Sistemas Distribuidos
Coordinación y acuerdo

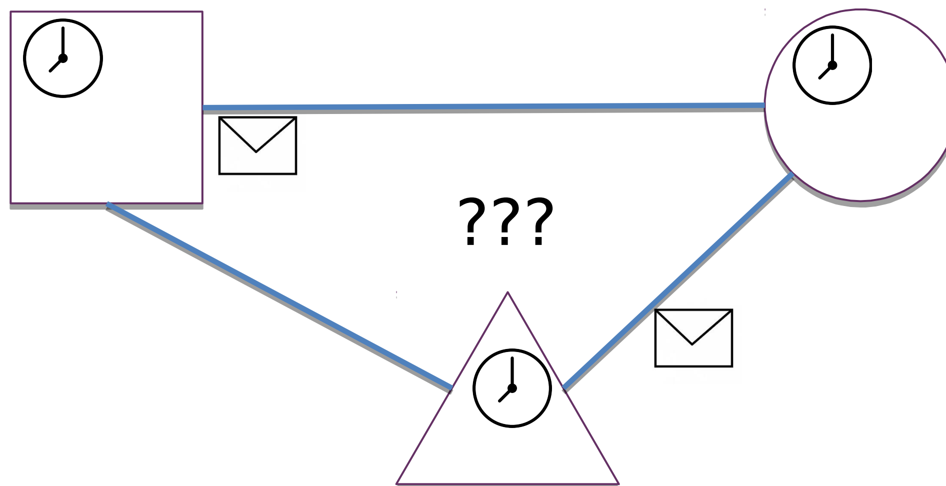
Rodrigo Santamaría

+ Coordinación y acuerdo

- Introducción
- Exclusión mutua distribuida
- Elección distribuida
- Multidifusión
- Consenso distribuido

Introducción

- Hasta ahora hemos visto cómo
  - **Comunicar** procesos en distintos nodos: **middleware**
  - **Sincronizar** procesos en distintos nodos: **tiempos lógicos**
- Con estas herramientas podemos plantearnos cómo
  - **Coordinar** procesos en distintos nodos



Introducción

Asunciones

- **Sincronismo**: asincronía
- **Fallos de proceso:** los procesos no fallan
- **Fallos de comunicación**: canales fiables
  - Los mensajes se terminan recibiendo
  - El fallo de un proceso no evita que el resto puedan comunicarse
  - En un sistema síncrono, además con un límite de tiempo

Se harán siempre estas asunciones para cada solución discutida salvo que se indique lo contrario

+ Introducción

Objetivos

- Dado un conjunto de procesos en un SD, vamos a necesitar
  - **Coordinar** sus acciones
  - Llegar a un **acuerdo** en uno o más valores
- Formas de coordinación y acuerdo:
  - **Acceso a recursos**: exclusión mutua distribuida
  - **Selección de valores**: algoritmos de elección
  - **Comunicación distribuida**: algoritmos de multidifusión
  - **Toma de decisiones**: algoritmos de consenso

+ Coordinación y acuerdo

- Introducción
- Exclusión mutua distribuida
- Elección distribuida
- Multidifusión
- Consenso distribuido

Exclusión mutua distribuida

- **Sección crítica (SC)**: porción de código que permite el acceso a un recurso compartido por varios procesos
- **Exclusión mutua**: el acceso a la sección crítica se regula por medio de *variables compartidas*, por ejemplo semáforos
- **Exclusión mutua distribuida**: el acceso a la sección crítica se basa en *paso de mensajes*

Exclusión mutua distribuida

Algoritmo básico

1. entrarSC() // bloqueo del proceso si SC ocupada
2. accesoRecursos() // uso de recursos compartidos
3. salirSC() // liberación de procesos bloqueados

Exclusión mutua distribuida

Requisitos

- **Seguridad**
  - A lo sumo un proceso puede estar ejecutándose a la vez en la SC
- **Pervivencia**
  - Las peticiones de entrada/salida de la SC al final son concedidas
    - Sin interbloqueos ni inanición
- **Ordenación**
  - Si una petición para entrar en la SC ocurrió "antes que" otra, entonces la entrada en la SC se garantiza en ese orden

RECUERDA: un evento sucede "antes que" otro si
1) ocurren en ese orden en el mismo proceso, o
2) es el envío correspondiente a una recepción

Tema 5 (diap 28 y sigs.)

Exclusión mutua distribuida

Criterios de evaluación

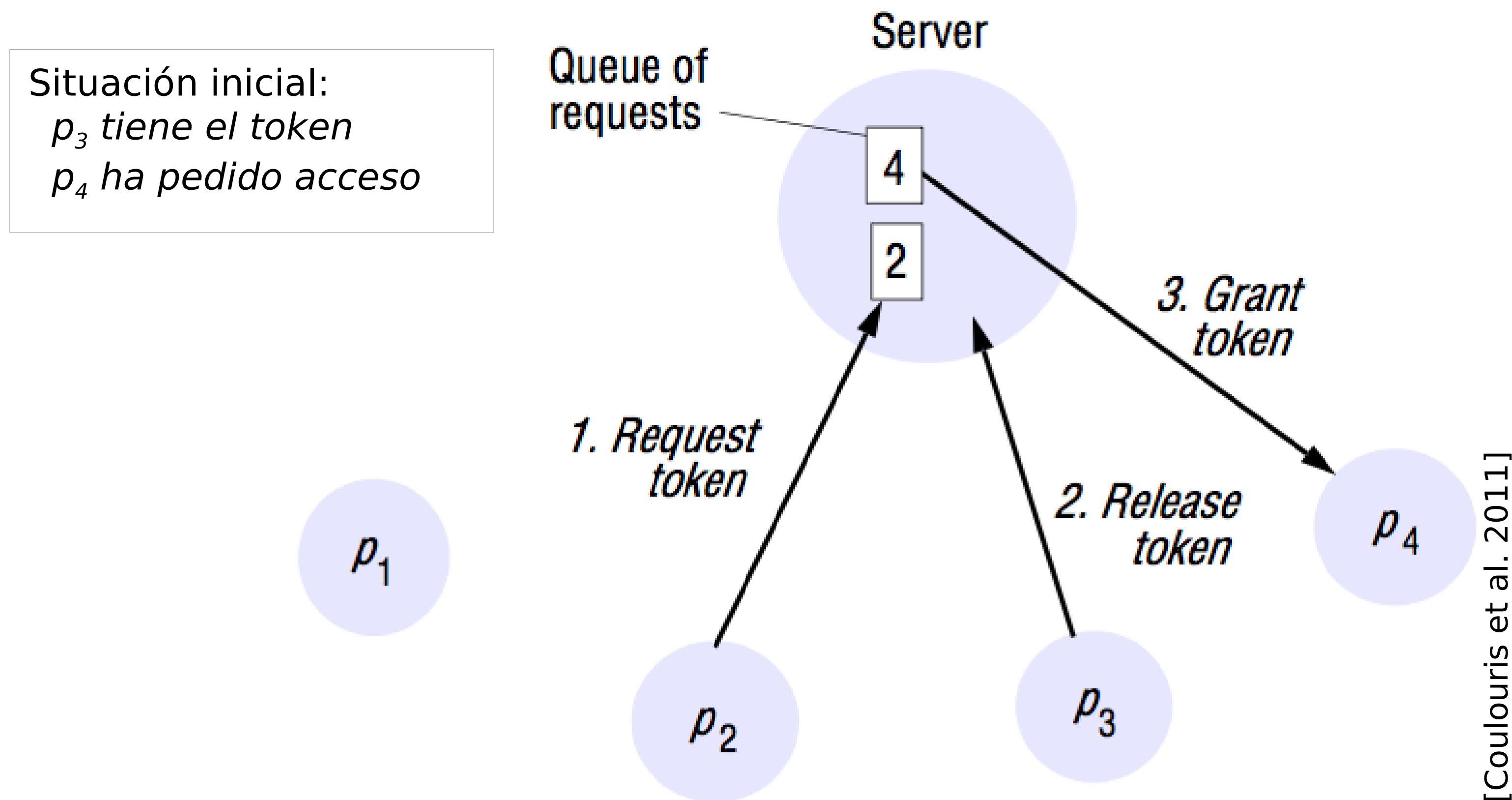
- **Retraso de entrada**
  - Cuánto tarda un proceso en entrar en la SC desde que lo solicita
- **Retraso en el “relevo”**
  - Tiempo que pasa entre que un proceso sale de la SC y el siguiente que esté esperando entra
- **Ancho de banda** consumido
  - Proporcional al número de mensajes enviados en cada operación de entrada y salida de la SC
  - En sistemas asíncronos, va a ser la medida del retraso

Exclusión mutua distribuida

Algoritmo con servidor central

- Un servidor concede los permisos para entrar en la SC
  - Mediante el uso de un testigo (*token*) que se envía por mensajes

Situación inicial:
p_3 tiene el token
p_4 ha pedido acceso



[Coulouris et al. 2011]

Exclusión mutua distribuida

Algoritmo con servidor central

- Cumplimiento de requisitos
  - **Seguridad**: Sí
    - El servidor se encarga de ello a través de un *token* único
  - **Pervivencia**: Sí
    - Todas las peticiones se registran en la cola
  - **Ordenación**: No
    - No considera los tiempos locales en que se enviaron los mensajes

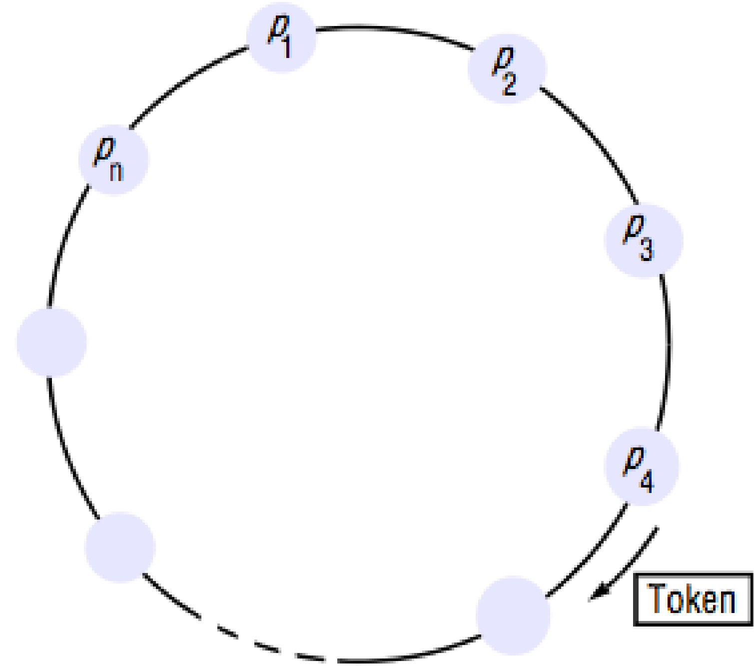
- Rendimiento
  - Entrada: 2 mensajes (petición y concesión)
  - Relevo: 2 mensajes (liberación y nueva concesión)
  - El servidor actúa de cuello de botella
    - Realiza los mensajes de liberación y concesión

¿Encuentras alguna otra desventaja a este método?
¿Y si no consideramos las asunciones previas?

