FACULTAD DE INGENIERÍA

Universidad Nacional de La Pampa

# TOLERANCIA A FALLOS

**Dra. Carolina Salto**

Fac. de Ingeniería – UNLPam

2017

**Sistemas Distribuidos I**

Carrera: Ingeniería en Sistemas

# Conceptos Básicos



Tolerante a fallas ⇔ sistemas fiables ("*dependable*")

Incluye

| Disponibilidad | Confiabilidad | Seguridad | Mantenibilidad |
|---|---|---|---|
| • se define como la propiedad de que un sistema esté disponible para ser usado inmediata-mente | • se refiere a la propiedad de que un sistema se ejecute continuamente (24/7) sin fallas | • se refiere a la situación en la que un sistema falla temporal-mente y no ocurre nada catastrófico | • se refiere a cuan fácil puede ser reparado un sistema que ha fallado |

¿Un sistema altamente disponible es necesariamente altamente confiable? NO

# Conceptos Básicos






- Un **sistema falla** cuando no puede cumplir con su propósito, no satisface a sus usuarios
- Un **error** es parte del estado de un sistema que lleva a una falla del sistema
- La causa de un error es una **falla externa** (fault)

Determinar una "falla externa" es importante, pero no siempre ayuda (cambiar las condiciones del tiempo para prevenir un error, no parece una opción!!!!)

# Conceptos Básicos





- Los sistemas tolerantes a fallas no previenen las fallas, más bien las controlan

**Tolerancia a fallas** implica que el sistema puede proveer sus servicios aún en presencia de fallas

- Las fallas externas, en general, se pueden clasificar en:
  - Transitorias: ocurren una vez y desaparecen
  - Intermitentes: ocurren, cuando se desvanece, ocurren nuevamente y así sucesivamente
  - Permanentes: la falla permanece

# Redundancia



- La técnica más usada para enmascarar las fallas es la **redundancia**
- Hay tres tipos de redundancia:
  - Redundancia de información

    Agregar información adicionales para detectar fallas (código de Hamming, bits de paridad, etc) y permitir la recuperación
  - Redundancia de tiempo

    Si es necesario realizar nuevamente una acción (modelo de transacciones)
  - Redundancia física

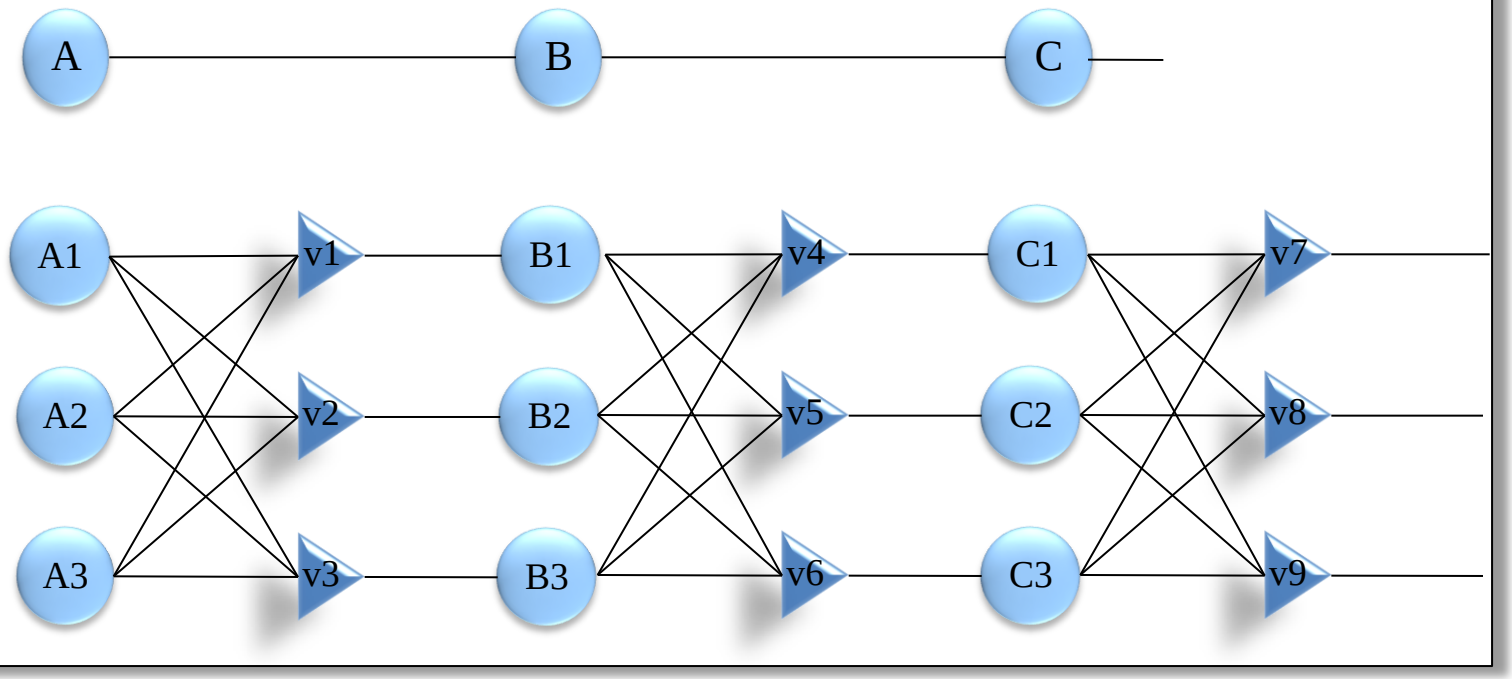
    Equipamiento o procesos extras para sustituir al componente que falla

# Redundancia



- La **redundancia física** es la más usada para proveer tolerancia a fallas: en biología, aviones, deportes y circuitos electrónicos

El modelo de circuitos electrónicos TMR (Tripe Modular Redundancy) es un buen modelo para los sistemas distribuidos



# Recuperación de procesos

# Recuperación de procesos



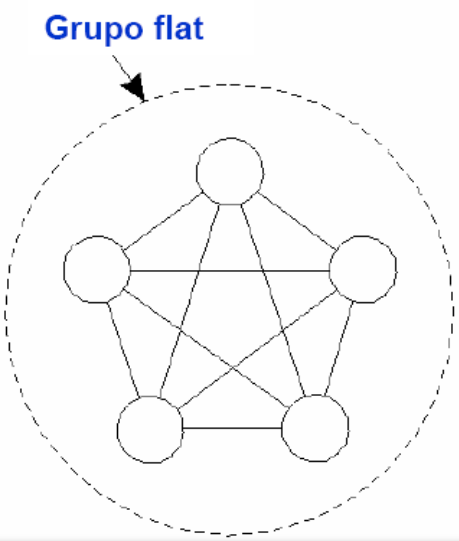
- Enmascarar la falla de procesos por replicación
- Los grupos de procesos y la comunicación en grupo se pueden usar para implementar redundancia
- Organizar procesos idénticos en **grupos**
  - Propósito: permitir a los procesos tratar con una colección de procesos de una manera abstracta
  - Todos los miembros del grupo reciben todos los mensajes
    - Si un miembro falla, otro proceso realizará la tarea
- Los grupos son dinámicos
  - Manejo de grupos y membrecía de grupo

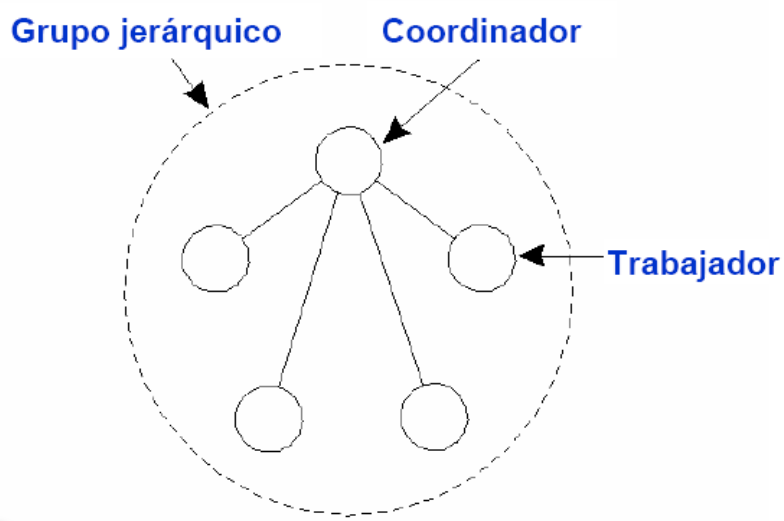
# Recuperación de procesos





## Grupos planos versus Grupos jerárquicos




**Ventajas**

Simétrico

No tiene único punto de falla

**Desventaja**

La toma de decisiones es complicada

**Ventaja**

La toma de decisiones está centralizada

**Desventaja**

Pérdida del coordinador

# Recuperación de procesos





## Membresía de grupo

- Métodos para crear y eliminar grupos, para permitir a los procesos unirse o dejar grupos
- Opciones:
  - Servidor de grupo: recibe todos los requerimientos

    Mantiene una base de datos de todos los grupos y sus miembros

    Método eficiente, simple y fácil de implementar

    Desventaja: único punto de falla
  - Manejo distribuido de grupo

    Posible si está disponible multicast confiable
- Punto de diseño: dejar o unirse a un grupo debe ser síncrono con el envío y recepción de mensajes

# Enmascaramiento de Fallas y Replicación

## Indice



- Reemplazar un único proceso con un grupo de procesos tolerante a fallas
- ¿Cómo se logra la replicación?:
  - Protocolos basados en el primario (protocolo de backup primario)

    Procesos organizados en forma jerárquica: un coordinador primario coordina todas las escrituras. Si el primario cae, los backups ejecutan un algoritmo de elección para elegir el nuevo primario
  - Protocolos de escrituras replicadas

    Procesos organizados en grupos planos: replicación activa vs protocolos basados en quorum

### ¿Cuánta replicación es necesaria?

Se dice que un sistema es ***k tolerante a fallas*** si puede sobrevivir a *k* componentes con fallas y aún reunir sus especificaciones

- Teniendo $k+1$ sin fallas es suficiente
- Si los procesos exhiben fallas bizantinas, es necesario un mínimo de $2k+1$ procesos sin fallas

# Acuerdo en Sistemas con Fallas



- Necesidad de llegar a acuerdos en SD
  - Lider, sincronización, acordar una transacción, etc.
- **Objetivo de los algoritmos de acuerdo distribuidos:** todos los procesos sin fallas alcancen un consenso en un número finito de pasos
  - Procesos perfectos, canales perfectos: situación perfecta
  - Procesos perfectos, canales con fallas: dos-armadas
  - Procesos con fallas, canales perfectos: generales bizantinos
- Suposiciones sobre el sistema subyacente:
  - Sistemas síncronos vs asíncrono
  - Retraso de la comunicación está limitado o no
  - La entrega de mensajes está ordenada o no
  - La trasmisión de mensajes se realiza con unicast o multicast

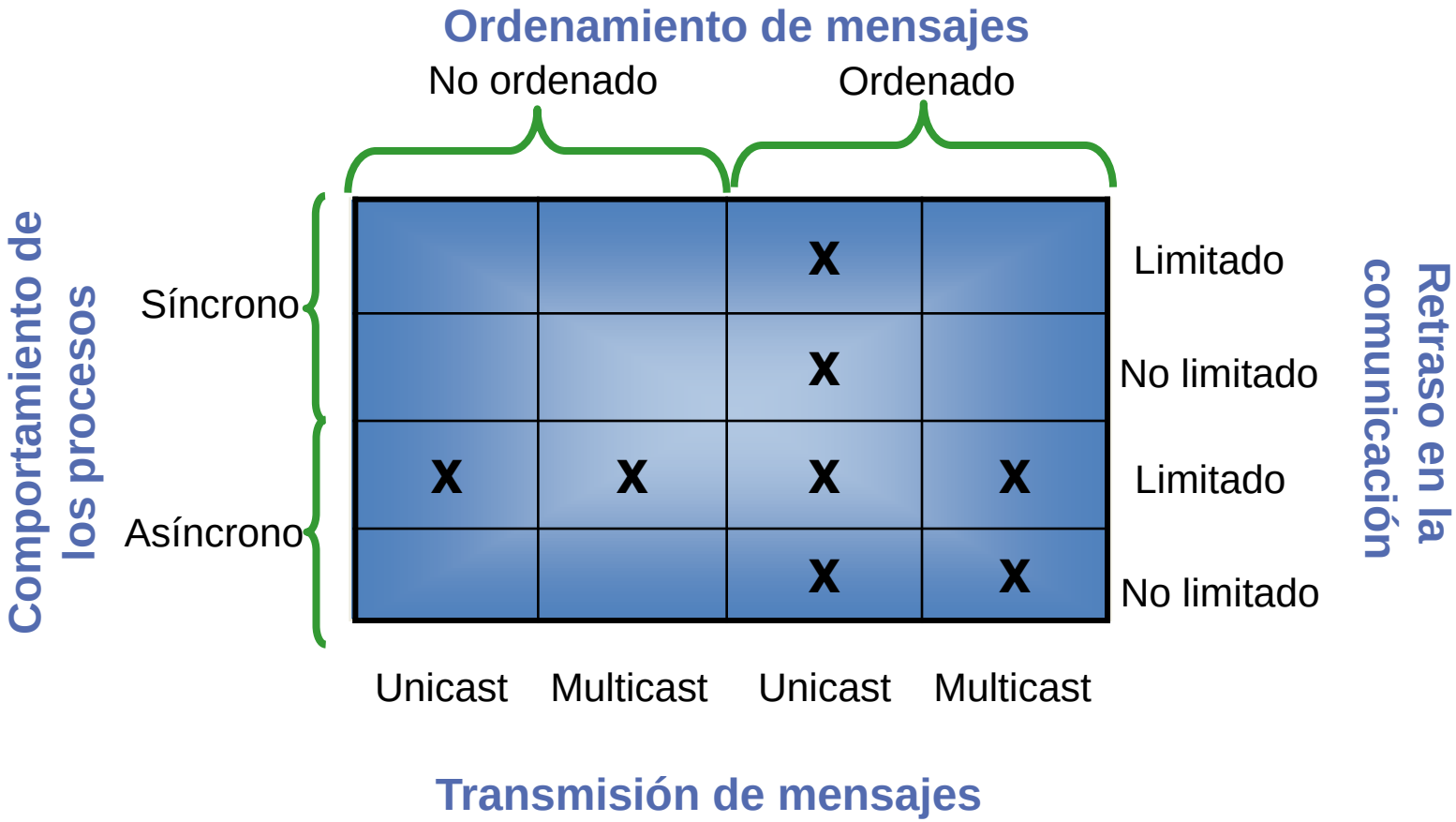
Requieren soluciones distintas

# Acuerdo en Sistemas con Fallas





Circunstancias bajo las cuales se puede alcanzar un acuerdo distribuido cuando hay procesos que fallan

