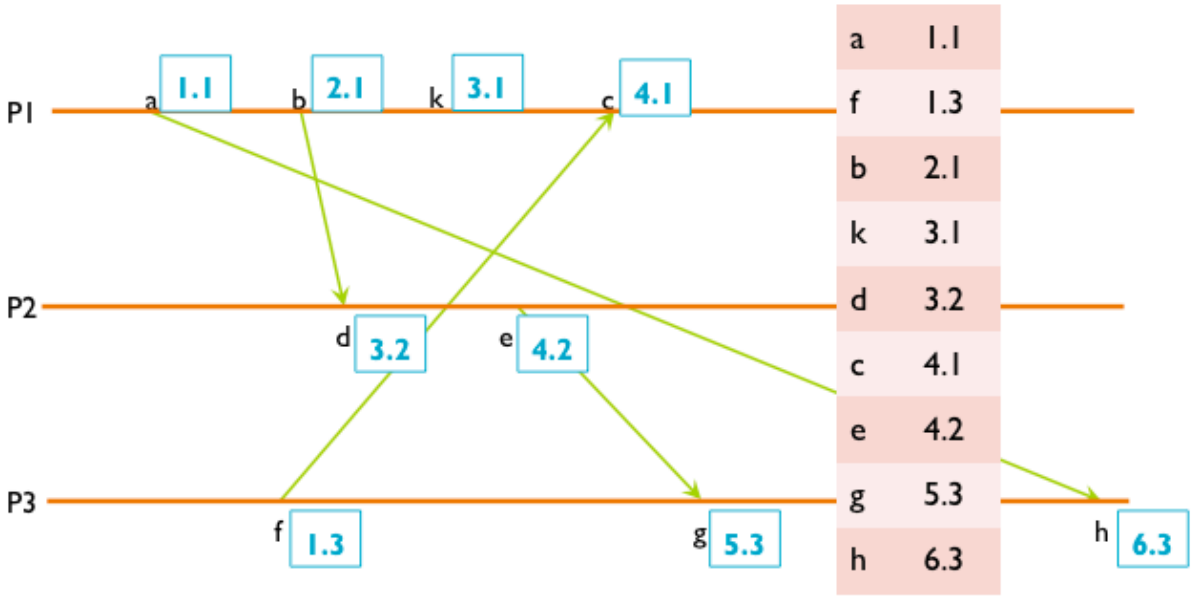
1. Cada ejecución de un evento (envío de mensaje o evento interno) en un nodo ***p***, incrementa el valor de su contador C_p en 1.
2. Cada mensaje **m** enviado por un nodo **p** es etiquetado (C_m) con el valor de su contador C_p, es decir $C_m = C_p$
3. Cuando un nodo ***p*** recibe un mensaje, actualiza su reloj acorde a $C_p = \max(C_p, C_m) + 1$

Relojes lógicos: Relojes lógicos de Lamport



Relojes lógicos: Relojes lógicos de Lamport

- Características:
  - Establece un orden parcial entre los eventos.
    - Los eventos concurrentes no guardan orden entre sí.
  - Puede convertirse en total añadiendo el identificador del nodo como sufijo



  - Si **a** → **b**, el algoritmo garantiza que **C(a) < C(b)**
  - Pero, si **C(a) < C(b)** no se puede decidir si **a** → **b** o **a || b**

Relojes vectoriales

- El reloj de Lamport garantiza que si a → b , entonces C(a) < C(b).

- Sin embargo, si C(a) < C(b), no se desprende nada. Es decir, la simple observación de los relojes no implica nada acerca de su ordenación.

- En algunas aplicaciones resulta necesario detectar cuándo dos eventos han sido **concurrentes**. Con relojes lógicos no podemos saberlo.

Relojes vectoriales

- Solución: uso de **relojes vectoriales**
  - Los relojes vectoriales asocian un valor vectorial **V(x)** a cada evento **x** que sucede en un sistema distribuido
  - Nos permitirán saber si un evento ocurre **antes** que otro o si los dos eventos son **concurrentes**
  - Dados **N** nodos, cada nodo **p** mantiene un vector de **N** marcas temporales $\mathbf{V_p}$ de forma que:
    - $\mathbf{V_p[p]}$ es el número de eventos que han ocurrido en **p**
    - Si $\mathbf{V_p[q]=k}$, entonces **p** sabe que al menos han ocurrido **k** eventos en el nodo **q**

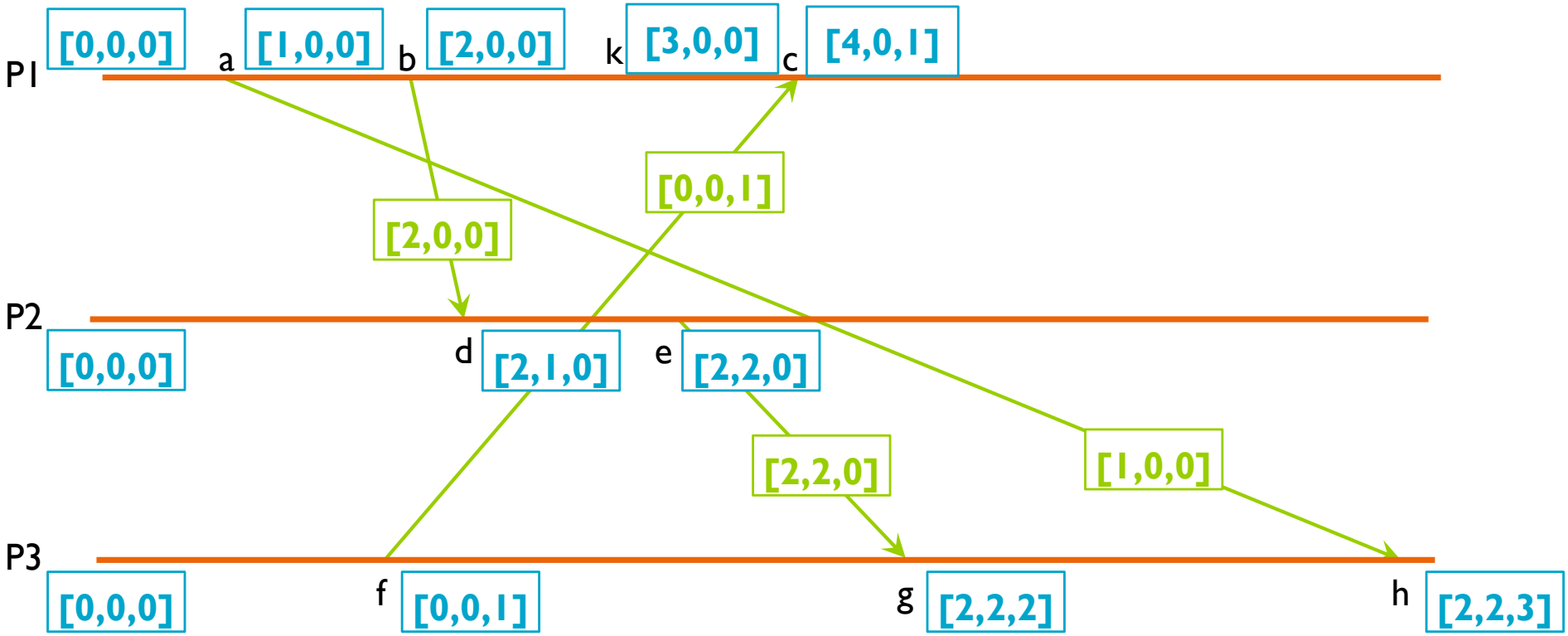
Relojes vectoriales: algoritmo

- Cada nodo **p** dispone de un reloj vectorial $\mathbf{V_p}$ inicializado con todos sus elementos a 0

1. El nodo ***p*** incrementa $\mathbf{V_p[p]}$ en 1 cada vez que envía un mensaje o ejecuta un evento interno.
2. Un mensaje ***m*** enviado por un nodo ***p*** lleva asociado su reloj vectorial. Es decir, $\mathbf{V_m=V_p}$.
3. Al recibir **p** un mensaje *m*, incrementa $\mathbf{V_p[p]}$ en 1 y además actualiza su reloj seleccionando el máximo entre el valor local y el valor del reloj del mensaje, para cada una de sus componentes:

$$\forall i, 1 \le i \le N, V_p[i] = \max(V_p[i], V_m[i])$$

Relojes vectoriales: ejemplo



Relojes vectoriales: características

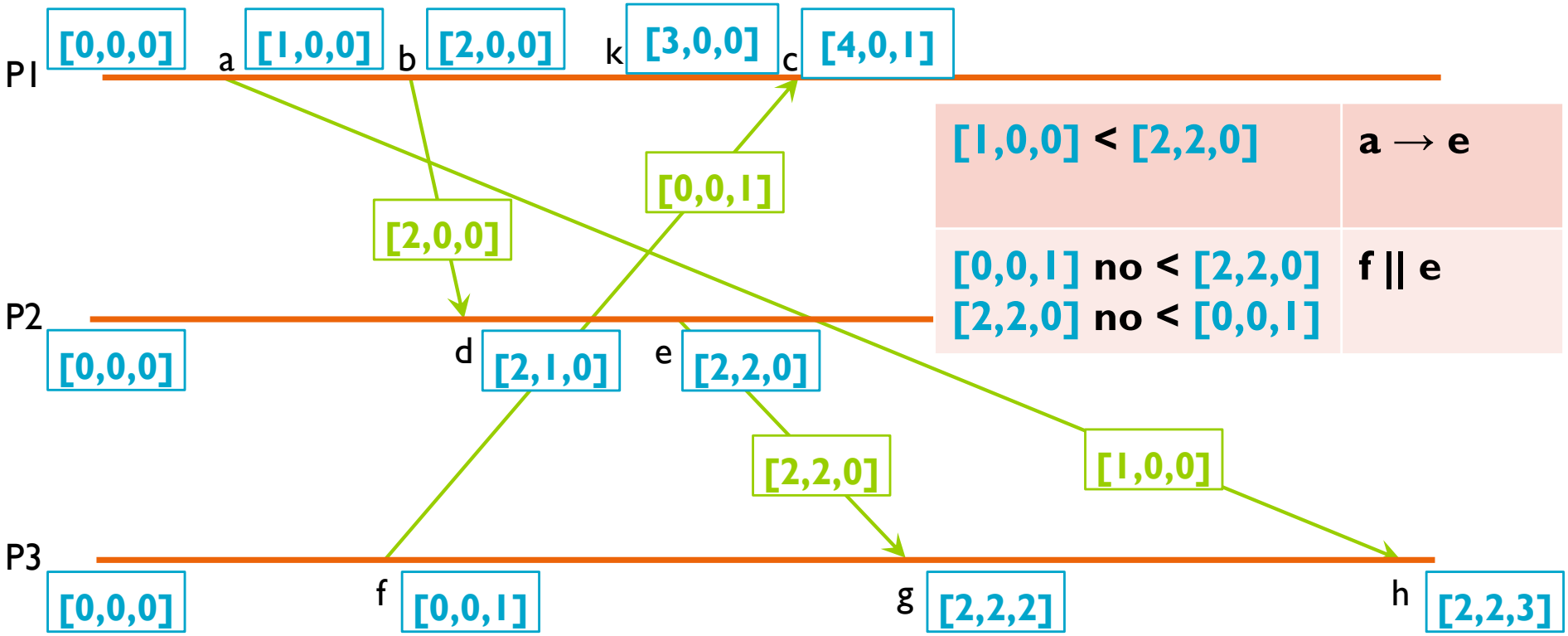
- **V(a) <V (b)** si:
  - cada una de las componentes de V(a) es menor o igual que la respectiva componente de V (b)
  - y además debe haber al menos una de ellas que sea estrictamente menor.

$$V(a) \leq V(b) \Leftrightarrow \forall i, 1 \leq i \leq N, V(a)[i] \leq V(b)[i]$$

$$V(a) < V(b) \Leftrightarrow V(a) \leq V(b) \wedge \exists i, 1 \leq i \leq N, V(a)[i] < V(b)[i]$$

- Si **a** → **b** entonces **V(a) <V (b)**
- Si **V(a) <V (b)** entonces **a** → **b**
- Si no se cumple que **V(a) <V (b)** y tampoco que **V(b) <V (a)** entonces **a||b**

Relojes vectoriales: ejemplo



Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Estado global: necesidad

- **Estado global** formado por:
  - El estado de cada nodo: sólo las variables que nos interesen
  - Y los mensajes enviados y todavía no entregados.