Exclusión mutua distribuida

Algoritmo basado en anillo (token-ring)

- Paso de testigos sin servidor central
- Requisitos
  - Seguridad, pervivencia: Sí
  - Ordenación: No
- Rendimiento
  - Tiempo de entrada: de 0 a N mensajes
  - Tiempo de relevo: de 1 a N-1 mensajes
  - Ancho de banda: consumo continuo salvo cuando un proceso está en SC



Exclusión mutua distribuida

Algoritmo de Ricart y Agrawala [1981]

- Descentralizado: evita cuellos de botella
- Uso de multidifusión y relojes lógicos
  - Asegura seguridad, pervivencia *y ordenación*
- Funcionamiento
  - Sean N procesos $p_1, \ldots, p_N$ con identificadores distintos
  - Todos los procesos pueden comunicarse entre sí
  - Cada proceso mantiene **un reloj lógico** de Lamport C_i
  - Mensaje de **solicitud de entrada:** $\boldsymbol{<C_i(t), p_i>}$
  - Cada proceso mantiene una **variable de estado**
    - LIBERADA → fuera de la SC
    - BUSCADA → fuera de la SC e intentando entrar en la SC
    - TOMADA → dentro de la SC

Exclusión mutua distribuida

Algoritmo de Ricart y Agrawala: pseudocódigo

En la inicialización

estado = LIBERADA;
cola vacía;

Para entrar en la SC

estado = BUSCADA;
T = marca temporal de la petición
Multidifusión de la petición de entrada en SC
Espera hasta que (nº de respuestas = (N-1));
estado = TOMADA;

Al recibir una petición **<T_j, p_j>** ***en*** p_i

si (estado = TOMADA o
(estado = BUSCADA y (T, p_i) < (T_j, p_j)*))
pon en la cola la petición de p_j
si no
responde inmediatamente a pj

Al salir de la SC

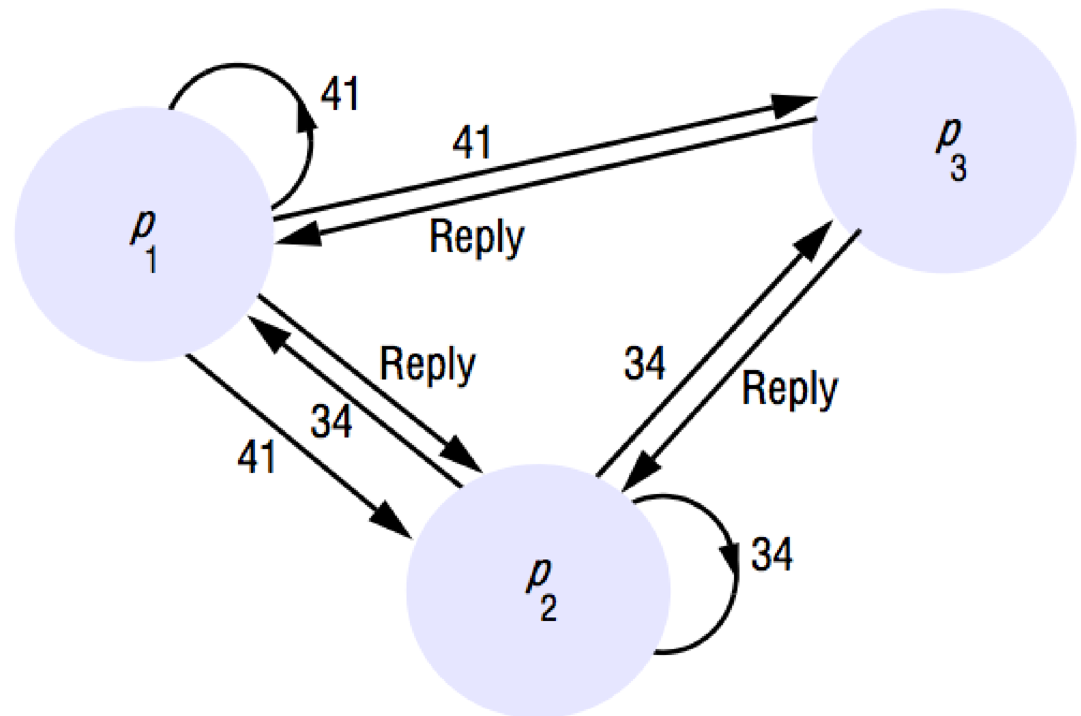
estado = LIBERADA;
responder a todas las peticiones en la cola;
cola vacía;

*(T, p_i) < (T_j, p_j) implica que T < T_j o que T = T_j y $p_i < p_j$
Para ello, los identificadores de proceso deben ser comparables (p. ej. $p_1<p_2$)

Exclusión mutua distribuida

Algoritmo de Ricart y Agrawala: ejemplo

- p_1 multidifunde una petición para entrar en la SC, en su tiempo C_1=41
- p_2, concurrentemente, multidifunde su petición en su tiempo C_2=34
- p_3 no está interesado en la SC, así que responde inmediatamente a ambos
- p_1, observando que su tiempo lógico es mayor que el de p_2, le responde
  - p_2 no responde a p_1, pero lo mantiene en cola para responderle cuando salga de la SC



Exclusión mutua distribuida

Algoritmo de Ricart y Agrawala: análisis

- Cumple seguridad, pervivencia
  - y *ordenación* (uso de relojes lógicos)
- Rendimiento
  - Retraso de entrada
    - (N-1) mensajes de petición de entrada
    - (N-1) mensajes de concesión de entrada
  - Retraso de relevo
    - 1 mensaje (el del proceso actualmente en la SC)
  - Ancho de banda
    - 2N-2

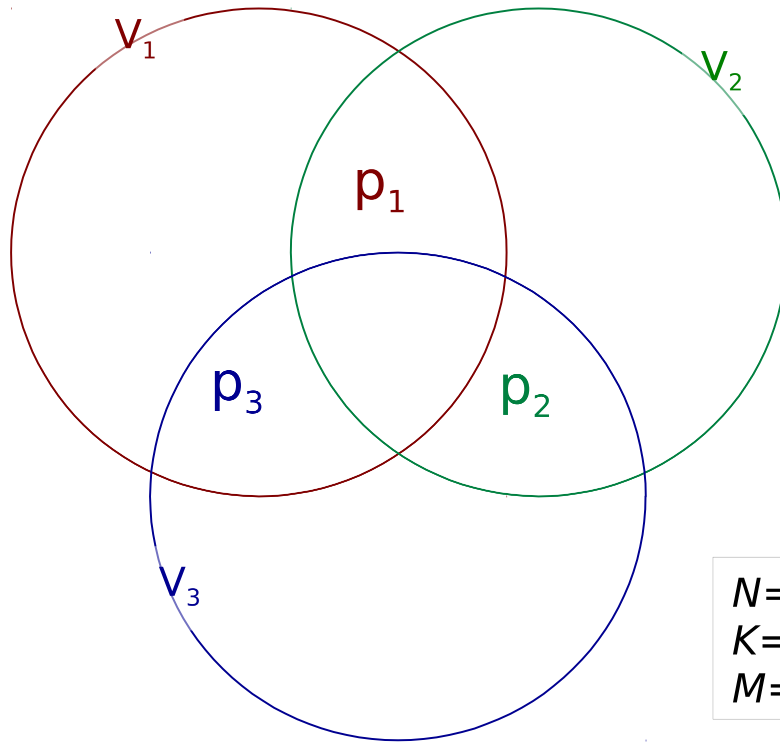
Exclusión mutua distribuida

Algoritmo de Maekawa [1985]

- Fundamento
  - No necesitamos que todos los procesos nos permitan el acceso
  - Basta con obtener el **permiso de un subconjunto de *K* procesos**, siempre que los subconjuntos usados por cualquier par de procesos se solapen con unos determinados criterios
- Elección de los subconjuntos de voto V_i para cada proceso p_i:
  - Para todo i, j = 1, 2, …, N
    - p_i debe pertenecer a V_i
    - $V_i \cap V_j \neq \emptyset$
    - $|V_i| = K$
    - p_j está contenido en M de los subconjuntos de voto
- Maekawa demuestra que la solución óptima es $K = M \sim \sqrt{N}$

Exclusión mutua distribuida

Algoritmo de Maekawa: ejemplo



N=3	*número de procesos*
K=2	*procesos por conjunto de voto*
M=2	*conjuntos de voto por proceso*

¿Cómo sería el esquema con N=4?

Exclusión mutua distribuida

Algoritmo de Maekawa: pseudocódigo

En la inicialización
estado = LIBERADA;
votado = FALSO;

Cuando p_i quiere entrar en la SC
estado = BUSCADA;
Multidifusión de la petición de entrada en SC a los procesos en V_i - {p_i};
Espera hasta que (nº de respuestas = (K-1));
estado = TOMADA;

Al salir de la SC
estado = LIBERADA;
Multidifunde *liberar* a los procesos en V_i - {p_i};

Cuando p_i recibe una petición de p_j (i≠j)
si (estado = TOMADA o votado = CIERTO o
(estado = BUSCADA y $(T, p_i) < (T_j, p_j)$*))
pon en la cola la petición, por parte de p_j
si no
responde inmediatamente a p_j
votado = CIERTO;

Al recibir en p_i *liberar* de parte de p_j (i≠j)
si (la cola de peticiones no está vacía)
quita la cabeza de la cola (p. ej. p_k);
envía respuesta a p_k;
votado = CIERTO;
si no
votado = FALSO;

En morado los cambios respecto a Ricart y Agrawala

Exclusión mutua distribuida

Algoritmo de Maekawa: análisis

- Requisitos
  - Cumple con la **seguridad**
  - No cumple con la **pervivencia**, pues puede producir interbloqueos
    - Solución mejorada por Sanders [1987] mediante el uso de una ordenación “antes que” → cumple con pervivencia y ordenación

- Rendimiento
    - Retraso de entrada:
      - Petición de entrada en SC: $\sqrt{N}$
      - Concesión de entrada en SC: $\sqrt{N}$
    - Retraso en el relevo: $\sqrt{N}$
    - Ancho de banda: $3\sqrt{N}$
      - Mejora Ricart y Agrawala si N>4

Esta solución con conjuntos de voto solapados su usará posteriormente para direccionamiento P2P o sistemas de replicación

Exclusión mutua distribuida

Comparativa

Algoritmo	Entrada	Relevo	Ancho de Banda	Ordenación
Servidor central	2	2	3	No
Anillo	N	N-1	Consumo continuo	No
Ricart & Agrawala	2N-2	1	2N-2	Sí
Maekawa	2√N	√N	3√N	Sí*

*Con la mejora de Sanders, 1987

+ Coordinación y acuerdo

- Introducción
- Exclusión mutua distribuida
- Elección distribuida
- Multidifusión
- Consenso distribuido

+ Elección distribuida

Objetivo

Elegir un proceso único para que tome un determinado rol o para decidir una determinada acción

- Aplicaciones
  - Elegir un nuevo servidor si se cae el actual
  - Elegir un nuevo proceso para entrar en una sección crítica
  - Elegir el proceso menos activo (balanceo de carga)
  - Elegir el proceso con la copia más reciente (réplicas)

Elección distribuida

Consideraciones

- Un proceso *convoca* elecciones cuando lleva a cabo una acción que inicia el algoritmo de elección
- Puede haber N elecciones *concurrentes*
- Un proceso siempre tiene uno de estos dos roles:
  - *Participante*: comprometido en una ejecución del algoritmo
  - *No participante*: no comprometido en ninguna ejecución
- El proceso elegido debe ser único, incluso en elecciones concurrentes

Elección distribuida

Consideraciones

- Identificadores
  - Todos los procesos tienen un identificador
    - Único para el conjunto
    - Totalmente ordenados
  - El proceso elegido es aquél de mayor identificador
- Variable *elegido*
  - Cada proceso p_i mantiene una variable e_i que contiene el identificador del proceso elegido
  - Cuando el proceso se convierta en participante, fija la variable al valor especial $\perp$, indicando que no hay consenso todavía

+ Elección distribuida

Requisitos y rendimiento

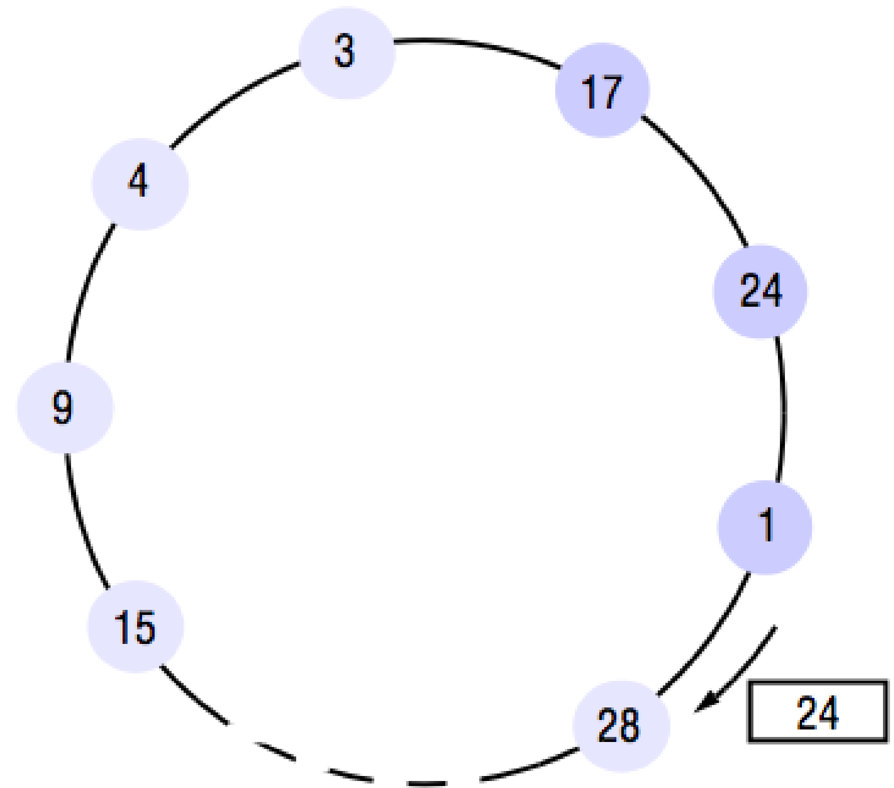
- Requisitos
  - **Seguridad**
    - Un proceso participante p_i tiene $e_i = \perp$ o $e_i = P$, donde **el proceso elegido *P* es aquél con identificador mayor** que no se ha caído al final de la ejecución del algoritmo de elección
  - **Pervivencia**
    - Todos los procesos p_i participan y, al final, fijan $e_i \neq \perp$; o bien se han caído.

