



Desarrollo de Sistemas Distribuidos

Tema 3 **Coordinación**

Contenidos

1. Tiempo lógico

2. Algoritmos Distribuidos

- Exclusión Mutua
- Elección
- Consenso

Tiempo Lógico

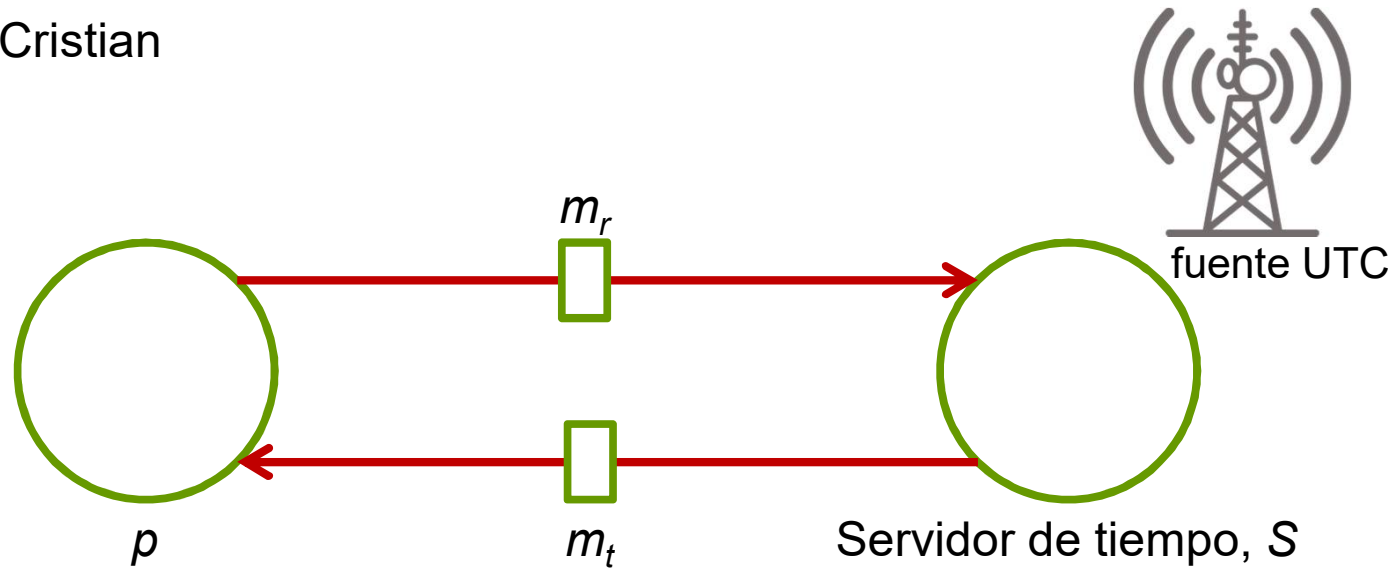
- Cuestiones de tiempo importantes en SD por:
 1. **Medida** que deseamos obtener con precisión para:
 - a) **Sincronización externa**: cuándo ocurrió un evento concreto (a qué hora sucedió, p.ej. transferencia bancaria). Para ello es necesario sincronizar la hora de la máquina donde ocurrió el evento con algún **reloj o fuente externa autorizada** (relojes atómicos que transmiten por radio terrestre o satélite, o red telefónica)
 - b) **Sincronización interna**: se trata de obtener las mismas referencias de tiempo o intervalo entre dos eventos ocurriendo en dos computadoras diferentes **conociendo precisión**, para un instante dado (p.ej., tiempos de transmisión de un mensaje entre máquinas → dos marcas de tiempo: una en origen y otra en destino)
 2. **Problemas lógicos** debidos a la distribución (p.ej., para mantener la consistencia en un gestor de transacciones bancarias, auditorías...)
- **Evento**: Acción que parece ocurrir indivisiblemente (p.ej. envío de mensaje)
- El **orden de la ocurrencia de eventos** puede ser **crítico** en aplicaciones distribuidas (p.ej., servidor de datos replicados)

Tiempo Lógico

- Requisitos y tipos de aplicaciones:
 - **Centralizadas:** Sólo necesitan conocer el **orden de los eventos**, con lo que basta asociar un **reloj** (tiempo absoluto) o **contador** (tiempo relativo) a cada evento
 - **Distribuidas:**
 - Conocer el desplazamiento relativo del tiempo (reloj) de una máquina con respecto a otra, e idéntica velocidad del pulso → casi imposible
 - Otra opción es que exista un reloj físico compartido (sistemas síncronos)
 - Servidor de tiempo sobre peticiones (sistemas asíncronos):
 - Existe un método (**Cristian**) para sincronizar relojes que se basa en el tiempo universal coordinado (estándar internacional) y en la existencia de un servidor de tiempo. **Problema:** fallo del servidor, o de una replica de éste o impostor (que responda a los mensajes *multicast*)
 - ¿Contador? ¿centralizado o distribuido?

Tiempo Lógico

Método Cristian



$$T_{round} = T_{recibir_mt} + T_{enviar_mr}$$

- Asumiendo iguales tiempos de envío y recepción, p puede fijar su reloj a: $t + T_{round}/2$

