06 - Coordinación en sistemas distribuidos

IIC2523 - Sistemas Distribuidos

Cristian Ruz – cruz@ing.puc.cl

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Semestre 2-2020

Contenidos

- Relojes físicos
- Relojes lógicos
- Relojes vectoriales
- Exclusión mutua distribuida
- Algoritmos de elección

Necesidad de coordinación

Necesidad

- Access simultáneos
- Relojes independientes
- Ausencia de un coordinador global

Relojes físicos

Cuando se requiere un tiempo exacto

Relojes físicos

Cuando se requiere un tiempo exacto

Universal Coordinated Time (UTC)

- Basado en transiciones por segundo de un átomo de Cesio 133
- Mantenido por alrededor de 50 relojes atómicos distribuidos en el mundo
- Sin embargos los días se están alargando: *leap seconds*
- Valor puede enviarse vía *broadcast* de radio.
- Satélites tiene exactitud (*accuracy*) de ± 0.5ms

Sincronización de relojes físicos

$C_p(t)$: valor de reloj de máquina p en tiempo UTC t

Precision

La desviación entre dos relojes en máquinas distintas p, q debe estar acotada por un valor de **precisión** π.

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

Accuracy

Accuracy (exactitud) se refiere a mantener acotada la desviación de un reloj respecto al tiempo real.

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

Dos tipos de sincronización física

- Sincronización **iterna** mantiene relojes **precisos** (*precision*)
- Sincronización **externa** mantiene relojes **exactos** (*accurate*)

Clock drift

Clock drift

- Cada reloj posee un **maximum clock drift rate** ρ
- $F(t)$ frecuencia de oscilamiento de reloj en tiempo t
- F frecuencia ideal de reloj (constante)

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

Relojes de *software* y relojes de *hardware*

Un reloj de *software* está asociado a una cantidad de oscilaciones de un reloj de *hardware* y, por lo tanto, a su *drift rate*

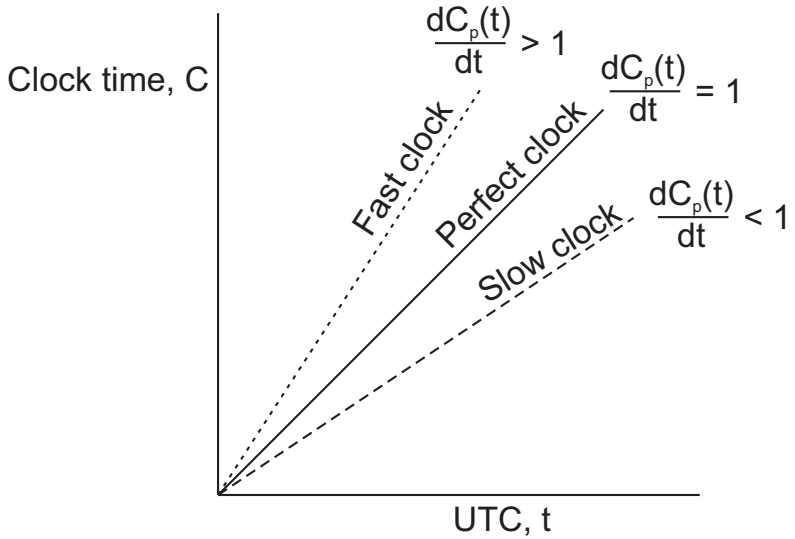
$$C_p(t) = \frac{1}{F} \int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

$$\forall t : (1 - \rho) \leq \frac{dC_p(t)}{dt} \leq (1 + \rho)$$

Clock drift

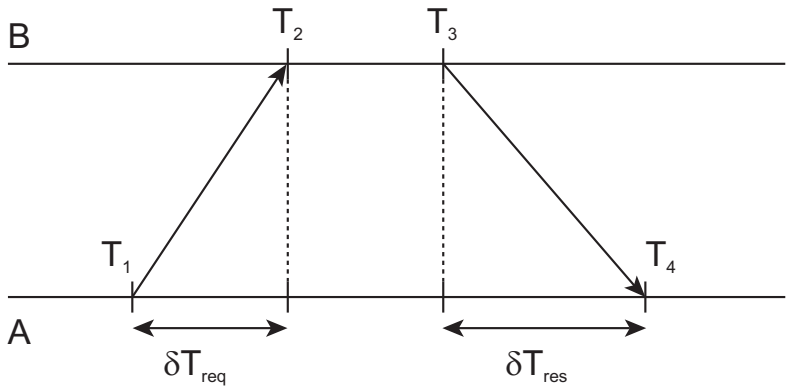
Relación de relojes: rápidos, perfectos, lentos

$$\forall t : (1 - \rho) \leq \frac{dC_p(t)}{dt} \leq (1 + \rho)$$



Ajustando el tiempo

¿Cómo obtener el tiempo desde un *time server*?



Se obtienen valores para *offset* θ (diferencia entre relojes) y *delay* δ (tiempo de propagación)
Supuesto: $\delta T_{\text{req}} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{\text{res}}$

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T4)}{2}$$

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

Network Time Protocol (NTP)

Toma ocho pares (θ, δ) y elige un θ cuyo δ es mínimo

Relojes lógicos

¿Necesitamos estar sincronizados con el tiempo real?

Relojes lógicos

¿Necesitamos estar sincronizados con el tiempo real?

Orden de eventos

En la práctica nos interesa saber **qué ocurrió antes de qué**

Relojes lógicos

¿Necesitamos estar sincronizados con el tiempo real?

Orden de eventos

En la práctica nos interesa saber **qué ocurrió antes de qué**

Relación *happens-before*: $\rightarrow$, (Lamport 1978)

- Si a y b del mismo proceso, y a ocurre antes que b, entonces $a \rightarrow b$.
- Si m es un mensaje, a es un send(m), y b un recv(m), entonces $a \rightarrow b$
- Si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$

Relojes lógicos

¿Necesitamos estar sincronizados con el tiempo real?

Orden de eventos

En la práctica nos interesa saber **qué ocurrió antes de qué**

Relación *happens-before*: $\rightarrow$, (Lamport 1978)

- Si a y b del mismo proceso, y a ocurre antes que b, entonces $a \rightarrow b$.
- Si m es un mensaje, a es un send(m), y b un recv(m), entonces $a \rightarrow b$
- Si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$

Orden parcial

La relación *happens-before* introduce un **orden parcial de eventos** en un sistema con procesos concurrentes.

Relojes lógicos

¿Cómo mantener una "visión global" del sistema que sea consistente con *happens-before*?