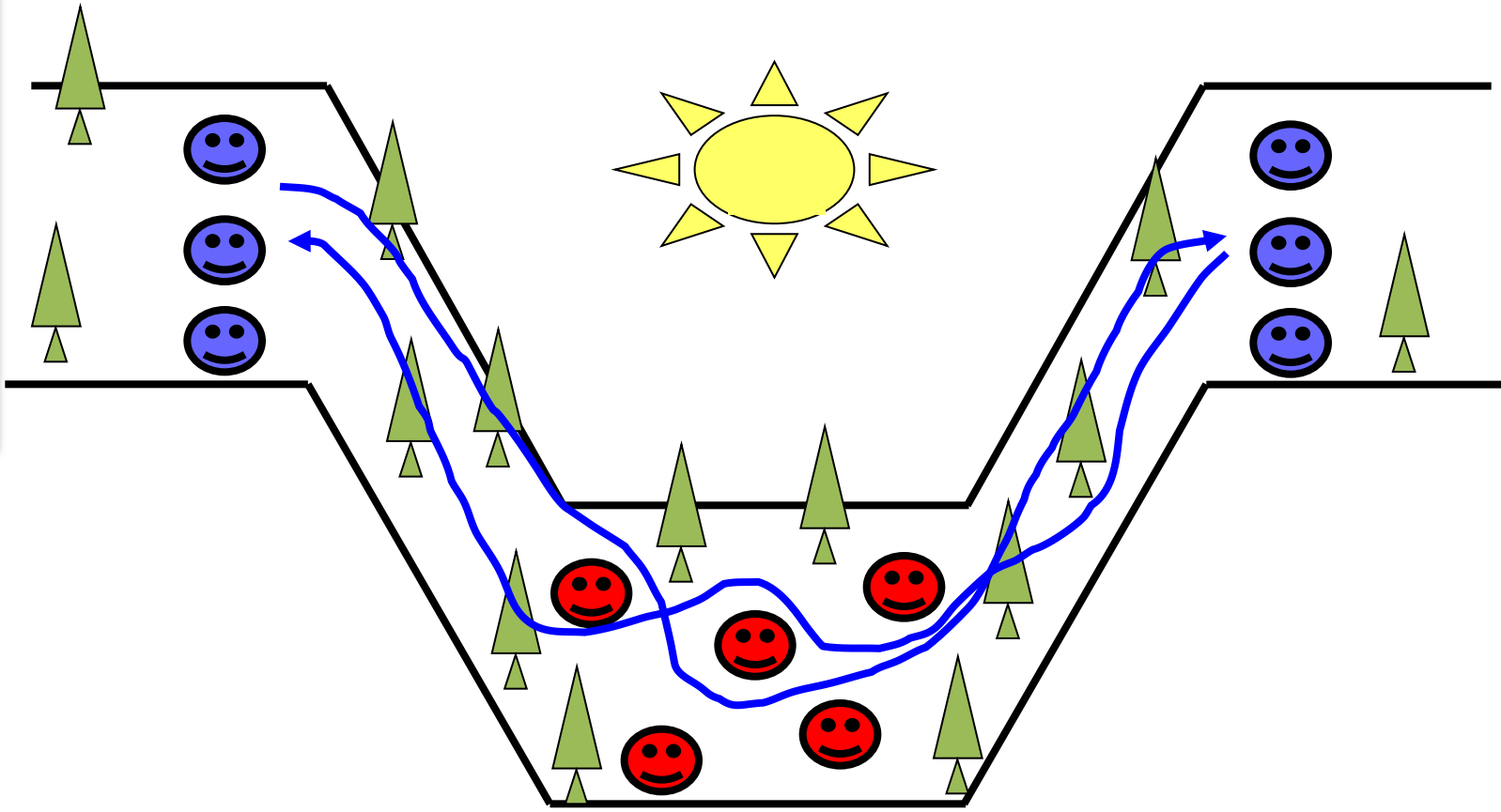


# Acuerdo en Sistemas con Fallas





## Problema de dos armadas



# Consenso imposible





- Acuerdo **imposible** en **SD asíncronos,** aún si sólo un proceso es el que falla

- SD asíncronos:
  - No se puede garantizar la entrega de los mensajes en un tiempo conocido y finito
  - no se puede distinguir un proceso lento de uno caído

# Consenso posible





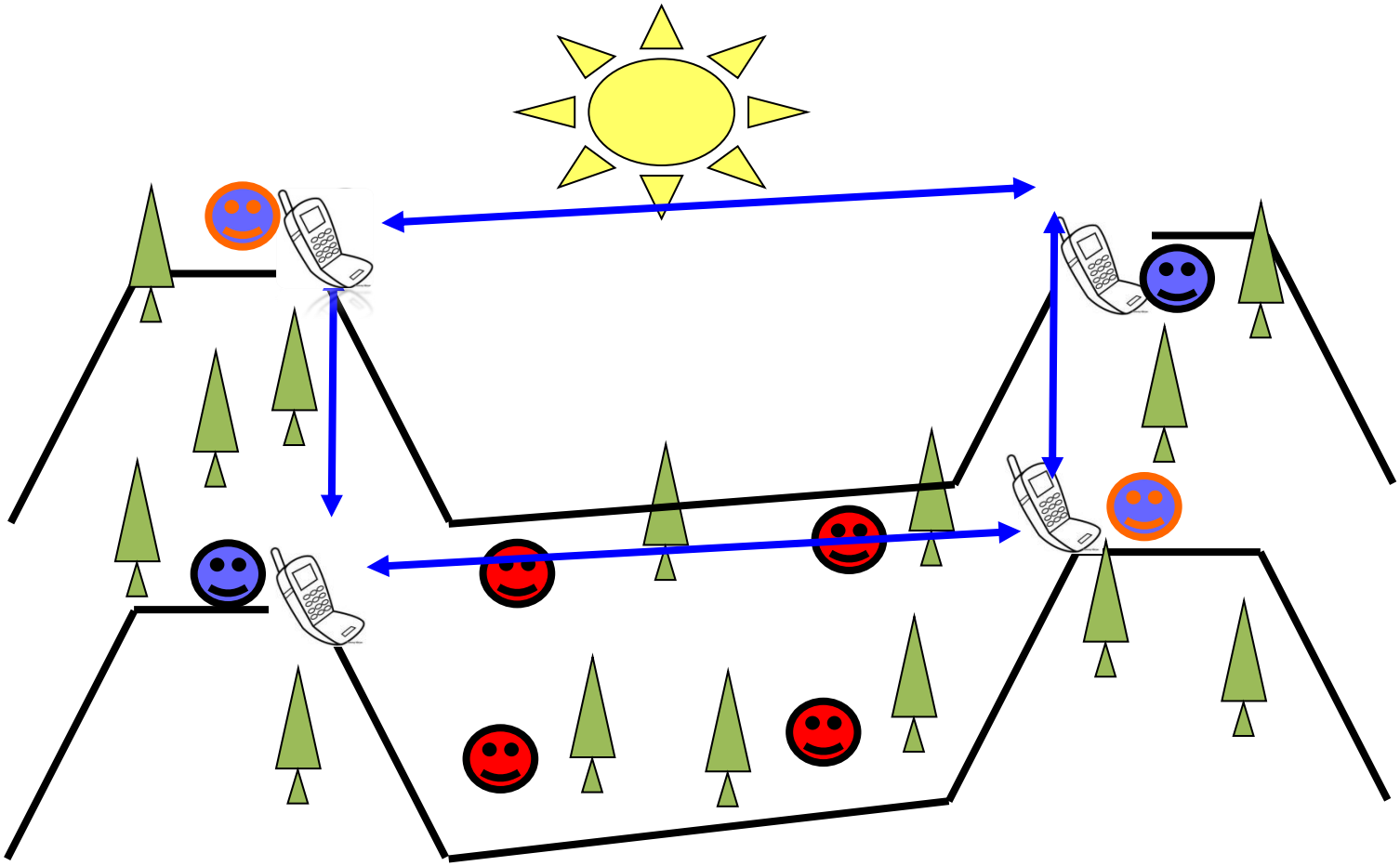
- El acuerdo es **posible** en **SD síncronos**
  - Se puede garantizar la entrega de mensajes en tiempo finito y conocido
  - Problemas de generales bizantinos
- Un SD síncrono puede distinguir un proceso lento de uno que ha caído

# Acuerdo en Sistemas con Fallas

## Indice



## Problema de Generales Bizantinos



# Acuerdo en Sistemas con Fallas





## Problema de Acuerdo Bizantino

*Reaching agreement in the presence of faults*
Autores: Pease, Shostak and Lamport
Journal of the Association for Computing Machinery-1982

- Estudiado por Lamport en 1982
- Se asume:
  - Procesos síncronos
  - Mensajes enviados por unicast
  - Preservación en el orden de los mensajes
  - Comunicación con retraso limitado
- Hay *N* procesos, cada proceso *i* envía un valor $v_i$ al resto

**Objetivo**: cada proceso construye un vector V de largo *N*, tal que si un proceso no falla $V[i]= v_i$. Sino, $V[i]$ es indefinido

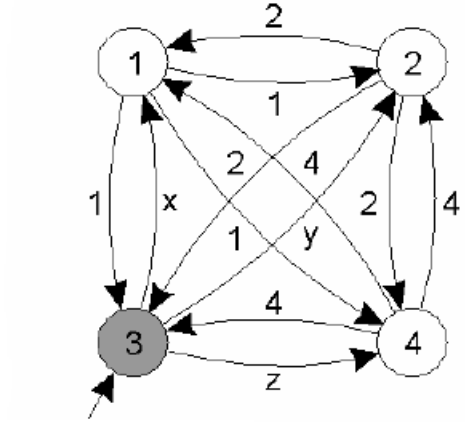
Se asume al menos *k* procesos con fallas

# Acuerdo en Sistemas con Fallas





- El problema de los generales Bizantinos para tres generales leales y un traidor (N=4, k=1 y $v_i = i$ )




Proceso fallado

Los generales anuncian la fortaleza de su tropa ( en unidades de “kilosoldados”)

1 Obt(1, 2, x, 4)
2 Obt(1, 2, y, 4)
3 Obt(1, 2, 3, 4)
4 Obt(1, 2, z, 4)

Los vectores que cada general arma basados en (a)

| 1 Obt | 2 Obt | 4 Obt |
|---|---|---|
| (1, 2, y, 4) | (1, 2, x, 4) | (1, 2, x, 4) |
| (a, b, c, d) | (e, f, g, h) | (1, 2, y, 4) |
| (1, 2, z, 4) | (1, 2, z, 4) | (i, j, k, l) |

Los vectores que cada general recibe en el paso 3

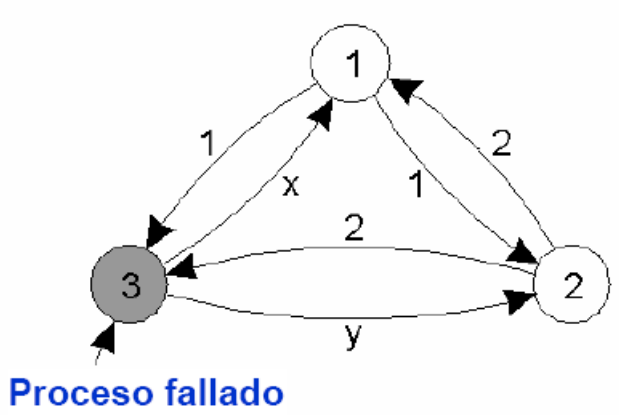
El objetivo del acuerdo bizantino es que el consenso se alcanza sobre el valor de los procesos que no fallan

# Acuerdo en Sistemas con Fallas





- En este ejemplo, ahora con dos generales leales y un traidor




1 Obt (1, 2, x)
2 Obt (1, 2, y)
3 Obt (1, 2, 3)

1 Obt
(1, 2, y)
(a, b, c)

2 Obt
(1, 2, x)
(d, e, f)

¿Se alcanza consenso?

