# Teoría de Sistemas Distribuidos

# Contenido

- Tiempo físico y tiempo lógico
  - NTP
  - Relojes lógicos
- Coordinación y acuerdo
  - Detectores de fallos
  - Exclusión mutua distribuida
  - Elecciones

# Tiempo Físico y Tiempo Lógico

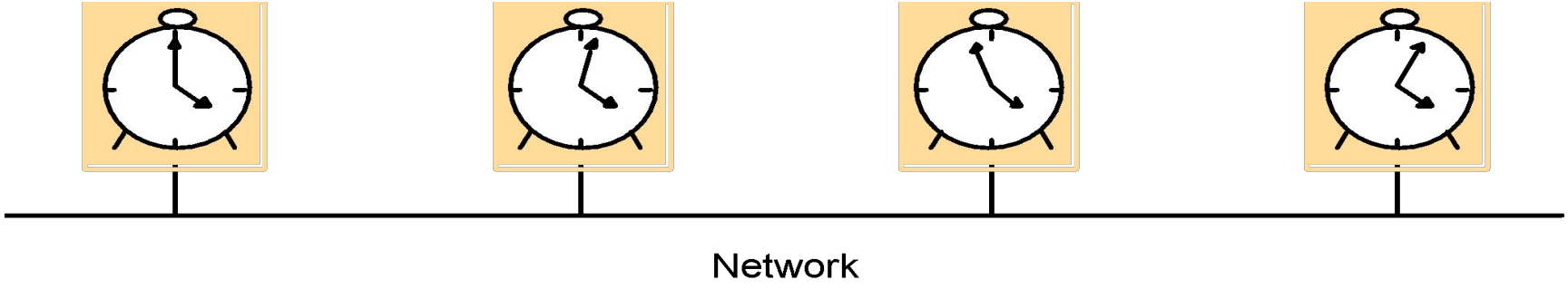
# Planteamiento del Problema

- Se debe poder medir el tiempo de manera precisa
  - Para saber cuándo ocurrió un evento en una máquina
  - Necesario sincronizar reloj con reloj externo confiable
- Algoritmos de sincronización de relojes, útiles para
  - Control de concurrencia basado en ordenamiento de timestamps
  - Autenticidad de pedidos, por ejemplo, en Kerberos
- No hay un reloj global en un sistema distribuido
  - Esta sección: precisión de relojes y sincronización
- Tiempo lógico es una alternativa
  - Permite ordenar eventos
  - También útil para consistencia de datos replicados

# Relojes de Computadoras

- Cada computadora en un SD tiene un reloj interno
  - Usado por procesos locales para obtener hora actual
  - Procesos en diferentes computadoras pueden obtener estampas de tiempo para sus eventos
  - Relojes en diferentes computadoras pueden diferir
  - Relojes se desfasan del tiempo exacto; ratios de desfase varían
    - Ratio de desfase del reloj: Cantidad relativa que un reloj difiere de un reloj perfecto
- Aún si los relojes en todas las computadoras en un SD se setean con el mismo tiempo, sus relojes eventualmente variarán significativamente
  - Necesario aplicar correcciones apropiadas

# Variaciones de los Relojes




- Relojes generalmente marcan horas diferentes
- Desviación: Diferencia entre tiempos en 2 relojes (en cualquier instante)
- Desfase del reloj: Relojes miden el paso del tiempo en ratios diferentes
- Ratio de desfase del reloj: Diferencia por unidad de tiempo de algún reloj referencial ideal
- Relojes de quartz ordinario se desfasan en aprox. 1 segundo cada 11-12 días ($10^{-6}$ segundos/segundo)
- Relojes de quartz de alta precisión se desfasan $10^{-7}$ or $10^{-8}$ segundos/segundo

# Tiempo Universal Coordinado (UTC)

- T. atómico basado en relojes físicos de alta precisión
- UTC: estándar internacional para llevar el tiempo
- Se basa en tiempo atómico; se ajusta ocasionalmente para igualarse con el tiempo astrológico
- Se anuncia en estaciones de radio y satélites (GPS)
- Computadoras con receptores pueden sincronizar sus relojes con estas señales
- Señales de radio son precisas en 0.1-10 milisegundos
- Señales de GPS son precisas en 1 microsegundo