Relojes lógicos

¿Cómo mantener una "visión global" del sistema que sea consistente con *happens-before*?

¿Cómo mantener la relación *happens-before*?

Agregar un *timestamp* $C(e)$ a cada evento e tal que:

- **P1** Si a y b son dos eventos del mismo procesos, entonces $a \rightarrow b$, y debe cumplirse $C(a) < C(b)$
- **P2** Si a es send(m) y b es recv(m), entonces debe cumplirse $C(a) < C(b)$

Relojes lógicos

¿Cómo mantener una "visión global" del sistema que sea consistente con *happens-before*?

¿Cómo mantener la relación *happens-before*?

Agregar un *timestamp* $C(e)$ a cada evento e tal que:

- **P1** Si a y b son dos eventos del mismo procesos, entonces $a \rightarrow b$, y debe cumplirse $C(a) < C(b)$
- **P2** Si a es send(m) y b es recv(m), entonces debe cumplirse $C(a) < C(b)$

Sin embargo, no hay un reloj global que permita mantener los relojes lógicos consistentes.

Relojes lógicos

Algoritmo para mantener relojes lógicos

Para cada procesos P_i, mantener un **contador local** C_i, y ajustarlo:

1. Por cada evento nuevo que ocurre en P_i, incrementar C_i en 1
2. Cada vez que P_i envía un mensaje m, el mensaje recibe un *timestamp* $ts(m) = C_i$
3. Cada vez que P_j recibe un mensaje m, P_j ajusta C_j a $\max\{C_j, ts(m)\}$, y ejecuta el paso 1 antes de entregar el mensaje.

Relojes lógicos

Algoritmo para mantener relojes lógicos

Para cada procesos P_i, mantener un **contador local** C_i, y ajustarlo:

1. Por cada evento nuevo que ocurre en P_i, incrementar C_i en 1
2. Cada vez que P_i envía un mensaje m, el mensaje recibe un *timestamp* $ts(m) = C_i$
3. Cada vez que P_j recibe un mensaje m, P_j ajusta C_j a $\max\{C_j, ts(m)\}$, y ejecuta el paso 1 antes de entregar el mensaje.

¿Se cumplen las propiedades?

P1 Si a y b son dos eventos del mismo procesos, entonces $a \rightarrow b$, y debe cumplirse $C(a) < C(b)$

P2 Si a es send(m) y b es recv(m), entonces debe cumplirse $C(a) < C(b)$

Relojes lógicos

Algoritmo para mantener relojes lógicos

Para cada procesos P_i, mantener un **contador local** C_i, y ajustarlo:

1. Por cada evento nuevo que ocurre en P_i, incrementar C_i en 1
2. Cada vez que P_i envía un mensaje m, el mensaje recibe un *timestamp* $ts(m) = C_i$
3. Cada vez que P_j recibe un mensaje m, P_j ajusta C_j a $\max\{C_j, ts(m)\}$, y ejecuta el paso 1 antes de entregar el mensaje.

¿Se cumplen las propiedades?

P1 Si a y b son dos eventos del mismo procesos, entonces $a \rightarrow b$, y debe cumplirse $C(a) < C(b)$

P2 Si a es send(m) y b es recv(m), entonces debe cumplirse $C(a) < C(b)$

Paso 1 mantiene la propiedad **P1**

Pasos 2 y 3 mantienen la propiedad **P2**

Relojes lógicos

Algoritmo para mantener relojes lógicos

Para cada procesos P_i, mantener un **contador local** C_i, y ajustarlo:

1. Por cada evento nuevo que ocurre en P_i, incrementar C_i en 1
2. Cada vez que P_i envía un mensaje m, el mensaje recibe un *timestamp* $ts(m) = C_i$
3. Cada vez que P_j recibe un mensaje m, P_j ajusta C_j a $\max\{C_j, ts(m)\}$, y ejecuta el paso 1 antes de entregar el mensaje.

¿Se cumplen las propiedades?

P1 Si a y b son dos eventos del mismo procesos, entonces $a \rightarrow b$, y debe cumplirse $C(a) < C(b)$

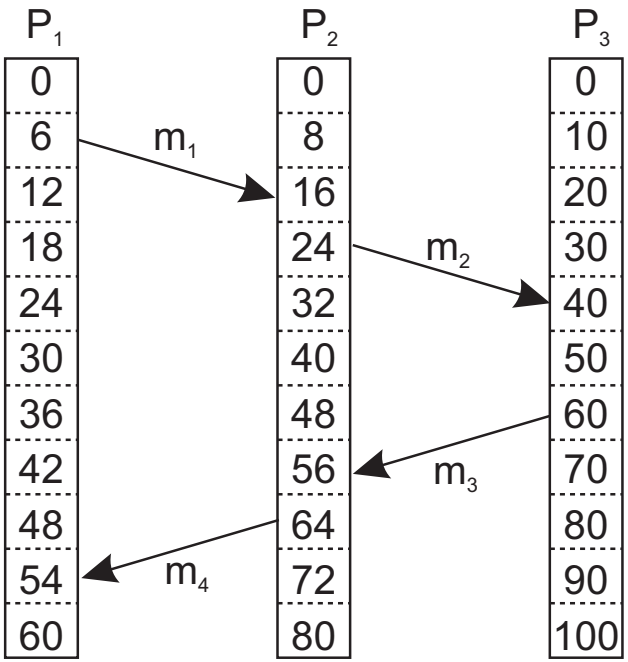
P2 Si a es send(m) y b es recv(m), entonces debe cumplirse $C(a) < C(b)$

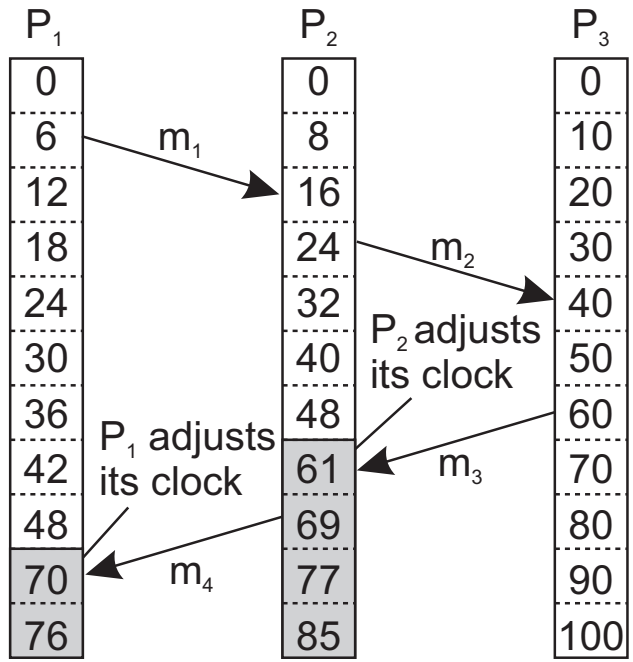
Paso 1 mantiene la propiedad **P1**

Pasos 2 y 3 mantienen la propiedad **P2**

¿Pueden dos eventos tener el mismo $C_k(e)$?