- Rendimiento
  - **Ancho de banda**: proporcional al número de mensajes enviados
  - **Tiempo de ronda** (*turnaround*): tiempo pasado desde que se convocan elecciones hasta que se elige un proceso

Elección distribuida

Algoritmo en anillo

- Chang y Roberts [1979]
- Los procesos se consideran dispuestos en un anillo lógico
- Un proceso convoca elecciones y pasa su identificador al siguiente en el anillo
  - Si el identificador del proceso es mayor que el recibido, cambia el identificador
  - Cuando llega el mensaje al proceso convocante, finaliza la elección, transmitiendo en el mismo orden el identificador elegido



Por qué vemos soluciones en anillo? Aunque superada por Ethernet, durante un tiempo las redes en anillo eran uno de los competidores para LANs (https://www.networkworld.com/article/2287366/lan-wan/ethernet-vs--token-ring.html) No obstante, las redes lógicas siguen siendo una opción.

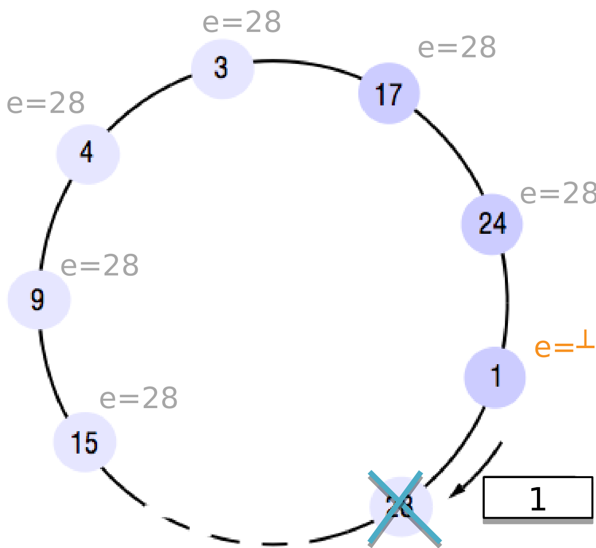
+
Elección distribuida

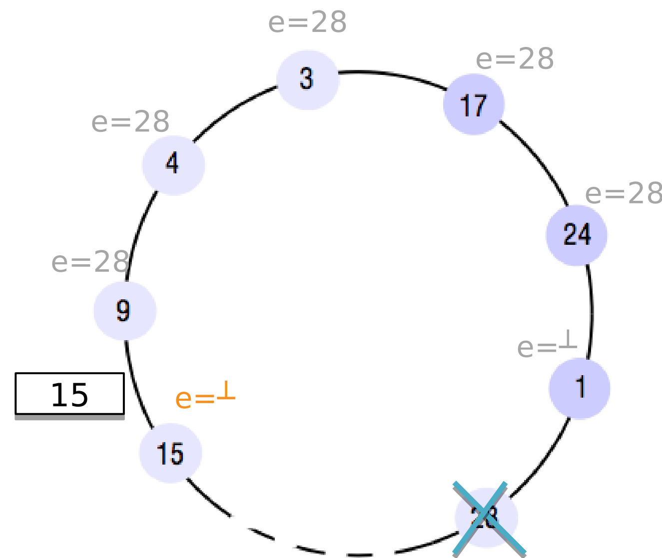
Algoritmo en anillo

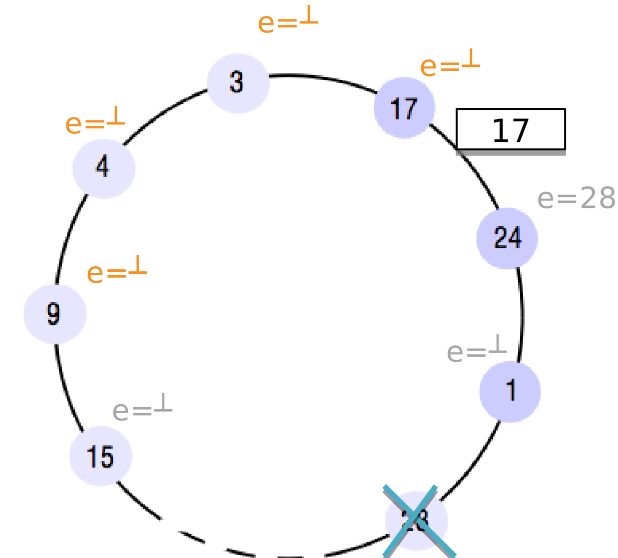
- Cada proceso se etiqueta como *no participante*
- Cualquier proceso convoca elecciones
  - Se marca a sí mismo como *participante*, y pone e=⊥
  - Pone su identificador en un mensaje *elección* y lo envía al vecino
- Un proceso recibe un mensaje *elección* con identificador
  - Mayor al suyo: se etiqueta como *participante*, pone e=⊥ y reenvía el mensaje
  - Menor al suyo: si es *no participante*, pone su identificador en el mensaje y lo reenvía, etiquetándose como *participante* y poniendo e=⊥
  - Igual al suyo: se convierte en *coordinador*, se marca como *no participante* y envía el mensaje *elegido* al vecino
- Un proceso recibe un mensaje *elegido*
  - Se marca como *no participante* y fija e_i al valor del identificador del mensaje
  - Envía un mensaje *elegido* al siguiente vecino, a no ser que sea el coordinador

Elección distribuida

Algoritmo en anillo: ejemplo







El proceso 1 detecta que el proceso 28 (nodo líder) ha caído

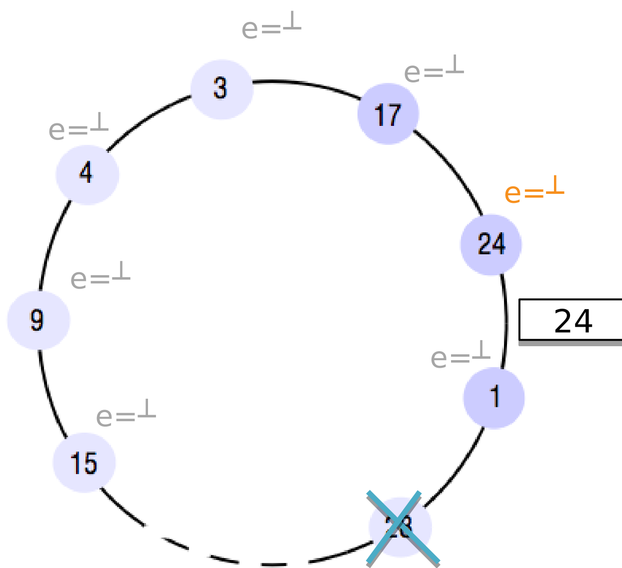
Envía un mensaje de elección (con su id) al siguiente nodo en la red

Cuando el mensaje llega al proceso 15, como 15>1, cambia el identificador del mensaje y lo reenvía

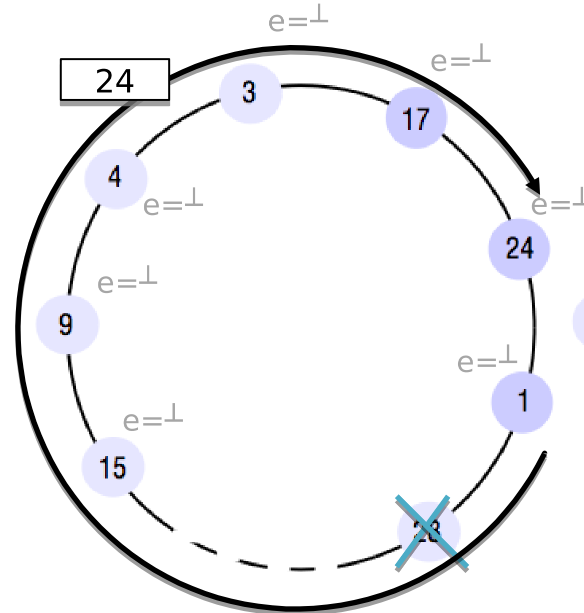
9,4,3 reenvían el mensaje 17 lo modifica y reenvía

Elección distribuida

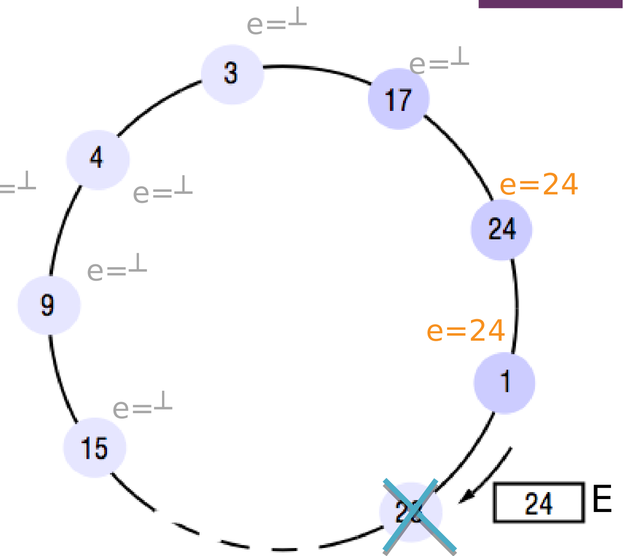
Algoritmo en anillo: ejemplo



24 igualmente cambia el mensaje, pero aún no se erige como proceso elegido



El mensaje tiene que dar todavía una vuelta completa (en el peor caso) en la que no hay cambios en los estados



Cuando el mensaje llega de nuevo a 24, este detecta que nadie lo ha modificado así que debe ser el nuevo coordinador.

Envía un mensaje *elegido*(24) que da la vuelta a todo el anillo de nuevo estableciendo el coordinador

Elección distribuida

Algoritmo en anillo: análisis

- Requisitos:
  - **Seguridad**: al cabo de la primera vuelta se obtiene el proceso activo con identificador más alto.
  - **Pervivencia**: la última vuelta asegura que todos los procesos activos conocen el resultado de la elección.

- Rendimiento
  - Peor caso: el nuevo elegido es el vecino antihorario del convocante
    - 3N-1 mensajes
      - N-1 mensajes *elección* hasta alcanzar el vecino antihorario
      - Otros N mensajes *elección* con id del vecino antihorario
      - N mensajes *elegido*

Elección distribuida

Algoritmo abusón (bully)

- García-Molina [1982]
- Consideraciones adicionales
  - Permite la caída de procesos durante la elección
    - Utiliza timeouts para detectar fallos de procesos
  - Cada proceso conoce qué procesos tienen identificadores mayores y puede comunicarse con ellos
- Funcionamiento resumido
  1. El convocante envía mensajes *elección* a los procesos de id mayor
  2. Si ninguno le responde, multidifunde que es el nuevo *coordinador*
  3. Si alguno le responde, el convocante inicial queda en espera, y los procesos que responden inician un nuevo proceso de elección como convocantes (vuelta al paso 1)

Elección distribuida

Algoritmo abusón: mensajes y timeouts

- Tipos de mensaje
  - *Elección*: anuncia un proceso de elección
  - *Respuesta*: respuesta a un mensaje de elección
  - *Coordinador*: anuncia la identidad del proceso elegido
- Inicio del algoritmo
  - La elección comienza cuando un proceso se da cuenta (debido a los timeouts) de que el coordinador ha fallado
    - Varios procesos pueden descubrirlo de forma concurrente
    - Timeout = $T = 2 \cdot T_{\text{transmisión de un mensaje}} + T_{\text{procesado de un mensaje}}$

Elección distribuida

Algoritmo abusón

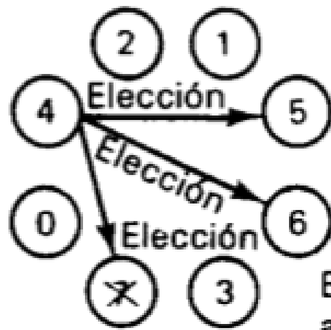
1. Si un proceso sabe que tiene el id (no fallido) más alto
   - Se elige a sí mismo coordinador enviando el mensaje *coordinador* a todos los de identificador más bajo (proceso abusón)

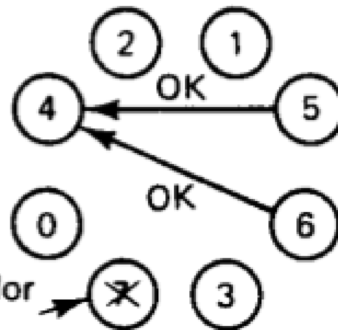
2. Si no tiene el id (no fallido) más alto
   - Manda un mensaje *elección* a todos los de id más alto y espera un mensaje *respuesta*
     - Si tras un tiempo T no recibe ningún mensaje *respuesta*, ir al paso 1
     - Si recibe un mensaje *respuesta*, espera un mensaje *coordinador*
       - Si recibe un mensaje *coordinador*, fija su variable e_i al id que está contenido en dicho mensaje
       - Si no recibe ningún mensaje, comienza otra nueva elección