Tiempo Lógico

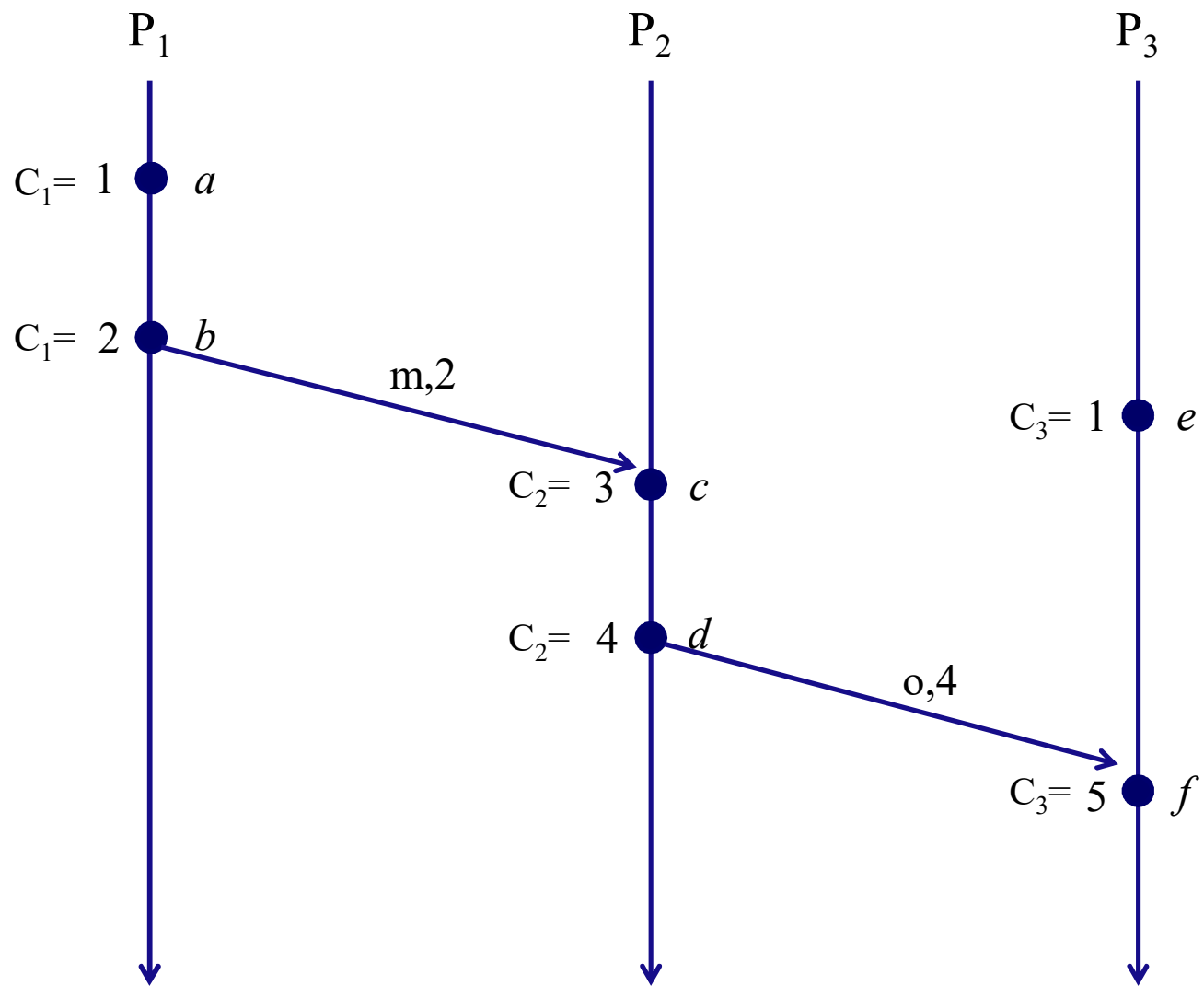
- **Relación de orden (*ocurrió-antes*):**
 - **Esquema de ordenación** de eventos basado en dos puntos:
 1. Si dos eventos ocurren en el mismo proceso, entonces ocurren en el orden que se observan
 2. Si se envía un mensaje, entonces el evento asociado al envío **ocurre antes** que el evento de recepción de dicho mensaje
 - *Lamport* generalizó estas dos relaciones en una **relación de orden causal** denominada ***ocurrió-antes* (\rightarrow)**:
 1. Si $\exists p: x \xrightarrow{p} y$ (en p) entonces $x \rightarrow y$
 2. $\forall m \in \text{Mensajes}, \text{send}(m) \rightarrow \text{receive}(m)$
 3. Siendo $x, y, z \in \text{Eventos}$: $x \rightarrow y$ e $y \rightarrow z$ entonces $x \rightarrow z$

Tiempo Lógico

- **Relojes lógicos:**
 - Mecanismo simple que propuso *Lamport* para capturar numéricamente la relación causal *ocurrió antes*
 - Un **reloj lógico** es un contador software que se incrementa monótonamente:
 - C_p : nota el reloj lógico C del proceso p
 - $C_p(a)$: nota la marca de tiempo del evento a en el proceso p
 - $C(b)$: nota la marca de tiempo del evento b en cualquier proceso donde haya ocurrido

Tiempo Lógico

- **Relojes lógicos:**
 - Para capturar la relación *ocurrió-antes*, los procesos actualizan sus relojes lógicos y transmiten sus valores en los mensajes:
 1. C_p se incrementa antes de cada evento que ocurre en p
 2. Cuando un proceso p envía un mensaje le añade el valor $t = C_p$
 3. Cuando un proceso q recibe un mensaje entonces:
 - computar $C_q = \max(C_q, t)$ y
 - aplicar *acción 1* antes de marcar el evento $receive(m, t)$
 - Fácil demostrar que si $a \rightarrow b$ entonces $C(a) < C(b)$
 - Extensión a relación de orden total:
 - $C(a) < C(b) \Leftrightarrow C_p(a) < C_q(b) \vee (C_p(a) = C_q(b) \wedge p < q)$



Algoritmos Distribuidos

- **Necesarios en muchas aplicaciones:** telecomunicaciones, procesamiento de información distribuida, computación científica, control de procesos en tiempo real, etc.
- **Requieren** realizar diseño, implementación y análisis.
- **Clases de algoritmos distribuidos según atributos:**
 - **Modelo de temporización de eventos:** asíncrono (sin suposiciones de tiempo), completamente síncrono (límites en el retardo de mensajes, ejecución por pasos, y derivas de velocidad del reloj), o parcialmente síncrono (con plazos de tiempo).
 - **Método de comunicación (IPC):** memoria compartida, punto-punto, grupos, RPC, ...
 - **Modelo de fallos:** completamente fiable o tolerante a ciertos fallos (procesador o comunicaciones).
 - **Problemas abordados:** asignación recursos, comunicación datos, consenso, control concurrencia, detección de bloqueos, etc.

Algoritmos Distribuidos

- **Mayor distinción** en base a modelo de temporización:
 - **Síncrono** (componentes dan pasos simultáneamente):
 - El más simple de describir, programar y razonar
 - No es la realidad pero ayuda a entender cómo resolver el problema abordado
 - Es imposible o ineficiente implementarlo en muchos sistemas distribuidos
 - **Asíncrono** (componentes dan pasos arbitrariamente):
 - Razonablemente simple de describir; con complicaciones sobre vivacidad
 - Más difícil de implementar por incertidumbre en el orden de los eventos, pero ello resulta en una solución más general y portable.
 - A veces no proporcionan soluciones, o no son eficientes
 - **Parcialmente síncrono** (con ciertas restricciones en temporización relativa de los eventos):
 - Modelo más realista, pero difícil de programar.
 - Pueden ser eficientes pero también frágiles ya que no funcionan correctamente si no se cumplen las restricciones de temporización de los eventos.

Algoritmos Distribuidos

- **Exclusión Mutua:**
 - Cuando **no existe núcleo central local** para basar la exclusión mutua en variables u otras facilidades compartidas
 - **Ejemplos:**
 - Existencia de servidores o recursos que no tienen mecanismos de sincronización incorporados, e.g.: NFS se diseña como servidor sin estado y por tanto no soporta bloqueo de archivos, por ello UNIX proporciona un demonio aparte (*lockd*) que sirve las peticiones de los clientes.
 - Coordinación distribuida en servicios replicados o distribuidos
- **Elección:**
 - Método para escoger un único proceso que realice un **rol concreto**
 - **Ejemplo:** elección de servidor replicado primario cuando el anterior falla
- **Consenso:**
 - Método para que un grupo de procesos acuerden un **mismo valor**
 - **Ejemplo:** orden de los mensajes, generales bizantinos, ...