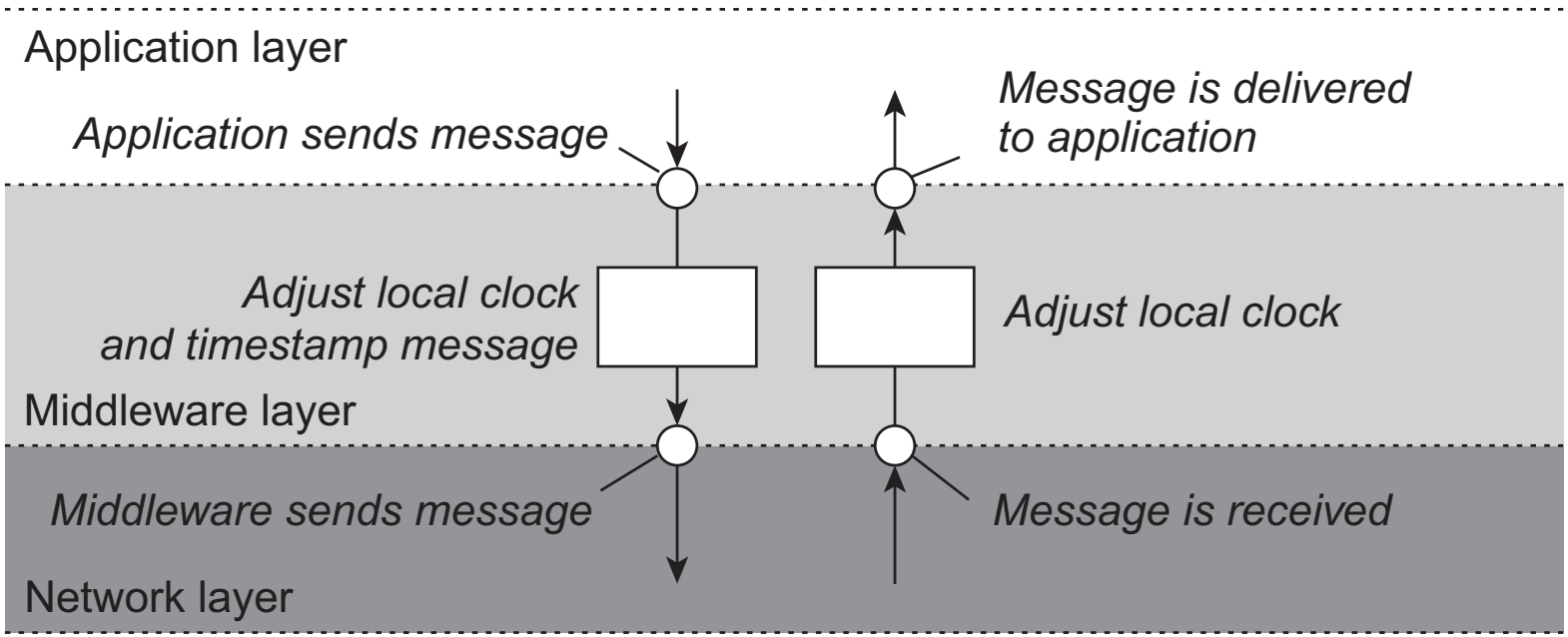
Relojes lógicos





Relojes lógicos

¿Cuándo se actualizan los relojes lógicos?

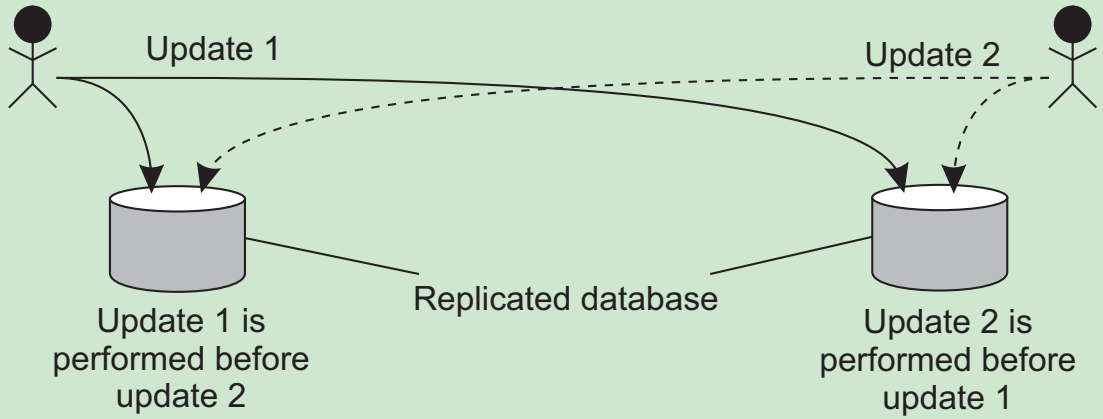


Ejemplo: *Totally-ordered multicast*

Cuando una BD replicada recibe *updates* concurrentes, éstos deben verse en el mismo orden para todos.

Dos *updates* concurrentes a una BD replicada

- Cuenta tiene \$1000
- P_1 suma \$100 a la cuenta
- P_2 incrementa cuenta en 1%



¿Cuál es el resultado final sobre la BD?

Ejemplo: *Totally-ordered multicast*

Relojes lógicos al rescate

- P_i envía mensaje m_i con *timestamp* a todos los demás (incluido él mismo)
- m_i queda en una cola local $queue_i$
- Cualquier mensaje P_j que llegue, se encola en $queue_j$ de acuerdo a su *timestamp* y es *acknowledged* por todos los otros procesos.

P_j pasa el mensaje m_i a su aplicación si:

(1) m_i está al inicio $queue_j$

(2) Para cada proceso P_k hay un mensaje m_k en $queue_j$ con *timestamp* $m_k > m_i$

Ejemplo: *Totally-ordered multicast*

Relojes lógicos al rescate

- P_i envía mensaje m_i con *timestamp* a todos los demás (incluido él mismo)
- m_i queda en una cola local $queue_i$
- Cualquier mensaje P_j que llegue, se encola en $queue_j$ de acuerdo a su *timestamp* y es *acknowledged* por todos los otros procesos.