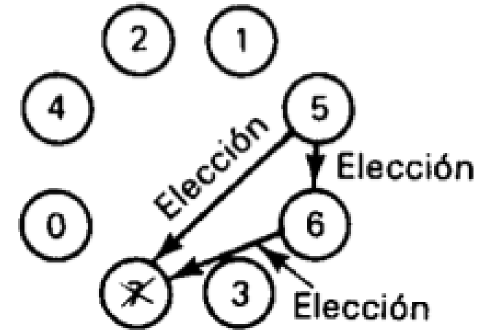
- Si un proceso se recupera o se lanza un proceso sustituto con el mismo id, éste comienza una nueva elección, aunque el coordinador actual esté funcionando

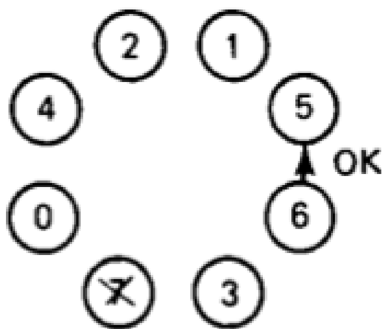
Elección distribuida

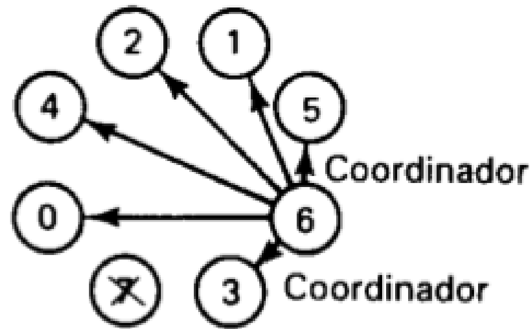
Algoritmo abusón: ejemplo





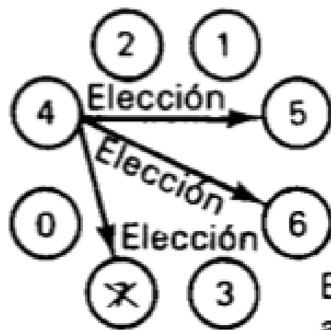


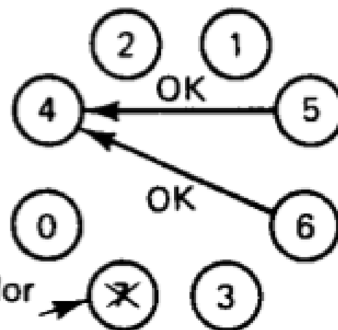


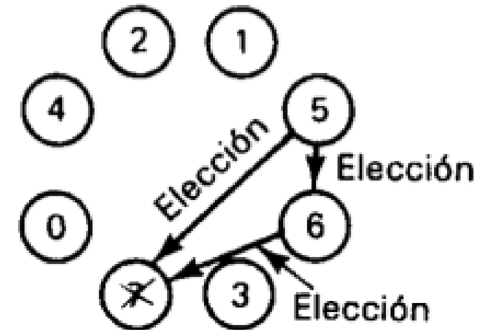


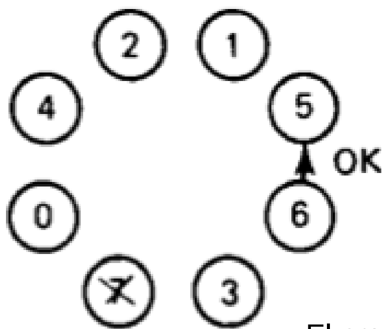
Elección distribuida

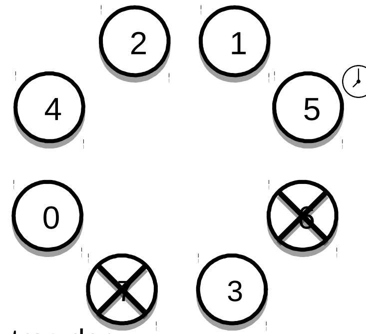
Algoritmo abusón: caída intermedia



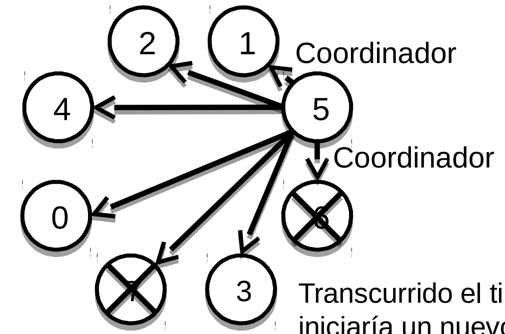








El proceso 6 falla tras dar respuesta a 5
5 permanece a la espera…



Transcurrido el timeout, 5 iniciaría un nuevo proceso (f), no recibiría ningún ok (g) y se multidifundiría coordinador (h, dibujado)

Elección distribuida

Algoritmo abusón: análisis

- Requisitos
  - **Pervivencia**: todos fijan e_i mediante la multidifusión final
  - Algunos problemas con la **Seguridad** (e_i indica el id mayor no caído)
    - Problema 1: cae *p* (coordinador), pero se recupera al mismo tiempo que otro proceso *q* decide ser el coordinador → algún proceso puede recibir dos mensajes *coordinador* con distintos identificadores*
    - Problema 2: los valores de timeout son imprecisos (el sistema no es síncrono)
- Rendimiento
  - Mejor caso: el proceso con el segundo identificador más alto detecta el fallo del coordinador → N-2 mensajes *coordinador*
  - Peor caso: el proceso con identificador más bajo detecta el fallo del coordinador → $O(N^2)$

** Pregunta a resolver cuando veamos todo el tema:*
¿Cómo podríamos solucionar este problema, al menos parcialmente?

Elección distribuida

Comparativa

Algoritmo	Anillo	Abusón
Mensajes	3N-1	N^2
Caída intermedia	No	Sí
Seguridad	Sí	No*
Pervivencia	Sí	Sí
Conocimiento de la estructura	Proceso siguiente	Todos los procesos

*En casos de caídas y recuperaciones intermedias, o fallos de timeout

+ Coordinación y acuerdo

Introducción

Exclusión mutua distribuida

Elección distribuida

Multidifusión

Consenso distribuido

+ Multidifusión

- **Objetivo**: entrega de mensajes a un grupo de procesos distribuidos, garantizando
  - Fiabilidad: todos los procesos del grupo reciben el mensaje
  - Orden: el orden de entrega del mensaje es acordado y se respeta
- Fundamento
  - Un proceso realiza sólo una operación *multicast* para enviar un mensaje a todos los miembros del grupo
  - Un proceso obtiene mensajes mediante una orden *entrega*, que no implica una recepción instantánea del mensaje

Multidifusión

Modelo del sistema

- Consideraciones
  - Los procesos se comunican a través de canales fiables uno-a-uno
  - Los procesos sólo pueden fallar por caída
  - Los procesos son miembros de grupos, que son los destinos de las operaciones *multicast*
  - Un proceso puede pertenecer a varios grupos
- Operaciones
  - multicast(g, m) → recibir(m)
    - Donde *g* es el grupo y *m* es un mensaje que porta
      - emisor(m): identificador único del proceso remitente
      - grupo(m): identificador único del grupo destinatario

Multidifusión

Multidifusión básica

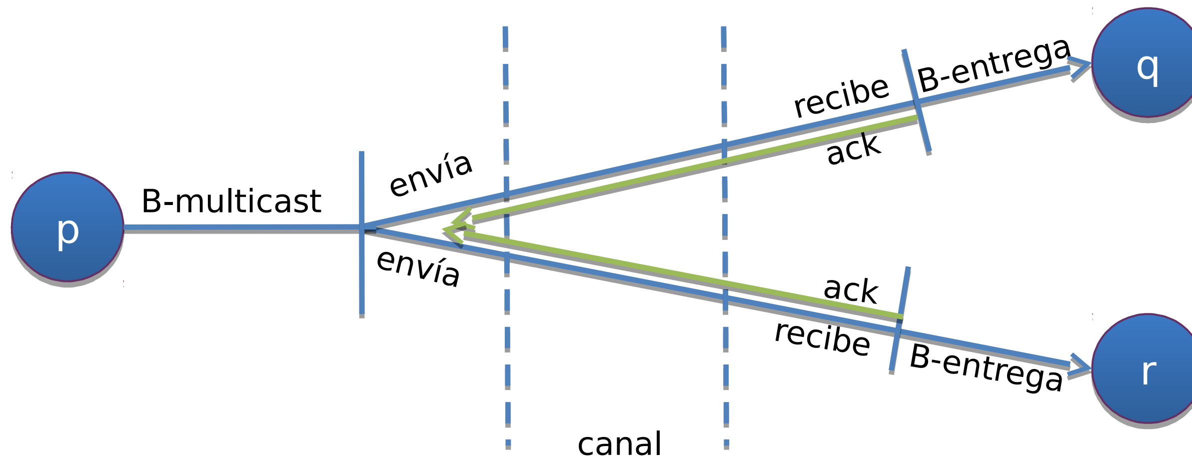
- **Operaciones**: B-multicast y B-entrega
  - “Envuelven” a las operaciones de envío y recepción
- Los procesos pueden pertenecer a varios grupos
- Cada mensaje se destina a un grupo en particular
- Implementación usando la operación *envía* fiable uno-a-uno

```
B-multicast(g,m) → p ∈ g, envía(p,m)
Al recibir(m) en p → B-entrega(m) en p
          Y acuse de recibo al remitente (ack)
```

- Implementación real: hilos para envío concurrente
  - Problema: ack-implosion (colapso por exceso de acuses de recibo)
    - Solución: multidifusión IP sobre UDP

Multidifusión

Multidifusión básica



Multidifusión

Multidifusión fiable

- Debe cumplir con las condiciones de:
  - **Integridad**: un proceso entrega un mensaje a lo sumo una vez
    - En comunicación fiable uno-a-uno, era al menos una vez
  - **Validez**: todo mensaje multidifundido es entregado al remitente
    - Garantiza que el remitente sigue vivo tras la multidifusión
  - **Acuerdo**: si un proceso recibe un mensaje, todo el grupo lo recibe
    - Concepto de "*atomicidad*": o todos, o ninguno
    - Esta condición no se cumple en B-multicast, que solo garantiza la comunicación fiable uno a uno.

- Operaciones (Hadzilacos y Toueg, 1994) (Chandra y Toueg, 1996)
  - F-multicast y F-entrega

Multidifusión

Multidifusión fiable sobre B-multicast: teoría

En la inicialización

recibidos = {};

F-multicast (g, m) //del proceso p

B-multicast(g, m) // $p \in g$ se incluye como destino

B-entrega(m) //en el proceso q

si (m ∉ recibidos)

entonces

    recibidos = recibidos ∪ {m};

    si (q ≠ p)

        entonces

        B-multicast(g, m);

        fin si

    F-entrega(m)

fin si

* En teoría, un sistema como este aseguraría:
  - **-Integridad**: `si (m ∉ recibidos)`
  - **-Validez**: el remitente se envía *m* a sí mismo
  - **-Acuerdo**: se multidifunde de nuevo antes de hacer la entrega

* En la práctica no se puede implementar porque requeriría un número exagerado de re-multidifusiones: cada mensaje *m* se manda |*g*| veces a cada proceso

Multidifusión

Multidifusión fiable sobre IP-multicast

- IP-multicast
  - Multidifusión mediante una dirección IP para el grupo
    - Direcciones 224.0.0.0 a 239.255.255.255
  - El emisor envía un datagrama con su IP y el router se encarga de mandar copias a las IPs adheridas a la IP multicast
    - Muy fiable
- Funcionamiento
  - Como IP-multicast es muy fiable, los acuses de recibo (*ack*) no se envían por separado, si no adjuntos (*piggybacked*) en otros mensajes que envíen al grupo
    - “Te envío xxx y de paso te digo que recibí zzz”
  - Sólo se envían acuses de recibo por separado cuando se ha detectado que se ha perdido algún mensaje
    - Acuses de recibo *negativos* (nack)

Multidifusión

Multidifusión fiable sobre IP-multicast

- Variables
  - Cada proceso *p* mantiene un nº de secuencia S_g^p por cada grupo *g* al que pertenece (inicialmente a 0)
  - Cada proceso *p* almacena R_g^q, números de secuencia del último mensaje que ha sido entregado por *p* y que fue enviado desde el proceso *q* del grupo *g* (inicialmente a 0)
- F-multicast de *p* para el grupo *g*
  - Adjunta al mensaje el valor S_g^p
  - Adjunta acuses de recibo sobre el mensaje del tipo $<q, R_g^q>$
  - $S_g^p = S_g^p + 1$