# Sincronizando Relojes Físicos

# Relojes Correctos

- Se dice que un reloj de HW *H* es correcto si su ratio de desfase está dentro de un límite $\rho > 0$ (ej.: $10^{-6}$ segs/seg)
- ⇒ error al medir el intervalo entre tiempos reales *t* y *t'* tiene un límite:
  - $(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$ donde $t' > t$
  - Lo cual no permite saltos en las lecturas del tiempo de relojes de hardware

# Relojes Correctos

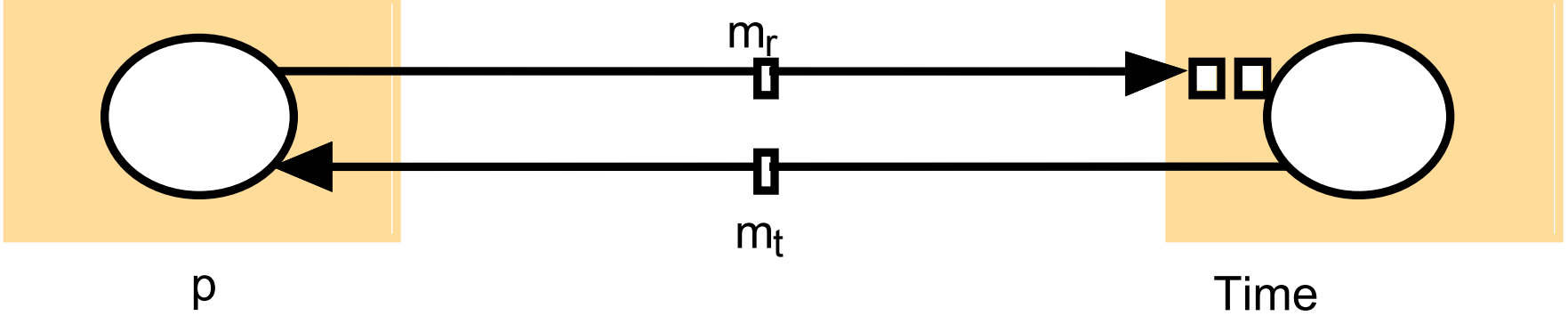
- Se puede requerir que reloj de SW cumpla medida anterior
- Generalmente, basta con cumplir condición de monotonía
  - $t' > t \Rightarrow C(t') > C(t)$
  - Ejemplo, *make* en Unix requiere esta condición
  - Se puede lograr monotonía con un reloj de hw que se adelanta, al ajustar los valores α y β de $C_i(t) = \alpha H_i(t) + \beta$

# Relojes Correctos

- Híbridos: monotonía dentro de puntos de sincronización
- Un reloj con fallos es uno que no es correcto (según alguna definición de correcto previamente acordada)
  - Fallas crash: reloj deja de avanzar (tics)
  - Fallas arbitrarias: cualquier otro tipo de fallas; ej.: saltos en el tiempo