El algoritmo falla en producir acuerdo

# Acuerdo en Sistemas con Fallas



En un sistema con *k* procesos erráticos, el acuerdo se alcanza sólo si están presentes $2k+1$ procesos que funcionan bien, de un total de $3k+1$ (Lamport 82)

- El **acuerdo** es **posible** sólo si **más de las dos terceras parte de los procesos está trabajando correctamente**

- Alcanzar un acuerdo en un sistema distribuido puede ser peor sino se cumplen las condiciones planteadas (garantías sobre los tiempos de entrega de los mensajes)
  - Procesos lentos no se pueden diferenciar de procesos que han caído

# Detección de Fallas



- Pilar de la tolerancia a fallas en un sistema distribuido
- Si se quiere enmascarar fallas, es necesario detectarlas
- Para un grupo de procesos, los miembros no erráticos deben ser capaces de decidir quien es un miembro y quien no
- Mecanismos para detectar procesos con fallas:
  - Enviar mensajes "estas vivo?"
  - Esperar pasivamente mensajes de los distintos procesos

La más usada (pinging)

# Detección de Fallas



- Mecanismos de timeout: usado para chequear si un proceso ha fallado
- Problema
  - Puede ser erróneo creer que un proceso ha fallado porque no ha retornado una respuesta a un mensaje ping

    redes no confiables
- Puntos de diseño a tener en cuenta para un subsistema de detección de fallas:
  - Detección de fallas por gossiping
  - El sistema debe poder distinguir entre procesos que han fallado y fallas en la red
  - Cuando se detecta un proceso fallado, ¿cómo se informa esta situación al resto?

# Comunicación confiable

# Comunicación cliente-servidor confiable





- Tolerancia a fallas se concentra en **procesos que fallan**, pero también se deben considerar **fallas en la comunicación**
- Los canales de comunicación pueden exhibir:
  - Crash
  - Omisión
  - Timing
  - Fallas arbitrarias

Mensajes duplicados

Fallas más importantes que se tratan de enmascarar

En una conexión punto a punto, el uso de protocolo confiable (TCP )

- Enmascara fallas por omisión
  - pérdidad de mensajes → ack y retransmisión
- Fallas por crash no son enmascaradas
  - conección TCP abruptamente cortada

# Semántica RPC en la presencia de fallas





Pueden ocurrir cinco clases de errores en un sistema RPC:

1. El cliente no puede ubicar al servidor
2. Se pierde el requerimiento del cliente al servidor
3. El servidor se cae después de recibir un requerimiento
4. Se pierde la respuesta del servidor al cliente
5. El cliente cae luego de enviar un requerimiento

# Semántica RPC en la presencia de fallas



## Cliente no puede localizar al servidor

- El servidor puede estar caído
- El cliente puede estar usando una versión antigua del servidor
  - La versión ayuda a detectar accesos a copias obsoletas
  - Cómo indicar el error al cliente?
    - Devolviendo un código de error (-1)
      - No es transparente
        - Ejemplo: sumar(a,b)
    - Elevando una excepción
      - Necesita un lenguaje que soporte excepciones
      - No es transparente

# Semántica RPC en la presencia de fallas



- **Pérdida de mensajes de requerimiento del cliente**
  - Es la más fácil de tratar
  - Se activa una alarma (*timeout*) después de enviar el mensaje
  - Si no se recibe una respuesta se retransmite
  - Depende del protocolo de comunicación subyacente

# Semántica RPC en la presencia de fallas



## Fallos en los servidores

- El servidor no ha llegado a ejecutar la operación
  - Se podría retransmitir
- El servidor ha llegado a ejecutar la operación
- El cliente no puede distinguir las dos situaciones
- ¿Qué hacer?
  - No garantizar nada
  - Semántica al menos una vez
    - Reintentar y garantizar que la RPC se realiza al menos una vez
  - Semántica a lo sumo una vez
    - No reintentar, puede que no se realice ni una sola vez
  - ***Semántica de exactamente una***
    - Sería lo deseable.

# Semántica RPC en la presencia de fallas





- ***Semántica tal-vez***
  - Procedimiento remoto puede ejecutarse una vez o ninguna
  - Cliente puede recibir una respuesta o ninguna
  - Funcionamiento:
    - Cliente envía petición y queda a la espera un tiempo
    - Si no llega respuesta dentro del tiempo de espera, continúa su ejecución
    - Cliente no tiene realimentación en caso de fallo (no sabe que pasó)
  - Sólo admisible en aplicaciones donde se tolere la pérdida de peticiones y la recepción de respuestas con retaso (fuera de orden)

# Semántica RPC en la presencia de fallas





- ***Semántica al-menos-una-vez***
  - Procedimiento remoto se ejecuta una o más veces
  - Cliente puede recibir una o más respuestas
  - Funcionamiento:
    - Cliente envía petición y queda a la espera un tiempo
    - Si no llega respuesta o ACK dentro del tiempo de espera, repite la petición
    - Servidor no filtra peticiones duplicadas -> procedim. remoto puede ejecutarse repetidas veces
    - Cliente puede recibir varias respuestas
  - Sólo es aplicable cuando se usan exclusivamente operaciones idempotentes (repetibles)
    - Una operación es idempotente si se puede ejecutar varias veces resultando el mismo efecto que si se hubiera ejecutado sólo una
      - Nota: en ocasiones una operación no idempotente puede implementarse como una secuencia de operaciones idempotentes
  - Admisible en aplicaciones donde se tolere que se puedan repetir invocaciones sin afectar a su funcionamiento

# Semántica RPC en la presencia de fallas





- ***Semántica como-máximo-una-vez***
  - El procedimiento remoto se ejecuta exactamente una vez o no llega a ejecutarse ninguna
  - Cliente recibe una respuesta o una indicación de que no se ha ejecutado el procedimiento remoto
  - Funcionamiento:
    - Cliente envía petición y queda a la espera un tiempo
    - Si no llega respuesta o ACK dentro del tiempo de espera, repite la petición
    - Servidor filtra las peticiones duplicadas y guarda historial con las respuestas enviadas (servidor con memoria) -> procedim. remoto sólo se ejecuta una vez
    - Cliente sólo recibe una respuesta si la petición llegó y se ejecutó el procedim., si no recibe informe del error
  - Semántica más próxima a la de llamadas a procedimiento locales (semántica sólo una vez) en presencia de fallos

# Semántica RPC en la presencia de fallas





## Resumen

| Semántica | Funcionamiento |
|---|---|
| **tal-vez** | ■ cliente no retransmite sus peticiones (no usa ACK)<br>■ servidor no filtra peticiones duplicadas |
| **al-menos-una** | ■ cliente retransmite sus peticiones (usa ACK + temporizador)<br>■ servidor no filtra peticiones duplicadas<br>■ ante peticiones repetidas, servidor repite ejecución |
| **como-máximo-una** | ■ cliente reintenta retransmitir peticiones (usa ACK + temporizador)<br>■ servidor filtra peticiones duplicadas<br>■ ante peticiones repetidas, servidor retransmite las respuestas pasadas |

# Semántica RPC en la presencia de fallas





## Pérdidas de mensajes de respuesta

- Más difícil de tratar
- Se pueden emplear alarmas y retransmisiones, pero:
  - ¿Se perdió la petición?
  - ¿Se perdió la respuesta?
  - ¿El servidor va lento?
- Algunas operaciones pueden repetirse sin problemas (operaciones idempotentes)
  - Una transferencia bancaria no es idempotente
- Solución con operaciones no idempotentes es descartar peticiones ya ejecutadas
  - Un número de secuencia en el cliente
  - Un campo en el mensaje que indique si es un pedido original o una retransmisión

# Semántica RPC en la presencia de fallas





## Fallos en los clientes

- La computación está activa pero ningún cliente espera los resultados (computación huérfana)
  - Gasto de ciclos de CPU
  - Lock de archivos
  - Apropiación de recursos valiosos
  - Si el cliente re arranca y ejecuta de nuevo la RPC, se pueden crear confusiones
- Soluciones:
  - Exterminación de computación huérfana (log entry por RCP)
  - Reencarnación (dividir en épocas)
  - Reencarnación gentil
  - Expiración (timeout por RPC)

# Comunicación en Grupo Confiable





- Los niveles de transporte raramente proveen comunicación a un grupo de procesos

Establecer comunicaciones punto a punto con cada uno de los procesos con los que se desea comunicar

- Organización no eficiente (mucho gasto de ancho de banda)

Pero si la cantidad de procesos es pequeña, lograr confiabilidad de esta forma es una solución posible

**Multicast confiable:** un mensaje que es enviado a un grupo de procesos es entregado a cada miembro del grupo

- ¿Qué pasa si durante la comunicación se une un proceso?, ¿deberá recibir el mensaje?
- ¿Qué ocurre si un proceso cae durante la comunicación?

# Multicast confiable



Procesos que fallan **vs** procesos correctos

Multicast es considerado confiable cuando se garantiza que todos los miembros del grupo que no fallan reciben el mensaje
**Consideraciones**: alcanzar acuerdo sobre conformación de grupo y restricciones de ordenamiento

Multicast es considerado confiable si todos los mensajes son entregados a cada miembro del grupo
**Consideraciones**: los procesos no se unen ni salen del grupo mientras se realiza una comunicación
En algunos casos no hay consideraciones de ordenamiento de mensajes

**Multicast atómico**

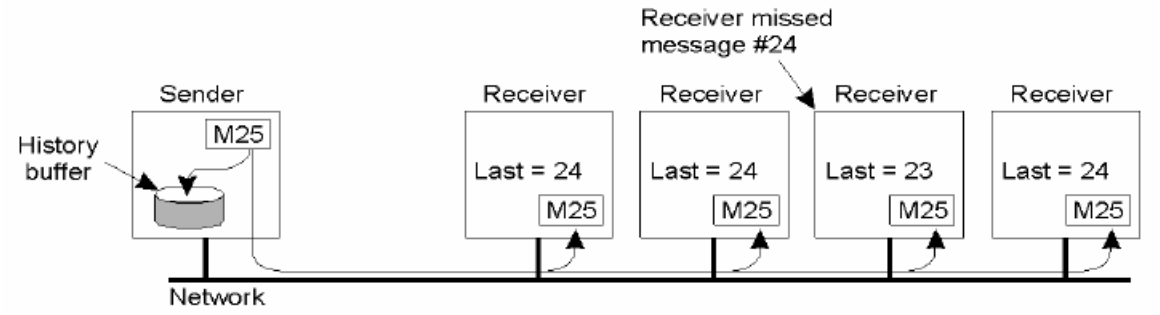
**Multicast confiable débil**

# Multicast confiable débil

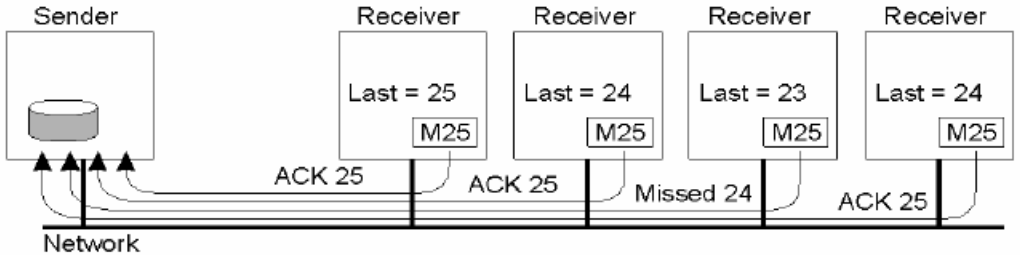




- Relativamente fácil de implementar (#receptores limitado)
- Consideraciones:
  - El sistema subyacente provee sólo multicast no confiable
  - Mensajes recibidos en el orden en que son enviados
- El proceso emisor asigna un nro. de secuencia a cada mensaje




Transmisión del mensaje




Reportando feedback

# Multicast confiable débil



## Mejoras

- Reducir la cantidad de mensajes → ack incluídos en otros mensajes
- Retrasmisión se puede realizar usando comunicación punto a punto

## Desventajas:

- No soporta gran cantidad de receptores

  Si hay *N* receptores, el emisor deberá recibir *N* ack (implosión feedback)

  Solución: no enviar ack por la recepción de un mensaje, sólo informar en el caso de que se pierda un mensaje (ack negativos)

  - El emisor deberá mantener los mensajes en su buffer de historia

# Multicast confiable





**Soluciones escalables para multicast confiable**: reducir la cantidad de mensajes feedback que deben ser retornados al emisor

Modelo Popular

**Supresión Feedback**

opciones

- **Opción no jerárquica**
  - Multicast confiable escalable
- **Opción jerárquica**
  - Control de feedback jerárquico

# Multicast confiable





## Multicasting confiable escalable (SRM)

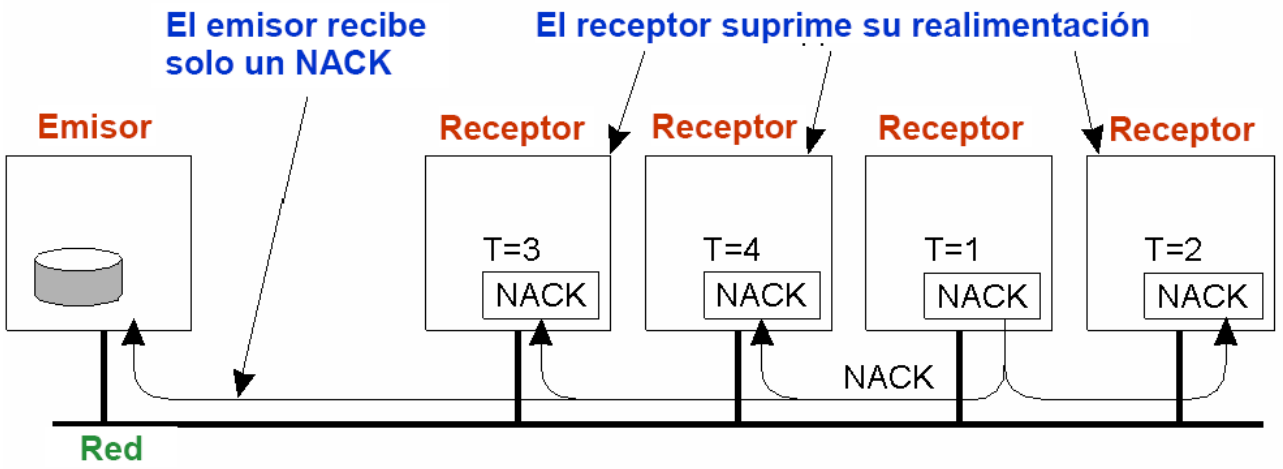
Los receptores no envian ack

Reportan sólo la pérdida de un mensaje (ack negativos)