Algoritmos Distribuidos

- Exclusión Mutua:
 - Requisitos básicos y comunes:
 - Propiedades de **seguridad y vivacidad**
 - Adicional: **Orden causal** en la entrada a la sección crítica
 - **Soluciones:**
 1. Servidor centralizado
 2. Algoritmo distribuido basado en relojes lógicos
 3. Algoritmo basado en anillo

Algoritmos Distribuidos

- Exclusión mutua con **servidor centralizado**:
 - Para **entrar en sección crítica** se envía una petición al servidor y se espera la respuesta (***testigo*** o ***token***)
 - El servidor **encola peticiones** cuando no dispone del *testigo*
 - Cuando un proceso **sale de la sección crítica** envía un mensaje de liberación (devuelve el *testigo*), si el servidor tiene mensajes de petición encolados le envía el *testigo* al primero de ellos y lo saca de la cola

Algoritmos Distribuidos

- Exclusión mutua con **servidor centralizado**:
 - El servidor se puede convertir en un **cuello de botella**
 - El servidor es punto crítico de fallo
 - Hay que **regenerar el testigo** si el cliente que lo tiene **falla**

Algoritmos Distribuidos

- Exclusión mutua con **algoritmo distribuido basado en relojes lógicos (Ricart/Agrawala):** Tiene como maximo $(2 \cdot (n-1))$

- **Idea básica:** los procesos que desean entrar en la **sección crítica** envían un mensaje *multicast* a los otros $n-1$ procesos. Un proceso puede entrar si todos los demás le responden, es decir, la obtención del *testigo* requiere n mensajes.
- **Suposiciones:**
 - Los procesos **conocen las direcciones** de los demás
 - Paso de mensajes **fiable**
 - Cada proceso mantiene su **reloj lógico**

Algoritmos Distribuidos

- **Algoritmo distribuido basado en relojes lógicos (proceso P_i):**

- **Inicialización:**

estado := *LIBERADO*

- **Obtención del toquen:**

estado := *INTENTANDO*;

Envío selectivo de petición a los demás procesos;

T_i := marca de tiempo de la petición;

wait until (número de respuestas recibidas = $(n-1)$);

estado := *EN_SECCION_CRITICA*;

- **Recepción de una petición $\langle T_j, P_j \rangle$ en P_i ($i \neq j$):**

if (*estado*=*EN_SECCION_CRITICA* or (*estado*=*INTENTANDO* and $(T_i, P_i) < (T_j, P_j)$))

then encolar petición de P_j ;

else enviar mensaje de respuesta a P_j ;

- **Liberación del Testigo:**

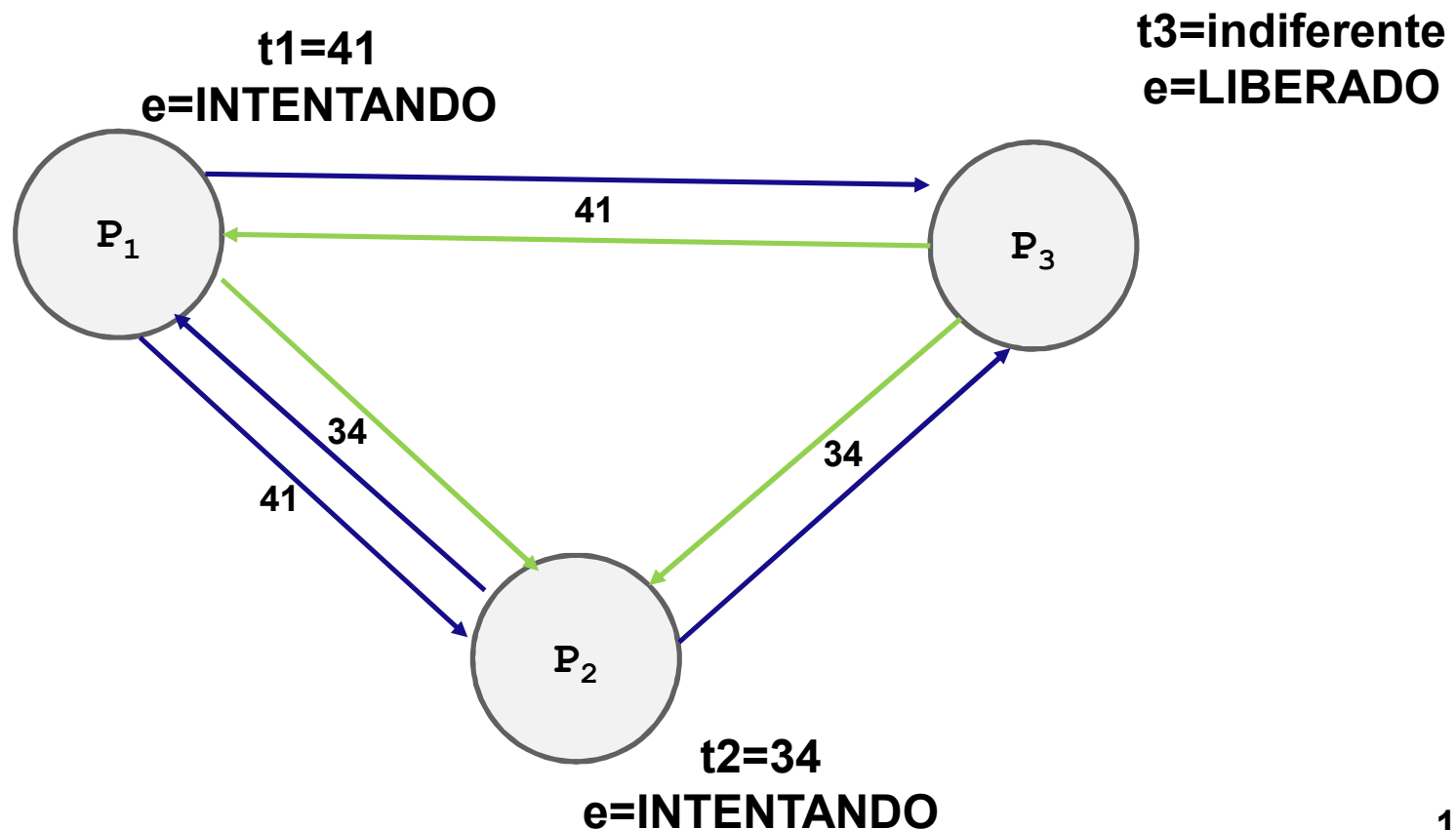
estado := *LIBERADO*;

Enviar mensaje de respuesta a todas las peticiones encoladas y eliminarlas;

Aplazada la recepción
y el procesamiento de
peticiones

Algoritmos Distribuidos

- Ejemplo:



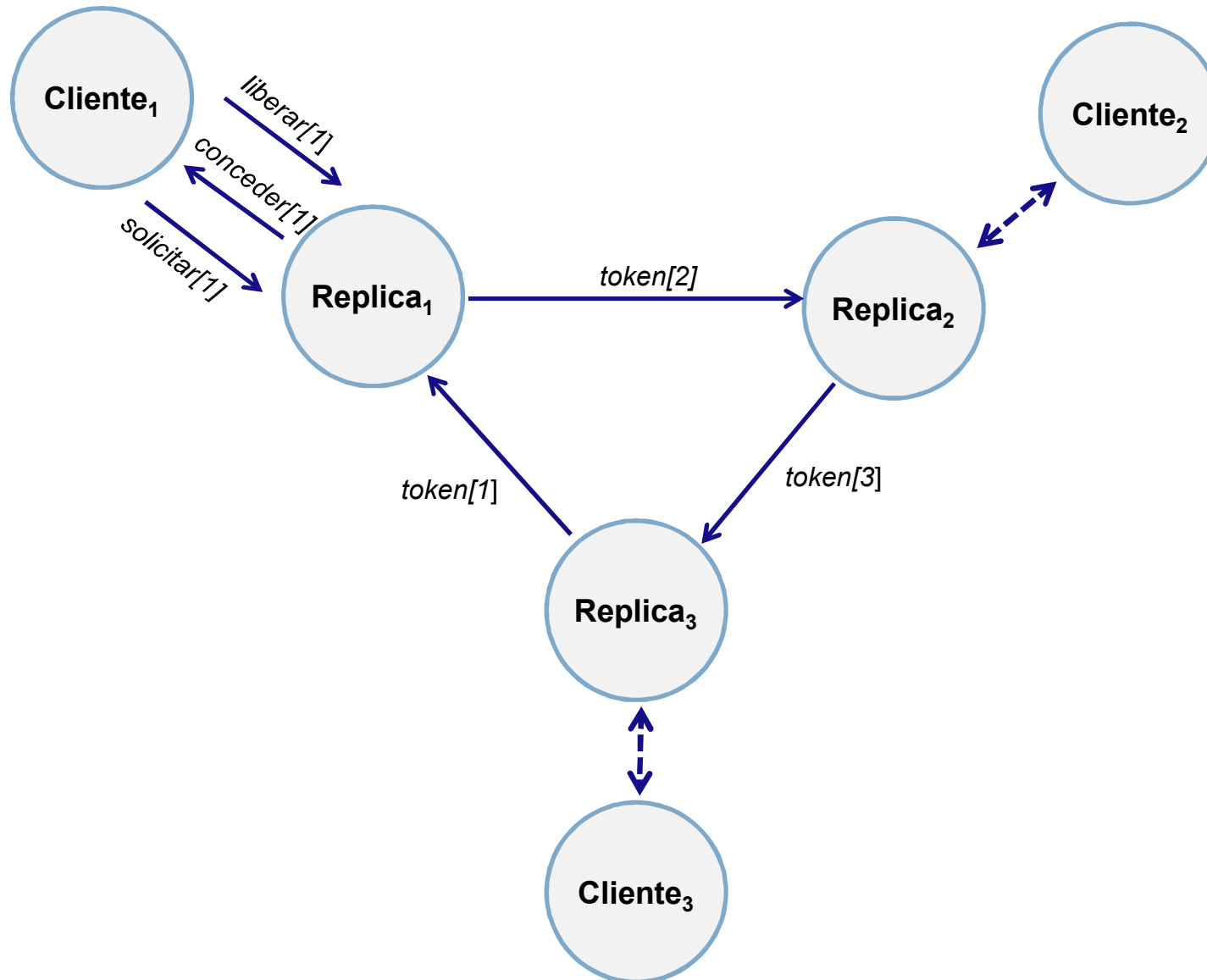
Algoritmos Distribuidos

- Exclusión mutua con **algoritmo distribuido basado en relojes lógicos**:
 - Obtener el testigo requiere ? mensajes
 - **Más costoso** que el algoritmo centralizado
 - Cualquier proceso es punto crítico de fallo
 - Cada proceso recibe peticiones y envía respuestas, por tanto, el **cuello de botella** puede ocurrir
 - Variante: algoritmo de votación de *Maekawa* donde cada proceso obtiene permiso de su subconjunto de procesos asociados, cada proceso responde a la primera petición recibida, y las intersecciones de los conjuntos no pueden estar vacías.

Algoritmos Distribuidos

- Exclusión mutua con **algoritmo basado en anillo**:
 - Los procesos se configuran en un anillo lógico (cada proceso conoce la dirección de sus vecinos)
 - El *testigo* circula en una sola dirección y el proceso que lo tiene puede acceder a la sección crítica; en caso contrario ha de esperar
 - **Suposiciones**:
 - Cada proceso conoce la dirección de su vecino por la derecha
 - Paso de mensajes **fiable**

Exclusión Mutua en anillo



Algoritmos Distribuidos

- Exclusión mutua con **algoritmo basado en anillo**:
 - Obtener el testigo requiere máximo $n-1$ mensajes
 - El testigo está **continuamente circulando**
 - Si un proceso falla se ha de **reconfigurar el anillo**
 - **Regenerar testigo** si se pierde
 - **No** es posible asegurar el cumplimiento de la relación *ocurrió-antes*

Algoritmos Distribuidos

- **Elección:**
 - Se trata de escoger un único proceso de un conjunto de ellos, por ejemplo, debido al fallo de otro proceso
 - Principal **requisito** → que el proceso sea **único** (con mayor identificador) incluso si varios solicitan la **elección** simultáneamente
 - **Soluciones:**
 1. Algoritmo del Valentón (*Bully*)
 2. Algoritmo basado en anillo

Algoritmos Distribuidos

- Elección con el **algoritmo del valentón**
- **Requisitos:**
 - Los miembros del grupo se conocen, aunque puede haber caído más de un proceso aparte del coordinador
 - Paso de mensajes fiable
- Tres tipos de mensajes:
 - **Elección:** Para anunciar una elección
 - **Respuesta:** Se envía como respuesta a un mensaje de elección
 - **Coordinador:** Se envía para anunciar el *identificador* del nuevo proceso coordinador
- Costoso: se requieren hasta n^2 mensajes

Algoritmos Distribuidos

- **Pasos algoritmo del valentón:**
 1. Un proceso comienza una elección cuando detecta que un proceso ha fallado → Envía un mensaje de **“elección”** a los procesos con identificador más alto que él mismo
 2. Espera un mensaje de **“respuesta”**
 3. **Si no llega el mensaje de respuesta**, se proclama coordinador y envía un mensaje **“coordinador”** a todos los procesos con id más bajo que él; **en otro caso**, espera un tiempo a que llegue un mensaje **“coordinador”** del proceso elegido; si éste no llega comienza una nueva elección
 4. Si un proceso recibe un mensaje **“coordinador”** graba el identificador contenido en dicho mensaje
 5. Si recibe un mensaje de **“elección”** devuelve un mensaje de **“respuesta”** y comienza una elección (a menos que ya haya iniciado una)
 6. Cuando se restablece un proceso que había fallado, comienza una nueva elección. Si tiene el id más alto será el nuevo coordinador junto con el actual hasta que éste reciba el mensaje **“coordinador”**