# Algoritmo de Cristian (1989)

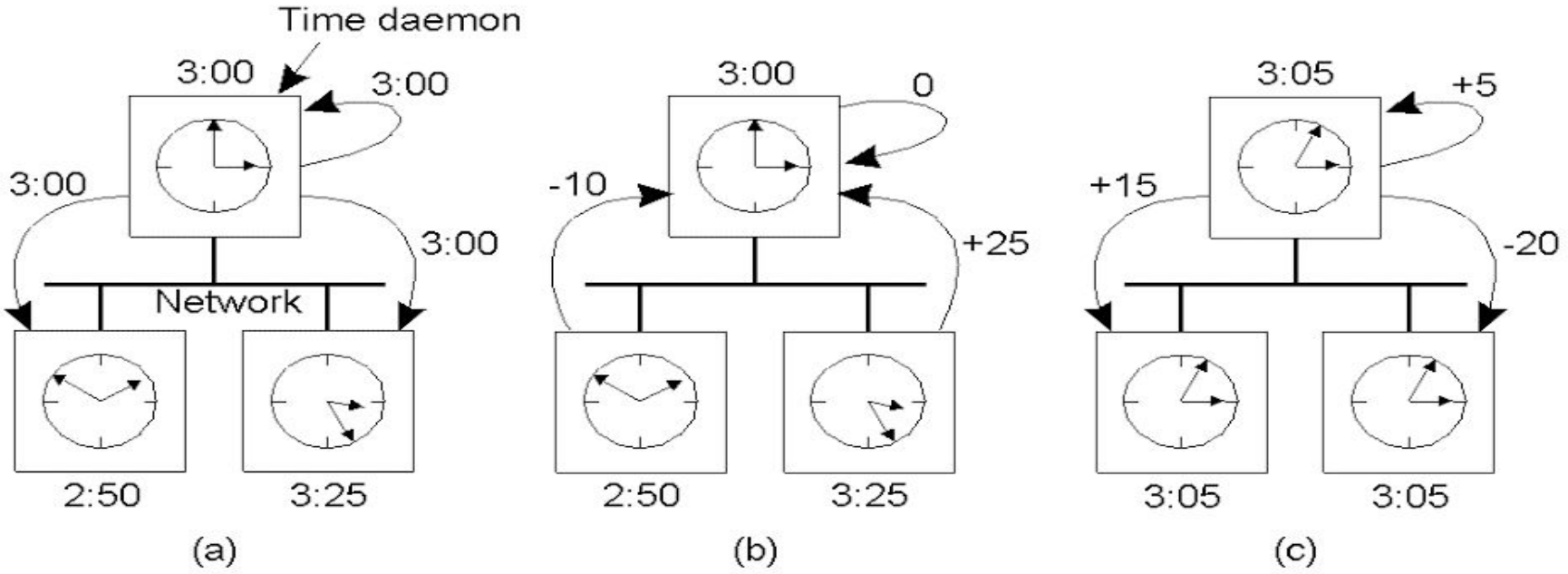
- Para sistemas síncronos
- Servidor de tiempo *S* envía señales de una fuente UTC
  - Proceso *p* pide tiempo en $m_r$ y recibe *t* en $m_t$
  - *p* setea su reloj a $t + T_{rtt}/2$
  - Precisión ± ($T_{rtt}/2$ – min):
    - Si *S* coloca *t* en $m_t$ lo más pronto, será *min* después de que *p* recibe $m_r$
    - Lo más tarde es *min* antes de que $m_t$ llegue a *p*
    - Momento según S cuando $m_t$ llega, está en rango [$t + min$, $t + T_{rtt} - min$]



# Algoritmo de Berkeley (1989)

- Algoritmo para sincronización interna de grupos
- *Maestro* pide valores de relojes del resto (*esclavos*)
- Maestro usa RTTs para estimar relojes de esclavos
- Obtiene un promedio de los relojes
  - Se eliminan valores mayores a un RTT máximo
- Tolerancia a fallos:
  - Solo se consideran relojes que no difieren más de *x* entre sí
- Maestro envía *ajuste* de tiempo a esclavos
  - No envía nuevo tiempo *t*, sino valor a sumar (o restar)
- Si maestro falla, se puede *elegir* un nuevo maestro

# Algoritmo de Berkeley (1989)



# NTP

- Algoritmos de Cristian y Berkeley diseñados para intranets
- NTP: Protocolo de Tiempo de Red
  - Servicio de sincronización de relojes para Internet
    - Arquitectura para el servicio
    - Protocolo de distribución
  - Sincroniza clientes a UTC