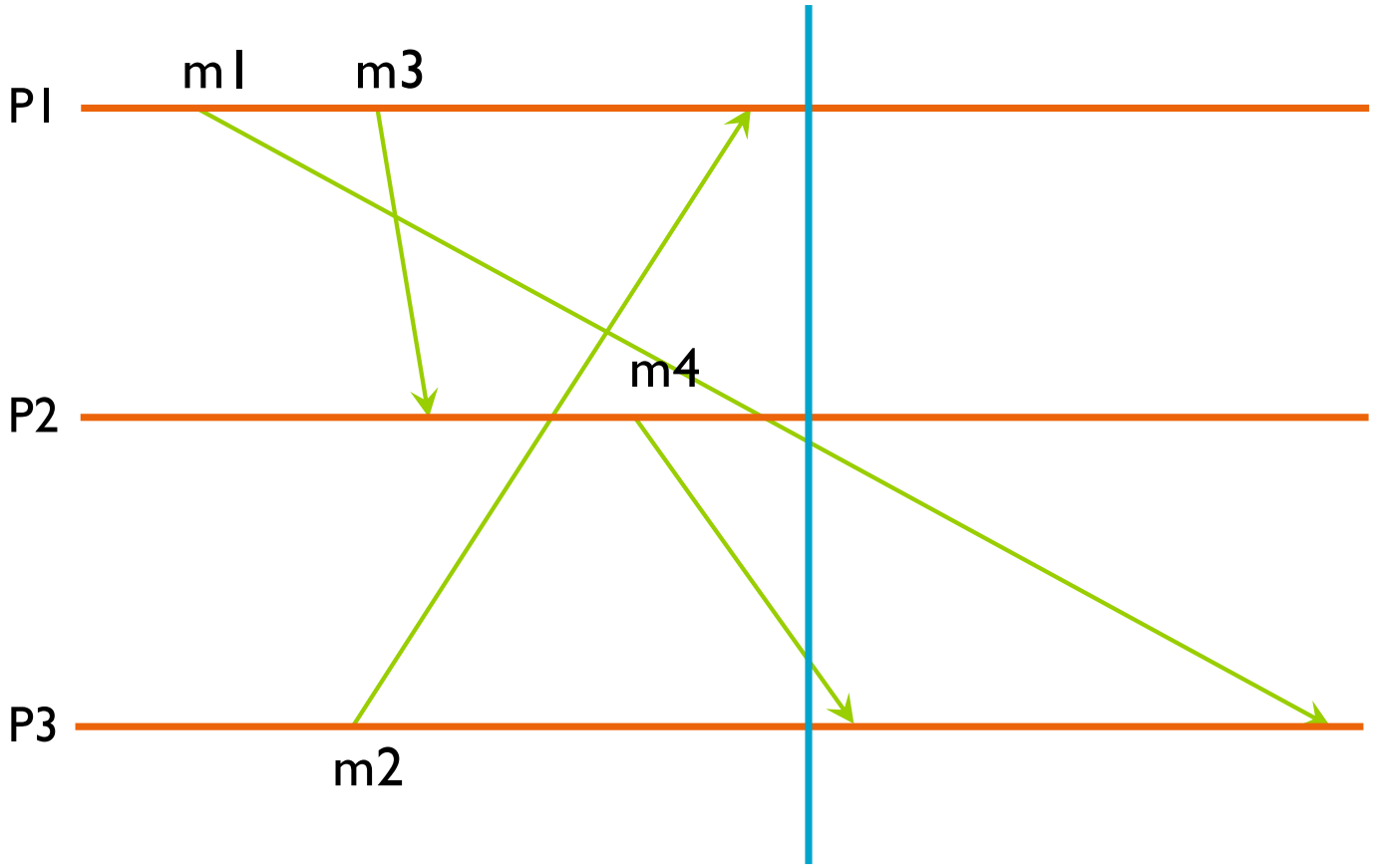
- Ejemplos de uso:
  - Recolector de objetos remotos no utilizados
    - Se averigua si algún nodo mantiene alguna referencia al objeto remoto
    - Y si en algún mensaje en tránsito hay alguna referencia a dicho objeto
    - Si no se encuentra ninguna referencia, el objeto se elimina
  - Detección de deadlock distribuido
    - ¿Está el sistema en un estado de bloqueo mutuo?
  - Detección de terminación distribuida
    - ¿Ha terminado el algoritmo distribuido? Todos los procesos pueden estar detenidos.. pero puede haber un mensaje en camino.

Estado global: necesidad

- Conceptos previos:
  - Una **instantánea distribuida** refleja el estado que pudo haberse alcanzado en el sistema.
  - Las instantáneas deben reflejar sólo estados consistentes:
    - Si P ha recibido un mensaje de Q en la instantánea, Q envió antes un mensaje a P.

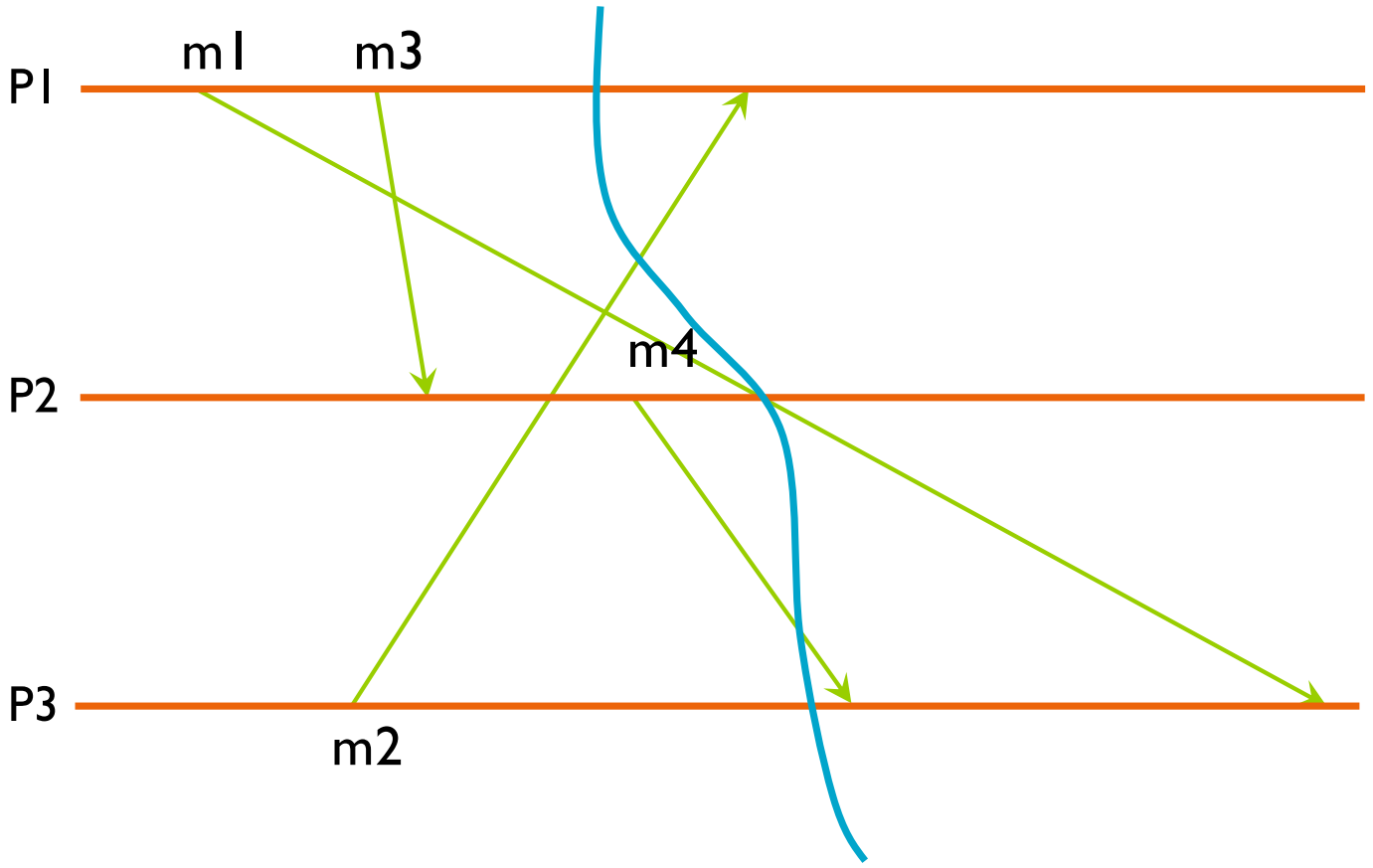
Estado global
Instantáneas: **instantánea precisa**

- La instantánea precisa no es factible, haría falta que los nodos tuviesen sus relojes perfectamente sincronizados



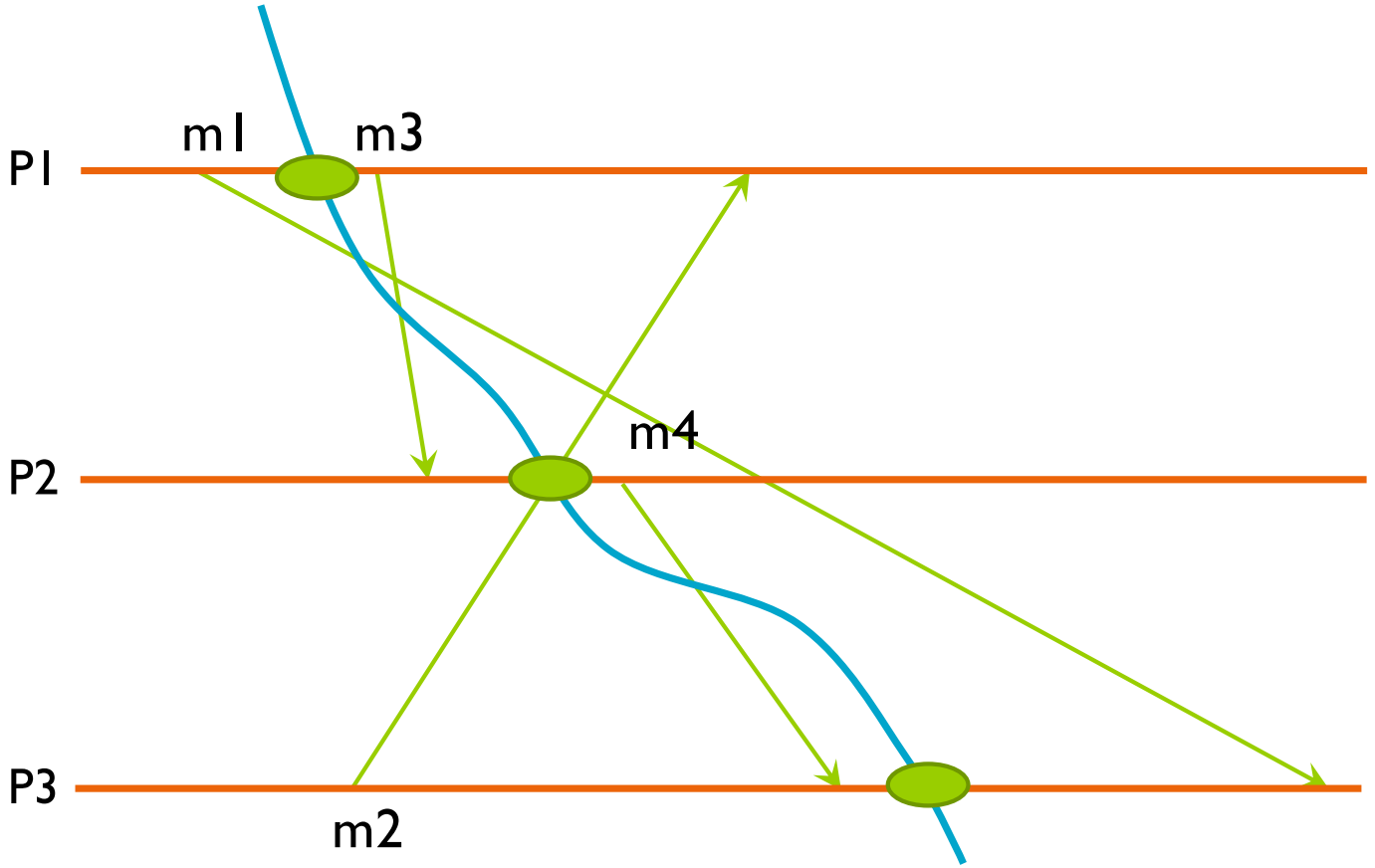
Estado global
Instantáneas: instantánea **consistente**

- No se ha reflejado ninguna entrega de mensaje que no esté precedida por su respectivo envío.