Multidifusión

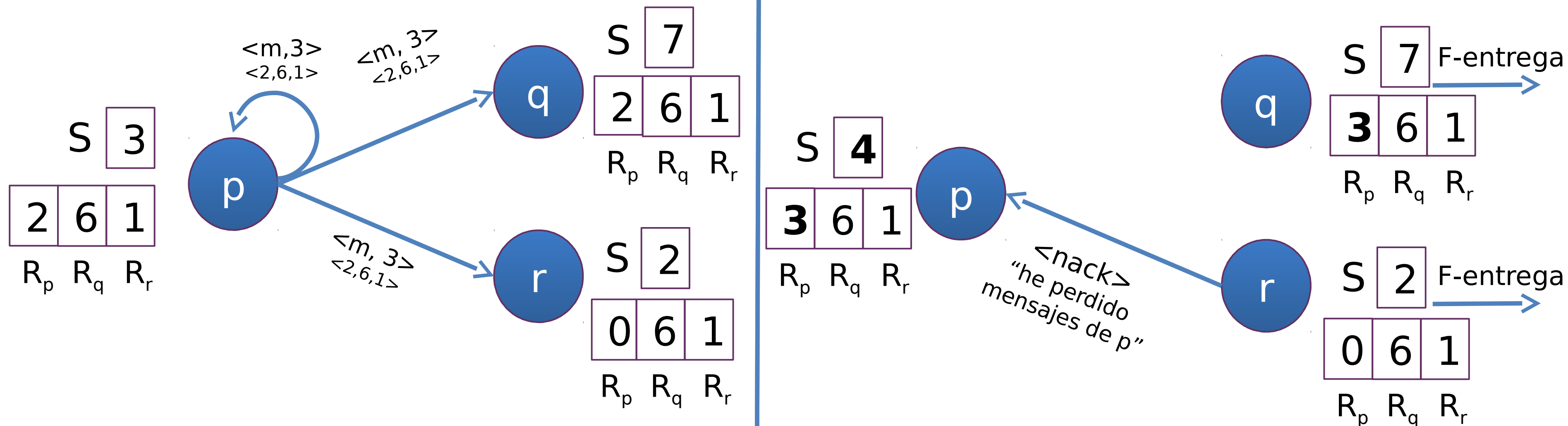
Multidifusión fiable sobre IP-multicast

- Recepción en *q* de un mensaje de *p* para el grupo *g* con nº de secuencia *S* y acuses de recibo adjuntos $<q, R>$
  - Si S = $R_g{}^p + 1$
    - F-entrega el mensaje
    - $R_g{}^p = R_g{}^p + 1$
  - Si $S \leq R_g{}^p$
    - El mensaje se descarta (ya ha sido recibido con anterioridad)
  - Si $S > R_g{}^p+1$ o $R > R_g{}^q$ para cualquier acuse de recibo $<q, R>$ adjunto, significa que se han perdido uno o más mensajes
    - Solicitud mediante acuse de recibo negativo

- A continuación se presentan ejemplos con un solo grupo de 3 procesos {p,q,r}

Multidifusión

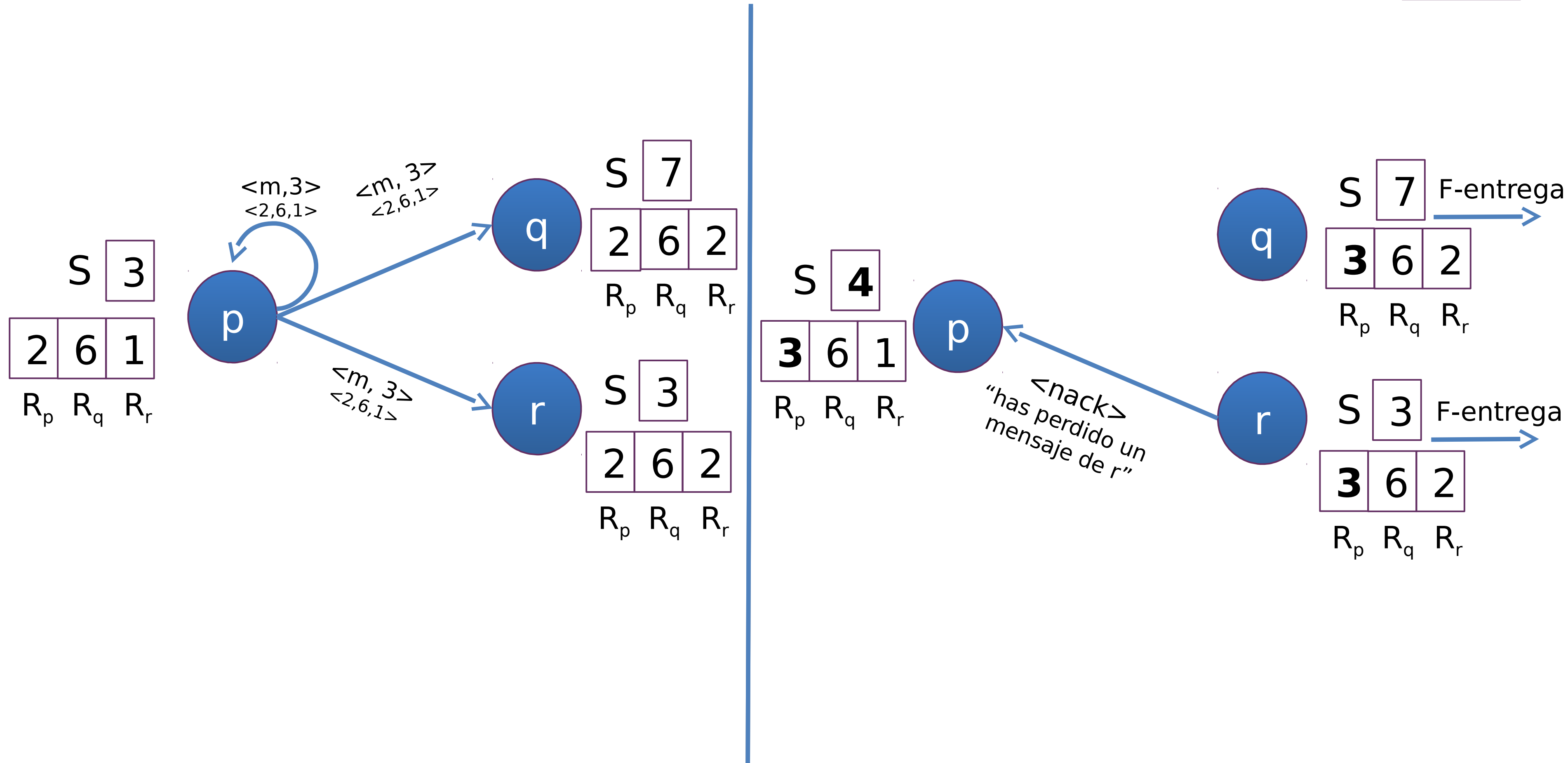
Multidifusión fiable sobre IP-multicast



Nótese que la entrega se realiza para el mensaje actual aunque falten mensajes anteriores. Esta situación implica que los mensajes se entreguen en distinto orden en distintos procesos, siendo la principal diferencia con, p. ej., la multidifusión ordenada que veremos posteriormente

Multidifusión

Multidifusión fiable sobre IP-multicast



Multidifusión

Multidifusión ordenada

- B-multicast y derivadas no garantizan que los mensajes lleguen siempre en el mismo orden a los miembros del grupo
  - Debido a retrasos arbitrarios en los envíos individuales subyacentes
  - La ordenación puede ser un requisito para algunas aplicaciones
- **Ordenación FIFO**: si un proceso realiza multicast(*g*,*m*) y luego multicast(*g*,*m’*), *m* debe entregarse siempre antes de *m’*
  - Respecto al proceso emisor
- **Ordenación causal**: si multicast(*g*,*m*) → multicast(*g*,*m’*) entonces *m* debe entregarse siempre antes que *m’*
  - Respecto a la relación ‘antes que’
- **Ordenación total**: si un proceso entrega *m* antes que *m’*, entonces todo proceso entrega *m* antes que *m’*
  - Respecto a la entrega

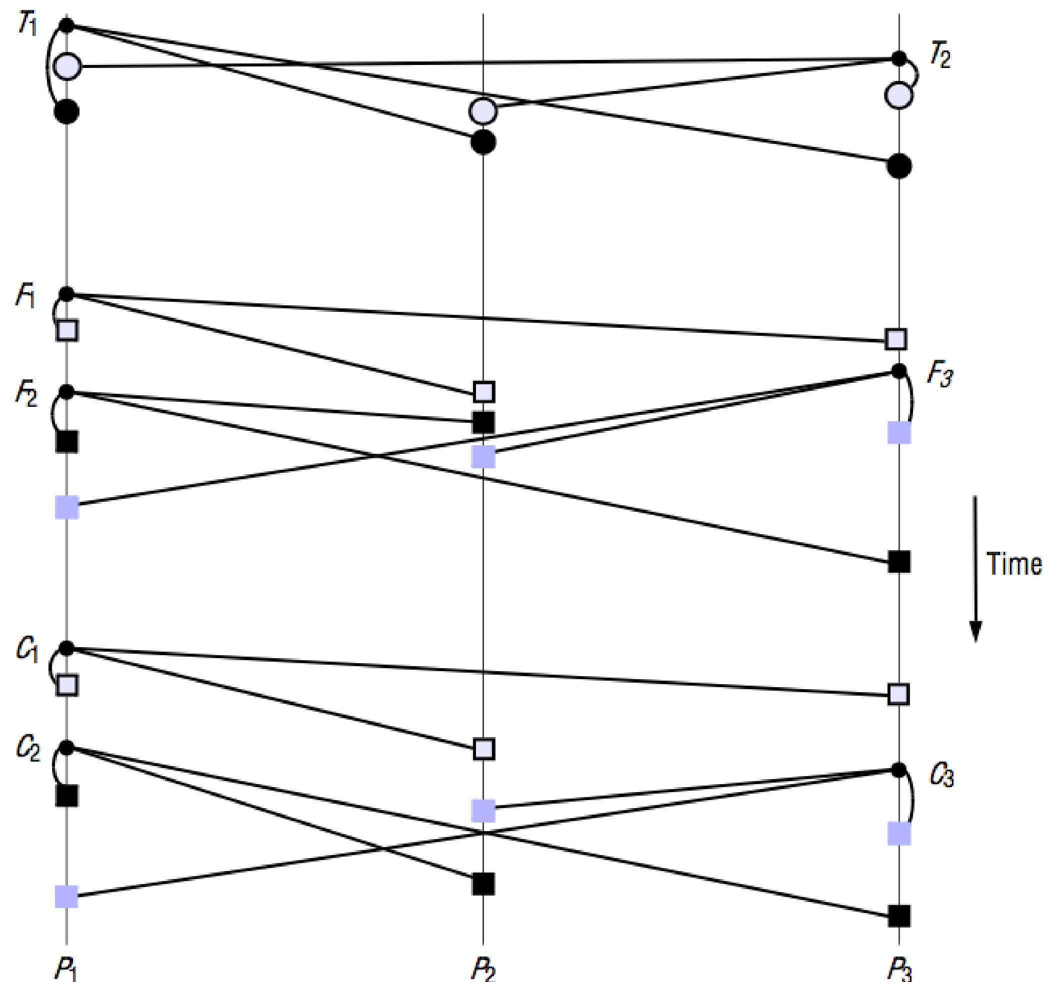
+ Multidifusión

Multidifusión ordenada: ejemplos

Total: como T_2 se entrega antes que T_1 en algún proceso, T_2 se entrega antes en todos los procesos.

FIFO: F_1 se envía antes que el otro mensaje F_2 de P_1, así que se reciben en ese orden en todos los procesos. Indep. de F_3

Causal: Sean $C_1 \rightarrow C_2$ y $C_1 \rightarrow C_3$. C_1 se entrega antes que C_2 y C_3, pero no hay relación 'antes que' ni orden de entrega entre C_2 y C_3



Multidifusión

Multidifusión ordenada

- Observaciones
  - En una ordenación total, no importa el tiempo físico en el que se enviaron los mensajes (en el ejemplo anterior, T_2 ocurre un poco después que T_1)
  - Una ordenación total no respeta orden FIFO o causal
    - Por ejemplo, si F_2 se entregara antes que F_1 en P_2, ese sería el orden de entrega para todos, independientemente de que localmente, en P_1 se difundiera F_1 antes que F_2

Multidifusión

Multidifusión ordenada FIFO

- Variables en el proceso *p*
  - S_g^p contador de mensajes que el proceso *p* ha enviado al grupo *g*
  - R_g^q número de secuencia del último mensaje que *p* ha recibido del proceso *q* enviado al grupo *g*
- OF-multicast de *p* para el grupo *g*
  - Adjuntar S_g^p al mensaje
  - B-multicast
  - $S_g^p = S_g^p + 1$
- Recepción
  - Si $S = R_g^q + 1$ → OF-entrega fijando $R_g^q = S$
  - Si $S > R_g^q + 1$ → **retención** del mensaje hasta que los mensajes intermedios hayan sido entregados y se cumpla $S = R_g^q + 1$

Principal diferencia con F-multicast

Multidifusión

Multidifusión ordenada causal

- Variables en el proceso *p*
  - V_p^g vector con el nº de mensajes recibidos de cada proceso
  - Se adjunta con cada multidifusión
  - Se compara el vector adjunto a un mensaje multidifundido con el vector propio para decidir si los mensajes deben:
    - Entregarse
    - Mantenerse sin entregar a la espera de mensajes no recibidos todavía
      - Bien del remitente o de un tercer proceso

¿Te recuerdan estos vectores a alguna solución vista en temas anteriores?