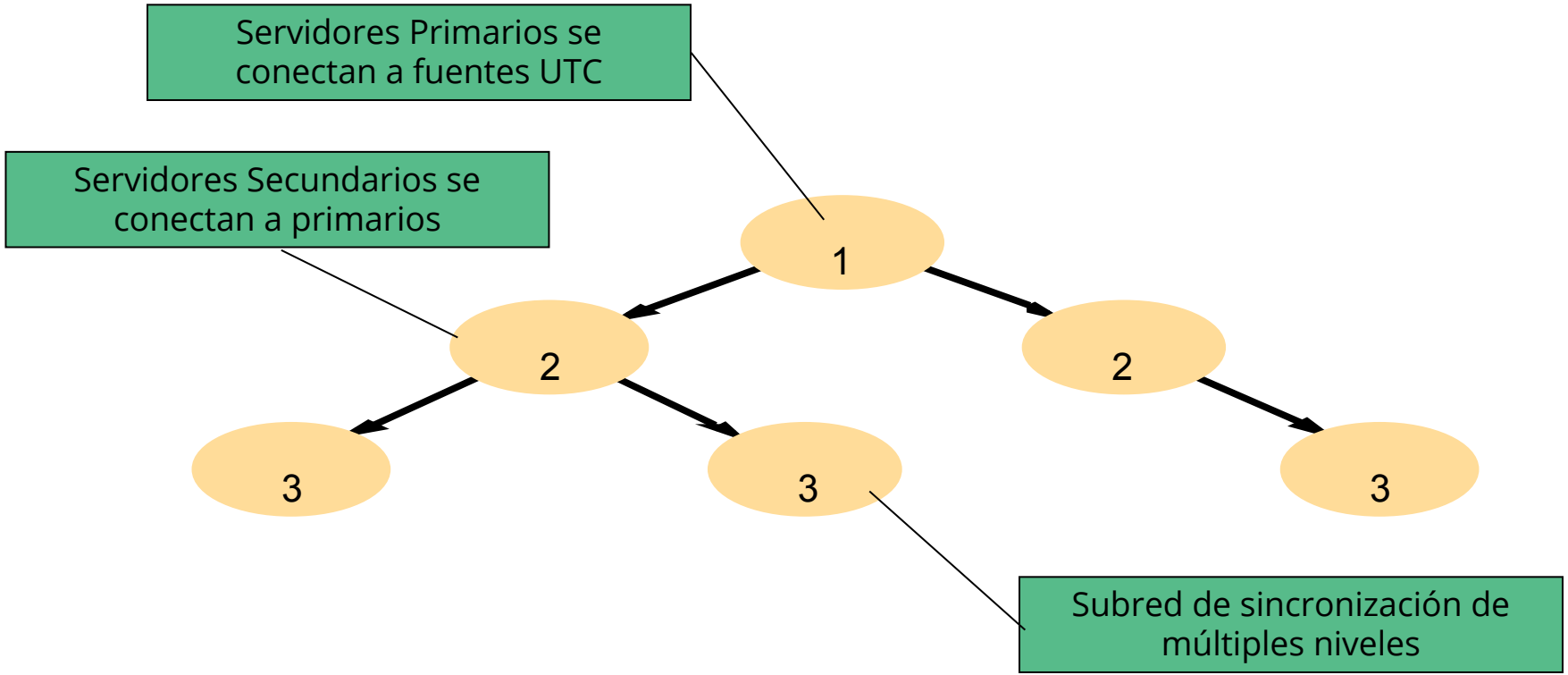
# NTP: Cronología

- Varios predecesores, desde 1979
- Versión 0 (1985) a 4 (1994), por David Mills
- Desde Versión 2, basado en algoritmo de Marzullo (tesis de PhD, Stanford, 1984)
  - Mejoras fueron necesarias para eliminar jitter (variación en delay)

# NTP: Metas del Diseño

- Permitir sincronizarse a UTC, en Internet
- Confiabilidad; sobreviva largas desconexiones
  - Varios servidores y varias rutas a c/u; continúan si uno falla
- Permitir a re-sincronizaciones frecuentes
  - Escalabilidad
- Proporcionar protección contra interferencias maliciosas o accidentales
  - Autenticación

# NTP: Organización



# NTP

- Subred de sincronización se puede reconfigurar si ocurren fallos, como por ejemplo:
  - Un primario pierde su fuente UTC y puede convertirse en secundario
  - Si secundario pierde primario, usa otro primario
- Ejemplo de primario: ntp.nasa.gov
- Ejemplo de secundario: ntp0.cornell.edu
- Puede mantener sincronización de 10 ms en Internet (1 ms en LANs)

# NTP: Modelos de Sincronización

- Multicast
  - Un servidor con una red LAN de alta velocidad envía tiempo actual a otros usando multicast, los cuales ajustan sus relojes asumiendo una demora (delay) – poca precisión
- Llamadas a procedimientos
  - Un servidor acepta pedidos de clientes – mayor precisión
  - Útil si multicast no es disponible
- Simétrico
  - Pares de servidores intercambian mensajes
  - Alta precisión (ej.: para niveles altos de la jerarquía)

# Tiempo Lógico y Relojes Lógicos

# Relojes, Eventos y Estados

- Un SD se define como una colección *P* de *N* procesos $p_i$, $i = 1, 2, \ldots N$
- Cada proceso $p_i$ tiene un estado $s_i$ que consiste de sus variables (que son transformadas conforme se ejecuta)
- Procesos se comunican únicamente via mensajes (en la red)
- Acciones de procesos:
  - Enviar, recibir y cambiar su propio estado

# Relojes, Eventos y Estados

- Evento: ocurrencia de una acción que un proceso realiza mientras se ejecuta. Ej.: enviar, recibir y cambiar estado
- Eventos en un proceso $p_i$ pueden colocarse en un orden total denotado por la relación $\rightarrow_i$ entre los eventos
  - e $\rightarrow_i$ e' si y solo si *e* ocurre antes de *e'* en $p_i$
- Un historial de procesos $p_i$ es una serie de eventos ordenados por $\rightarrow_i$
  - historial($p_i$) = $h_i = <e_i^0, e_i^1, e_i^2, ...>$
- Concepto clave: *OCURRIÓ-ANTES*