Estado global
Instantáneas: instantánea **inconsistente**

- Se ha registrado la entrega de algunos mensajes (m3 y m4) pero no su envío



Estado global
Algoritmo de **Chandy-Lamport** (1985)

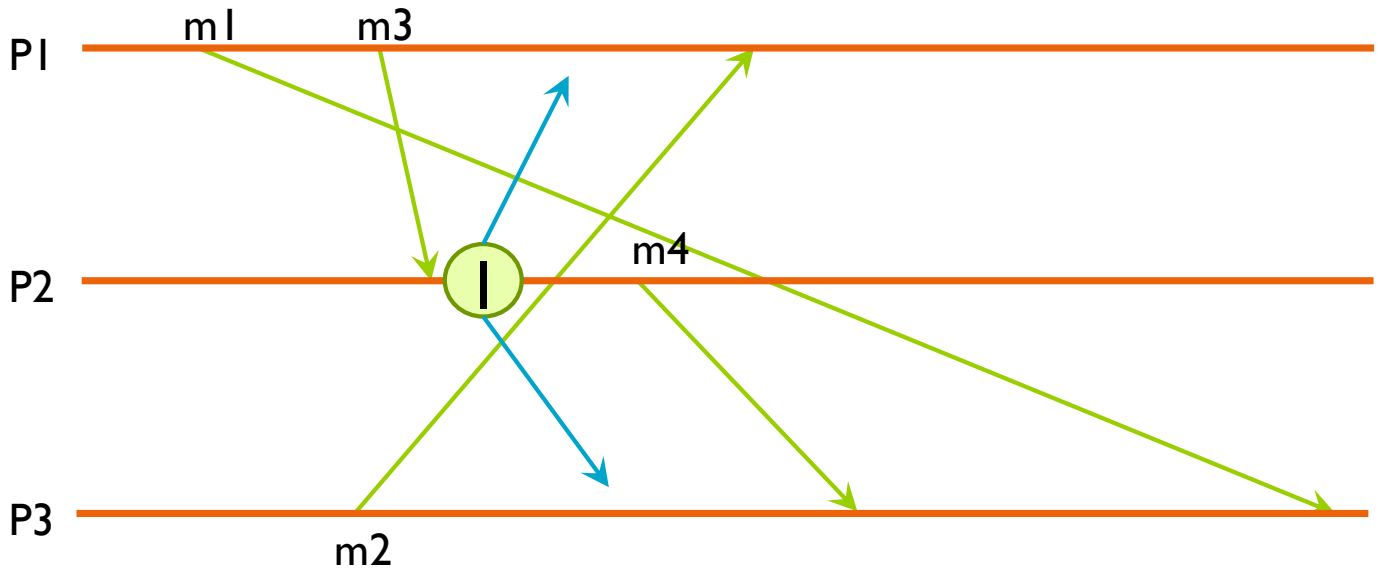
- Este algoritmo crea una instantánea consistente del estado global del sistema distribuido
- Asumimos:
  - Sistema distribuido formado por varios nodos
  - Red con topología completa, existe un canal entre todo par de nodos
  - Los canales son:
    - fiables
    - Transmiten sus mensajes en orden FIFO
    - Son unidireccionales, entre dos nodos **p** y **q** existen dos canales (**p**,**q**) y (**q**,**p**)

Estado global
Algoritmo de Chandy-Lamport

- **Algoritmo de Chandy-Lamport:**
  1. El nodo iniciador *p* guarda su estado local y envía un mensaje MARCA al resto de nodos.
  2. Cuando un nodo ***p*** recibe el mensaje MARCA por el canal **c**:
     a) Si aún no ha registrado su estado:
        - Registra su estado local y anota como vacío el canal **c**.
        - Luego, pero antes de enviar cualquier otro mensaje, envía **MARCA** al resto de nodos y comienza a registrar los mensajes que lleguen por otros canales distintos a **c**.
     b) Si ya había registrado su estado:
        - Anota todos los mensajes que ha recibido por el canal **c** (puede no haber ninguno) y deja de registrar la actividad en dicho canal.
  3. Cuando un nodo **p** recibe MARCA por todos sus canales de entrada
     - Envía su estado local previamente guardado y el de sus canales de entrada al nodo iniciador (salvo que sea el propio iniciador) y finaliza su participación en el algoritmo

Estado global
Algoritmo de Chandy-Lamport

- El nodo iniciador
  - Guarda su estado local y envía un mensaje MARCA al resto de nodos (1)

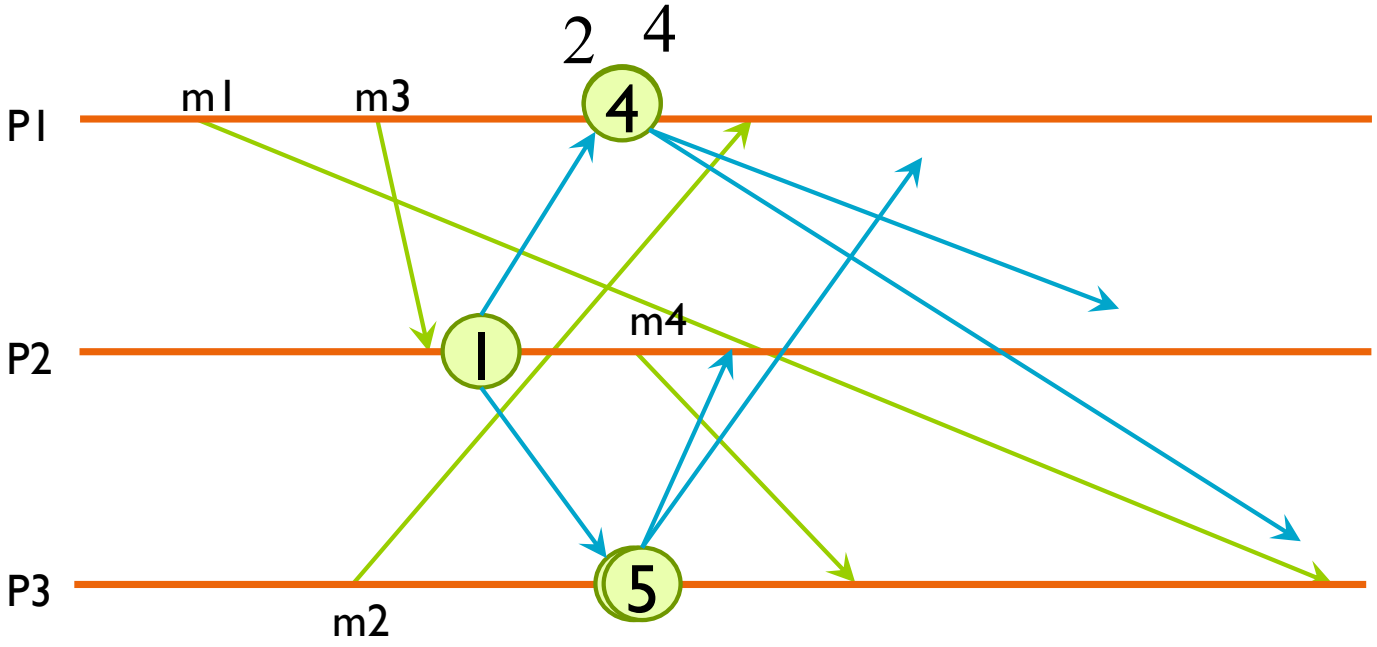


Estado local	Paso	Valor
P1		
P2	I	Est. P2
P3		

Estado Canal	Paso	Valor
P1,P2		
P2,P1		
P1,P3		
P3,P1		
P2,P3		
P3,P2		

Estado global
Algoritmo de Chandy-Lamport

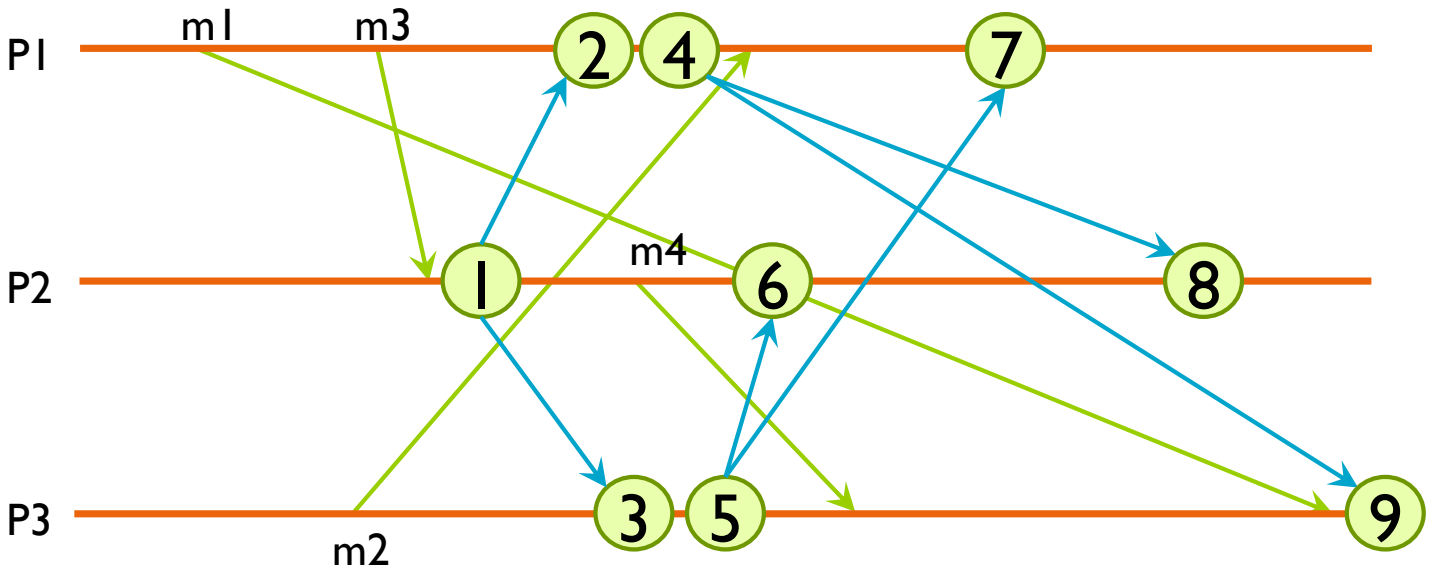
- Cuando un nodo **p** recibe MARCA por el canal **c**, si aun no ha registrado su estado
  - Registra su estado local y anota como vacío el canal **c** (2,3)
  - Luego, pero antes de enviar cualquier otro mensaje, envía MARCA al resto de nodos y comienza a registrar los mensajes que lleguen por otros canales distintos a **c** (4,5)



Estado local	Paso	Valor
P1	2	Est. P1
P2	1	Est. P2
P3	3	Est. P3
Estado Canal	**Paso**	**Valor**
P1,P2		
P2,P1	2	Vacío
P1,P3		
P3,P1		
P2,P3	3	Vacío
P3,P2		