Algoritmo del Valentón

detecta p1

p2

p3



Coordinador

Leyenda



mensaje de elección

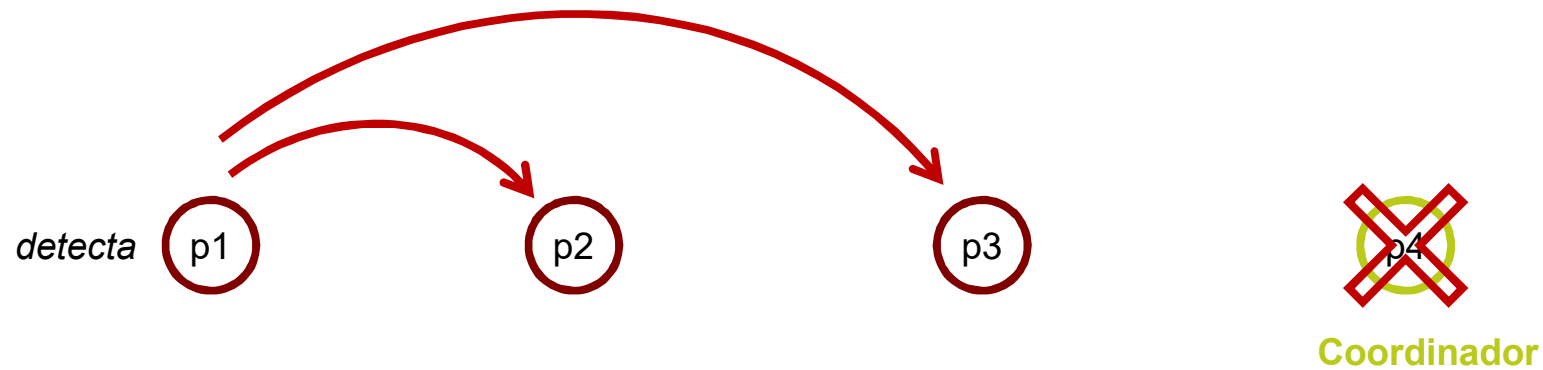


mensaje de respuesta

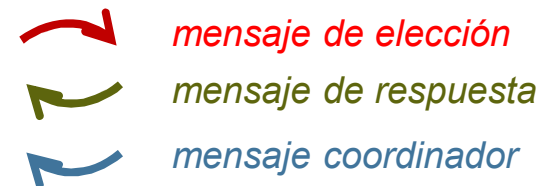


mensaje coordinador

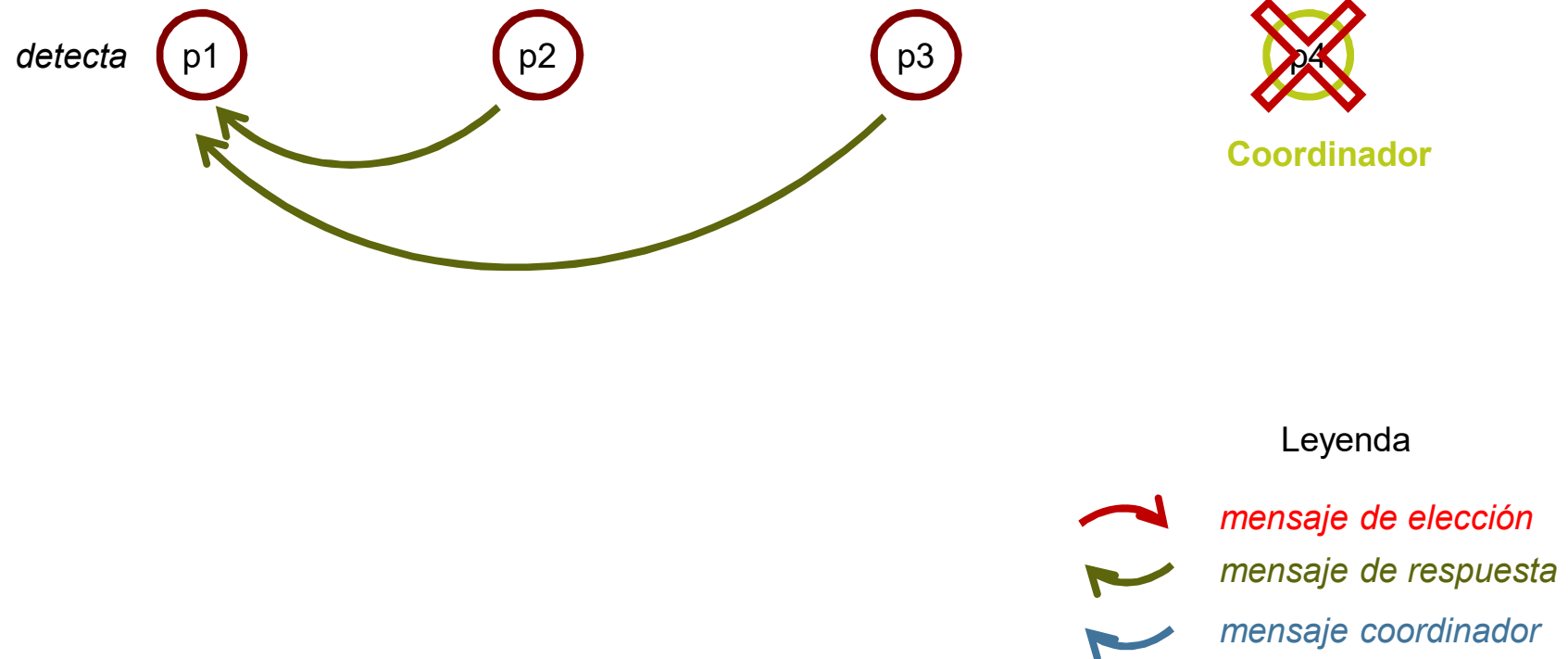
Algoritmo del Valentón



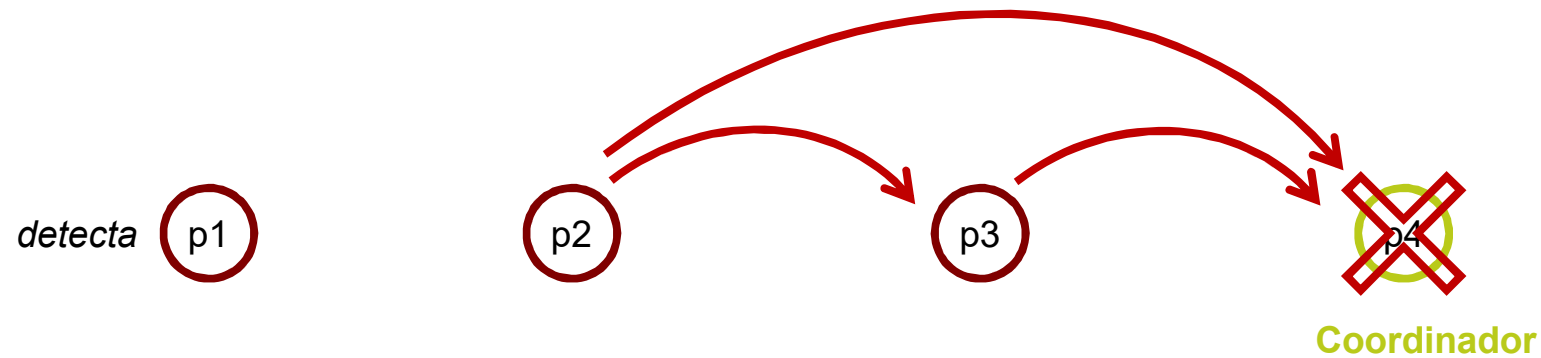
Leyenda






Algoritmo del Valentón



Algoritmo del Valentón



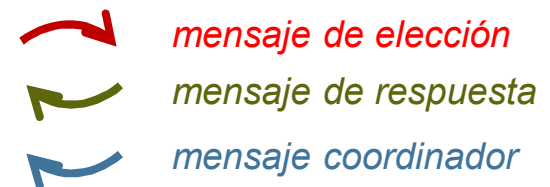
Leyenda

-  *mensaje de elección*
-  *mensaje de respuesta*
-  *mensaje coordinador*

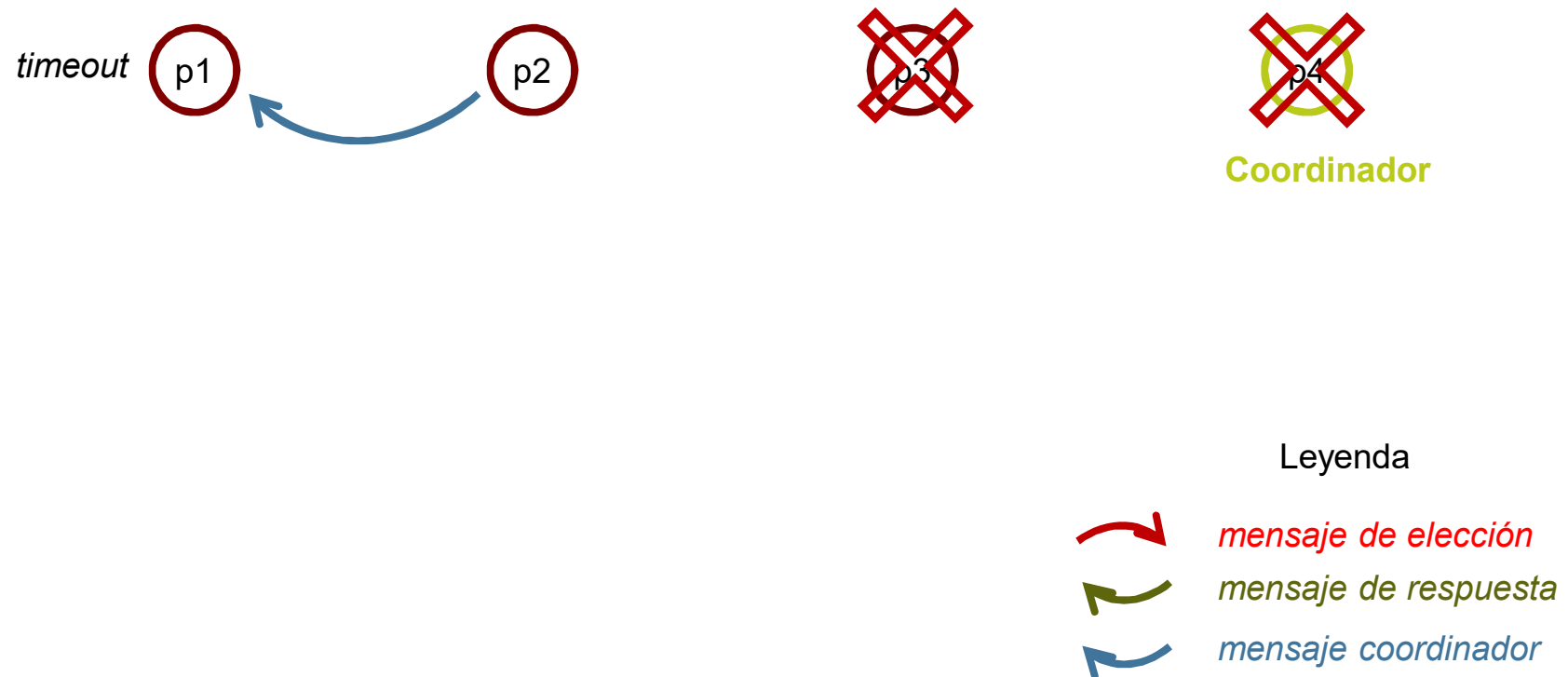
Algoritmo del Valentón



Leyenda



Algoritmo del Valentón



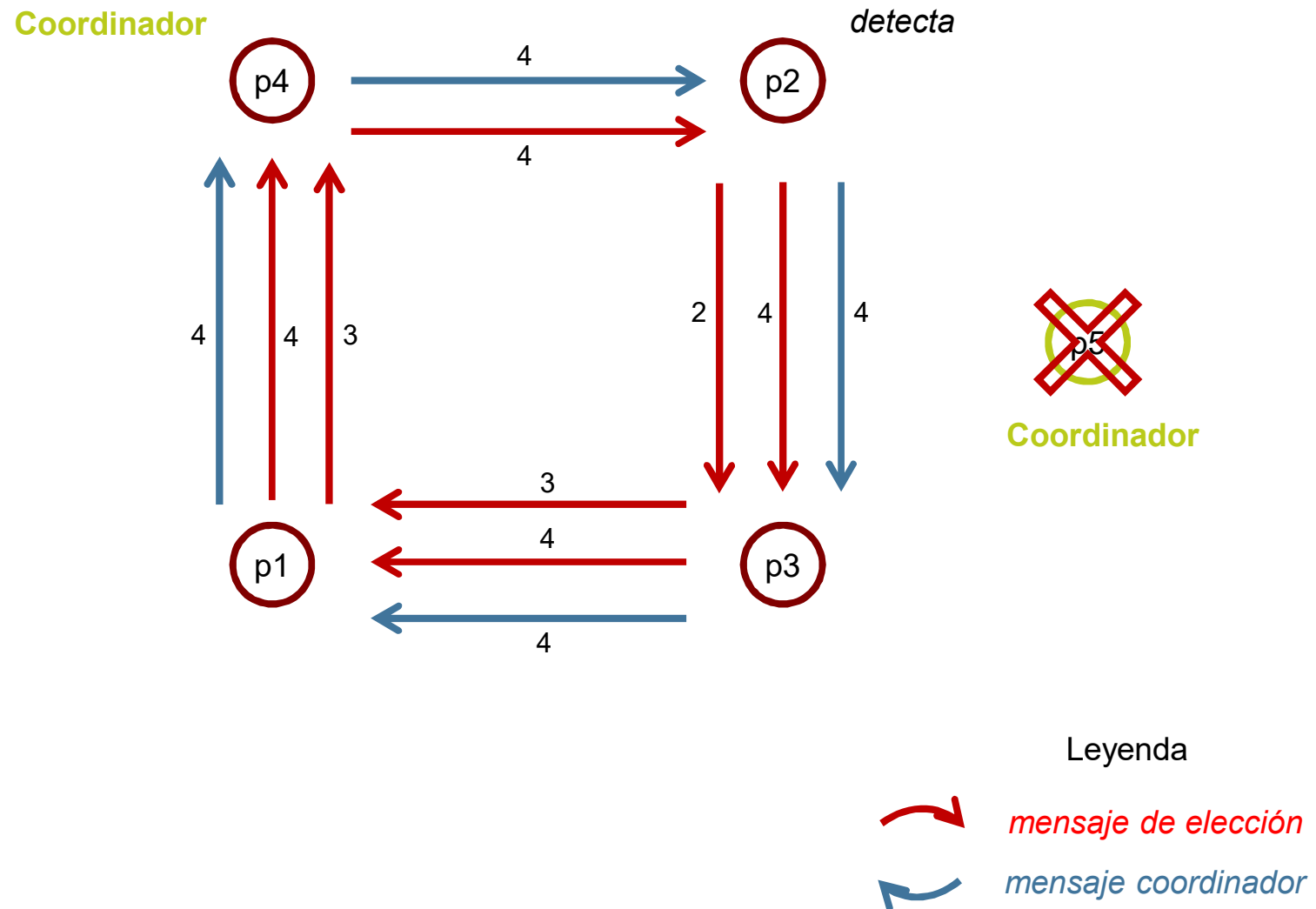
Algoritmos Distribuidos

- Elección con **algoritmo basado en anillo** (*Chang/Roberts*)
- **Requisitos:**
 - Los procesos están organizados en anillo lógico, aunque sin coincidir el orden con su identificador
 - Paso de mensajes fiable y los procesos no fallan durante la elección
- Los procesos indican su participación, tipos de mensajes:
 - **Elección**: Para anunciar que hay una elección en curso
 - **Coordinador**: Se envía para anunciar el id del nuevo proceso coordinador
- Costoso: se requieren hasta $3n - 1$ mensajes

Algoritmos Distribuidos

- Elección con el **algoritmo basado en anillo**
- **Pasos:**
 - Inicialmente, cada proceso está marcado como **no-participante**. Cualquiera puede comenzar la elección → se marca como **participante** y envía un mensaje de elección con su id de proceso a su vecino por la derecha