La aplicación es la encargada de detectar pérdida de mensajes

Ante la pérdida de un mensaje, el receptor multicast su feeedback al resto del grupo en un determinado delay random

Si durante ese tiempo, otro proceso generó una retransmisión, el proceso suprime su feedback



# Multicast confiable



## Multicasting confiable escalable (SRM) (cont.)

### Ventajas

- Escala razonablemente bien
- Usado para aplicaciones de Internet colaborativas
  - Pizarras (whiteboard) compartidas

### Problemas

- Planificación de mensajes feedback en cada receptor

  Setear los timers (T) en un grupo de procesos no es fácil
- Interrumpe a aquellos procesos que han recibido correctamente el mensaje
  - Solución: agrupar los receptores que no recibieron el mensaje

### Mejoras a la escalabilidad

- Los receptores asistan en la recuperación local

# Multicast confiable





**Soluciones escalables para multicast confiable**: reducir la cantidad de mensajes feedback que deben ser retornados al emisor

Modelo Popular

**Supresión Feedback**

opciones

- **Opción no jerárquica**
  - Multicast confiable escalable
- **Opción jerárquica**
  - Control de feedback jerárquico

# Multicast confiable



## Control de feedback jerárquico

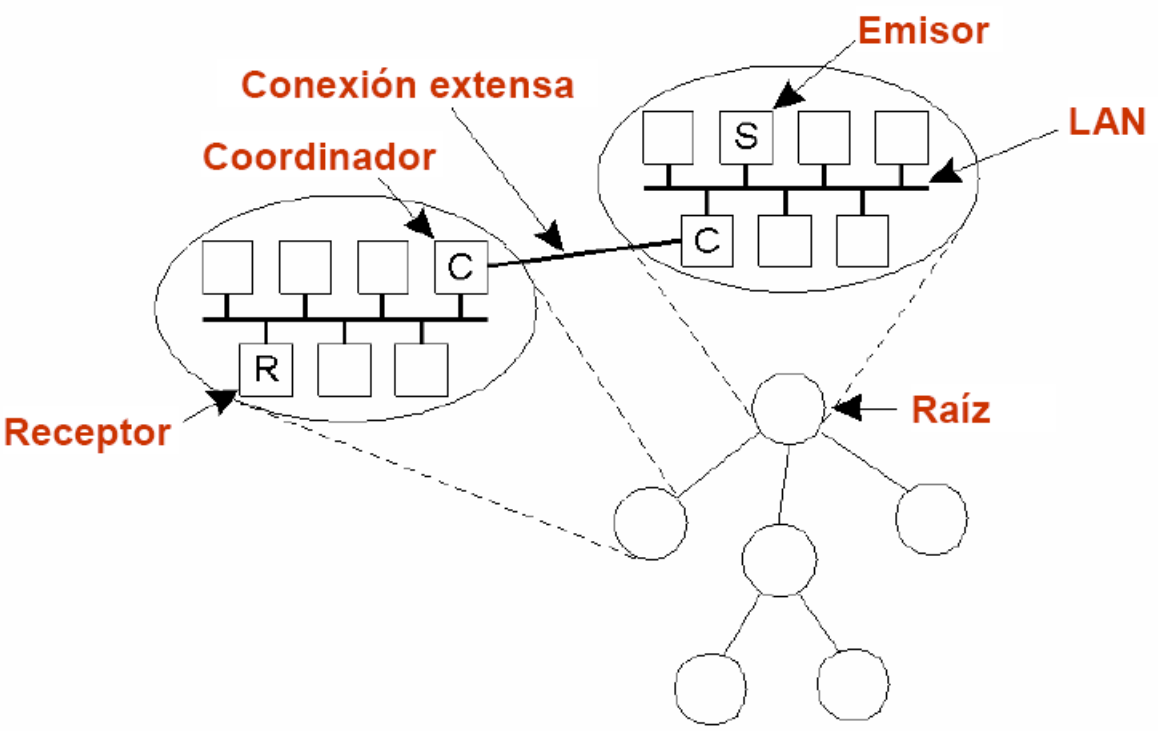
- Supongamos un emisor que necesita enviar un mensaje multicast a un gran grupo de receptores
- El grupo de receptores se divide en subgrupos, los cuales se organizan en un árbol
- Raíz del árbol: subgrupo que contiene al emisor
- Dentro de cada subgrupo, los mensajes se envían usando algún esquema de multicast confiable
- Cada subgrupo tiene un coordinador:
  - Gestiona los requerimientos de retransmisión dentro del subgrupo
  - Tiene un buffer de historia
  - En esquemas de ack, el coordinador local envía un ack a su padre si ha recibido el mensaje

# Multicast confiable





## Control de feedback jerárquico (cont)



# Multicast confiable





- **Control de feedback jerárquico (cont)**
  - **Principal problema:**
    - Construcción del árbol (construcción dinámica)

**CONCLUSIÓN**

construir esquemas de multicast confiable que escalen a una gran cantidad de receptores, **NO es tarea fácil**

# Multicast atómico



- Multicast confiable con procesos que fallan
  - Garantizar que el mensaje es entregado a todos los procesos o a ninguno de ellos
  - Ordenamiento de mensajes en todos los procesos

- Ej: base de datos replicada construida como una aplicación sobre un sistema distribuido

1. Ofrece facilidades de multicast confiable (grupo de procesos)
   Protocolo de replicación activa
2. Multicast atómico

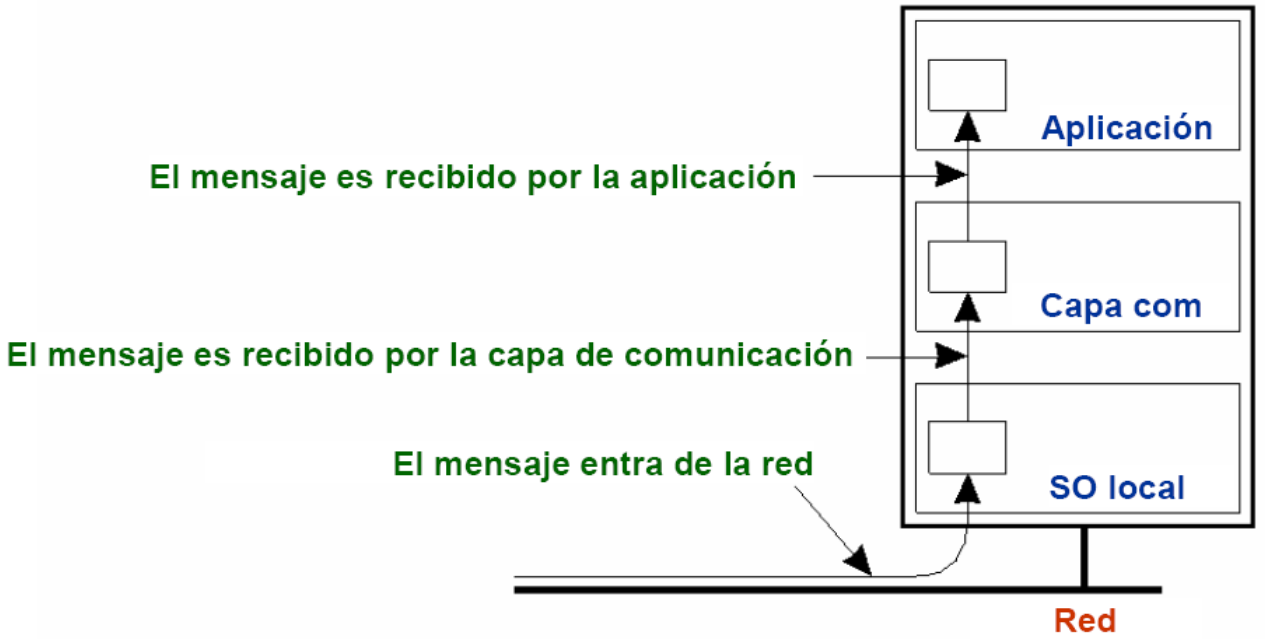
Supongamos una serie de actualizaciones y una de las réplicas cae

# Multicast atómico





**Multicast confiable en la presencia de fallas** se puede definir en término de grupo de procesos y cambios en la membresía de esos grupos



# Multicast atómico





- La idea de multicast atómico es que un mensaje *m* está únicamente asociado a una **lista de procesos**

**Vista de grupo**

- **Todos los procesos dentro del grupo** tienen la misma vista G

**Cambio de vista**

- El cambio en los integrantes del grupo es anunciado a todos los miembros (mensaje *vc*)

Puede haber dos mensajes *m* y *vc* (en ese orden), *m* es entregado antes que *vc* a todos los procesos o no es entregado a ninguno

comparable a multicast totalmente ordenado

# Multicast atómico



- Si *m* no es entregado, ¿podemos hablar de un protocolo de multicast confiable?
  - Caso en el que la entrega de *m* puede fallar:
    - Cuando el cambio en la membresía del grupo se debe a que el emisor de *m* falla
      - Todos o ninguno los miembros de G se enteran de la situación
      - *m* puede ser ignorado por todos los miembros (el emisor falla antes de enviar a *m*)

*Forma fuerte de multicast confiable:* garantiza que el mensaje multicast a la vista de grupo G es entregado a todos los procesos que no fallan en G

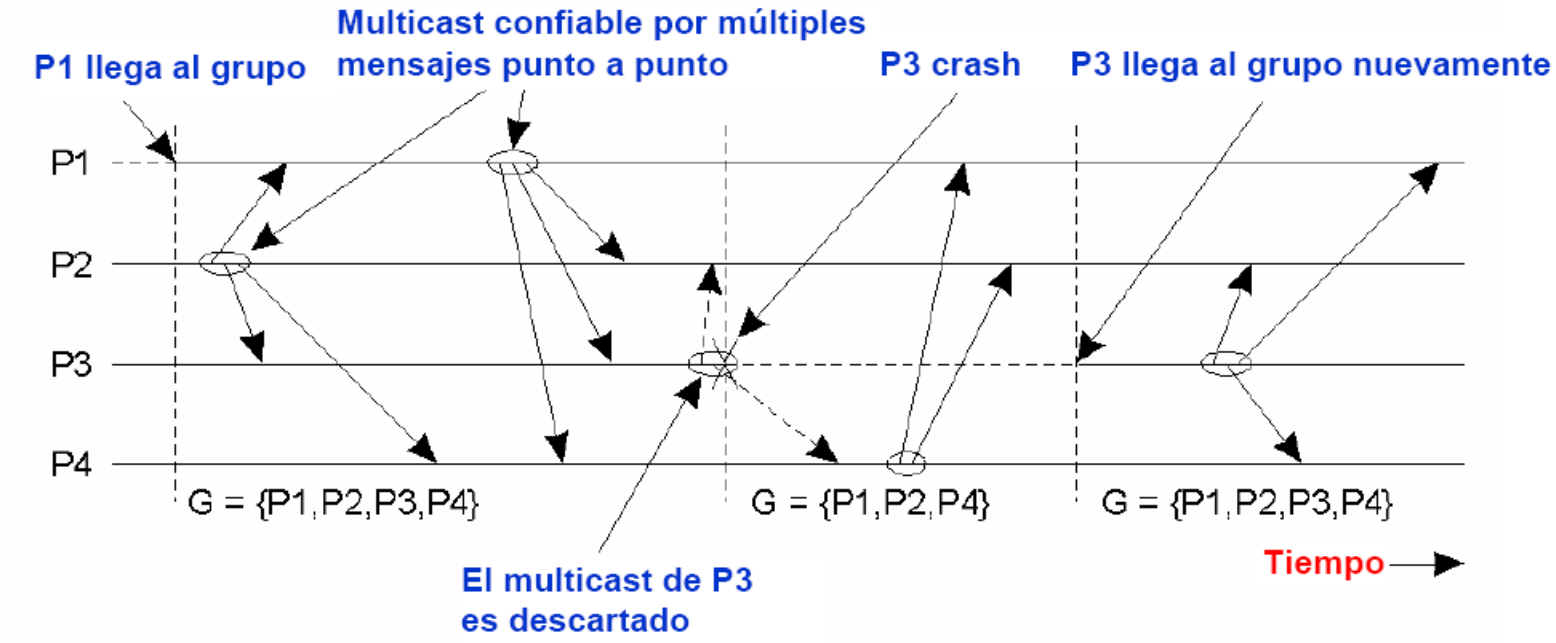
- También denominado **multicast síncrono virtual**

# Multicast atómico





## Multicast Síncrono Virtual




Todos los multicast se realizan entre cambios de vista

- Actúan como barriers
- No son fáciles de implementar

# Multicast atómico





- **Ordenamiento de mensajes**
  - Multicast sin orden
  - Multicast ordenados FIFO
  - Multicast causalmente ordenados
  - Multicast totalmente ordenados

# Multicast atómico





## 1. Multicast sin orden

- No se garantiza el orden en el cual los mensajes recibidos se entregan a los diferentes procesos

| Proceso P1 | Proceso P2 | Proceso P3 |
|---|---|---|
| send m1 | receive m1 | receive m2 |
| send m2 | receive m2 | receive m1 |

Tres procesos comunicándose en el mismo grupo. El orden de los eventos por proceso es mostrado a lo largo del eje vertical

# Multicast atómico





## 2. Multicast ordenados FIFO

- El nivel de comunicación está forzado a entregar los mensajes de entrada del mismo proceso en el mismo orden en el cual fueron enviados

| Proceso P1 | Proceso P2 | Proceso P3 | Proceso P4 |
|---|---|---|---|
| sends m1 | receives m1 | receives m3 | sends m3 |
| sends m2 | receives m3 | receives m1 | sends m4 |
| | receives m2 | receives m2 | |
| | receives m4 | receives m4 | |