Estado global
Algoritmo de Chandy-Lamport

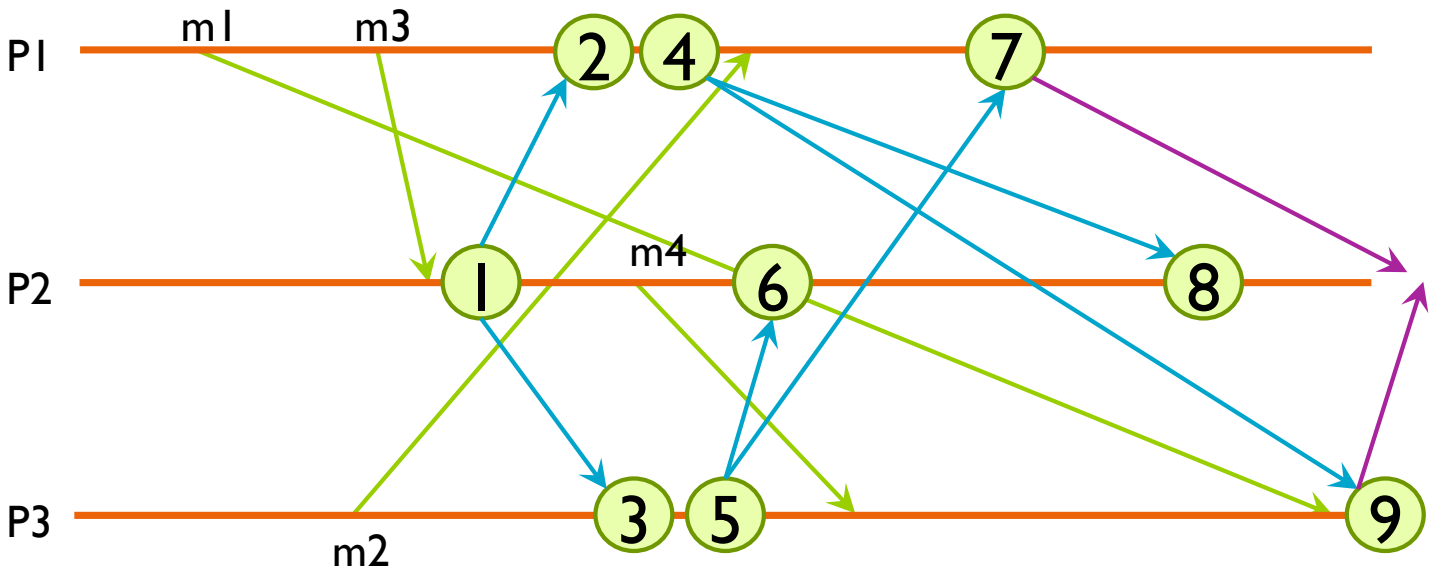
- Cuando un nodo **p** recibe MARCA por el canal **c**, y ya había registrado su estado
  - Anota todos los mensajes que ha recibido por el canal **c** (puede no haber ninguno) y deja de registrar la actividad en dicho canal (6,7,8,9)



Estado local	Paso	Valor
P1	2	Est. P1
P2	1	Est. P2
P3	3	Est. P3
Estado Canal	**Paso**	**Valor**
P1,P2	8	Vacío
P2,P1	2	Vacío
P1,P3	9	m1
P3,P1	7	m2
P2,P3	3	Vacío
P3,P2	6	Vacío

Estado global
Algoritmo de Chandy-Lamport

- Cuando un nodo **p** recibe MARCA por todos sus canales de entrada
  - Envía su estado local previamente guardado y el de sus canales de entrada al nodo iniciador (salvo que sea el propio iniciador) y finaliza su participación en el algoritmo (7,8,9)



Estado local	Paso	Valor
P1	2	Est. P1
P2	1	Est. P2
P3	3	Est. P3
Estado Canal	**Paso**	**Valor**
P1,P2	8	Vacío
P2,P1	2	Vacío
P1,P3	9	m1
P3,P1	7	m2
P2,P3	3	Vacío
P3,P2	6	Vacío

Estado global
Algoritmo de Chandy-Lamport

- Consideraciones adicionales:
  - Para que funcione el algoritmo conforme se ha descrito, se requiere añadir al mensaje MARCA el identificador del nodo iniciador.
    - De este modo, los nodos que reciben MARCA pueden conocer quién es el iniciador y así saber a qué nodo deben enviar la información de su estado y de los mensajes registrados
  - En el algoritmo original de Chandy-Lamport, los autores no especifican cómo se debe recolectar la información de estado que registra cada nodo
    - Se podría utilizar la técnica que hemos explicado
    - O bien se podrían utilizar otros algoritmos existentes para la recolectar información registrada
      - Por ejemplo, cada nodo podría enviar al resto de nodos su información registrada, de modo que todos los nodos podrían al final conocer el estado global del sistema.
    - *Referencia*:
      - K.M. Chandy & L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3 (1), pp. 63-75, 1985. http://research.microsoft.com/en-us/um/people/lamport/pubs/chandy.pdf

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Ejemplos de Problemas Algorítmicos en SD

- Ejemplos de problemas algorítmicos en sistemas distribuidos:
  - Encaminamiento
  - Difusiónes de mensajes a grupos
  - Garbage collection
  - Protocolos de replicación
  - Pertenencia a grupos
  - Detección de fallos
  - Compromiso distribuido
  - Elección de lider
  - Consenso
  - Exclusión mutua
  - Detección de interbloqueos
  - Detección distribuida de terminación
  - ….

Ejemplos de Problemas Algorítmicos en SD

- Nos centraremos en esta unidad en:
  - Exclusión mutua
  - Elección de lider
  - Consenso

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Exclusión mutua distribuida

- Problema a resolver → acceso a sección crítica (ej. un recurso compartido) por parte de distintos procesos que están en nodos diferentes.
  - Evitar inconsistencias
  - Las soluciones necesitan asegurar el acceso con exclusión mutua de los procesos.

- Soluciones propuestas:
  a) Algoritmo centralizado
  b) Algoritmo distribuido
  c) Algoritmo para anillos

Condiciones de corrección para exclusión mutua

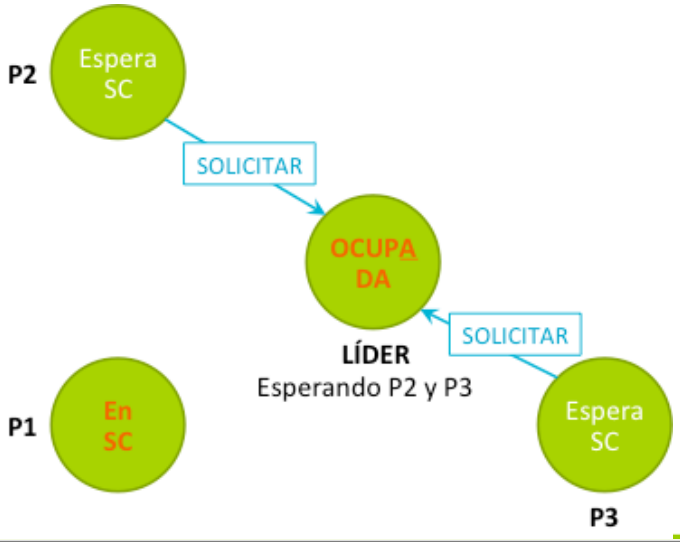
- **Seguridad**: como mucho, sólo un proceso puede estar ejecutándose dentro de la sección crítica en un momento dado (i.e. exclusión mutua)
- **Viveza**: todo proceso que quiere entrar en la sección crítica lo consigue en algún momento.
  - **Progreso:** si la sección crítica está libre y hay procesos que desean entrar, se selecciona en tiempo finito a uno de ellos.
  - **Espera limitada:** si un proceso quiere entrar en sección crítica, sólo debe esperar un número finito de veces a que otros entren antes que él.

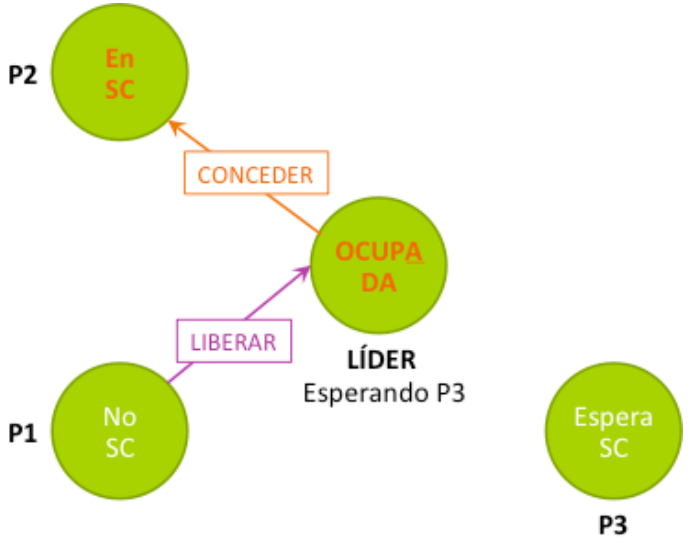
Exclusión mutua
a) Algoritmo centralizado (Lamport 1978)

- Se elige un nodo como coordinador (líder).
  - Este líder gobierna el acceso a la sección crítica

1. Cuando un nodo quiere entrar en la sección crítica, envía un mensaje **SOLICITAR** al líder, pidiendo permiso.
2. Si ningún otro nodo está en la sección crítica, el líder responde **CONCEDER**
   - El líder anota que la sección crítica está ahora ocupada
   - El nodo solicitante comienza a utilizar la sección crítica.
3. Si la sección crítica está ocupada, y otro nodo solicita al líder utilizarla
   - El líder anota la identidad del solicitante y no le contesta
   - El nodo solicitante permanece suspendido, esperando la respuesta

4. Cuando un nodo sale de su sección crítica, avisa al líder con un mensaje **LIBERAR**.
   - Si existe algún nodo suspendido esperando permiso para entrar en la sección crítica, el líder le envía **CONCEDER**
   - En caso de haber varios esperando, puede por ejemplo eleigir al nodo que primero solicitó la sección crítica
   - Si no existe ningún nodo suspendido esperando permiso para entrar en la sección crítica, el líder anota que la sección crítica vuelve a estar libre





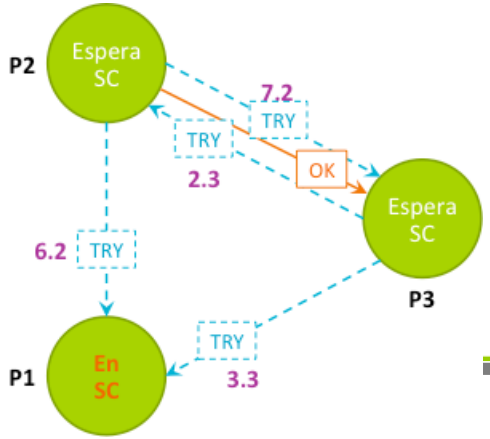
Exclusión mutua
b) Algoritmo distribuido (Ricart-Agrawala 1981)

- Suponemos que todos los eventos están ordenados (p.ej, usando relojes lógicos de Lamport junto al número de nodo).

1. Cuando un nodo quiere entrar en la sección crítica, difunde un mensaje **TRY** a todos los otros nodos
2. Cuando un nodo recibe un mensaje **TRY**:
   a) Si no está en su sección crítica, ni esperaba entrar, responde **OK**
   b) Si está en su sección crítica, no contesta y encola el mensaje.
   c) Si no está en su sección crítica, pero quiere entrar, compara el número de evento del mensaje entrante con el que él mismo envió al resto. Vence el número más bajo:
      1. Si el mensaje entrante es más bajo, responde **OK**.
      2. Si es más alto, no responde y encola el mensaje.
3. Un nodo entra en la sección crítica, cuando recibe **OK** de todos los demás nodos.
4. Cuando un nodo abandona la sección crítica, envía **OK** a todos los nodos que enviaron los mensajes que retuvo en su cola.