P_j pasa el mensaje m_i a su aplicación si:

(1) m_i está al inicio $queue_j$

(2) Para cada proceso P_k hay un mensaje m_k en $queue_j$ con *timestamp* $m_k > m_i$

Estamos haciendo un supuesto implícito:

Comunicación es **confiable** (**reliable**) y **FIFO-ordered**

Ejemplo: Exclusión mutua con relojes lógicos

```
1  class Process:
2    def __init__(self, chan):
3      self.queue      = []                          # The request queue
4      self.clock      = 0                           # The current logical clock
5
6    def requestToEnter(self):
7      self.clock = self.clock + 1                             # Increment clock value
8      self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
9      self.cleanupQ()                                         # Sort the queue
10     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request
11
12   def allowToEnter(self, requester):
13     self.clock = self.clock + 1                             # Increment clock value
14     self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other
15
16   def release(self):
17     tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
18     self.queue = tmp                                        # and copy to new queue
19     self.clock = self.clock + 1                             # Increment clock value
20     self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release
21
22   def allowedToEnter(self):
23     commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
24     return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

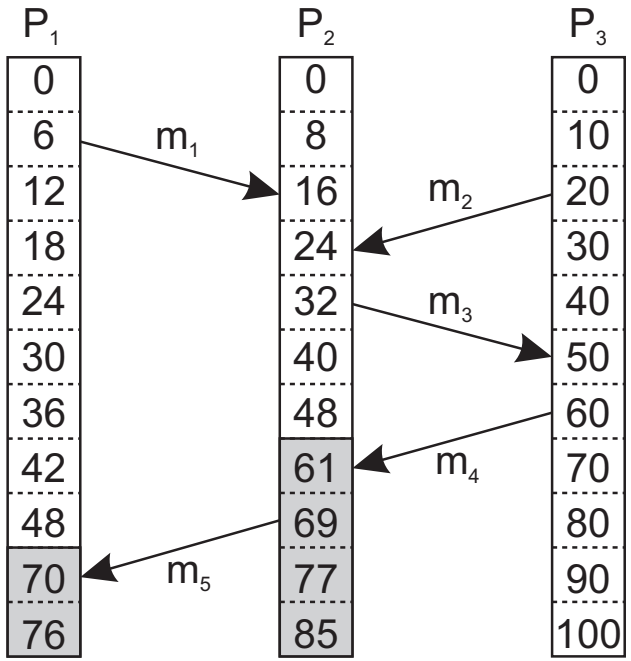
Ejemplo: Exclusión mutua con relojes lógicos