Multidifusión

Multidifusión ordenada causal

para el proceso p_i (i=1, 2, …, N)

En la inicialización

$V^g_i[j] = 0$ (j=1,2,…,N);

OC-multicast (g, m)

$V^g_i[i] = V^g_i[i] + 1$;

B-multicast(g, $<V^g_i,m>$);

B-entrega($<V^g_j,m>$**)** //viene de p_j

Colocar $<V^g_j,m>$ en la cola de retención

Esperar hasta que $V^g_j[j]=V^g_i[j]+1$ y $V^g_j[k] \leq V^g_i[k]$ $(k \neq j)$

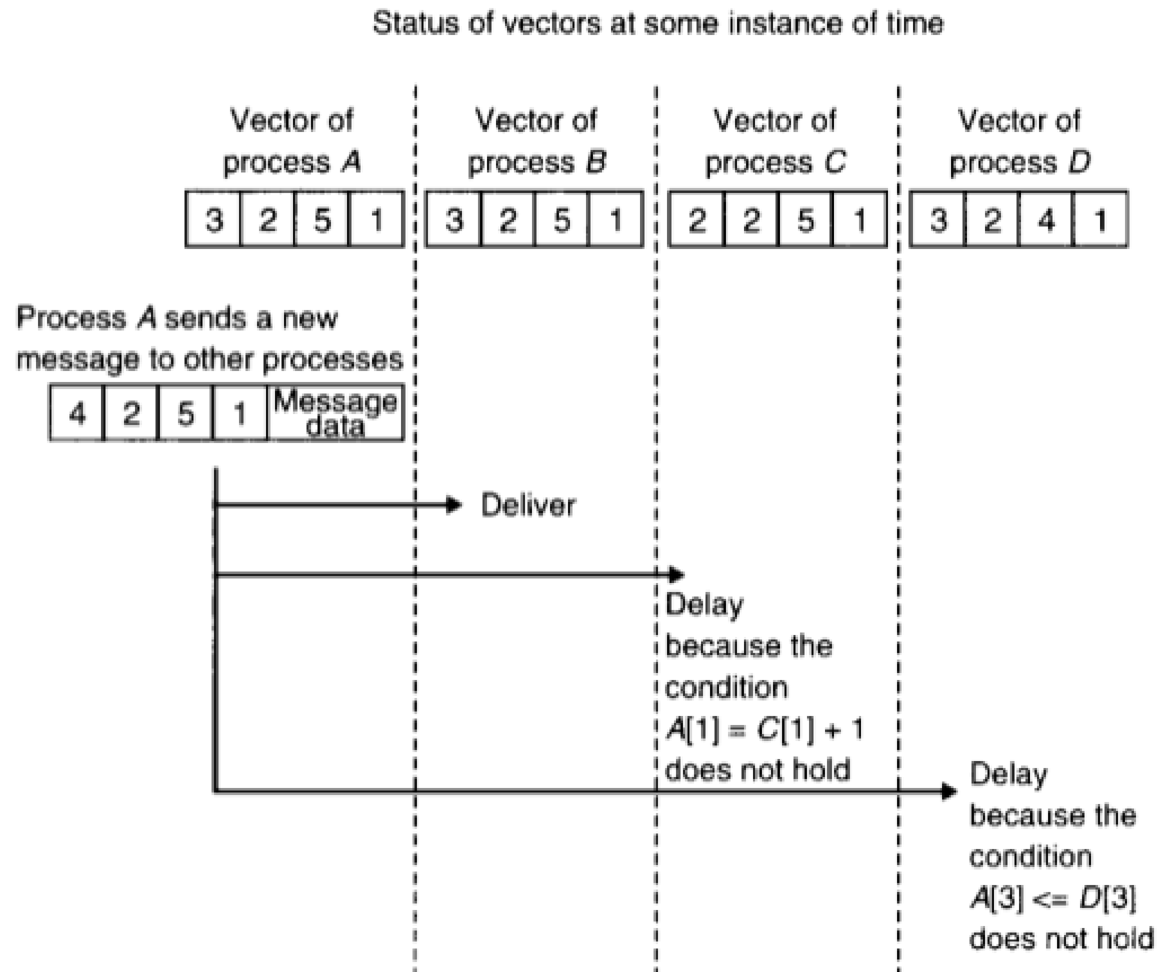
Eliminar m de la cola de retención;

OC-entrega(m);

$V^g_i[j]=V^g_i[j]+1$;

Multidifusión

Multidifusión ordenada causal

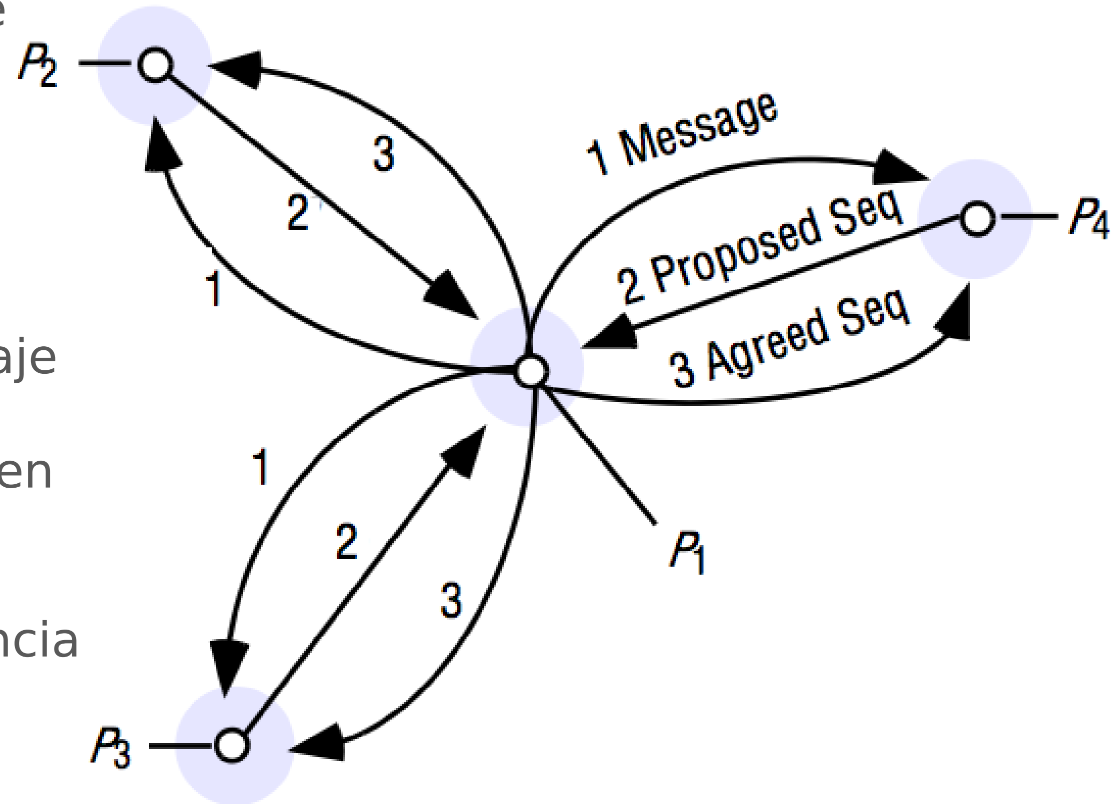


Para recibir un mensaje de A tengo que haber recibido (1) todos los mensajes previos de A y (2) todos los mensajes de otros procesos que haya recibido ya A

Multidifusión

Multidifusión ordenada total por acuerdo (ISIS)

- Desarrollado originalmente para la herramienta ISIS*
  - Birman y Joseph 1987



1. p_1 multidifunde un mensaje
2. Los receptores le proponen números de secuencia
3. p_1 decide el nº de secuencia definitivo a partir de los propuestos

*Aunque la empresa que comercializaba esta herramienta ya no existe, ISIS operó en la bolsa de valores de NY, y sigue usándose en el sistema de control de tráfico aéreo francés o en el navío de guerra americano AEGIS (http://en.wikipedia.org/wiki/Ken_Birman). Actualmente, ISIS ha evolucionado hacia una nueva versión más centrada en replicación llamada VSync (http://vsync.codeplex.com/)

Multidifusión

Multidifusión ordenada total por acuerdo (ISIS)

- Variables para el proceso *q*
  - A_g^q: mayor número de secuencia *acordado* para el grupo *g*
  - P_g^q: mayor número de secuencia *propuesto* para el grupo *g*

- Implementación
  1. *p* hace B-multicast<m,i> a *g* (i identificador único de m)
  2. Cada proceso *q* responde a *p* con una propuesta $P_g^q = \text{máx}(A_g^q, P_g^q) + 1$
     - Se asigna al mensaje *m* el nº de secuencia P_g^q de modo provisional
     - Se coloca y ordena en la cola de retención según ese nº provisional
  3. *p* recoge todos los números de secuencia propuestos y selecciona el mayor (*a*) como acordado
     - Cada proceso fija $A_g^q = \text{máx}(A_g^q, a)$ y se lo asigna al mensaje i
     - Reordena la cola si difiere al propuesto
     - Cuando el mensaje al inicio de la cola tenga nº acordado, se entrega

Multidifusión

ISIS: ejemplo – multidifusión, propuestas y acuerdo

P1 P2 P3 P4

m m' m m' m m' m' m

m 1.1
m' 2.1

m 1.2
m' 2.2

m' 1.3
m 2.3

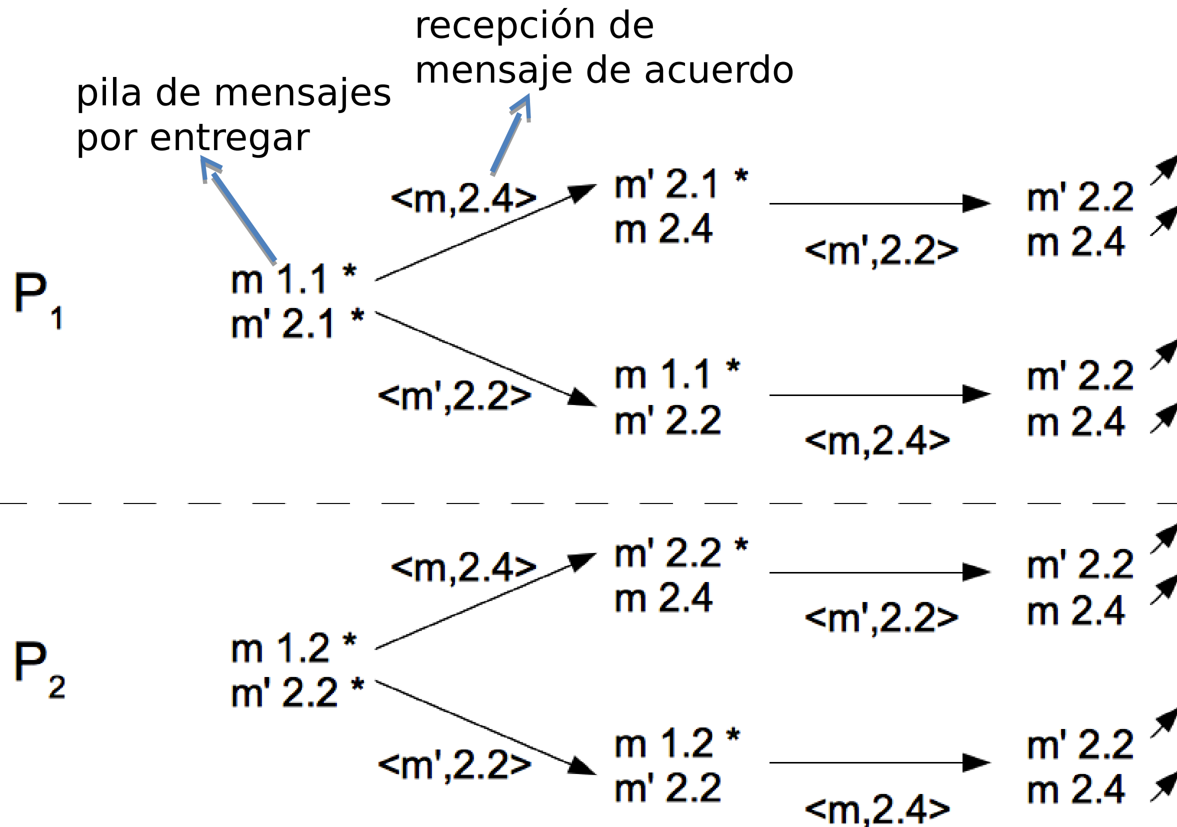
m' 1.4
m 2.4

Propuestas recibidas en

	P1: m propuestos	P2: m' propuestos
P_1	1.1	2.1
P_2	1.2	2.2
P_3	2.3	1.3
P_4	2.4	1.4
	acordado <m, 2.4>	acordado <m', 2.2>