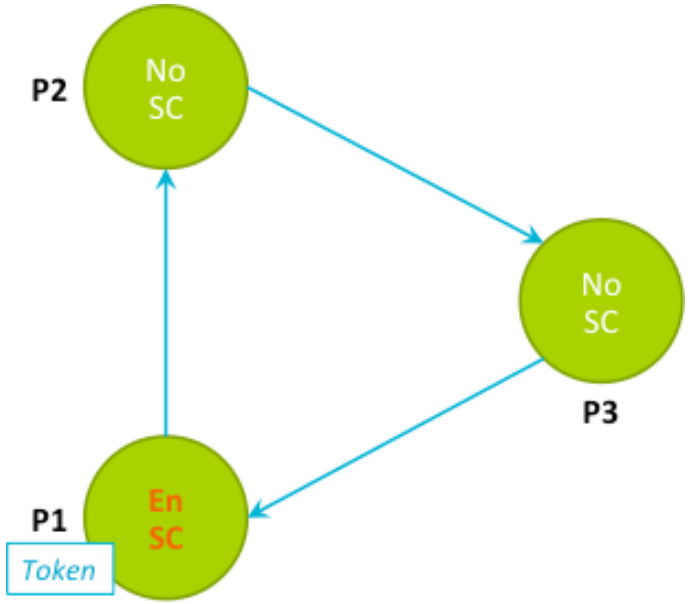
Exclusión mutua
b) Algoritmo distribuido

- **Características:**
  - Los mensajes están etiquetados con el reloj lógico de Lamport + Id. del nodo emisor
  - Cuando un nodo recibe un mensaje TRY, no está en su sección crítica, pero quiere entrar:
    - Si la etiqueta del mensaje entrante es más baja, responde OK
    - Si es más alta, no responde y anota la identidad del emisor del mensaje TRY
  - De esta forma, cuando la sección crítica se libere, el nodo que ha "ganado" será el primero en entrar



Exclusión mutua
c) Algoritmo para anillos (Le Lann 1977)

- No existe ningún nodo coordinador
- La coordinación se resuelve mediante comunicación en anillo y el uso de un *token* que circula por dicho anillo
  - Requiere comunicación fiable.
  - Si un nodo cae, hay que reconstruir el *token*.
- Sólo puede entrar en la sección crítica el nodo que tiene el *token*.



Exclusión mutua
c) Algoritmo para anillos

- **Algoritmo para anillos:**
  - Situación inicial: sección crítica libre y el *token* se encuentra en alguno de los nodos.

1. Si el nodo no espera entrar en su sección crítica, pasa el *token* al siguiente nodo.
2. Un nodo espera para entrar en su sección crítica si no tiene el *token*
3. Un nodo entra en su sección crítica cuando consigue el *token*
   - Y no lo pasa al siguiente nodo hasta que finaliza su sección crítica.

- En consecuencia, si un nodo quiere entrar en su sección crítica debe esperar a que el nodo que sí está en su sección crítica libere el *token* y este llegue eventualmente por el anillo.

Contenido

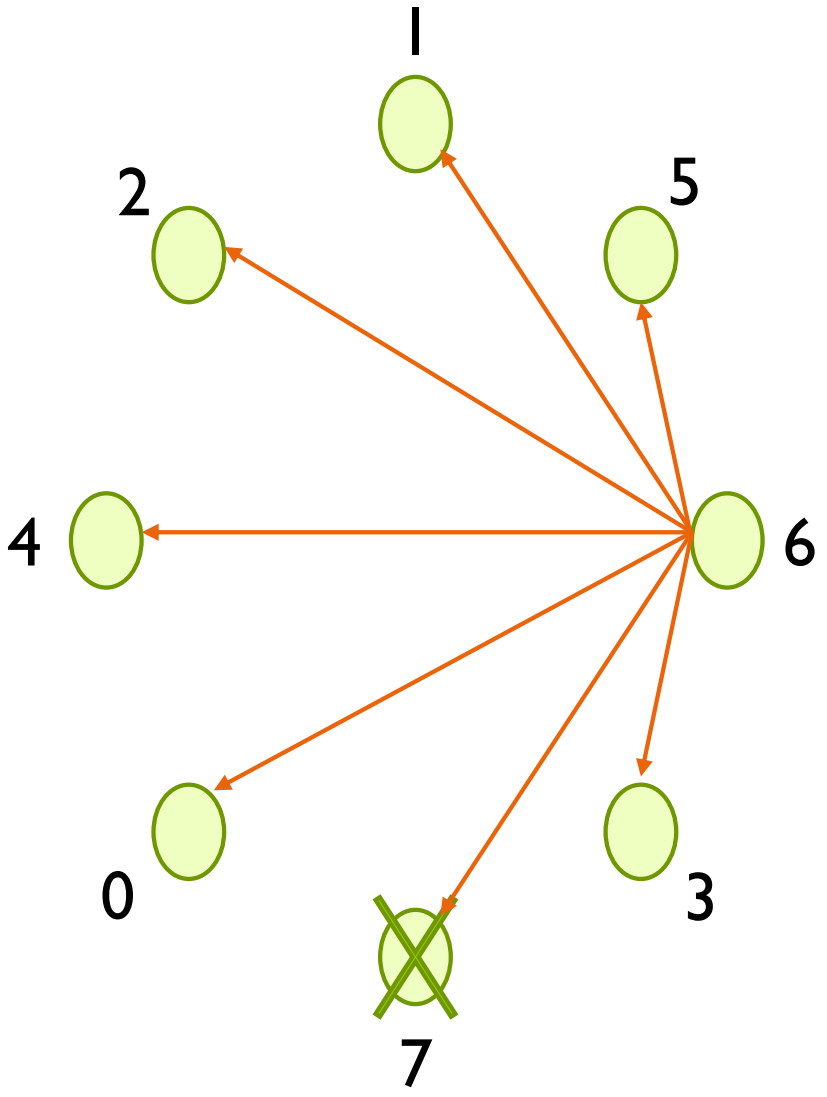
- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Elección de líder: concepto

- En muchos algoritmos distribuidos resulta ventajoso que un nodo actúe como líder o coordinador
  - Simplifica dichos algoritmos
  - Reduce el número de mensajes que se necesitan para llegar a un acuerdo
- La elección se produce al inicio o cuando se detecta que el líder deja de responder
- Se asume que los nodos conocen los identificadores del resto de nodos del sistema distribuido
- La elección de líder es un caso particular de consenso, pero que estudiamos por separado dada su importancia.
- **Ejemplos de algoritmos de elección de líder**:
  - Algoritmo Bully
  - Algoritmo para anillos

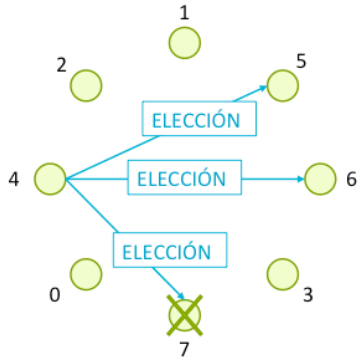
Elección de líder:
a) **Algoritmo Bully** (García-Molina 1982)

- Sirve para que, a iniciativa de uno de los nodos, se elija un nuevo líder
- El nuevo líder será el nodo activo con el identificador más alto
- Tras ser elegido, el líder lo notificará al resto

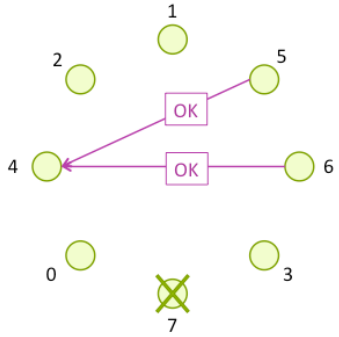


Elección de líder:
a) **Algoritmo Bully**

1. Cuando un nodo quiere comenzar una elección (p.ej., porque el líder actual no contesta), envía **ELECCIÓN** a todos los nodos con identificador mayor al suyo.



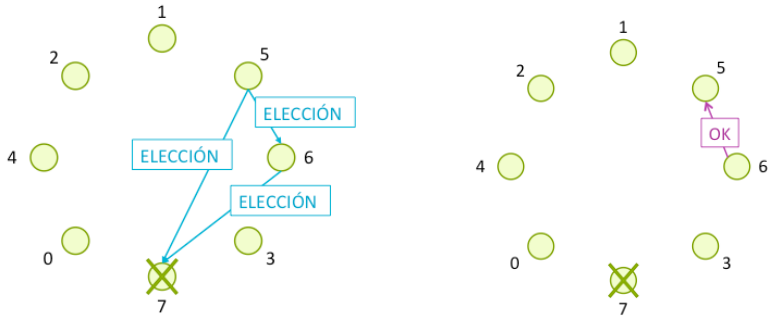
2. Cuando un nodo recibe ELECCIÓN, envía **OK** a quien se lo envió (para avisarle que participa y le ganará).



3. Cuando un nodo recibe al menos un mensaje **OK**, deja de participar.

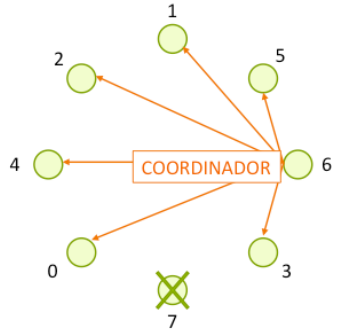
Elección de líder:
a) **Algoritmo Bully**

Los pasos anteriores se van repitiendo…



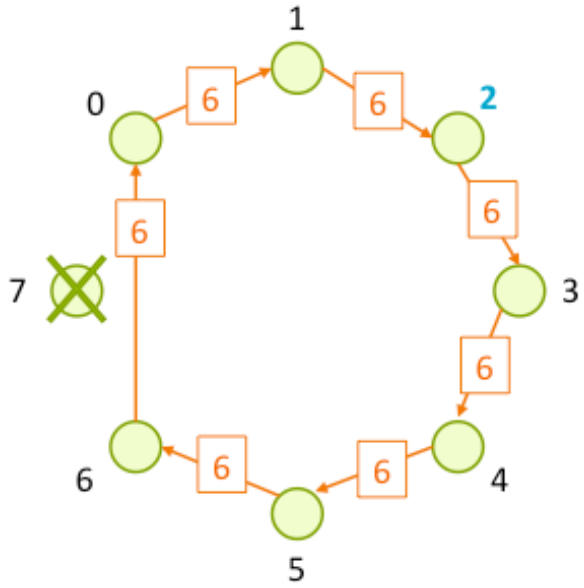
4. Hasta que si un nodo **no obtiene respuesta**, él es el nuevo líder.

5. El nuevo líder difundirá un mensaje **COORDINADOR** para comunicarlo al resto.



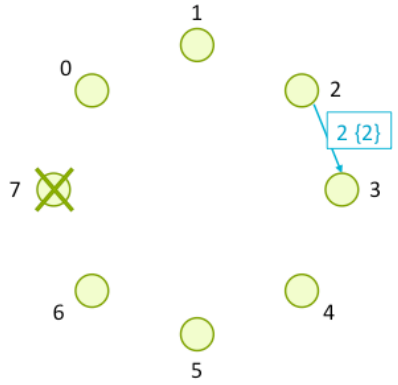
Elección de líder:
b) **Algoritmo para anillos**

- Los nodos están dispuestos en un anillo lógico y envían sus mensajes a través de los canales de dicho anillo
- Este algoritmo sirve para que, a iniciativa de uno de los nodos, se elija un nuevo líder
- Tras ser elegido, se propagará la identidad del líder por el anillo

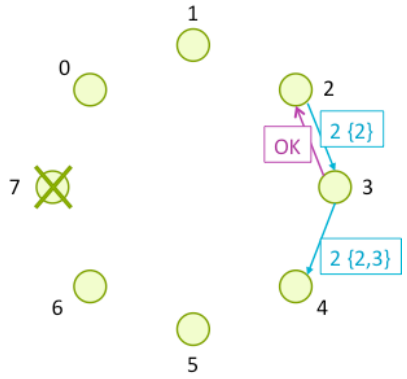


Elección de líder:
b) **Algoritmo para anillos**

1. Cuando un nodo quiere comenzar una elección de líder:
   - construye un mensaje ELECCIÓN con dos campos:
     - Iniciador
     - Lista de nodos participantes
   - Asigna su identificador a ambos campos y envía el mensaje al siguiente nodo del anillo