```
1   def receive(self):
2     msg = self.chan.recvFrom(self.otherProcs)[1]        # Pick up any message
3     self.clock = max(self.clock, msg[0])                # Adjust clock value...
4     self.clock = self.clock + 1                         # ...and increment
5     if msg[2] == ENTER:
6       self.queue.append(msg)                            # Append an ENTER request
7       self.allowToEnter(msg[1])                         # and unconditionally allow
8     elif msg[2] == ALLOW:
9       self.queue.append(msg)                            # Append an ALLOW
10    elif msg[2] == RELEASE:
11      del(self.queue[0])                                # Just remove first message
12    self.cleanupQ()                                     # And sort and cleanup
```

Relojes vectoriales

Los relojes lógicos funcionan en un solo sentido

$a \rightarrow b \Rightarrow C(a) < C(b)$, pero $C(a) < C(b) \not\Rightarrow a \rightarrow b$



¿Qué se puede decir de la recepción de m_1 y el envío m_2 ?
¿Están **causalmente** relacionados?

Dependencia causal

Dependencia causal

Decimos que b **puede depender causalmente** de a, si $\text{ts}(a) < \text{ts}(b)$.

Dependencia causal

Dependencia causal

Decimos que b **puede depender causalmente** de a, si $\text{ts}(a) < \text{ts}(b)$.

¿Cómo capturar potenciales dependencias causales?

Dependencia causal

Dependencia causal

Decimos que b **puede depender causalmente** de a, si $\text{ts}(a) < \text{ts}(b)$.

¿Cómo capturar potenciales dependencias causales?

Relojes vectoriales

Cada proceso P_i mantiene un reloj un vector VC_i.

- $VC_i[i]$ es el reloj lógico de P_i
- Si $VC_i[j] = k$, entonces P_i sabe que al menos k eventos han ocurrido en P_j.

Relojes vectoriales

Relojes vectoriales

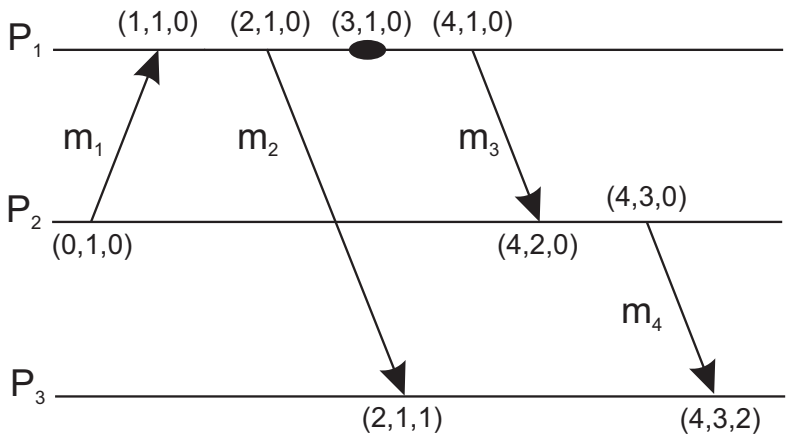
Cada proceso P_i mantiene un reloj un vector VC_i.

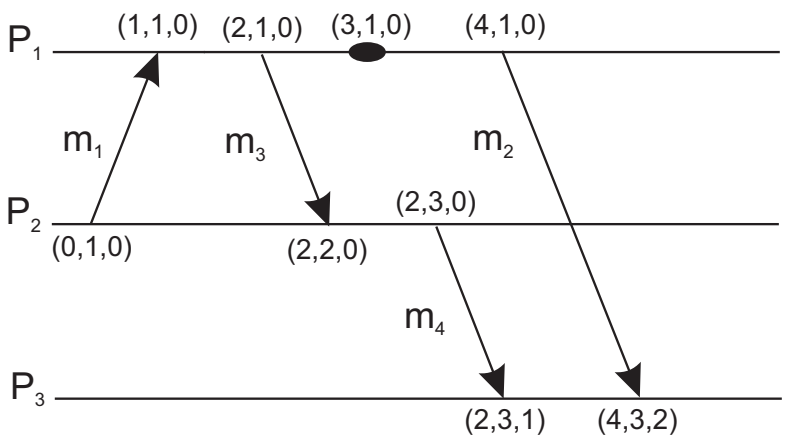
- $VC_i[i]$ es el reloj lógico de P_i
- Si $VC_i[j] = k$, entonces P_i sabe que al menos k eventos han ocurrido en P_j.