+ Multidifusión

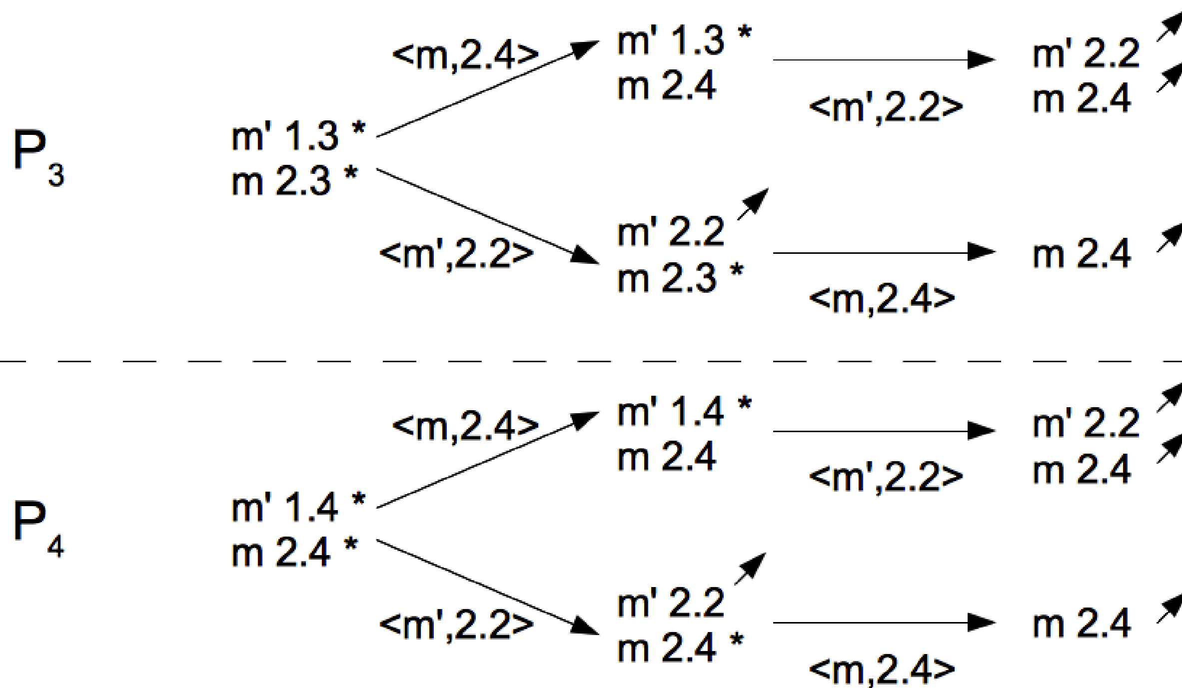
ISIS: ejemplo – acuerdo y entrega



* - provisional
↗ - entrega

+ Multidifusión

ISIS: ejemplo – acuerdo y entrega



* - provisional
↗ - entrega

+ Coordinación y acuerdo

- Introducción
- Exclusión mutua distribuida
- Elección distribuida
- Multidifusión
- Consenso distribuido

Consenso distribuido

- En un entorno distribuido, los procesos tienen dificultades para ponerse de acuerdo en un valor cuando uno o más procesos pueden proponer cuál debería ser
- Protocolos de acuerdo específicos
  - Exclusión mutua: consenso en quién entra en la SC
  - Elección: consenso en quién es el nuevo coordinador
  - Multidifusión ordenada: consenso en el orden de entrega
- En dichos protocolos, el consenso llega por un protocolo específico prefijado, no por una 'votación' de algún tipo.
  - Veremos a continuación aproximaciones más generales

Consenso distribuido

Modelo del sistema

- Sean N procesos p_i que se comunican por paso de mensajes
- La comunicación es fiable pero los procesos pueden fallar
  - Por caída
  - Por fallos arbitrarios (bizantinos)
- Hasta f de los N procesos pueden fallar
  - Se debe llegar a un consenso incluso en este caso
- Los procesos no firman sus mensajes
  - Esto puede ser relevante en algunos casos

Consenso distribuido

Definición

- Cada proceso p_i comienza en el estado *no decidido*
- Cada proceso p_i propone un único valor v_i de un conjunto de posibles valores D
  - Es el valor que quiere que tenga la variable d_i
- Los procesos se comunican entre sí intercambiando valores
  - Y buscan un valor de consenso, bien según un criterio de mayoría, mínimo, máximo, etc.
- Cada proceso fija el valor de la variable d_i sobre la que se busca consenso, y pasa al estado *decidido*
  - En el estado *decidido* ya no podrá cambiar el valor de d_i

+

Consenso distribuido

Condiciones

- **Terminación**
  - Cada proceso correcto ha de fijar su variable de decisión
- **Acuerdo**
  - El valor de decisión de los procesos correctos es el mismo
    - Si p_i y p_j son correctos y están en el estado *decidido* -> $d_i = d_j$ ($i,j = 1, 2, \ldots, N$)
- **Integridad**
  - Si todos los procesos correctos han propuesto el mismo valor, entonces cualquier proceso correcto en el estado *decidido* ha elegido dicho valor

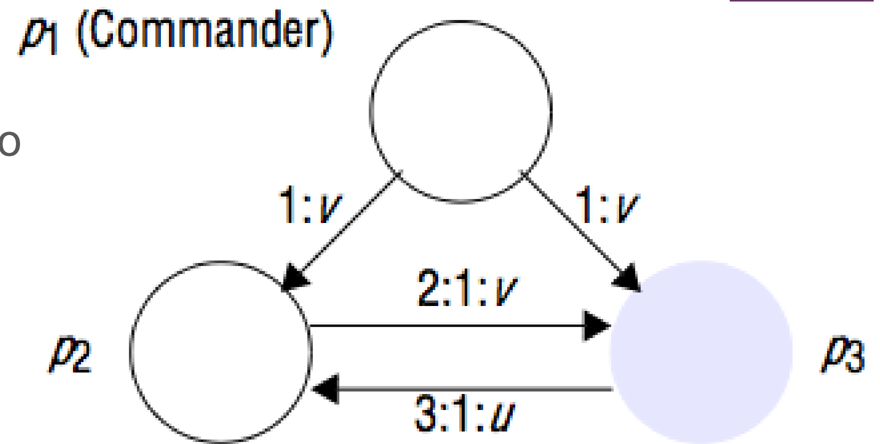
Consenso distribuido

El problema de los generales bizantinos

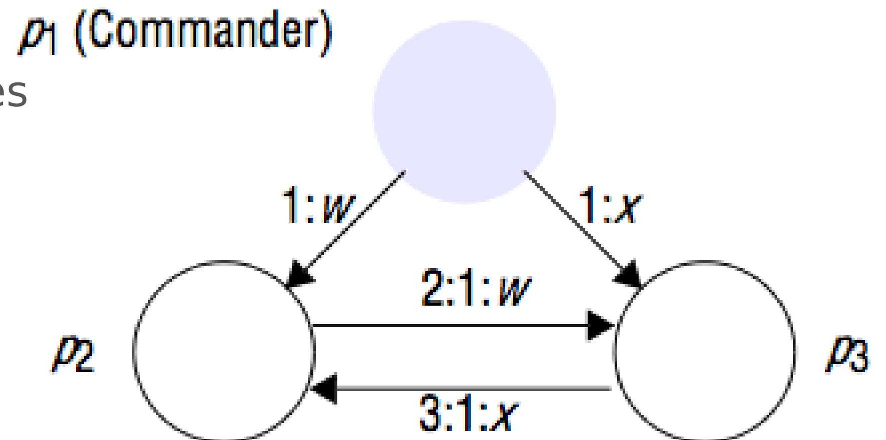
- Propuesto por Lamport et al. 1982
- *N* (>2) generales deben acordar si atacan o se retiran
  - Uno de ellos, el comandante, da la orden
- Se cumplen los supuestos de nuestro modelo de sistema
  - Especialmente, la posibilidad de *f* fallos arbitrarios
- **Punto clave**: el comandante define el valor de consenso, pero puede que dicho valor haya sido corrompido (fallo arbitrario) por el comandante o por algunos generales
- El problema tiene solución si $N \geq 3f+1$ y el sistema es *síncrono*

Consenso distribuido

Generales bizantinos: 3 procesos (1 falla)



- El comandante p_1 manda el valor correcto (v) pero p_3 falla y lo corrompe (u)
  - Tras de dos rondas, p_2 no puede determinar qué valor es el correcto



- El comandante p_1 falla y manda mensajes contradictorios a p_2 y p_3
  - De nuevo, p_2 no puede determinar qué valor es el correcto, ni qué proceso es el que ha fallado
- En ambos casos f=1 y N=3

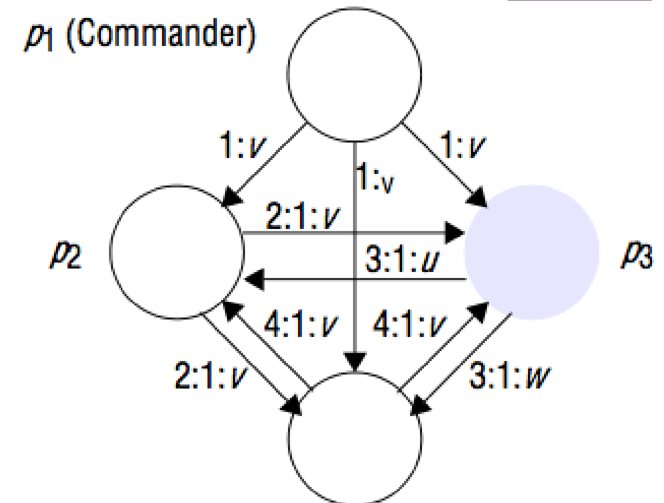
(procesos fallidos resaltados en azul)

(":" se lee "dice que")

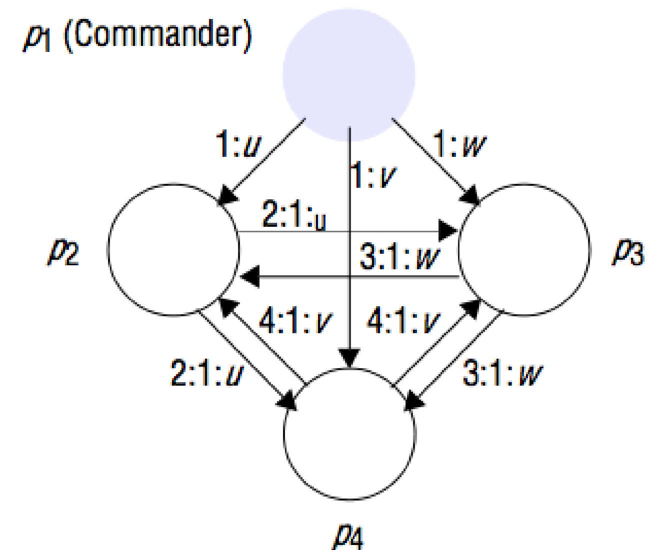
Consenso distribuido

Generales bizantinos: 4 procesos (1 falla)

- El comandante p_1 manda el valor correcto (v) pero p_3 lo corrompe (u)
  - Tras dos rondas, p_2 puede determinar qué valor es el correcto por mayoría



- El comandante p_1 falla y manda mensajes contradictorios a sus lugartenientes
  - Todos los procesos conocen el conjunto de mensajes enviados por el comandante, si coinciden y no hay consenso, hay un error en el comandante



Consenso distribuido

Generales bizantinos: análisis

- Medidas de eficiencia
  - Tiempo: nº de rondas de mensajes para llegar al consenso
  - Ancho de banda: nº de mensajes y tamaño
- En el algoritmo de Lamport, para $f \geq 1$ y mensajes sin firmar
  - $f + 1$ rondas
  - Cada ronda implica multidifusión: $O(N^{f+1})$ mensajes
- Fischer y Lynch [1982] probaron que $f + 1$ es el nº mínimo de rondas posible

Consenso distribuido

Imposibilidad de consenso en sistemas asíncronos

- Fischer et al. [1985] probaron que, en sistemas totalmente asíncronos* donde los procesos pueden fallar, un algoritmo nunca puede garantizar en todo caso el consenso
  - Basta con que un proceso falle por parada
- Incluso sin considerar caídas, demostraron que un algoritmo puede caer en una serie de decisiones repetitivas que eviten que se alcance el consenso
- Nótese que Fischer et al. no indican que el consenso no se alcance *nunca* sino la posibilidad de que no se alcance
  - En la práctica es improbable pues siempre hay algún grado de **aleatoriedad** que deshace situaciones de bloqueo

*Es decir, no podemos hacer asunciones de los tiempos de procesamiento o de los retardos de envío de mensajes

Consenso distribuido

Algoritmo Paxos: Roles

- Algoritmo de consenso distribuido propuesto por [Lamport1998]*
  - Se basa en tres roles (combinables) y dos fases

- Roles:
  - **Postulante(s)**: intentan conseguir que los oyentes elijan su propuesta**
    - A la propuesta se le asignará un número *N* en la fase 1 y se le asociará un valor *v* en la fase 2
  - **Oyentes**: actúan como una memoria 'a prueba de fallos' del protocolo
    - *Quórum*: conjunto de oyentes tal que si hay dos o más quórums, todos tengan intersección no nula (generalmente una mayoría)
  - **Ejecutores**: efectúan la propuesta que hayan acordado los oyentes
  - Los roles pueden ser compartidos por las mismas máquinas

* El nombre es una referencia a un sistema legislativo de consenso ficticio que se utilizaría en el parlamento de la isla griega de Paxos.