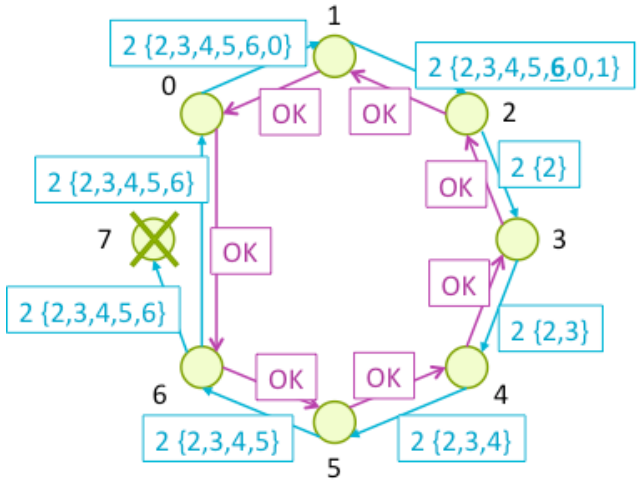


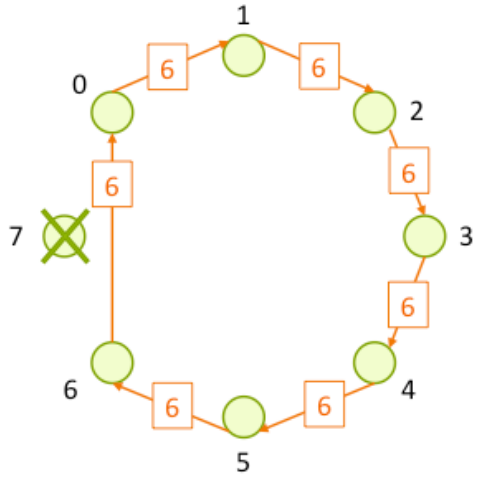
2. Cuando un nodo recibe un mensaje ELECCIÓN, si no es el iniciador:
   - Responde al nodo emisor con un mensaje OK
   - Se incluye en la lista de nodos participantes
   - Envía el mensaje al siguiente nodo
   - Si no llega confirmación (OK), lo envía al siguiente del nodo que no responde



Elección de líder:
b) **Algoritmo para anillos**

3. Cuando un nodo recibe un mensaje ELECCIÓN, y es el **iniciador**:
   - Selecciona como líder el nodo de la lista de participantes con mayor identificador
   - Construye un mensaje COORDINADOR
     - Nuevo líder
   - Propaga dicho mensaje por el anillo





Elección de líder: apreciaciones

- Apreciaciones sobre los algoritmos de elección de líder
  - El envío del mensaje **COORDINADOR** es opcional
    - A veces basta con que un nodo sepa que él NO es líder
    - Y que el coordinador sepa que él es el coordinador
  - En el algoritmo para anillos, ese envío del mensaje COORDINADOR puede realizarse a través del anillo o bien como un mensaje **broadcast**

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Consenso

- **Consenso:** definimos el problema como el **acuerdo** que deben alcanzar los nodos participantes en el valor de una **variable**.
- **Casos particulares de consenso:**
  - Elección de líder → ponerse de acuerdo en el valor de la "variable" "*nodo lider*"
  - Pertenencia a grupos → ponerse de acuerdo en el valor de la "variable" "*lista de nodos activos*".
  - Compromiso distribuido → ponerse de acuerdo en el valor de la variable booleana "*commit/rollback*"
  - Difusión ordenada de mensajes → ponerse de acuerdo en el valor de la(s) variables "*número de secuencia del mensaje*".
  - etc….

Algoritmos de Consenso

- Existen actualmente implementaciones muy relevantes y conocidas de consenso aplicado a replicación y eleccion de líder en sistemas en la nube
- **Ejemplos**
  - **Paxos** [Lamport, 1998] [Lamport, 2001]
    - Dominante durante las últimas décadas
    - Usado en Amazon…
  - **Raft** [Ongaro 2014] → usado en Facebook
    - Desarrollado para ser fácil de entender.
    - Equivalente a Paxos en tolerancia a fallos y desempeño.
    - Listado de más de 50 implementaciones open source de Raft

    (https://raft.github.io/)

Consenso

Definición del Problema del Consenso:

- N nodos tratan de ponerse de acuerdo en el valor de cierta **Variable V.**
- Todos los nodos reciben la "orden" **INICIAR** aproximadamente al mismo tiempo
  - Es decir partimos de la situación en que suponemos que todos los nodos quieren participar en el consenso al comenzar a ejecutar sus algoritmos
- Cada nodo "i" tiene una estimación inicial de la variable : **estimate (Vi)**
- Después de ejecutarse el algoritmo de consenso, todos deben proporcionar como salida "**decision (Vj)**", siendo "j" el nodo que propuso la estimación.

- No es necesario que se conozca "j", pero es necesario que "j" exista
  - El acuerdo debe haber surgido adoptando la estimación de algún nodo
  - No es consenso decidir un valor "derivado" o generado a partir de las estimaciones → en esos caso se trata de problemas diferentes a consenso.

Consenso

- **Condiciones de correción de consenso** → una solución correcta debe cumplir estas cuatro propiedades:
  - (viveza) **Terminación:** todo nodo correcto tarde o temprano decide algún valor.
  - (seguridad) **Integridad uniforme:** todo nodo decide como máximo una vez.
  - (seguridad) **Acuerdo:** ningún par de nodos correctos decide de manera diferente.
  - (seguridad) **Validez uniforme**: si un nodo decide **v**, entonces **v** fue propuesto por algún nodo.

Consenso

- **Algoritmo distribuido (simplificado)**
  - Suponemos N nodos en red totalmente conexa.
  - De los N nodos, algunos nodos pueden estar "apagados", por haber fallado previamente, pero se desconoce quienes.
  - Suponemos que durante la ejecución del algoritmo, ningún nodo falla.

  - **Ejemplo de algoritmo:**
    - Todos los nodos difunden su "estimate(V)"
    - Todos los nodos deciden como "decision(V)" el estimate propuesto por el nodo con identificador más bajo.

Contenido

- Estados Globales y Tiempo
  - Relojes, eventos y estados
  - Algoritmos de sincronización de relojes: Cristian, Berkeley
  - Relojes lógicos y Relojes vectoriales
  - Estados Globales
- Ejemplos de Problemas Algorítmicos en SD
  - Exclusión mutua
  - Elección de líder
  - Consenso
- Algoritmos en presencia de fallos

Algoritmos en presencia de fallos

- Es la mayor fuente de complejidad en el diseño de algoritmos.
  - En la mayoría de casos, hemos supuesto que no existen fallos (Chandy-Lamport, Berkely, Cristian, Exclusión mutua, etc).... o que existen nodos que han fallado previamente a la ejecución de los algoritmos (elección de líder, consenso)
- Es una suposición poco realista.
  - Los nodos fallan y pueden fallar durante la ejecución de los algoritmos.
- Los nodos pueden fallar de diversas formas.
  - El tipo de fallos más simple, y aun así bastante realista es el tipo de **fallos de parada** o "crash"
    - Los nodos fallan en cierto momento y cuando fallan, ya no vuelven a ejecutarse.
  - No consideramos fallos "arbitrarios", donde los nodos pueden comportarse de forma errónea e impredecible → esto da lugar a otra gama de algoritmos

Algoritmos de Consenso distribuido considerando fallos

- Suponemos N nodos en red totalmente conexa.
- De los N nodos, algunos nodos pueden estar "apagados", por haber fallado previamente, pero se desconoce quienes.
- **Suponemos que durante la ejecución del algoritmo, los nodos pueden fallar con fallo de parada.**
- Suponemos que los nodos disponen de un "**detector de fallos**", basado en "timeouts", para reconocer cuando algún nodo ha fallado
  - Nótese que un *timeout* muy ajustado puede hacer creer (erróneamente) a cierto nodo que otro nodo ha fallado.
- Para solucionar la mayoría de problemas en algoritmos distribuidos en presencia de fallos, suponemos que los temporizadores están "bien" ajustados
  - Más tarde o más temprano, todos los nodos "correctos" observarán que los demás nodos correctos, son correctos
  - Es decir, antes o después los temporizadores están bien ajustados.
  - Llamamos a estos temporizadores bien ajustados, como "**detectores de fallos eventualmente perfectos**".

- **Algoritmo básico** (esquema que se utiliza en Paxos)
  - Los nodos están numerados y ordenados, del 0 al N-1.
  - Los nodos pueden estar funcionando o estar "apagados", o fallar durante el algoritmo.
  - Se toleran $\lfloor (N-1)/2 \rfloor$ fallos
    - Es decir, suelo de (N-1)/2 ... así con 3 nodos toleramos 1 fallo; con 4 nodos, 1 fallo; con 5 nodos, 2 fallos…
  - Por tanto **solo funciona** si hay $\lceil (N+1)/2 \rceil$ **nodos correctos**, donde $\lceil X \rceil$ es el techo del número real x.
    - Ejemplo: con 5 nodos, el algoritmo funciona si hay 3 correctos.

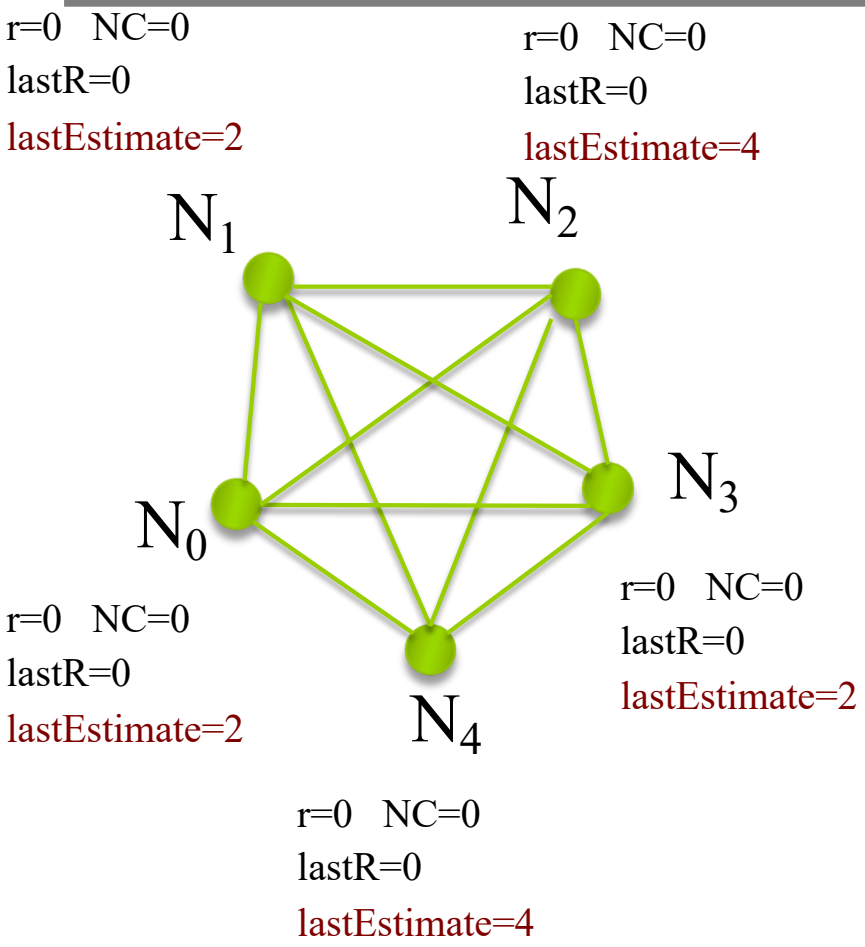
- **Algoritmo básico** (esquema que se utiliza en Paxos)
  - Se van ejecutando rondas hasta que se puede ejecutar una ronda donde participan exitosamente al menos $\lceil (N+1)/2 \rceil$ nodos
    - Sólo pueden fallar realmente $\lfloor (N-1)/2 \rfloor$ nodos, pero durante ciertas rondas, es posible que los diferentes detectores de fallos no estén bien ajustados.... hasta que eventualmente los detectores de fallos estén bien ajustados.
  - Si fallan más nodos de los permitidos, el algoritmo NO funciona, pues los nodos coordinadores se bloquean.

- **Funcionamiento General**
  - Los nodos irán ejecutando rondas hasta que emitan "decision(V)".
  - En cada ronda seleccionan como coordinador al nodo siguiente
    - Es decir, ronda 0 coordinador 0, ronda 1 coordinador 1... ronda 2 coordinador 2.... en ronda K, coordinador (K mod N)
  - En cada ronda distinguimos lo que hace el **coordinador** de la ronda y los demás nodos (**nodos ordinarios**).
  - Todos los nodos mantienen estas variables:
    - ronda actual (**r**)
    - coordinador actual (**NC**)
    - valor que el nodo propone a la variable (**lastEstimate**)
    - ronda más reciente que me hizo cambiar de propuesta (**lastR**)

- **Inicio**
  - Todos los nodos inician **r=0**, **NC=0**, **lastR =0**, **lastEstimate=** (valor que propone cada nodo)
- **En cada ronda "r":**
  - **Fase 1:** Todos los nodos envían al coordinador de la ronda NC: "**estimate (r, lastEstimate, lastR)**"
  - Los nodos ordinarios esperan como respuesta un mensaje "propose".
    - Esperan el mensaje de respuesta un tiempo máximo.
    - Si se excede el tiempo, decimos que su "detector de fallos" cree que el coordinador ha fallado
    - Nótese de nuevo que quizás no ha fallado, pero el *timeout* venció.