Cuatro procesos en el mismo grupo con dos diferentes emisores y un posible orden de recepción de mensajes bajo un multicast con ordenamiento

# Multicast atómico



3. Multicast causalmente ordenado confiable
   - Se preserva la causalidad entre mensajes
   - Implementación?

4. Multicast totalmente ordenado
   - Todos los mensajes son entregados en el mismo orden a todos los miembros del grupos
   - Sin importar si los mensajes están ordenados FIFO, causalmente o no estén ordenados

**Multicast síncrono virtual confiable + entrega totalmente ordenada de mensajes** obtenemos **multicast atómico**

# Multicast atómico





- Seis diferentes versiones de un multicasting confiable virtualmente sincrónico

| Multicast | Orden de mensajes básico | Recepción totalmente ordenada? |
|---|---|---|
| **Multicast confiable** | Ninguno | No |
| **Multicast FIFO** | Recepción FIFO-ordenada | No |
| **Multicast causal** | Recepción causalmente ordenada | No |
| **Multicast atómico** | Ninguna | Si |
| **Multicast atómico FIFO** | Recepción FIFO-ordenada | Si |
| **Multicast atómico causal** | Recepción causalmente ordenada | Si |

# Comunicación en Grupo Confiable



- **Implementación de un multicast confiable síncrono virtual**
- Sistema Isis
  - Sistema distribuido tolerante a fallas (1991)

- Usa **comunicación punto a punto** confiable de la red subyacente (TCP)
  - No hay garantías que todos los miembros reciban *m*
- Asume que el **nivel de comunicación recibe** los mensajes de un proceso **en el orden en el que fueron enviados**
  - Uso de conexiones punto a punto TCP

- **Principal problema a resolver:** garantizar que todos los mensajes enviados a la vista G se entreguen a todos los procesos sin fallas de G antes que se realice el cambio en la membresía del grupo

# Comunicación en Grupo Confiable



- Un proceso puede fallar antes de enviar *m* a todos los procesos en G. Esos procesos pueden obtener *m* desde algún lado
- Solución: cada proceso en G mantiene *m* hasta asegurar que ha sido entregado a todos los miembros
  - Mensaje estable
  - Para asegurar estabilidad, se elige un proceso en G y se le pide que envíe *m* al resto de los procesos
- Supongamos que la vista es $G_i$ y se realiza un cambio de vista a $G_{i+1}$
  - P se entera del cambio de vista al recibir el mensaje *vc*
  - *vc* puede provenir de un proceso que entra o deja el grupo, o de un proceso que ha detectado que otro proceso en $G_i$ ha fallado

# Comunicación en Grupo Confiable

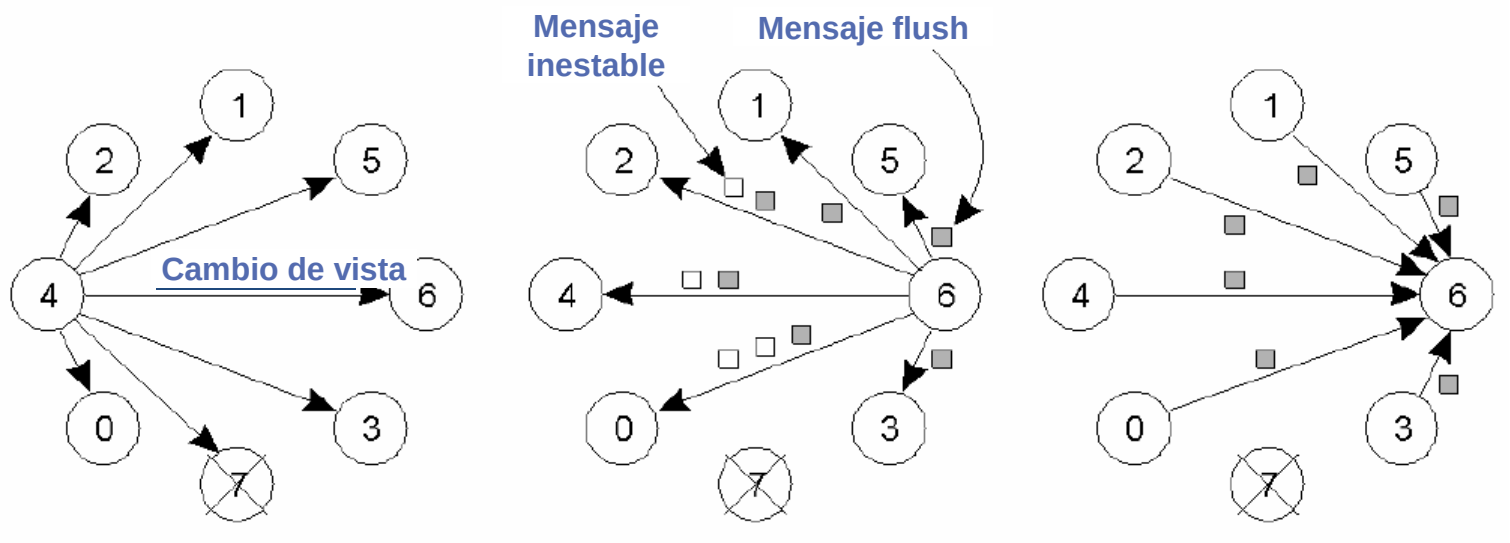




- Cuando P recibe el mensaje *vc* para $G_{i+1}$, envía una copia de todos los mensajes inestables y los marca como estables
  - Seleccionar un único coordinador para enviar mensajes inestables
- P indica que no tiene más mensajes inestables enviando un mensaje flush a todos en $G_{i+1}$
- Cuando P ha recibido un mensaje flush para $G_{i+1}$ del resto de los procesos, instala la nueva vista
- Cuando Q recibe *m* que fue enviado en $G_i$, y Q cree que $G_i$ es la vista actual, entrega *m* respetando las restricciones de ordenamiento.
- Si Q ha recibido *m* simplemente lo descarta

# Comunicación en Grupo Confiable






- El proceso 4 nota que el proceso 7 ha caído, envía un cambio de vista
- El proceso 6 emite todos sus mensajes inestables, seguidos de un mensaje flush
- El proceso 6 instala la nueva vista cuando ha recibido un mensaje flush de todos los demás

# Acuerdo distribuido

# Acuerdo distribuido



- Problema que involucra que todos los procesos de un grupo realicen determinada operación, o que ninguno la realice
  - Ejemplo: multicast atómico (la operación es la entrega de un mensaje)
- Existencia de un coordinador y varios participantes
- Esquemas:
  - Protocolo de commit de una fase
  - Protocolo de commit de dos fases
  - Protocolo de commit de tres fases

# Acuerdo distribuido (2PC)

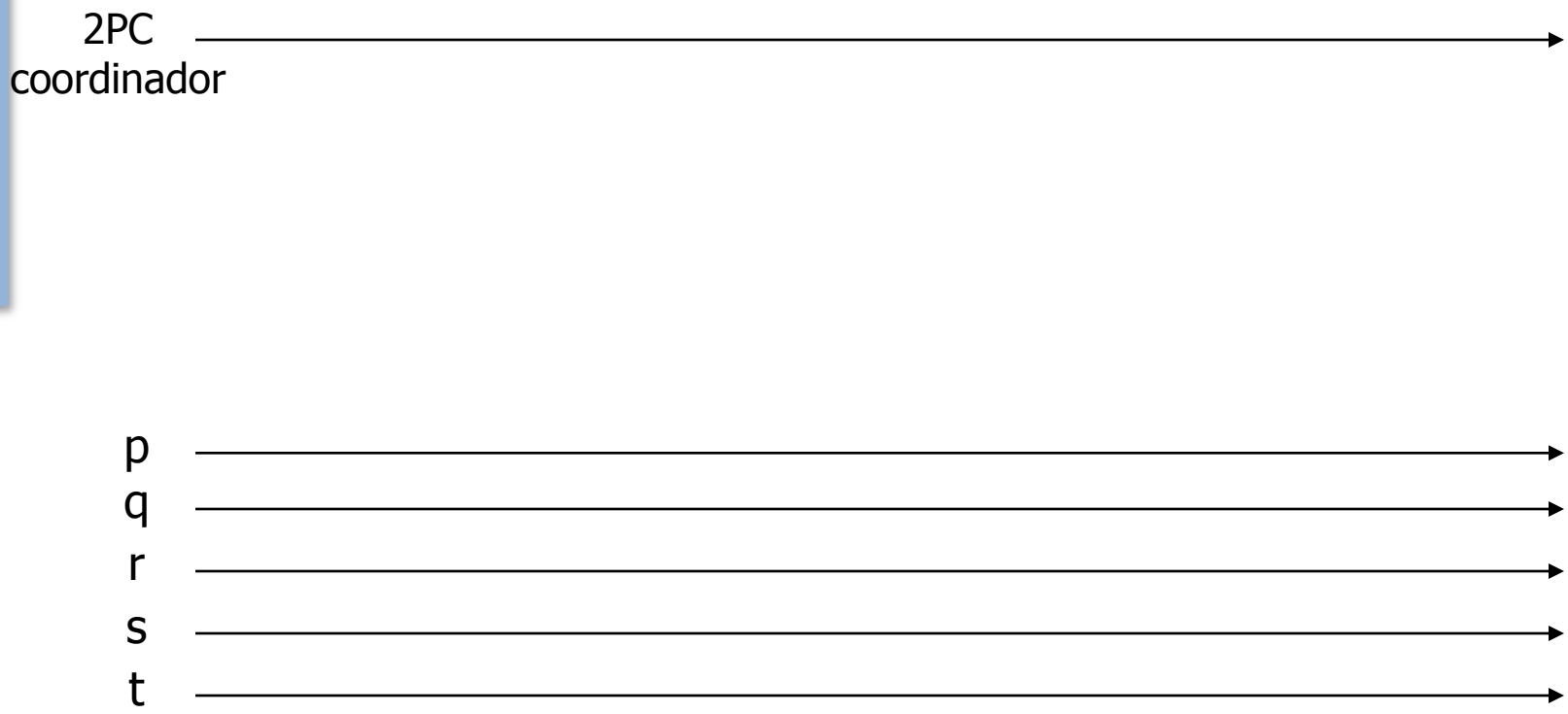


- Propuesto por Gray en 1978
- **Escenario**: transacción distribuida involucrando la participación de una cierta cantidad de procesos, cada uno ejecutando en una máquina distinta
- **Fases:**
  - **Fase 1: Votación**
    1. El coordinador envía VOTE_REQUEST
    2. Cada participantes envía:
       - VOTE_COMMIT si está preparado para realizar localmente su parte de la transacción, o
       - VOTE_ABBORT
  - **Fase 2: Decisión**
    3. Coordinador recolecta todos los votos
       - Si todos los participantes han votado para acordar la transacción, el coordinador envía GLOBAL_COMMIT a todos los participantes
       - Si un participante ha votado para abortar la transacción, el coordinador también decide abortar y multicast GLOBAL_ABBORT
    4. Cada participante queda a la espera del mensaje del coordinador