 - Si recibe un mensaje de **“elección”**, compara su id con el del mensaje:
 - Si su id es mayor que el recibido en el mensaje →
 - Si está marcado como **no participante**, sustituye el id del mensaje de elección por el suyo y lo envía
 - Si está marcado como **participante**, no envía el mensaje
 - Si es menor → pasa el mensaje de **“elección”** recibido a su vecino
 - En cualquier caso, si no lo estaba, se marca como **participante**
 - Si el *id* recibido es el del propio receptor (igual) → es el proceso con mayor id y se convierte en **coordinador**. Se marca como **no-participante** y envía un mensaje de **“coordinador”** a su vecino con su *id*
 - Si se recibe un mensaje de **“coordinador”**, se marca como **no-participante** y reenvía dicho mensaje a su vecino

Elección en anillo



Algoritmos Distribuidos

- **Consenso:**
 - Se trata resolver de forma precisa y generalizada problemas de consenso (acuerdo) y relacionados (i.e. acordar valores después de enviar diferentes propuestas):
 - Solución a los **generales Bizantinos** (un valor), **consistencia interactiva** (un vector), **multicast totalmente ordenado**.
 - Ejemplos de problemas de acuerdo específico:
 - transacciones (e.g. acordar una transferencia bancaria entre cuentas),
 - exclusión mutua (qué proceso entra en sección crítica),
 - elección (acordar proceso elegido), y
 - comunicaciones en grupo (acordar orden de los mensajes).

Algoritmos Distribuidos