- **En cada ronda "r":**
  - **Fase 2:** El coordinador, espera a recibir $\lceil (N+1)/2 \rceil$ mensajes "estimate".
    - Nótese que no se usan timeouts, ni detectores de fallos
    - El coordinador sabe que va a recibir como mínimo ese número de mensajes, pues no admitimos más número de fallos posibles.
    - Mientras no reciba esa cantidad de mensajes, el coordinador se queda bloqueado
      - El algoritmo ya no progresa
    - Ejemplo: Si tenemos 5 nodos, el coordinador esperará a recibir 3 mensajes "estimate", incluyendo el suyo propio.

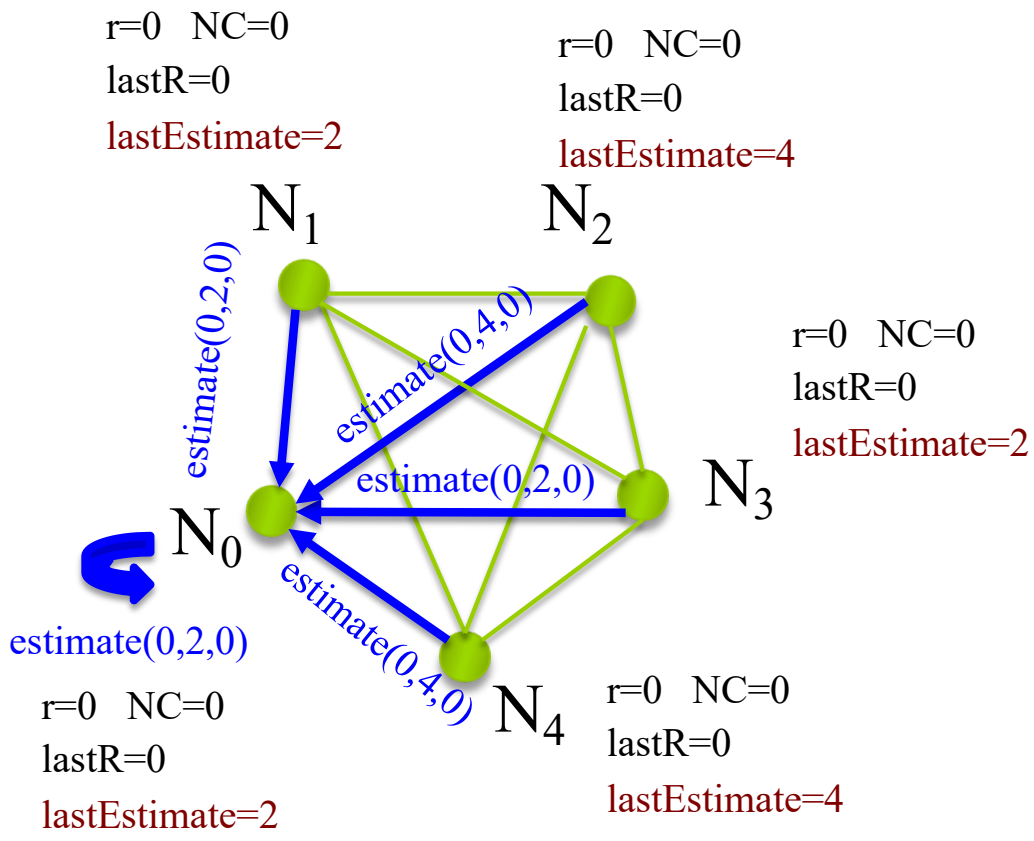
Algoritmos de Consenso distribuido considerando fallos - Ejemplo



INICIO

Funciona si al menos hay

|(5+1)/2|=3 nodos correctos

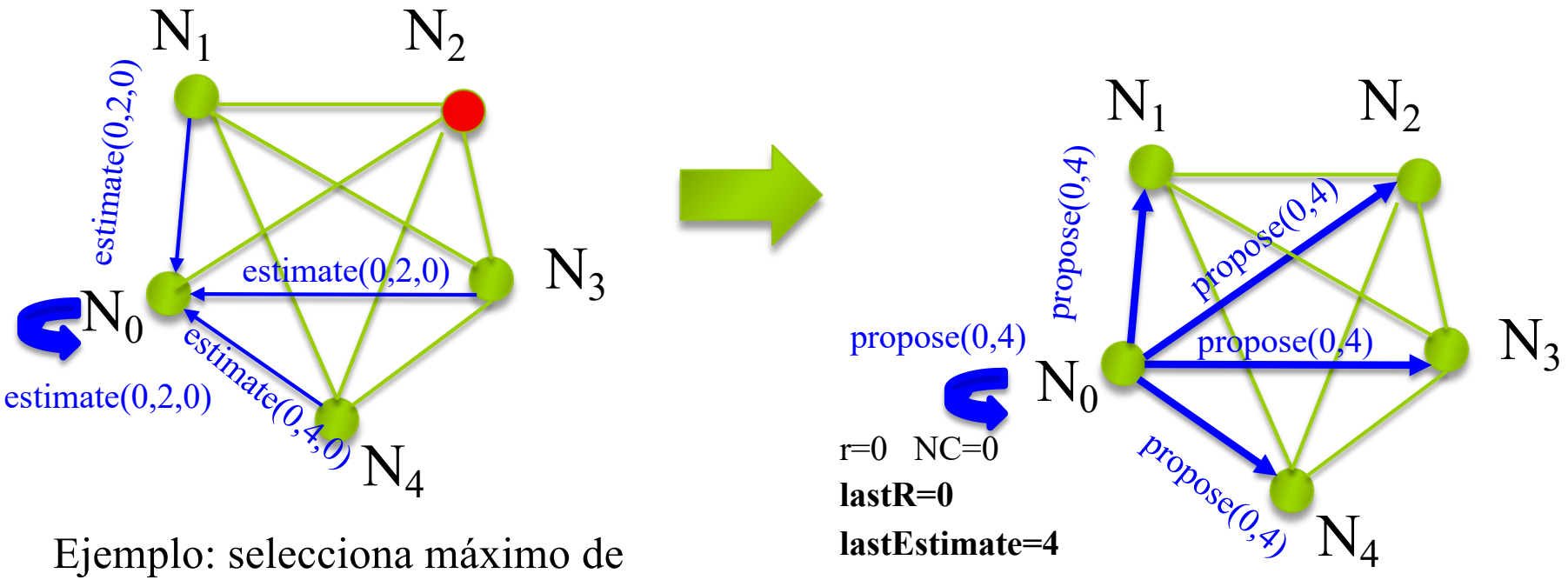


RONDA 0

Todos los nodos envían

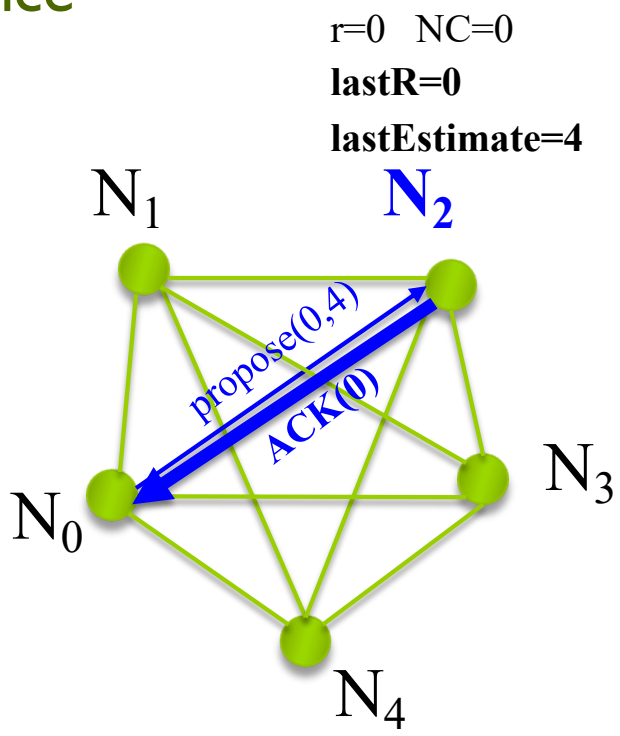
estimate(r,lastEstimate,lastR) a NC

- El coordinador elige uno de los valores "lastEstimate" que recibe, de entre los que tengan el máximo valor "lastR", y ésta será la propuesta de la ronda r
  1. Asigna su valor "**lastEstimate**" a este valor elegido
  2. Asigna **lastR = r** (su propia ronda, no el valor "lastR recibido")
  3. El coodinador difunde "**propose (r, lastEstimate)**".



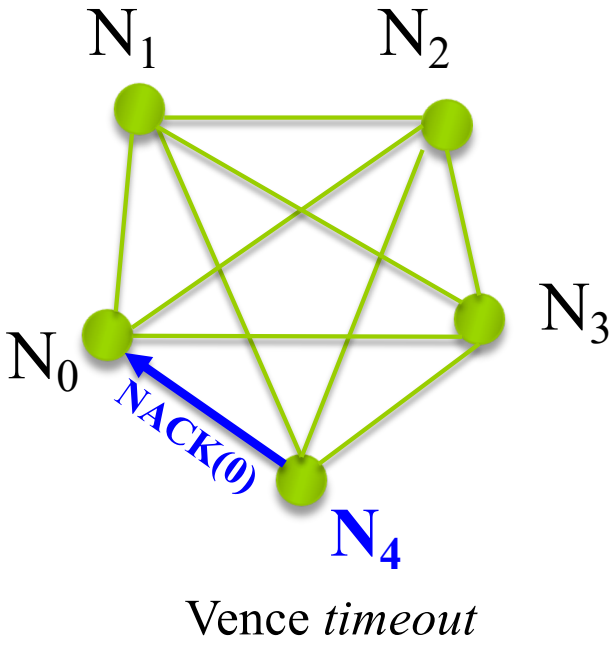
Ejemplo: selecciona máximo de valores recibidos → Valor 4

- **Fase 3:** Los nodos ordinarios esperan a recibir "propose (r, proposeR)" del coordinador o vence su timeout" de espera máxima.
- a) Si reciben "propose" entonces:
  1. Contestan con "**ACK (r)**" a NC.
  2. Actualizan **lastEstimate = proposeR**, y **lastR=r**.



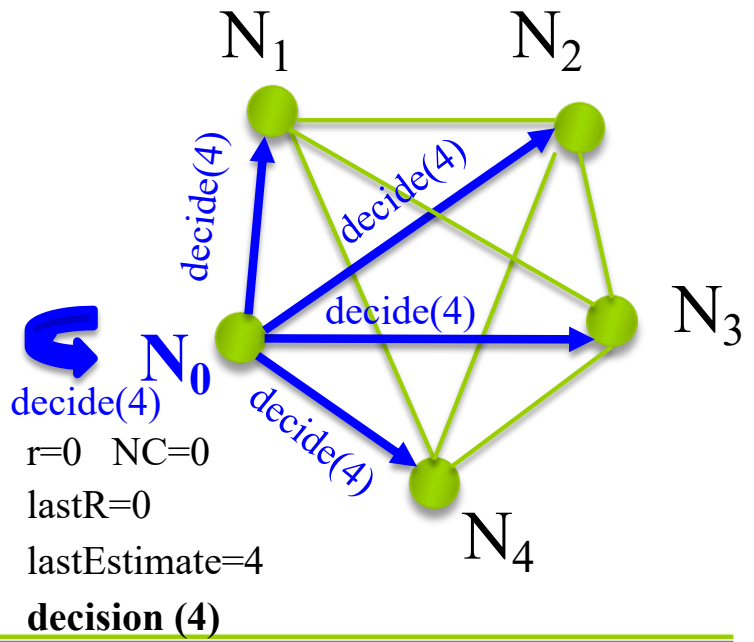
- Con esto el nodo ordinario sabe que el coordinador propuso un valor porque recibió el numero suficiente de *estimates*....
- Se trata de una buena estimación, que quizás ya fue aceptada
- El **nodo la adopta como su nueva estimación**, recordando la ronda en que sucedió.