¿Cómo mantener relojes vectoriales?

1. Antes de ejecutar un evento, P_i hace $VC_i[i] = VC_i[i] + 1$
2. Cuando P_i envía un mensaje m a P_j, establece el *timestamp* de m a VC_i, luego de haber ejecutado el paso 1.
3. Cuando P_j recibe el mensaje m establece su reloj a $VC_j[k] = \max\{VC_j[k], \text{ts}(m)[k]\}$ para cada k, luego ejecuta el paso 1, y pasa el mensaje a la aplicación.

Relojes vectoriales: ejemplo





¿Qué se puede deducir?

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
Figure 6.13(a)	$(2,1,0)$	$(4,3,0)$	Yes	No	m_2 may causally precede m_4
Figure 6.13(b)	$(4,1,0)$	$(2,3,0)$	No	No	m_2 and m_4 may conflict

Causally-ordered multicast

Usando relojes vectoriales podemos asegurar que: "un mensaje se entregue sólo si todos los mensajes causales precedentes han sido entregados"

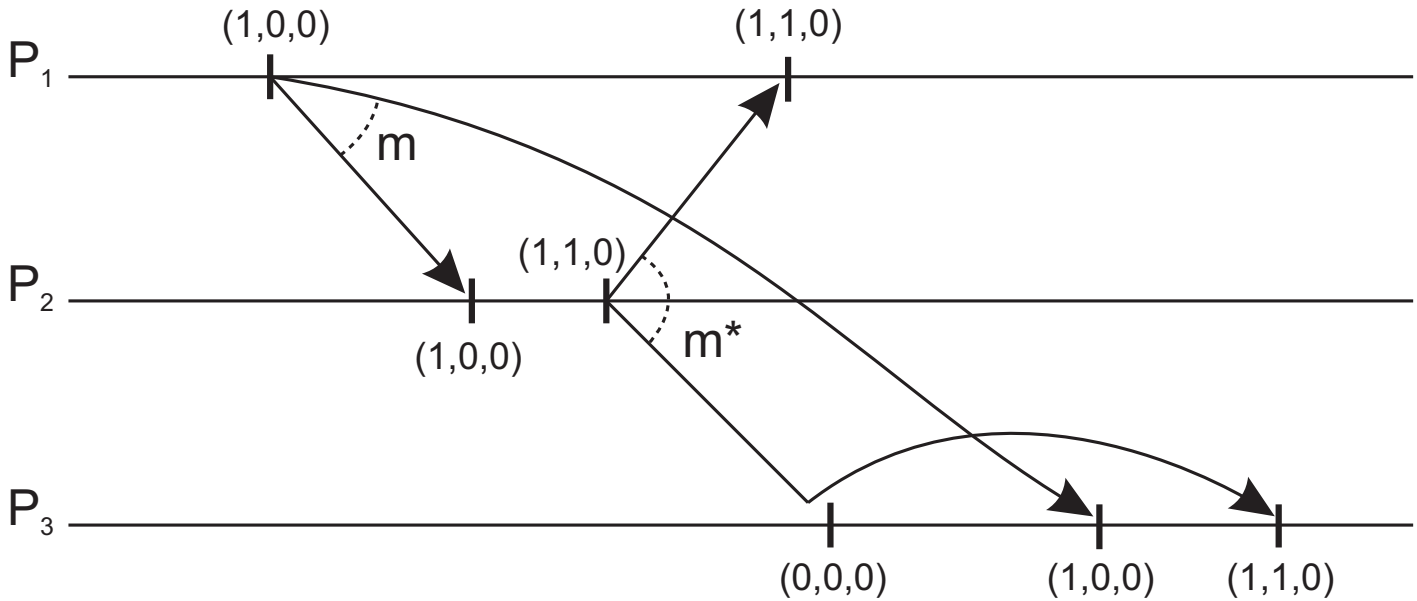
Ajuste . . .

P_i incremente $VC_i[i]$ solo cuando envía un mensajes, y P_j "ajusta' VC_j cuando recibe un mensaje (no modifica $VC_j[j]$)

P_j pospone la entrega de m hasta que ...

- $\text{ts}(m)[i] = VC_j[i] + 1$
- $\text{ts}(m)[k] \leq VC_j[k]$, para todo$k \neq i$

Causally-ordered multicast



Ejemplo

$VC_3 = [0, 2, 2]$, y ts$(m) = [1, 3, 0]$ desde P_1.
¿Qué información posee P_3, y qué hará al recibr m (desde P_1)?

Exclusión mutua

Tenemos un conjunto de procesos en un sistema distribuido que desean acceso exclusivo a algún recurso.

Exclusión mutua

Tenemos un conjunto de procesos en un sistema distribuido que desean acceso exclusivo a algún recurso.

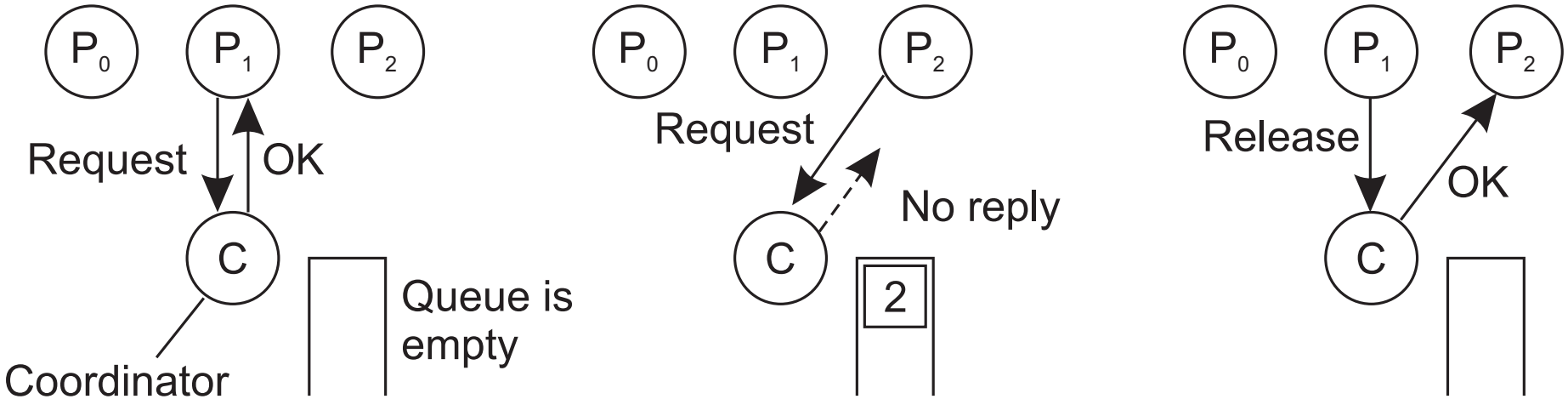
Dos tipos de soluciones

Usar permisos: Proceso debe solicitar permiso a otros procesos

Usar *token*: Procesos se pasan un *token* que confiere permiso. El que no lo quiere usar, lo pasa al siguiente.

Permisos. Solución centralizada

Usar un coordinador.



Ejemplo con coordinador

1. P_1 pide permiso al coordinador. Coordinador se lo otorga.
2. P_2 pide permiso al mismo recurso. Coordinador no responde.
3. P_1 libera el recurso y avisa al coordinador. Coordinador responde a P_2