# Acuerdo distribuido (2PC)



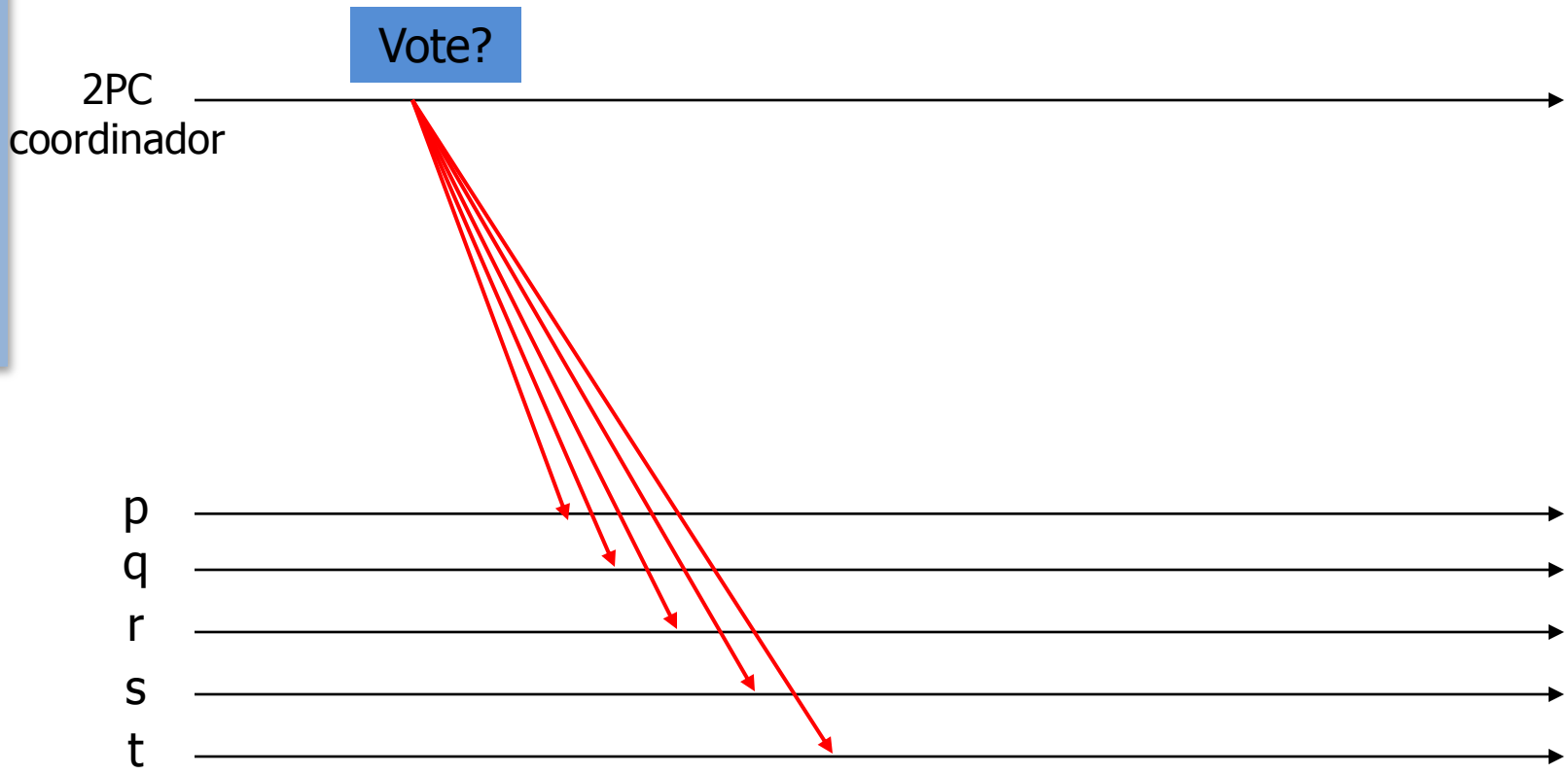




# Acuerdo distribuido (2PC)



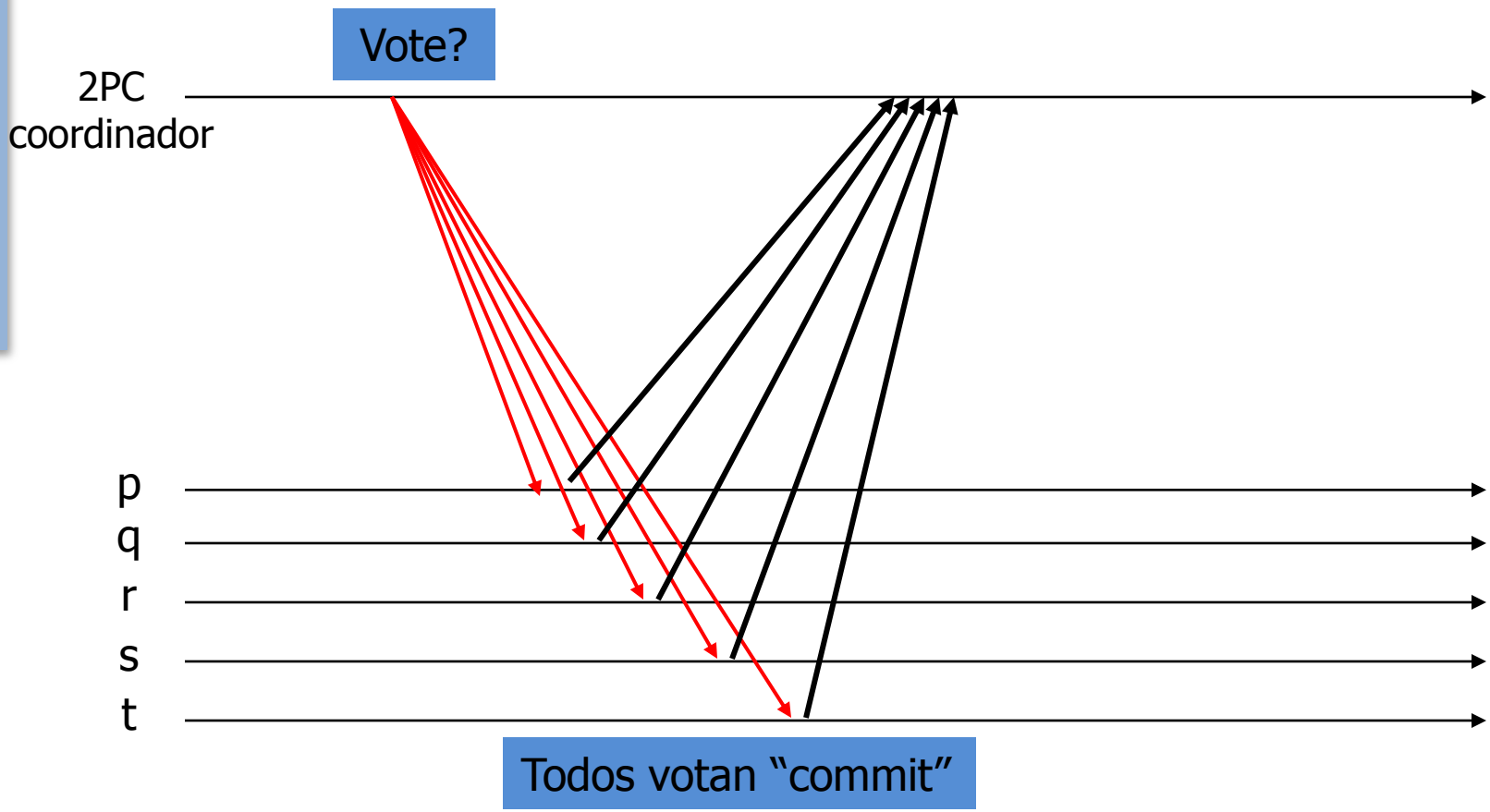




# Acuerdo distribuido (2PC)



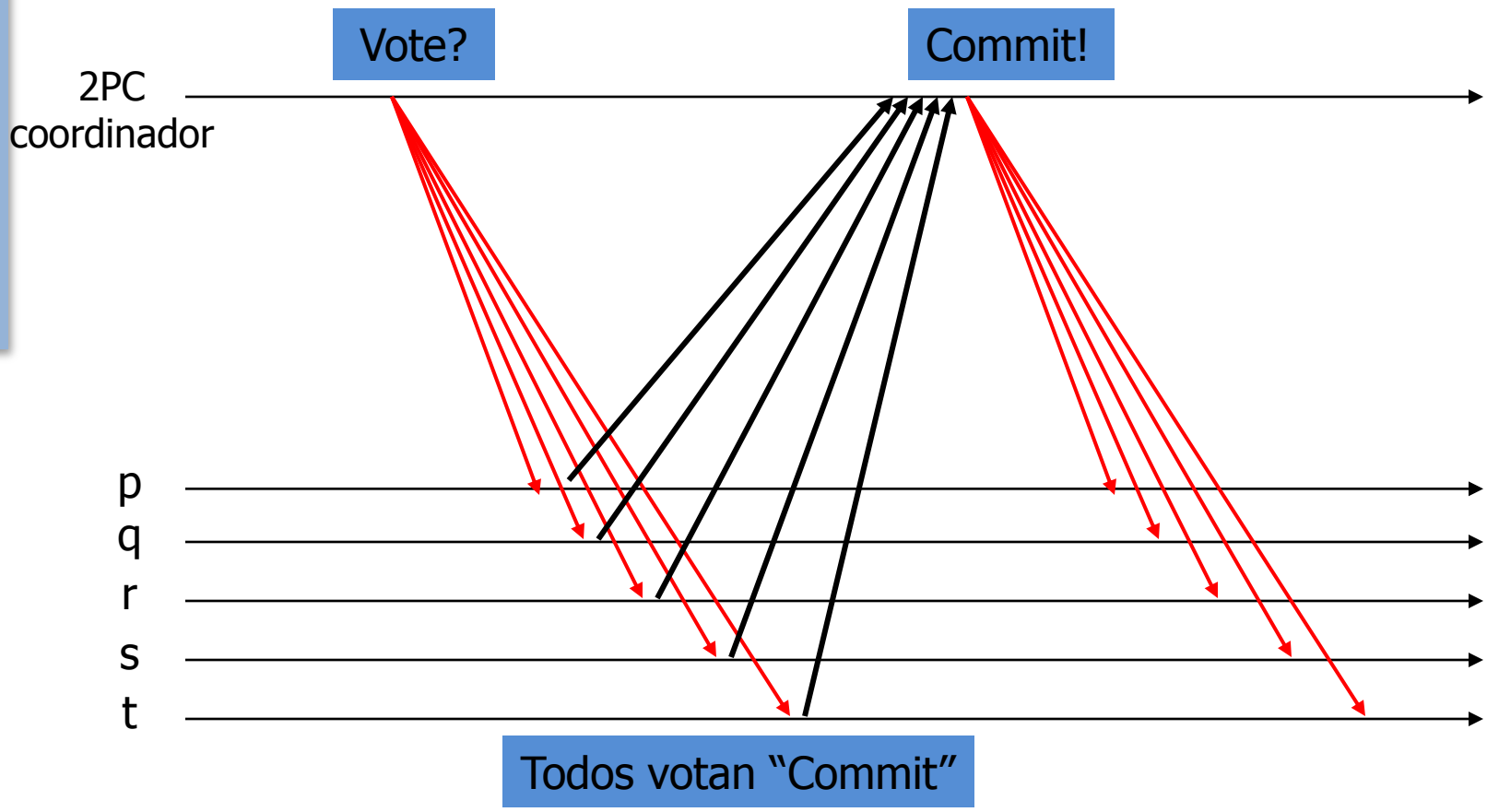




# Acuerdo distribuido (2PC)



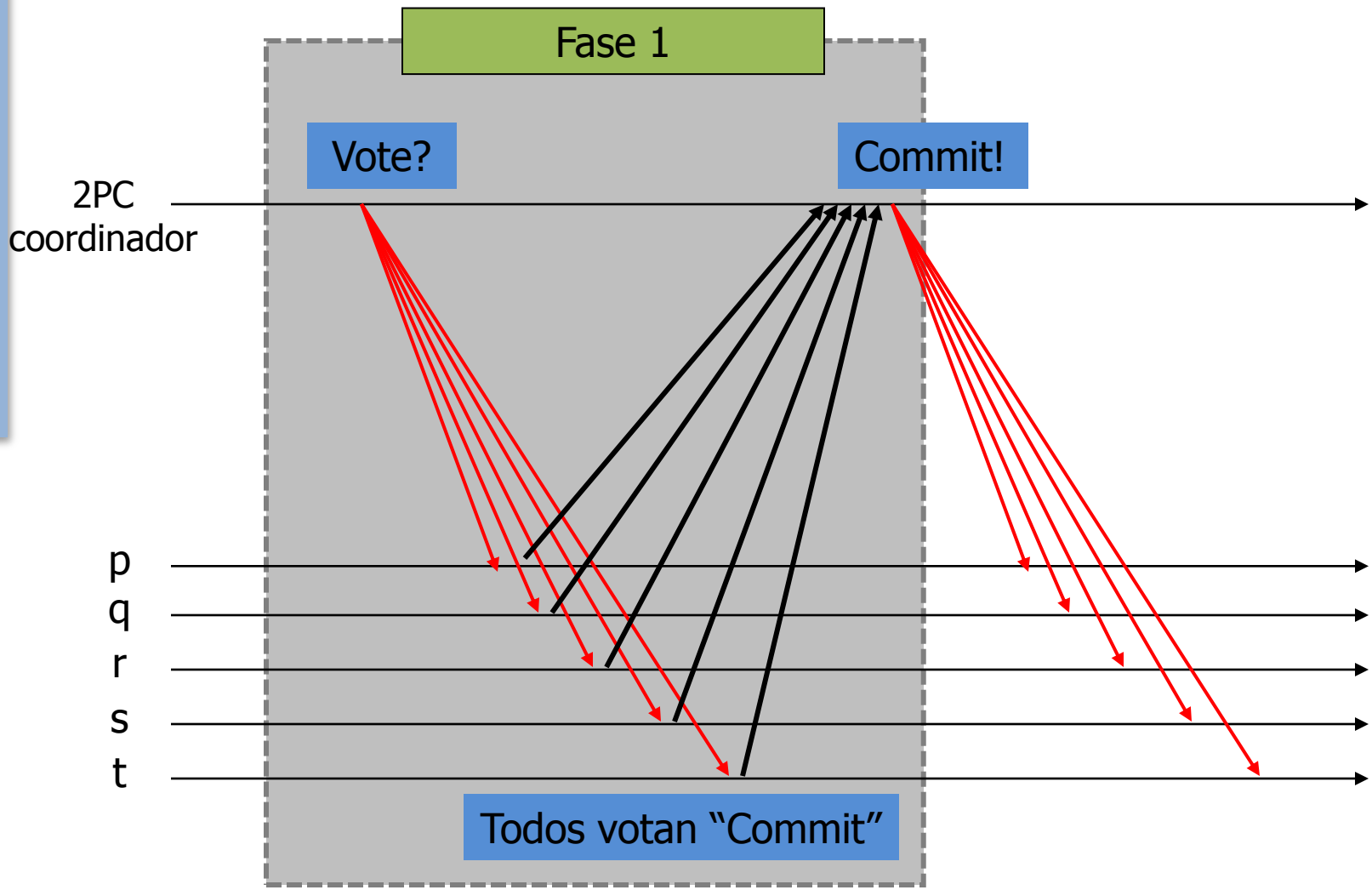




# Acuerdo distribuido (2PC)



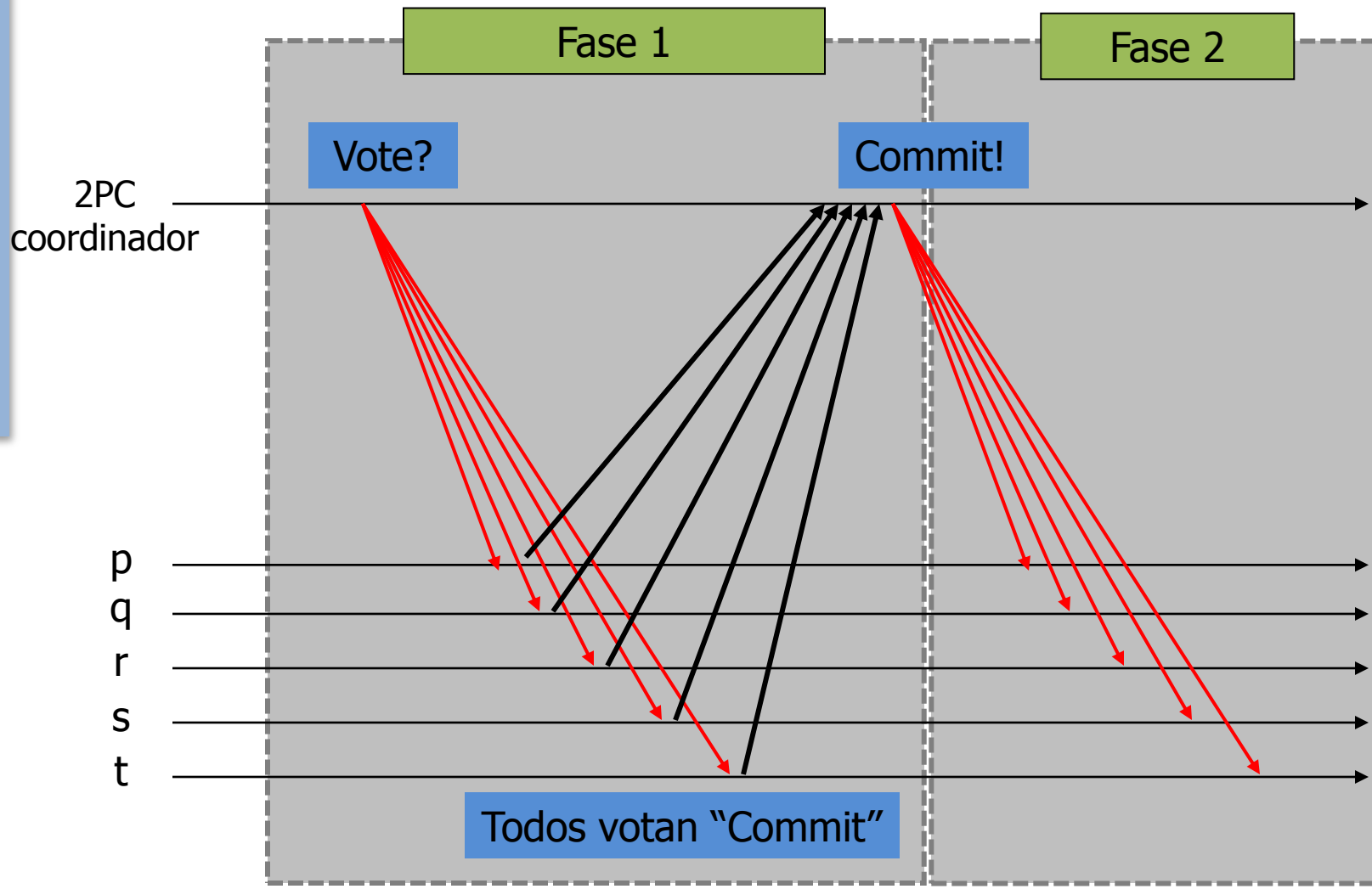




# Acuerdo distribuido (2PC)

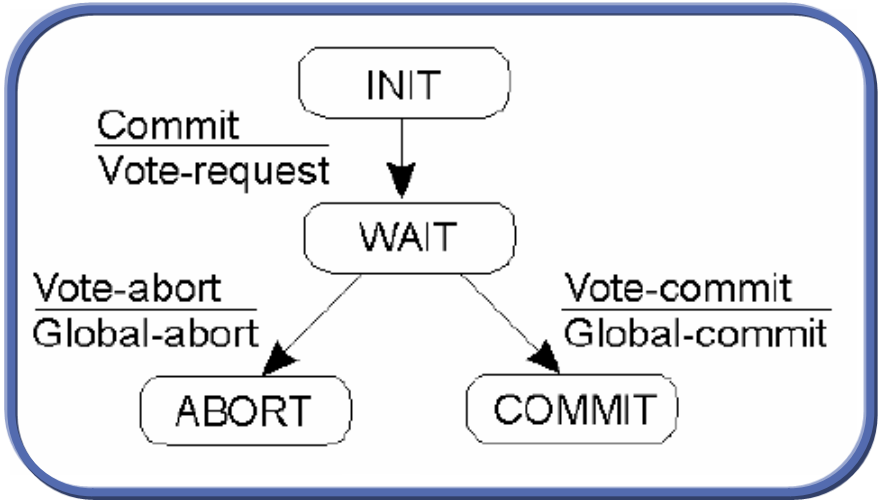




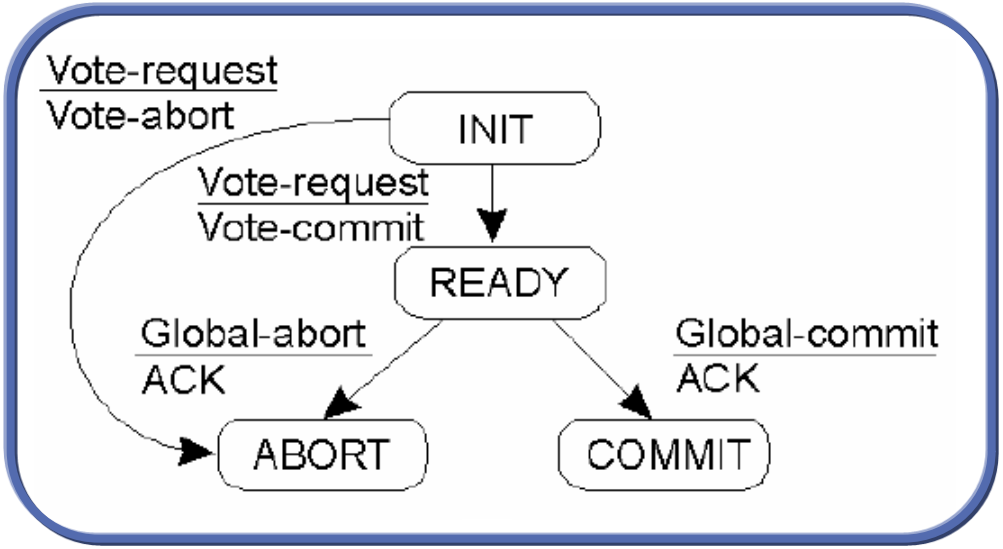


# Acuerdo distribuido (2PC)






Máquina de estados finita para el **coordinador** en 2PC



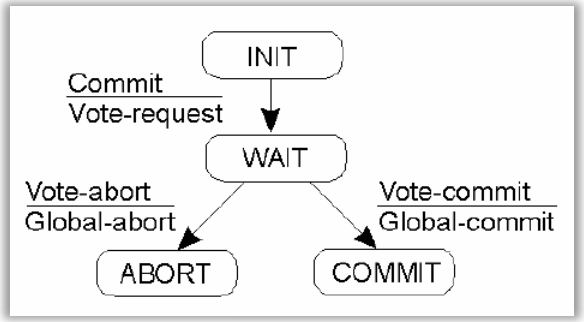

Máquina de estados finita para los **participantes** en 2PC

# Acuerdo distribuido (2PC)

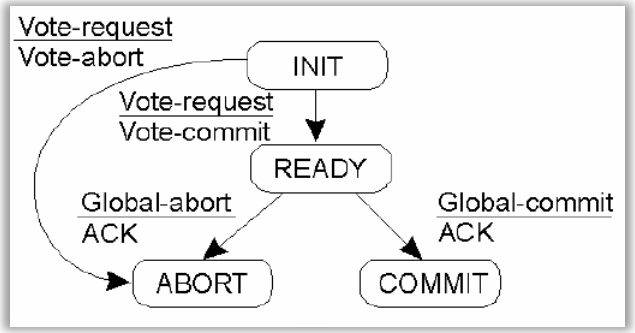




- **Problemas del 2SP en sistemas con procesos que fallan:**
  - Coordinadores y participantes se bloquean esperando por mensajes.-
    - Solución: Uso de timeouts.-




Máquina de estados finita para el **coordinador** en 2PC




Máquina de estados finita para los **participantes** en 2PC

Acciones tomadas por el participante P cuando está en estado READY y tiene contactado a otro participante Q

| Estado de Q | Acción de P |
|---|---|
| COMMIT | Hace la transición a COMMIT |
| ABORT | Hace la transición a ABORT |
| INIT | Hace la transición a ABORT |
| READY | Contacte a otro participante |

# Acuerdo distribuido (2PC)



- Los procesos deben grabar su estado a almacenamiento persistente (para recuperarse de caídas)