- **Fase 3:** Los nodos ordinarios esperan a recibir "propose (r, proposeR)" del coordinador o vence su "timeout" de espera máxima.
- b) Si vence su *timeout* esperando propuesta, contesta con "**NACK(r)**" a NC.
  - Nótese que envían NACK aunque su "detector" de fallos crea que el coordinador ha fallado
  - Esto garantiza que el coordinador recibirá $\lceil (N+1)/2 \rceil$ respuestas ACK o NACK, pues más nodos no pueden haber fallado.



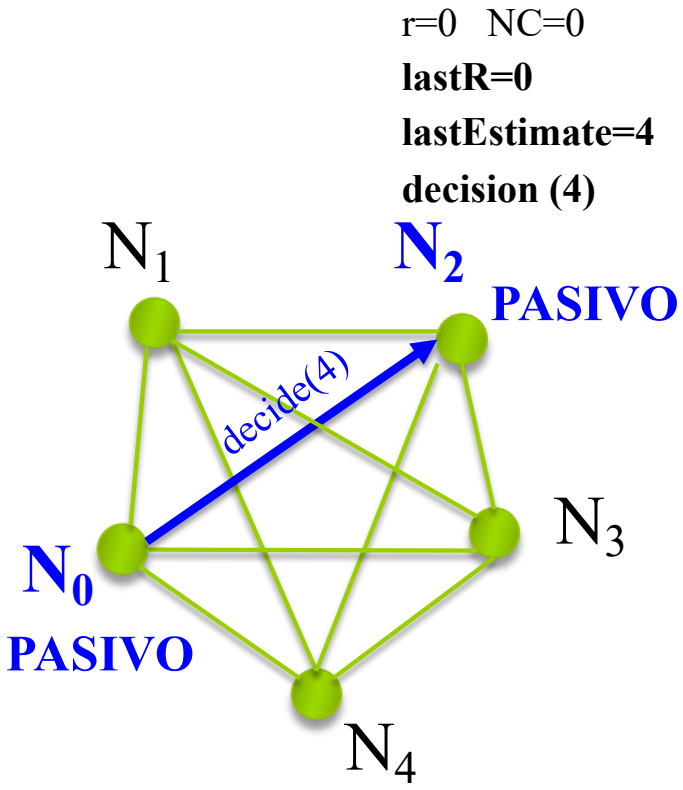
- **Fase 4:** El coordinador espera respuestas ACK o NACK de los nodos ordinarios.
  - Espera $\lceil (N+1)/2 \rceil$ mensajes, de nuevo sin usar timeout, pues sabe que al menos debe recibir esa cantidad de mensajes.
- Si el coordinador recibe $\lceil (N+1)/2 \rceil$ mensajes **ACK**:
  - difunde "**decide (lastEstimate)**" y genera "**decision (lastEstimate)**"
  - Este nodo ya sabe que se ha alcanzado consenso → este es el valor final.

Ejemplo: NC ha recibido 3 ACK (incluyendo el suyo)
Necesita recibir: |(5+1)/2|=3 mensajes ACK

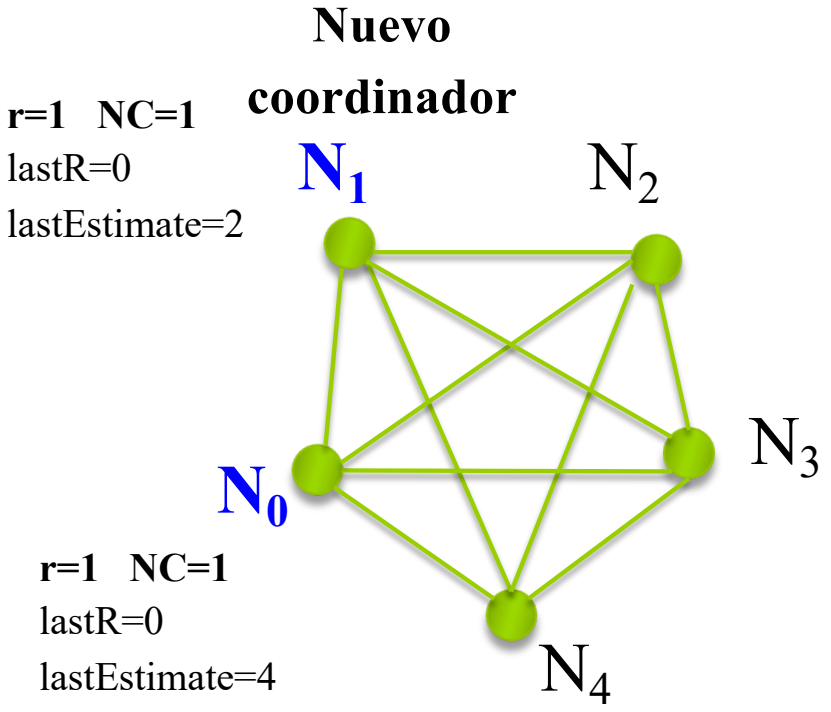


- Si un nodo ordinario recibe "**decide (lastEstimate)**", genera "**decision (lastEstimate)**" → este nodo ya sabe que se ha alcanzado consenso.

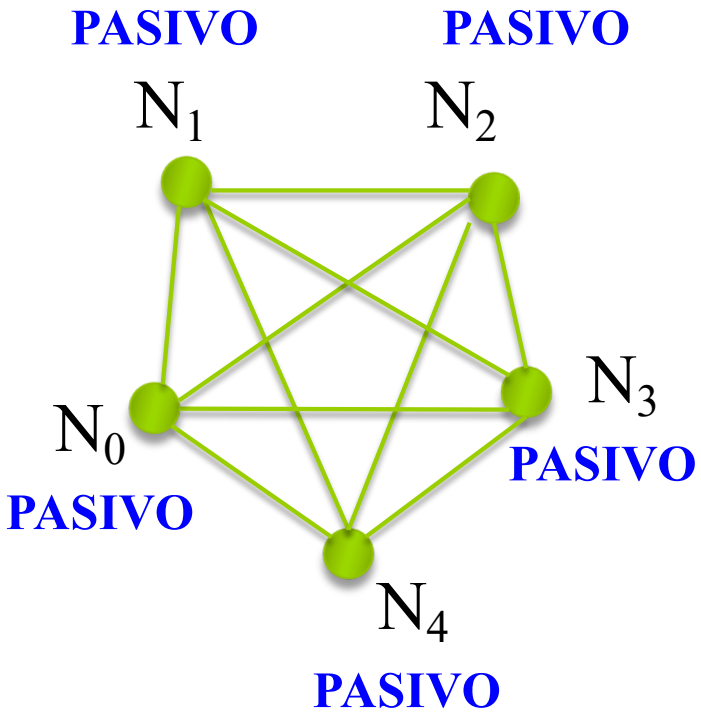
- Todos los nodos que han generado "decision" siguen participando en el algoritmo como nodos **PASIVOS**.
  - Cuando reciben un mensaje de tipo propose(r, proposeR) responden "**ACK**" siempre que **proposeR==lastEstimate** y **r>=lastR**.
  - Es decir, que coincida el *proposeR* con su decisión y la ronda del propose sea igual o superior a la ronda en la que decidió.
  - De esta forma ayudan a otros nodos correctos a decidir en sus futuras rondas.



- **Cambios de ronda**
  - Coordinador: tras Fase 4, si no recibe el número suficiente de "**ACK**", incrementa la ronda y ya no será coordinador.
  - Ordinario: tras Fase 3, los nodos ordinarios incrementan ronda y se convierten en el nuevo coordinador si les corresponde.



- **Finalización eventual**
  - El algoritmo termina cuando todos los nodos correctos están en **MODO PASIVO**.
    - Nótese que no se sabe cuándo termina el algoritmo, pero antes o después, **todo nodo correcto habrá generado la misma decisión**
    - Y antes o después todo nodo correcto estará en modo **PASIVO**, sin generar nuevos mensajes ni ejecutar más código.
    - Se puede observar que antes o después (eventualmente) todos los nodos generan "decision", sobre el mismo valor, tan pronto como los detectores de fallos de $\lceil (N+1)/2 \rceil$ nodos observen que el coordinador de esa ronda está operativo.



- **Conclusión**
  - Para alcanzar consenso en presencia de fallos, necesitamos tener "timeouts" bien ajustados, o al menos, timeouts eventualmente bien ajustados.
  - **Ejemplos de Consenso→** elección de lider, compromiso atómico, orden en mensajes, pertenencia a grupos, etc.
  - Para la gran mayoría de algoritmos realistas y sistemas realistas necesitamos **detectores de fallos** que "antes o después" sean bastante "buenos"
    - Detecten como fallos los nodos que realmente han fallado y no detecten como fallidos a los nodos correctos.
  - El tratamiento de fallos es complejo y su tratamiento completo escapa a esta asignatura.

Resultados de aprendizaje de la Unidad Didáctica

- Al finalizar esta unidad, el alumno deberá ser capaz de:
  - Describir los problemas que comporta la gestión del tiempo en un entorno distribuido.
    - Identificar las ventajas introducidas por una ordenación lógica de los eventos en una aplicación distribuida.
  - Ilustrar las soluciones clásicas para algunos problemas de sincronización en un entorno distribuido:
    - Imagen del estado global
    - Elección de líder
    - Exclusión mutua.
  - Describir el concepto de consenso e ilustrar soluciones de algoritmos de consenso.
  - Describir los algoritmos de consenso distribuido considerando fallos.

Bibliografía

- Relojes Atómicos:
  - **"El hombre que ajusta la hora en España"**. El Mundo, Martes 30 de Junio de 2015. Sección Ciencia, páginas 29 – 30.
- Algoritmo de Cristian:
  - Flaviu Cristian. **Probabilistic clock synchronization**. Distributed Computing, 3(3):146–158, 1989.
- Algoritmo de Berkeley:
  - Riccardo Gusella y Stefano Zatti. **The accuracy of the clock synchronization achieved by TEMPO in Berkeley** UNIX 4.3BSD. IEEE Trans. Software Eng., 15(7):847–853, 1989
- Relojes lógicos de Lamport:
  - Leslie Lamport. **Time, clocks, and the ordering of events in a distributed system**. Communications of the ACM, 21(7):558–565, 1978.
- Relojes vectoriales:
  - D. Stott Parker Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser y C. Kline. **Detection of mutual inconsistency in distributed systems**. IEEE Trans. Software Eng., 9(3):240–247, 1983.

Bibliografía

- Algoritmo de Chandy-Lamport (estado global):
  - K. Mani Chandy y Leslie Lamport. **Distributed snapshots: Determining global states of distributed systems**. ACM Trans. Comput. Syst., 3(1):63–75, 1985.
- Algoritmo Bully:
  - *Algoritmo clásico*: H. Garcia-Molina. **Elections in a distributed computing system**. IEEE Trans. Comput., 1982, C-13, 48-59.
  - *Mejoras al algoritmo Bully clásico:* G. Murshed, A. R. Allen. **Enhanced Bully Algorithm for Leader Node Election in Synchronous Distributed Systems**. Computers, Vol. 1, pp. 3-23, 2012.
- Algoritmo de Anillo para elección de líder:
  - E. Chang, R. Roberts. **An improved algorithm for decentralized extrema-finding in circular configurations of processes**. Communications of the ACM, 22(5): 281-283, ACM, 1979.
- Algoritmo distribuido de exclusión mutua:
  - Glenn Ricart y Ashok K. Agrawala. **An optimal algorithm for mutual exclusion in computer networks**. Commun. ACM, 24(1):9–17, 1981.

Bibliografía

- Algoritmos de consenso para sistemas distribuidos:
  - M. Bakhoff**. Consensus algorithms for distributed systems**.
  - D. Ongaro, J. Ousterhout. **In Search of an Understandable Consensus Algorithm**. Proc. USENIX ATC'14, Philadelphia, 2014.
  - L. Lamport, R. Shostak, M. Pease. **The Byzantine Generals Problem**. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 382-401, 1982.
  - T. D. Chandra, S. Toueg. **Unreliable Failure Detectors for Reliable Distributed Systems**. Journal of the ACM, Vol. 43, no. 2, pp. 225-267, 1996.