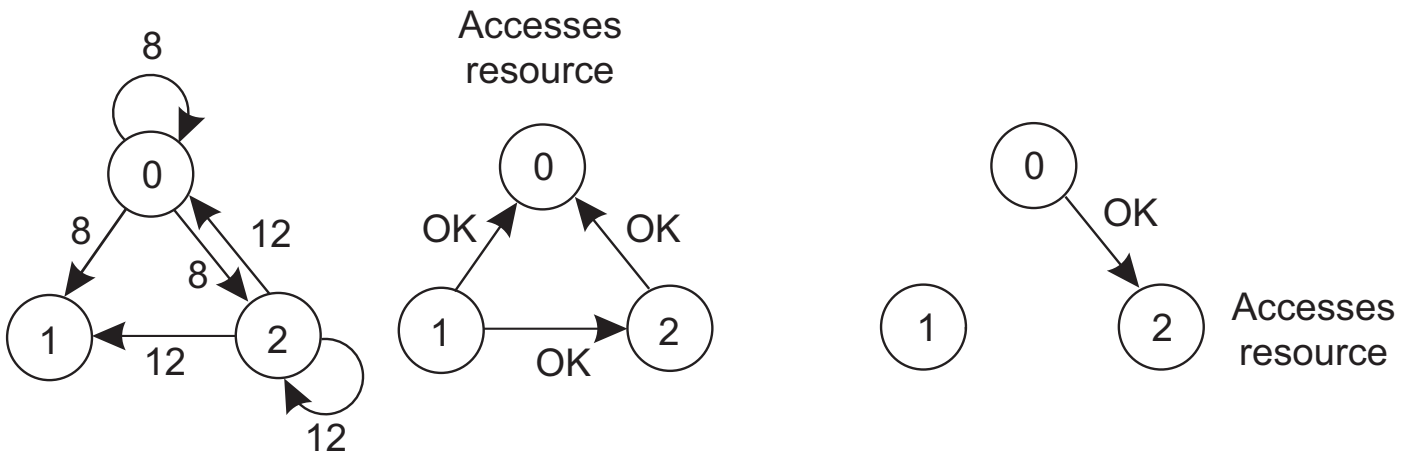
Permisos. Solución distribuida: Ricart & Agrawala

Similar al algoritmo de Lamport para *totally-ordered multicast*.

Algoritmo de Ricart & Agrawala

- Proceso(s) interesado(s) pide(n) acceso enviando solicitud a todos.
- Proceso receptor responde en dos casos:
  - El receptor no desea acceder al recurso
  - El receptor está esperando el recurso, pero posee menor prioridad. Prioridad se define de acuerdo a *timestamps*
- Si no se cumple alguna de las condiciones anterior, el receptor posterga la respuesta.

Permisos. Solución distribuida: Ricart & Agrawala

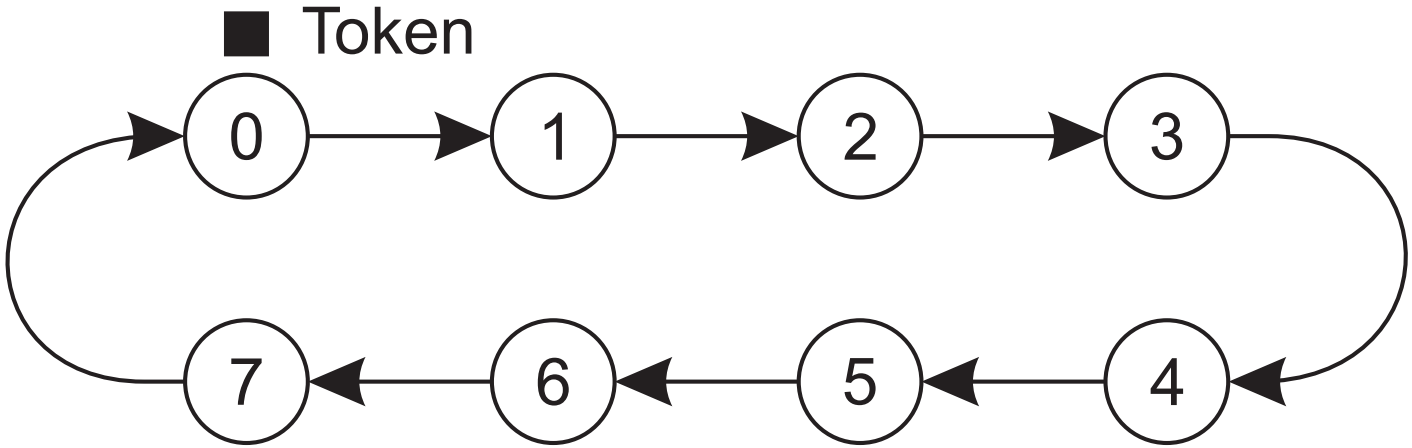


Ejemplo con coordinador

1. Dos procesos desde acceder al mismo recurso simultáneamente.
2. P_0 tiene menor *timestamp* y gana
3. Cuando P_0 termina, envía un *OK*, y P_2 puede continuar.

Token. Algoritmo *Token-ring*

Procesos organizados en un **anillo lógico**. El *token* se pasa entre ellos. El que posee el *token* puede entrar (si lo desea)



Exclusión mutua descentralizada

Idea

Cada recurso está replicado N veces, y cada réplica tiene su propio coordinador.
Para acceder se require un **voto de mayoría** de $m > \frac{N}{2}$ coordinadores.
Los coordinadores responden de manera inmediata.

Supuesto: si un coordinador falla, se recupera rápidamente. Sin embargo olvida los permisos que ha entregado.

Exclusión mutua descentralizada

¿Funciona?

- Sea $\rho = \Delta / T$ la probabilidad que un coordinador se "resetee" en un intervalo Δ durante un tiempo de vida T
- La probabilidad $\mathcal{P}[k]$ que k de entre m coordinadores se reseteen en el mismo intervalo es:

$$\mathcal{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- Si f coordinadores se "resetean", entonces la correctitud falla cuando hay un minoría de coordinador: $m - f \leq N/2$, o $f \geq m - N/2$
- Probabilidad de falla: $\sum_{k=m-N/2}^{N} \mathcal{P}[k]$

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-10}$
8	6	30 sec/hour	$< 10^{-11}$