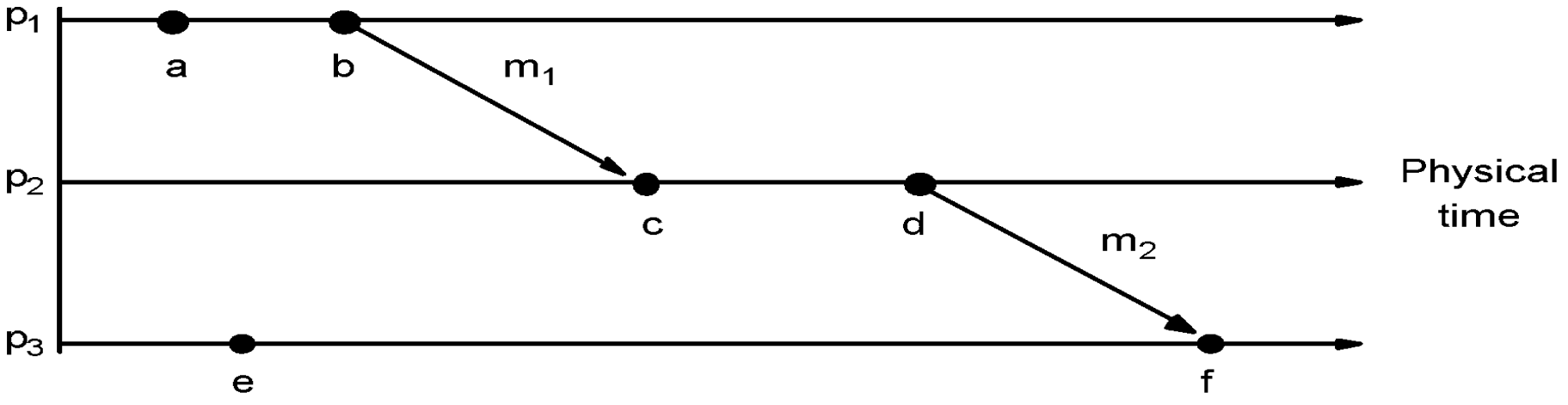
# Relojes Lógicos y Orden Causal

- Eventos en un proceso pueden ordenarse por tiempo de reloj físico local
- Lamport 1978
  - Relojes no pueden sincronizarse perfectamente en un sistema distribuido
  - No se debe depender de tiempo físico para ordenar un par arbitrario de eventos del SD
  - Usar relojes lógicos
    - Relación *ocurrió-antes*

# Orden Causal

- Basado en dos puntos simples:
  - Si dos eventos ocurren en el mismo proceso $p_i$ ($i$ = 1, 2, …, $N$), entonces ocurrieron en el orden en que se observaron en $p_i$ ($\rightarrow_i$)
  - Cuando un mensaje $m$ se envía entre dos procesos, *send*($m$) ocurre antes de *receive*($m$)
- Lamport: el orden *parcial* que se obtiene al combinar puntos anteriores lleva a relación *ocurrió-antes* ($\rightarrow$)
  - También llamada de *orden (potencialmente) causal*
- Además, la relación ocurrió-antes es transitiva
  - Si e $\rightarrow$ e’ y e’ $\rightarrow$ e”, entonces e $\rightarrow$ e”