** La propuesta depende de la aplicación, puede ser un valor para una variable, la versión de un documento, etc.

Consenso distribuido

Algoritmo Paxos: Fase 1

- **Fase 1a) Preparación:**
  - Un postulante crea una propuesta con número *N* mayor al de cualquier otra propuesta anterior suya
  - La envía en un mensaje *preparación*(*N*) a un quórum de oyentes
- **Fase 1b) Promesa:**
  - Si el número de la propuesta recibida por un oyente es mayor que cualquier otro número de propuesta recibido desde cualquier postulante
    - Responde con un mensaje *promesa*(*N,* [*v*]) de ignorar todas las futuras propuestas con número menor que *N*
      - Si ya había respondido a alguna propuesta anteriormente, adjunta su valor *v* al mensaje de respuesta
  - Si no, ignora la propuesta recibida (o puede responder con un nack)

Consenso distribuido

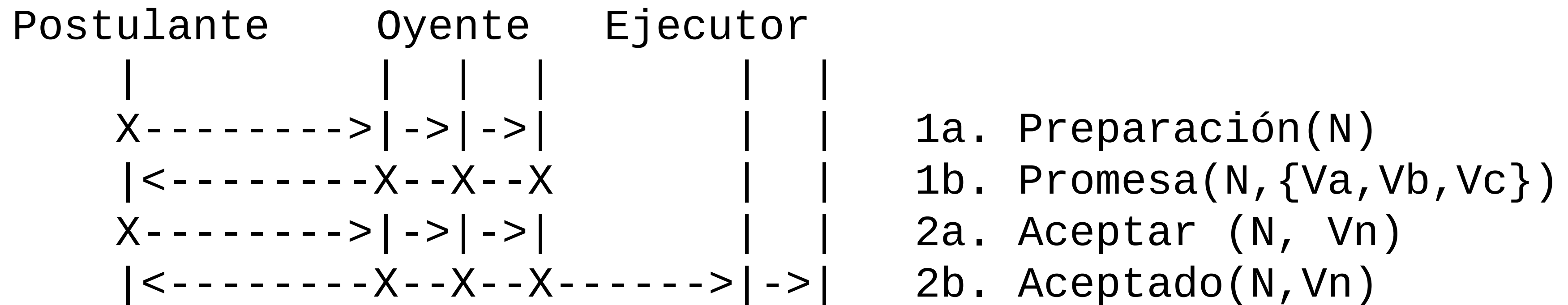
Algoritmo Paxos: Fase 2

- **Fase 2a) Aceptar la petición:**
  - Si el postulante recibe suficientes promesas del quórum, elige un valor *v* para su propuesta
    - Si algún oyente respondió indicando que ya había aceptado otras propuestas, debe elegir como valor el de la propuesta de número más alto comprometida por el quórum.
    - En caso contrario, puede elegir el valor que quiera
  - El postulante envía un mensaje *aceptar_petición(N, v)* al quórum

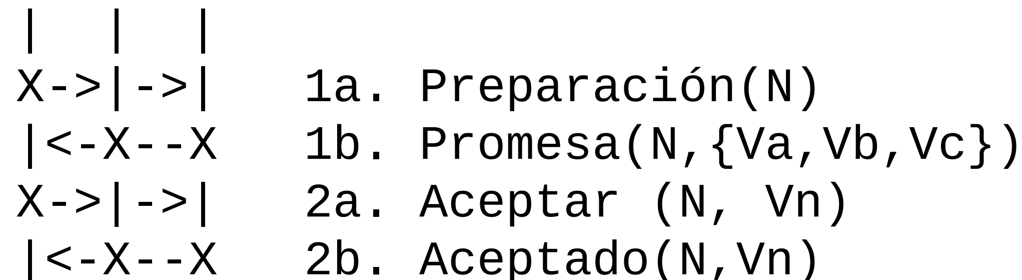
- **Fase 2b) Petición aceptada:**
  - Si el oyente no se ha comprometido con una petición *N'*>*N*:
    - Acepta la petición (*N*, *v*) y se la comunica a todos los ejecutores
  - Si se ha comprometido con una petición mayor, ignora la petición *N*

+ Consenso distribuido

Paxos: funcionamiento básico



(con roles colapsados)



Consenso distribuido

Paxos: ejemplo de falta de consenso

```
Postulante Oyentes Ejecutores
 |         |  |  |   | |
 X-------->|->|->|   | | Preparación(1)
 |<--------X--X--X   | | Promesa(1,{null,null,null})
 !         |  |  |   | | !! POSTULANTE FALLA
    |      |  |  |   | | !! NUEVO POSTULANTE(sabe que el último número era 1)
    X----->|->|->|   | | Preparación(2)
    |<-----X--X--X   | | Promesa(2,{null,null,null})
 |  |      |  |  |   | | !! ANTIGUO POSTULANTE se recupera!
 |  |      |  |  |   | | !! ANTIGUO POSTULANTE intenta 2, rechazado
 X-------->|->|->|   | | Preparación(2)
 |<--------X--X--X   | | Nack(2)
 | |       |  |  |   | | !! ANTIGUO POSTULANTE intenta 3
 X-------->|->|->|   | | Preparación(3)
 |<--------X--X--X   | | Promesa(3,{null,null,null})
 | |       |  |  |   | | !! NUEVO POSTULANTE  es rechazado
 | X------>|->|->|   | | Aceptar (2,Va)
 | |<------X--X--X   | | Nack(3)
 | |       |  |  |   | | !! NUEVO POSTULANTE intenta 4
 | X------>|->|->|   | | Preparación(4)
 | |<------X--X--X   | | Promesa(4,{null,null,null})
 | |       |  |  |   | | !! VIEJO POSTULANTE es rechazado
 X-------->|->|->|   | | Aceptar(3,Vb)
 |---------X--X--X   | | Nack(4)
 | |       |  |  |   | | ... Y así continuamente ...
```

Consenso distribuido

Influencia de Paxos

- Paxos funciona en dos fases: sondeo y ejecución
  - Este modo de funcionamiento es común a muchos algoritmos de acuerdo distribuido*:

Algoritmo	**Aplicación**	**Fase 1**	**Fase 2**
Ricart y Agrawala	Exclusión mutua	Sondeo uso SC	Entrada SC
Bully	Elección	Sondeo procesos activos mayores	Elección de coordinador
ISIS	Multidifusión ordenada	Sondeo de tiempos de recepción	Elección de tiempo de entrega
2PC	Transacciones	Petición de commit	Commit/rollback

*Google, Bitcoin, IBM, Heroku, VMWare, Amazon Web Services, Apache, Oracle, Azure etc. usan Paxos o derivados en algún contexto
https://en.wikipedia.org/wiki/Paxos_(computer_science)

Consenso distribuido

2PC: Protocolo de bloqueo del turno

- Utilizado en juegos online de estrategia por turnos [Baughman2001]
  - Una trampa frecuente es retrasar artificialmente mis decisiones para tomarlas cuando conozco las de los demás

- Funcionamiento:
  - Cada jugador decide su acción *A* para el turno *t*+1,
  - **Fase 1:** Confirma su acción, pero la revela encriptada: $H(A)=A'$
    - Mediante un hash criptográficamente seguro de dirección única
  - **Fase 2:** Cuando todos los jugadores han confirmado su *A'*
    - Se revelan las acciones *A*
  - Se comprueba fácilmente si $H(A)=A'$ para todos los jugadores

+ Resumen

- En un SD hay que **acordar decisiones**, sobre el **acceso a recursos** y el **valor de variables**
- Para el acceso a recursos hay algoritmos de **exclusión mutua** que tratan con secciones críticas accesibles mediante la **posesión de objetos distribuidos**, que se comunican de manera **centralizada**, en **anillo** o por **multifidusión (ricart & agrawala)**
- Para decidir sobre el valor de variables u otros asuntos debemos elegir un nodo **coordinador**. Los algoritmos de elección buscan el nodo con id más alto mediante comunicación en **anillo** o **multidifusión (bully)**
- **Multidifundir** mensajes a un grupo de procesos de manera **fiable** no es trivial, y menos si lo queremos hacer de manera **ordenada** (varios tipos de ordenación)
- Los algoritmos vistos consideran un **modelo sin fallos**, o sólo con fallos de ruptura, y **asíncrono**. Si consideramos que los procesos pueden tener **fallos arbitrarios**, los algoritmos necesitan llegar a **consensos**, como es el caso del problema de los **generales bizantinos** y el algoritmo **Paxos**
- En cualquier algoritmo de acuerdo, hay varias consideraciones a tener en cuenta: el **tráfico** de mensajes, el **tiempo** hasta llegar a un acuerdo y **cuestiones específicas** del problema/arquitectura en cuestión

+ Referencias

- G. Colouris, J. Dollimore, T. Kindberg and G. Blair. *Distributed Systems: Concepts and Design (5th Ed)*. Addison-Wesley, 2011
  - Capítulo 15

- Algoritmo de exclusión mutua distribuido de Ricart/Agrawala:
  - G. Ricart y A.K. Agrawala. *An optimal algorithm for mutual exclusion in computer networks*. Communications of the ACM. **1981.**
  - http://vis.usal.es/rodrigo/documentos/papers/Ricart1981.pdf

- Algoritmo de exclusión mutua distribuido de Maekawa:
  - M. Maekawa*. A $\sqrt{N}$ Algorithm for mutual exclusion in decentralized systems*. ACM Transactions on Computer Systems. **1985**
  - http://vis.usal.es/rodrigo/documentos/papers/Maekawa1985.pdf

+ Referencias

- Algoritmo abusón para elección distribuida (artículo original)
  - H. García-Molina. *Elections in a Distributed Computing System.* IEEE Trans. on Computers. **1982**
  - http://vis.usal.es/rodrigo/documentos/papers/GarciaMolina1982.pdf
- Sobre la explicación de la multidifusión total en ISIS (protocolo ABCAST sección 4.3.1)
  - K. Birman et al. *Reliable Communication in the Presence of Failures*. ACM Trans. on Computer Systems. **1987**.
  - http://vis.usal.es/rodrigo/documentos/papers/Birman87.pdf
- Aplicación de la multidifusión fiable en distintos contextos reales
  - K. Birman. *A Review of Experiences With Reliable Multicast,* **1998**
  - http://vis.usal.es/rodrigo/documentos/papers/Birman98.pdf

Referencias

- El problema de los generales bizantinos (artículo original)
  - L. Lamport et al. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems. **1982**
  - http://vis.usal.es/rodrigo/documentos/papers/Lamport82.pdf
- Algoritmo Paxos
  - L. Lamport. *The Part-Time Parliament*. ACM Transactions on Computer Systems 16, 2. **1998**
    - http://vis.usal.es/rodrigo/documentos/papers/lamport-paxos.pdf
  - L. Lamport. *Paxos Made Simple.* 2001:
    - http://vis.usal.es/rodrigo/documentos/papers/paxos-simple.pdf
- Algoritmo de bloqueo de turno para juegos online
  - N.E. Baughman and B.N. Levine. *Cheat-Proof Playout of Centralized and Distributed Online Games*. IEEE Infocom. **2001**
  - http://vis.usal.es/rodrigo/documentos/papers/Baughman2001.pdf

Belisario, general bizantino (505-565 dC)

http://listverse.com/2010/06/27/10-generals-who-got-in-trouble-with-their-chief/