16	9	30 sec/hour	$< 10^{-18}$
16	12	30 sec/hour	$< 10^{-24}$
32	17	30 sec/hour	$< 10^{-35}$
32	24	30 sec/hour	$< 10^{-49}$

Exclusión mutua: resumen

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

Algoritmos de elección

¿Cómo elegir dinámicamente a un proceso entre varios?

- Coordinador, líder, etc
- Si se elige estáticamente, se comporta como un punto centralizado de falla

¿Cómo estar seguro que **todos** los miembros están de acuerdo en el elegido?

Supuestos

- ID's únicos
- Cada proceso conoce los ID's de todos los procesos, pero no sabe cuales están presentes
- Una manera de hacer elección es elegir el de mayor ID

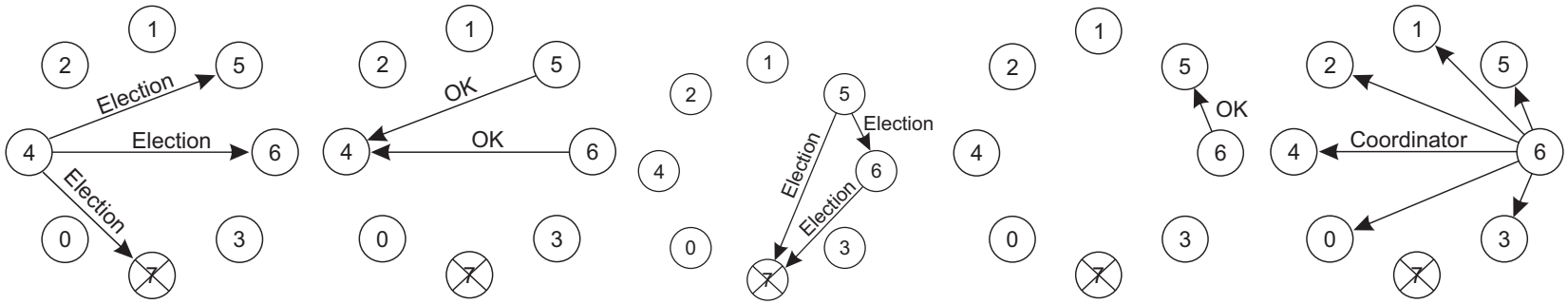
Algoritmos de elección: *bullying*

Elección por *bullying*

N procesos $\{P_0, P_1, \ldots, P_{N-1}\}$, con $\text{id}(P_k) = k$.

Cada proceso que detecta la falla de un coordinador inicia una elección:

- P_k envía mensaje `ELECTION` a todos los procesos con identificador mayor: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
- Si nadie responde, P_k es el coordinador.
- Si uno responde, P_k deja de intentar ser el coordinador, y el que responde continúa.

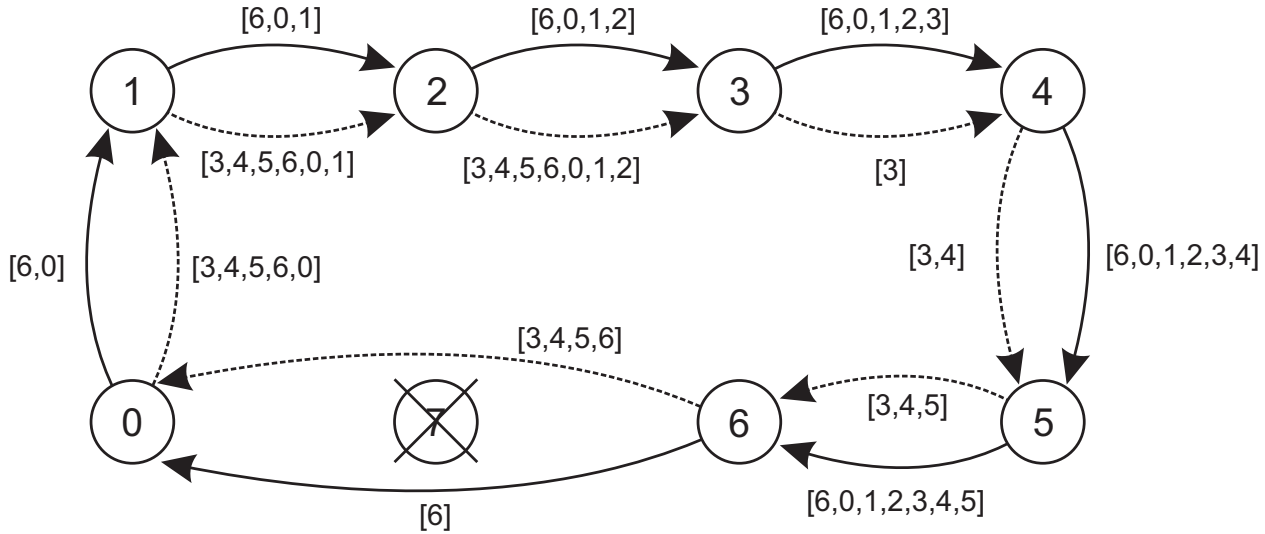


Algoritmos de elección: elección en anillo

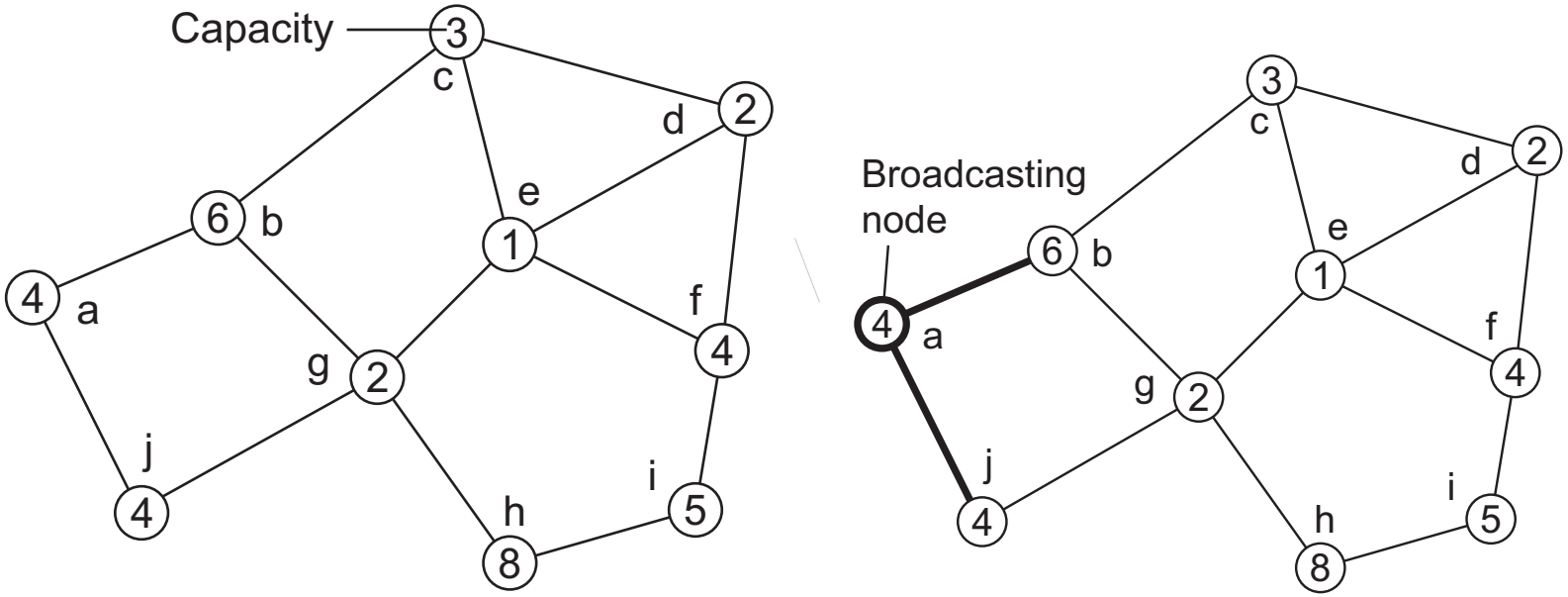
Elección por anillo

Procesos organizados en anillo lógico. Se debe elegir al de mayor ID

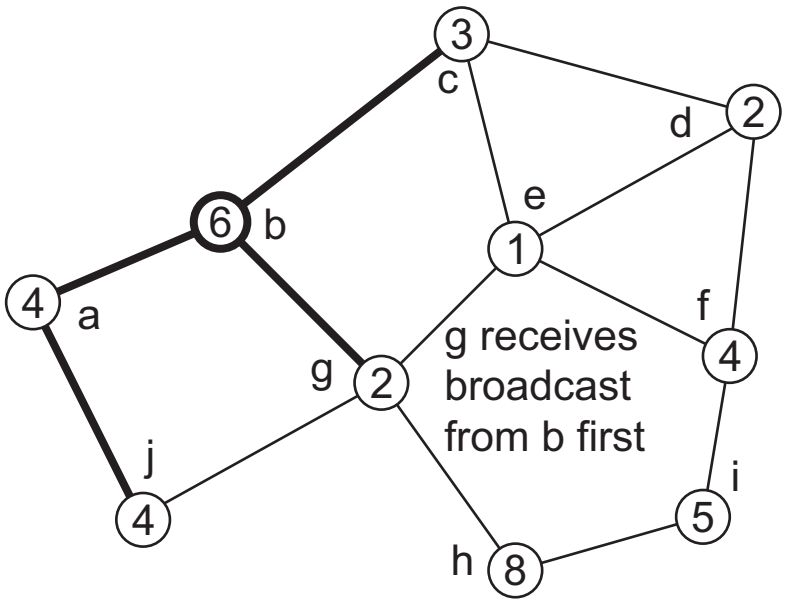
- Cualquiera puede iniciar la elección enviando un mensaje al sucesor. Si el sucesor está caído, se envía sucesor del sucesor.
- Cuando se reenvía un mensaje, el *sender* se agrega a una lista. Cuando la lista regresa al que inició el mensaje, se sabe que todos los miembros "vivos" están en la lista.
- El iniciador reenvía la lista a través del anillo con todos los miembros "vivos", y cada uno puede elegir el de mayor ID.

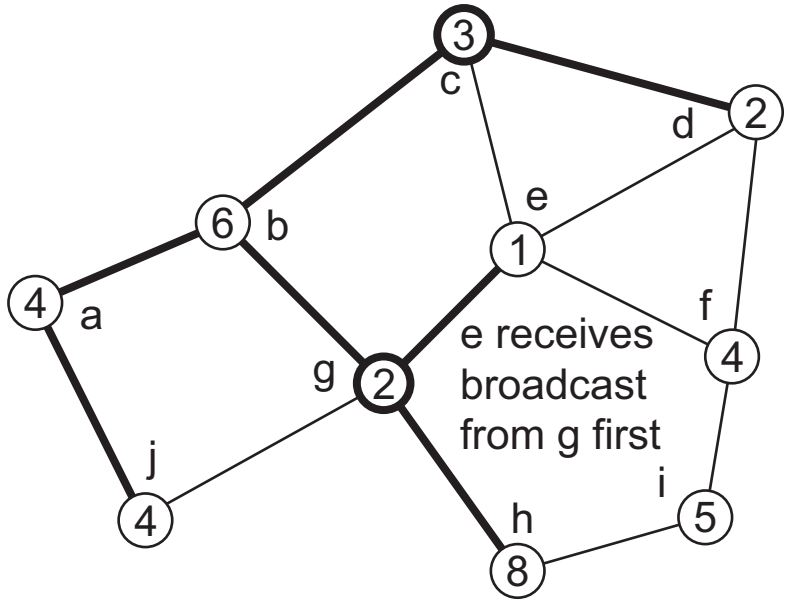


Algoritmos de elección: elección en *wireless*



Algoritmos de elección: elección en *wireless*





Algoritmos de elección: elección en *wireless*