# Ejemplo



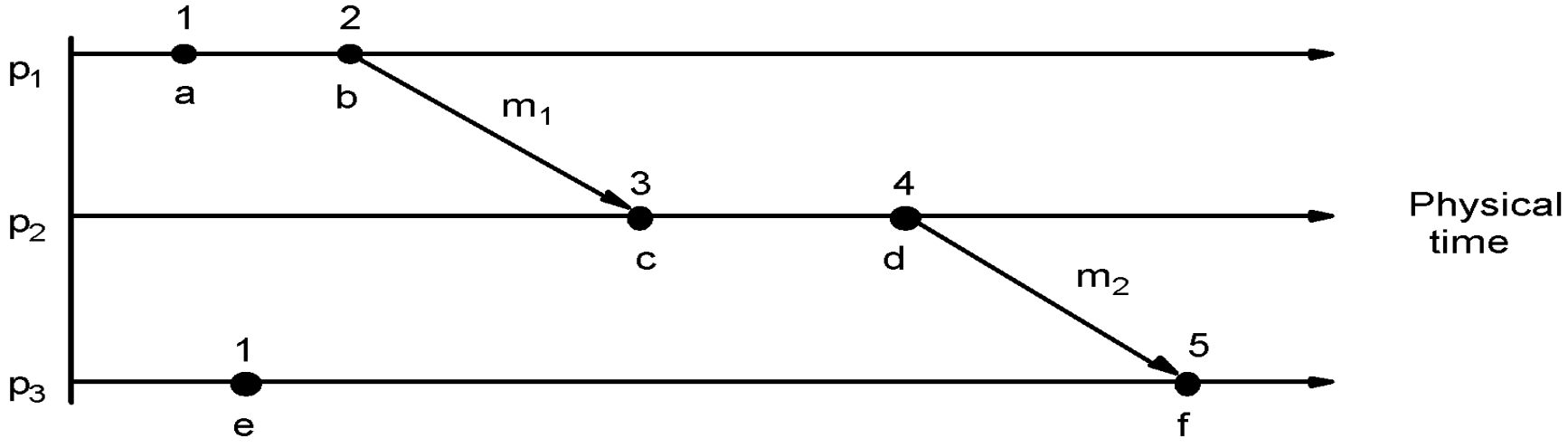

- a → b (en p1)
- c → d (en p2)
- e → f (en p3)
- b → c
- d → f

- Conclusión:
  - a → b → c → d → f
  - e → f
- No se puede concluir relación causal entre a y e
  - a || e

# Relojes Lógicos de Lamport

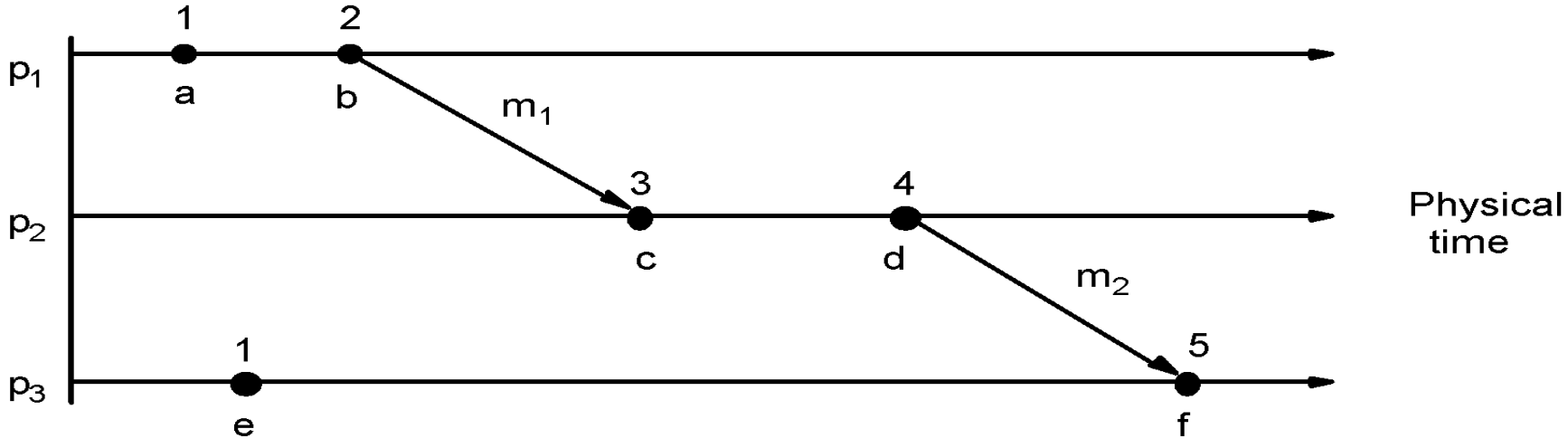
- Reloj lógico: contador de software que crece monotónicamente
  - No necesita tener relación con reloj físico
- C/ proceso $p_i$ tiene reloj lógico $L_i$ que se puede usar para aplicar estampas de tiempo lógicas a eventos:
  - LC1: $L_i$ se incrementa en 1 antes de cada evento en $p_i$
  - LC2:
    - Cuando $p_i$ envía mensaje $m$, adjunta $t = L_i$
    - Cuando $p_j$ recibe ($m$,$t$), setea $L_j = max(L_j,t)$ y aplica LC1 antes de colocar una estampa de tiempo al evento *receive*($m$)