- **Consenso:**
 - **Definición:** Cada proceso propone un valor que comunica a los otros procesos, y establece el valor de la decisión que ya no puede cambiar.
 - Principales **requisitos/propiedades:**
 - Comunicaciones **fiables**.
 - Procesos pueden fallar **arbitrariamente** (*Byzantine*) i.e.: los procesos pueden caer/detenerse, omitir pasos o darlos incorrectamente (confundir).
 - Los procesos correctos: acabarán eventualmente (**terminación**), deberán decidir el mismo valor (**acuerdo**), y si todos los correctos propusieron el mismo valor entonces cualquier proceso correcto ha escogido dicho valor (**validez o integridad**).

Algoritmos Distribuidos

- Algoritmo general Consenso:
 - Síncrono, *multicast* básico con *timeouts*, f procesos (del total de N) pueden caer durante las rondas:

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r ($1 \leq r \leq f+1$)

$B\text{-multicast}(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On $B\text{-deliver}(V_j)$ from some p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f+1)$ rounds

Assign $d_i = \text{minimum}(Values_i^{f+1})$;

[Fuente: *Distributed Systems: Concepts and Design (5th Edition)*; G. Coulouris et al, 2012]

Algoritmo general Consenso

<i>r</i>	Values	P1	P2	P3	P4	P5
	0					
1	1	7	4	5	×	3
2	2	7 5 4 3 ×	7 5 4 3	7 5 4 3		7 5 4 3
3	3		7 5 4 3 1	7 5 4 3		7 5 4 3
	Resultado después <i>r</i> = 3		$d_2 = \min (7,5,4,3,1)$	$d_3 = \min (7,5,4,3,1)$		$d_5 = \min (7,5,4,3,1)$