  - Participantes en estado INIT, puede decidir abortar la transacción cuando se recupera
  - Participantes que han tomado una decisión, retransmite la decisión al coordinador luego de la recuperación

- **Problemas** ante caídas según el estado:
  - Participantes en estado READY
    - Debe contactar a otro participantes para decidir qué hacer
  - Coordinador que inicia el protocolo 2PC, debe registrar que entra al estado WAIT
  - Coordinador en la segunda etapa, debe registrar la decisión para retransmitirla al recuperarse

# Acuerdo distribuido (2PC)



**Problemas:**

- Cuando el coordinador cae, los participantes no son capaces de alcanzar una decisión final
  - Los participantes deben permanecer bloqueados hasta que el coordinador se recupere

**Solución:** Commit distribuido de tres faces

La idea es agregar un estado extra de PRECOMMIT ("prepared to commit")

# Acuerdo distribuido (3PC)





- Los estados del coordinador y de cada participante satisfacen:
  - No hay un único estado desde el cual sea posible realizar una transición directa a un estado COMMIT o ABORT
  - No existe un estado en el que no sea posible tomar una decisión final, y desde el cual se puede realizar una transición al estado COMMIT

Esencia del protocolo

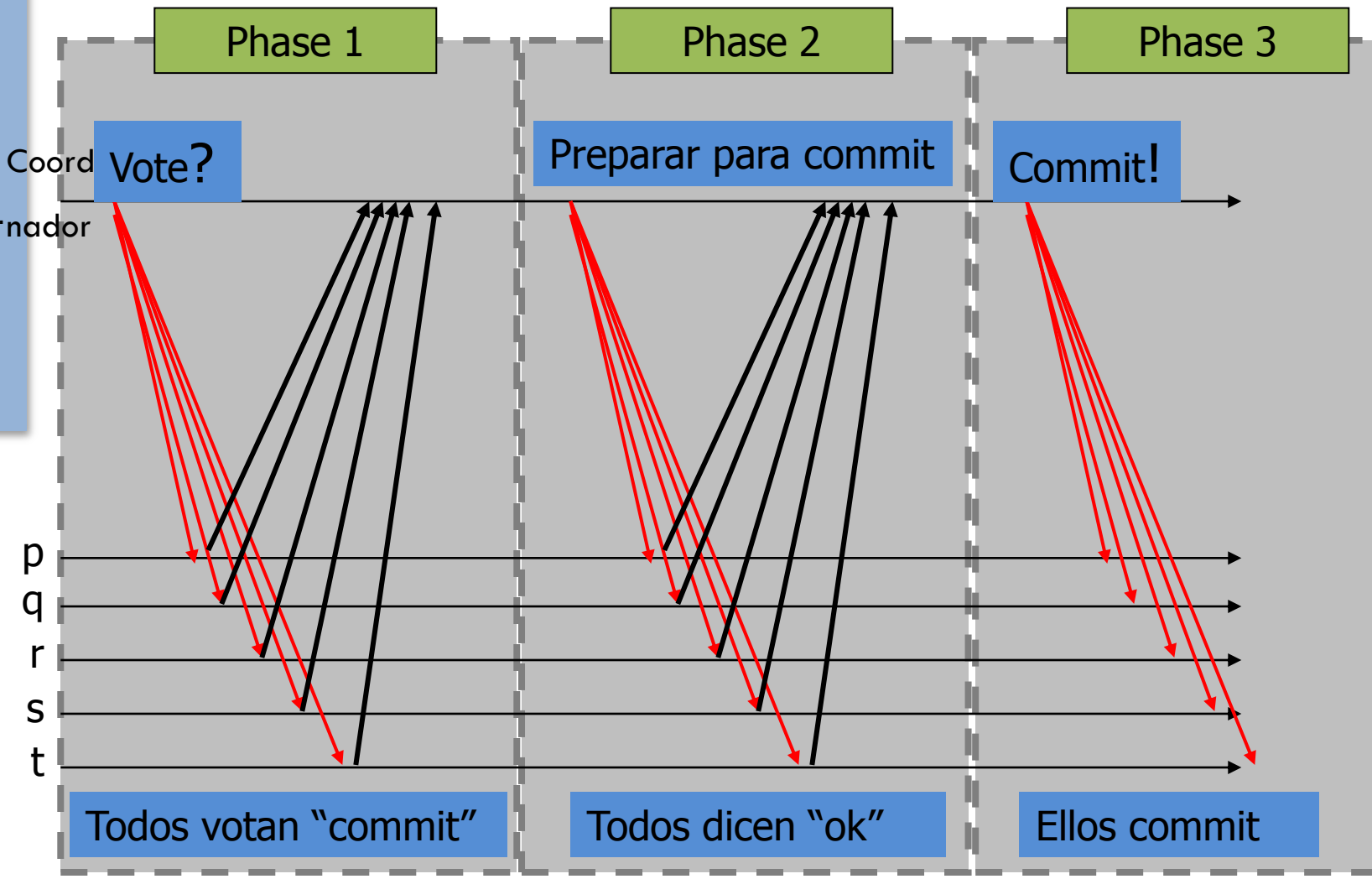
Evita que los procesos se bloqueen en la presencia de fallas
No es muy aplicado en la práctica

La principal diferencia con 2PC es que los procesos que se recuperan siempre pueden llegar a una decisión final