# Ejemplo




1. $L_1$=0, $L_2$=0, $L_3$=0
2. $m_1$ lleva “2” adjunto
3. Estampa de tiempo de c = *max*(0, 2) + 1 = 3
4. Estampa de tiempo de f = *max*(4, 1) + 1 = 5

# Ejemplo (cont ...)




- e → e' ⇒ $L(e) < L(e')$
- Pero, lo contrario no es cierto:
  - $L(e) < L(e')$ no implica e → e'
  - Ejemplo: $L(b) > L(e)$ pero $b \mid\mid e$

# Relojes Lógicos Totalmente Ordenados

- En ocasiones necesitamos aplicar un orden global y único a todos los eventos
- Relojes Lógicos *Totalmente* Ordenados:
  - Estampa de tiempo global de un evento en un proceso $p_i$ con estampa lógica local $T_i$ es $(T_i, i)$
  - $(T_i, i) < (T_j, j)$ si y solo si $(T_i < T_j)$ o $(T_i = T_j$ e $i < j)$
- Ejemplo de uso: ordenamiento de procesos que desean entrar a *sección crítica*

# Relojes Lógicos Vectoriales

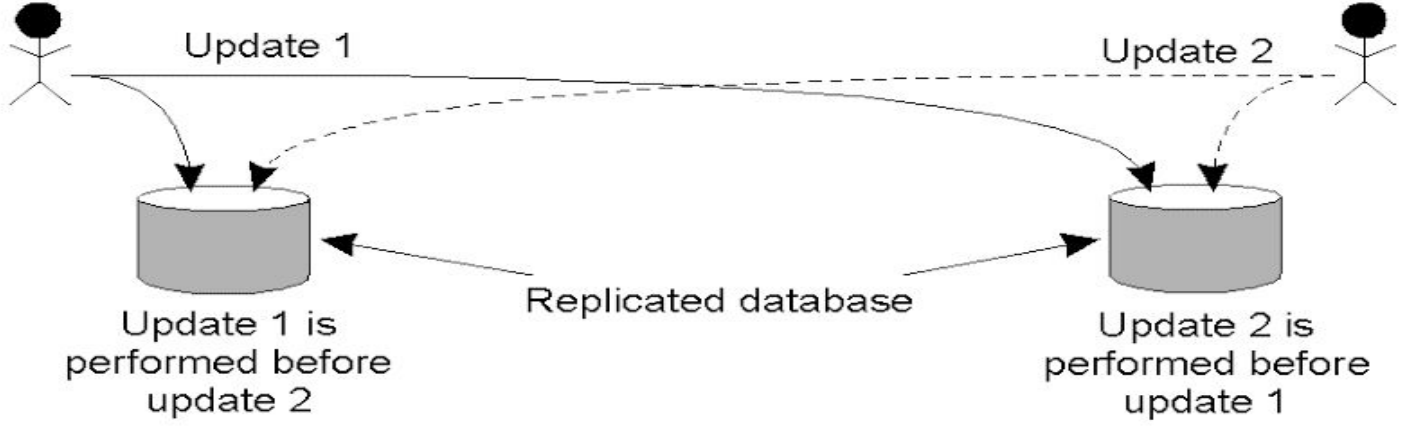
- Falencias de relojes lógicos de Lamport:
  - $L(e) < L(e')$ no implica e $\rightarrow$ e'
- Alternativa: relojes lógicos vectoriales (1989)
  - Estampas de tiempo local: $V_i$, un vector de $N$ enteros (si sistema tiene $N$ procesos)
  - Estampas de tiempo vectoriales se adjuntan a los mensajes que se envían los procesos