Leyenda

N: Valor enviado en *r*

×: Cae el proceso enviando valores

Algoritmos Distribuidos

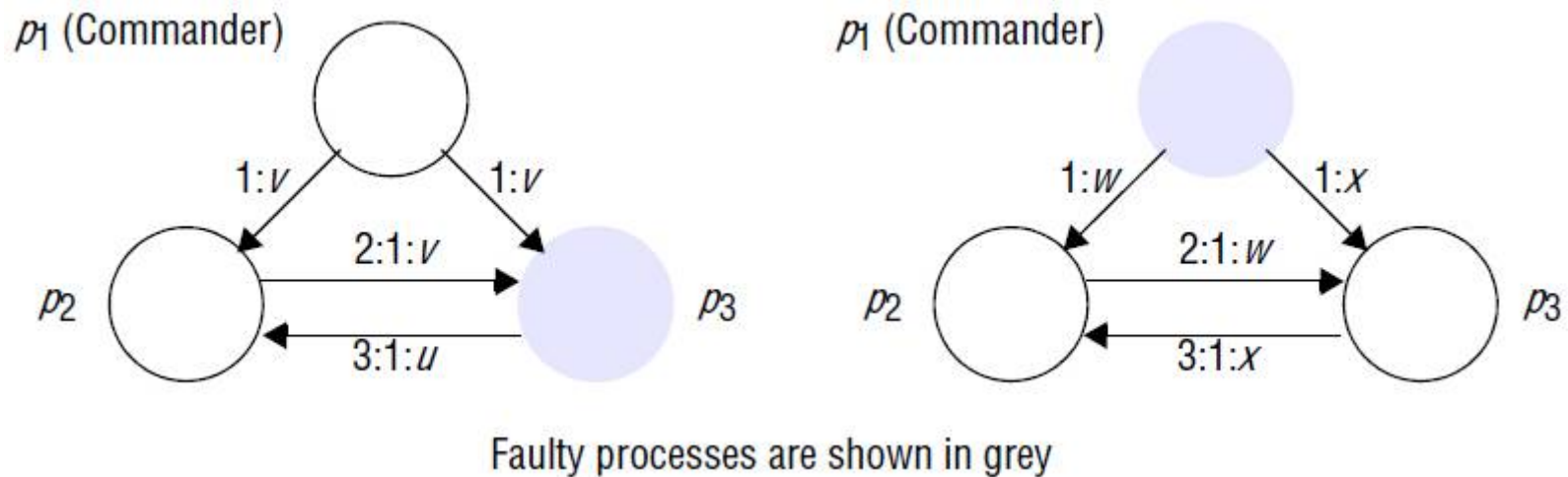
- **Propiedades algoritmo general de Consenso:**
 - **Terminación:**
 - Obvio ya que es sistema es al menos parcialmente **síncrono por *timeouts***
 - **Acuerdo:**
 - Cada proceso llega al **mismo conjunto de valores** cuando termina la **ronda final** y aplica la **misma función**
 - **Integridad:**
 - Variante porque se aplica a los **valores propuestos por todos los procesos** (no sólo por los correctos), y los correctos llegan al mismo conjunto de valores cuando termina la ronda final y aplica la misma función (*idem* anterior)

Algoritmos Distribuidos

- Consenso para **problema de generales Bizantinos**:
 - 3 o más generales deben acordar entre atacar o retirarse. Uno o más de los generales pueden ser "traidores" (defectuosos).
 - El comandante emite la orden y los tenientes deben decidir:
 - **Si comandante traicionero** → propone a uno atacar y a otro retirarse.
 - **Si teniente traicionero** → dice a uno de sus compañeros que la orden del comandante fue atacar y a otro que fue retirarse.
 - Caso especial de consenso porque **sólo un proceso propone el valor inicialmente, fallos arbitrarios de los procesos** (cualquier mensaje y valor, e incluso omisión), y procesos correctos con *timeouts*.
 - **Requisito integridad**: si el comandante es correcto → todos acuerdan el valor.

Algoritmos Distribuidos

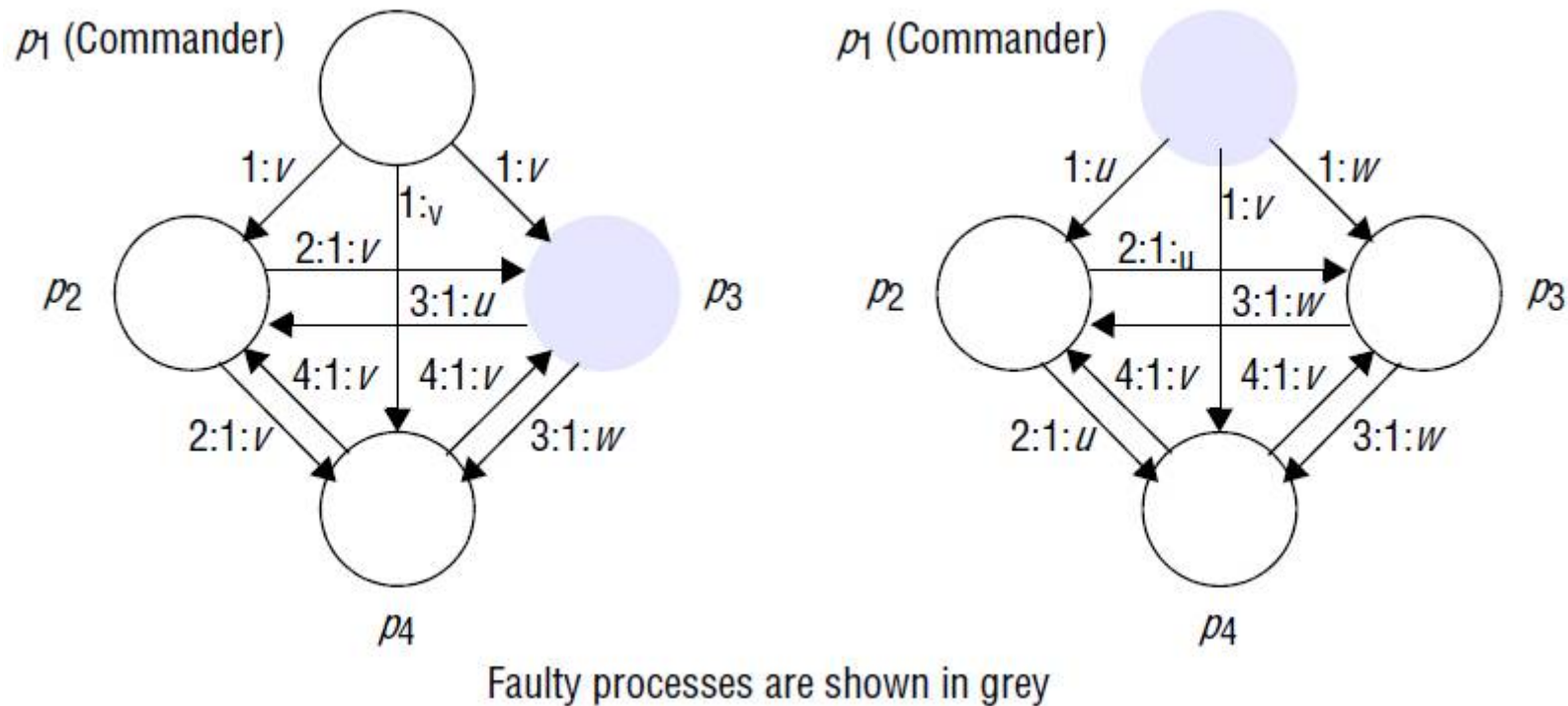
- Consenso para problema de generales bizantinos (*Lamport*):
 - 2 rondas de mensajes:
 - Los valores enviados por el comandante
 - Los valores que los tenientes envían a continuación a los demás tenientes
 - No hay solución para $N = 3$ ó $N \leq 3f$:



[Fuente: *Distributed Systems: Concepts and Design (5th Edition)*; G. Coulouris et al, 2012]

Algoritmos Distribuidos

- Consenso para problema de generales bizantinos (*Lamport*):
 - Solución para $N > 3f$, los procesos aplican la función *majority* al conjunto de valores recibidos, e implica $f + 1$ rondas.



[Fuente: Distributed Systems: Concepts and Design (5th Edition); G. Coulouris et al, 2012]