# Acuerdo distribuido (3PC)



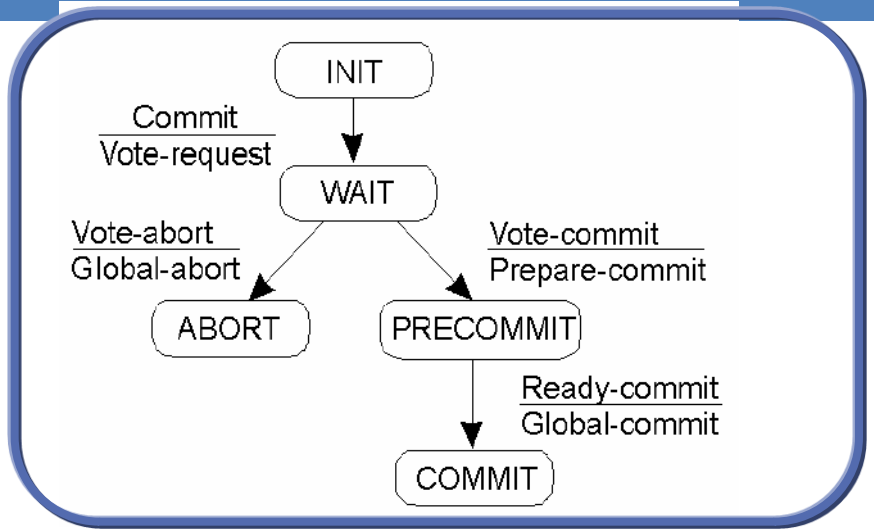




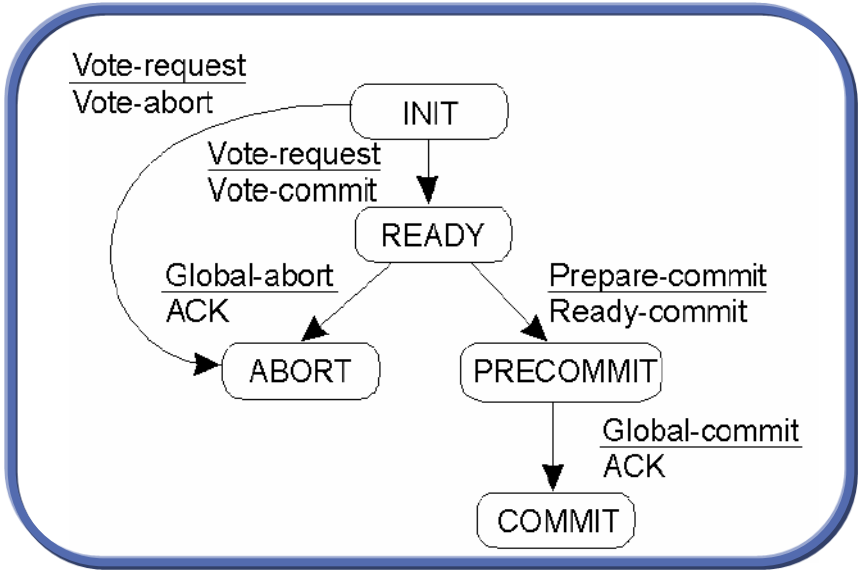
# Acuerdo distribuido (3PC)








Máquina de estados finita para el **coordinador** en 3PC

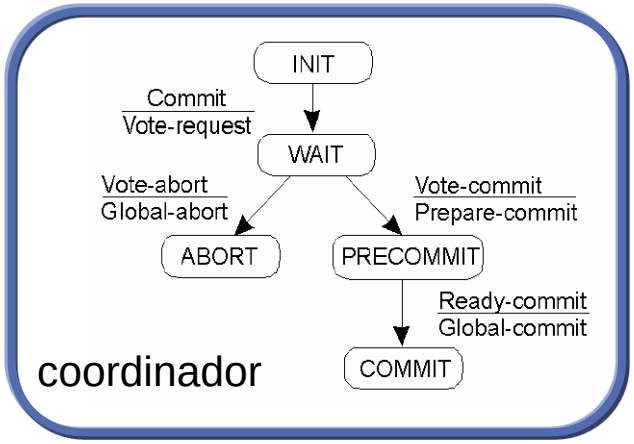


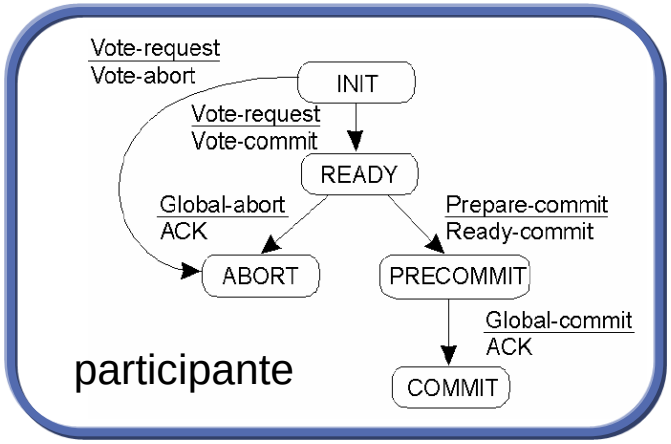

Máquina de estados finita para los **participantes** en 3PC

# Acuerdo distribuido (3PC)



- Participante en estado INIT espera por vote-request desde el coordinador
  - timeout -> transición estado ABORT
- Coordinador en estado WAIT
  - timeout -> multicast un mensaje de GLOBAL_ABORT
- Coordinador en estado PRECOMMIT
  - timeout -> puede concluir que uno de los participantes ha caído -> multicast GLOBAL_COMMIT (confía en el protocolo de recuperación)
- Participante bloqueado en estado READY o PRECOMMIT
  - timeout -> coordinador ha fallado -> contacta a otros participantes. Si todos los participantes están en PRECOMMIT, la transacción se puede finalizar



# Recuperación

# Recuperación





- Lo más importante para la TF es recuperarse de un error

**Error** es una parte de un sistema que puede producir una falla

- La **idea central** de **recuperarse de un error** es reemplazar el estado erróneo con un estado libre de error

- Formas de recuperación:
  - Recuperación backward

    Llevar al sistema desde el estado actual erróneo a un estado previo correcto

    Registros del estado del sistema (checkpoints) de vez en cuando
  - Recuperación forward

    Llevar el sistema a un estado correcto nuevo a partir del cual puede continuar ejecutando

    Problema: se deben conocer los posibles errores a ocurrir

# Recuperación





## Recuperación backward

- Ampliamente usada como mecanismo general para recuperarse de fallas en sistemas distribuidos
- **Ventaja:**
  - Método aplicable independiente de cualquier proceso o sistema específico
  - Se puede integrar en el nivel de middleware de un SD como un servicio de propósito especial
- **Problemas:**
  - Restablecer un sistema o un proceso a un estado previo es una tarea costosa en término de performance
  - No se puede garantizar que una vez que se realice la recuperación, no se produzca la misma falla o una similar
  - Algunos estados nunca se pueden “retroceder”

# Recuperación





- **Recuperación backward → Checkpoint**
  - Permite recuperar a un estado previo
  - Operación costosa

- Combinación de checkpoint con **registro (log) de mensajes**
  - Registros basados en el emisor
  - Registros basados en el receptor

Es más eficiente que tomar checkpoints continuos

# Almacenamiento estable

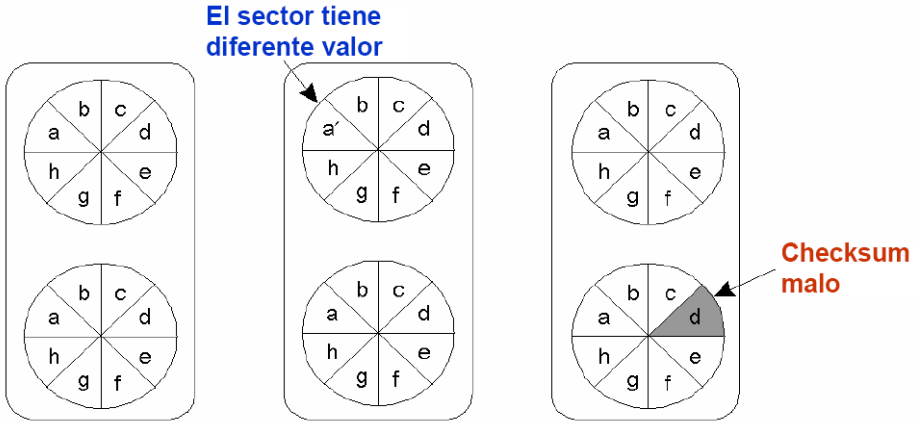




- Para recuperarse a un estado previo, es necesario que la información esté guardada en forma segura → **almacenamientos estables**

  Almacenamiento que sobrevive a fallas, excepto calamidades mayores

- Categorías de almacenamiento:
  - RAM
  - Discos
  - Almacenamiento estable
    - Se puede implementar con un par de discos comunes



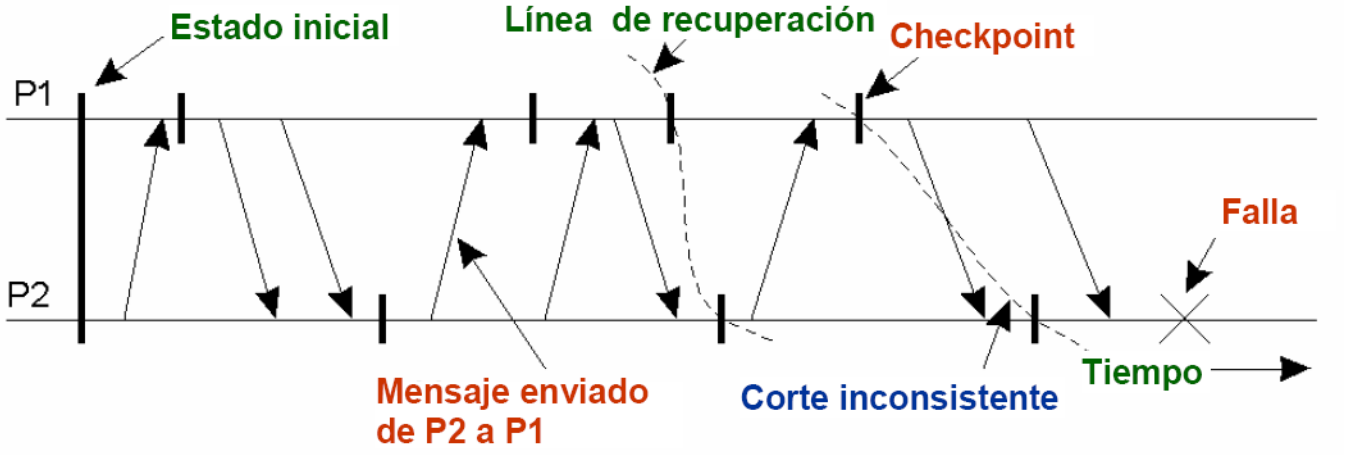
# Checkpoint





- Recuperación de error backward:
  - estados salvados en almacenamiento estables
  - Necesidad de registrar un estado global consistente
    - **Instantánea distribuida**

- Para recuperar un sistema o un proceso luego de una caída es necesario construir un estado global consistente desde los estados locales registrados por cada proceso



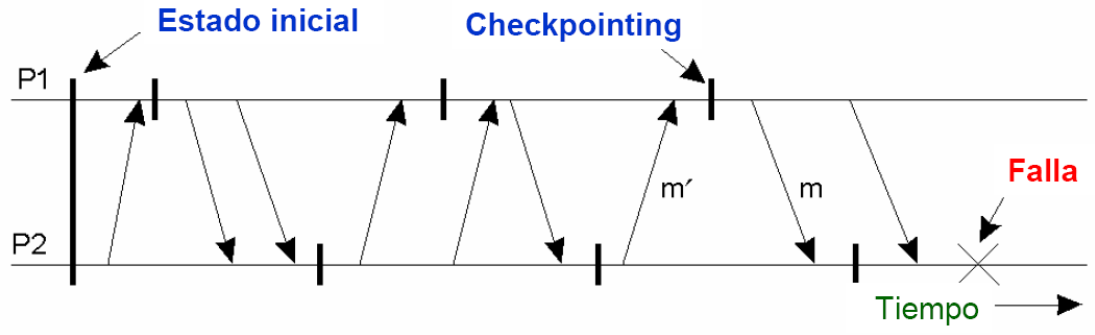
# Checkpoint





- Todos los estados locales deben formar una instantánea distribuida

- Los checkpoints pueden ser:
  - **Independientes**
    - Cada proceso realiza los checkpoints en forma independiente del resto
    - Encontrar una línea de recuperación – tarea difícil
    - Efecto dominó (posiblemente volver al estado inicial)




    - Limpieza periódica de los almacenamientos locales
    - Calcular la línea de recuperación requiere un análisis de dependencias registradas por cada proceso. Proceso complejo

# Checkpoint



- **Coordinados**
  - Los procesos se sincronizan para escribir en forma conjunta sus estados locales a almacenamiento local
  - Se evita el efecto dominó
  - El estado salvado es consistente globalmente
  - Algoritmo de instantánea distribuida (no bloqueante)
    - Solución más simple: protocolo bloqueante de dos fases

**Desventaja**

Operación muy cara (operaciones involucradas en escribir estados en almacenamiento estables)

# Log de mensajes





## Log de Mensajes

- Replicación de la transmisión de mensajes → alcanzar estados globales sin rehacerlos desde un almacenamiento estable
- Estado de checkpoint → punto de inicio y retransmisión de los mensajes enviados
- Modelo piecewise determinístico
  - La ejecución de un proceso se realiza como una serie de intervalos en la cual se producen eventos
    - Eventos discutidos en el contexto de la relación ocurre-antes de Lamport
  - Cada intervalo comienza con un evento no determinístico
  - Dentro del intervalo la ejecución es determinística
  - Un intervalo finaliza con el último evento antes de que ocurra un evento no determinístico
  - Un intervalo se puede replicar con resultados conocidos
  - Se registran todos los eventos no determinísticos