# Relojes Lógicos Vectoriales

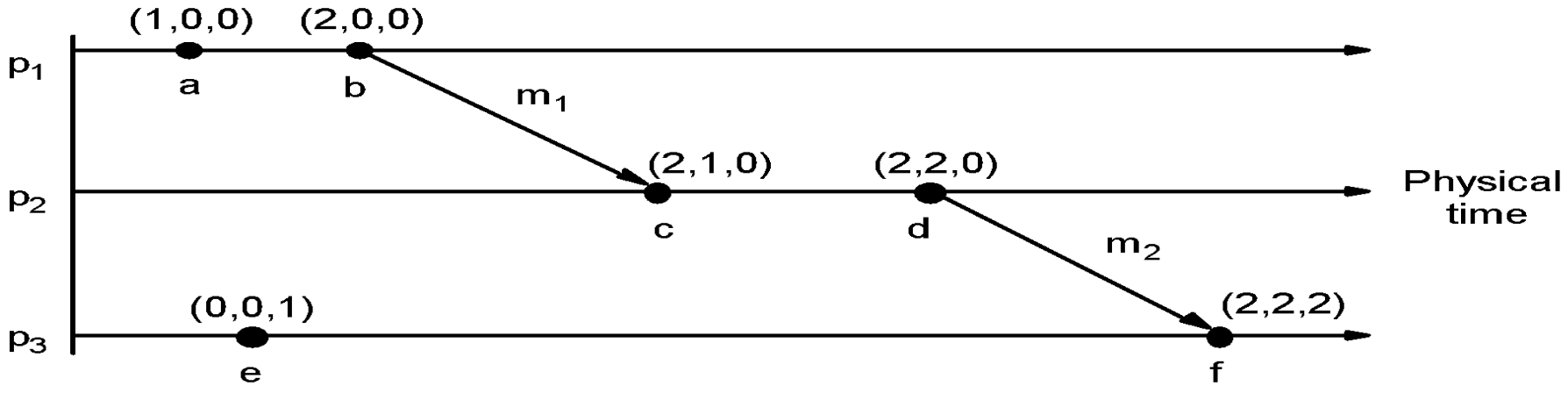
- Reglas de actualización de los relojes:
  - VC1: Inicialmente, $V_i[j] = 0$, para $j = 1, 2, ..., N$
  - VC2: Justo antes de que $p_i$ estampe un evento, setea $V_i[i] = V_i[i] + 1$
  - VC3: $p_i$ incluye valor $t = V_i$ en cada mensaje
  - VC4: Cuando $p_i$ recibe una estampa $t$ en un mensaje, setea $V_i[j] = max(V_i[j], t[j])$, para $j = 1, 2, ..., N$
    - A esta operación se la conoce como merge

# Relojes Lógicos Vectoriales

- Se los usa en esquemas de replicación de datos, BD, multicast causal, etc.

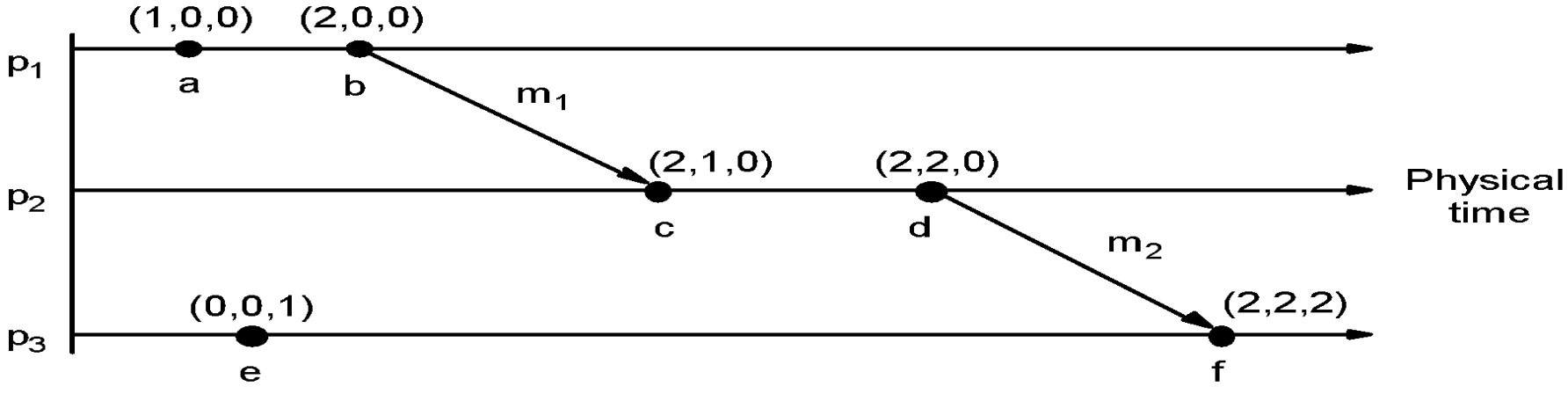


# Ejemplo




- **Reglas para comparar las estampas de tiempo vectoriales**:
  - $V = V'$ si y solo si $V[j] = V'[j]$ para $j = 1, 2, \ldots, N$
  - $V \leq V'$ si y solo si $V[j] \leq V'[j]$ para $j = 1, 2, \ldots, N$
  - $V < V'$ si y solo si $V \leq V'$ y $V \neq V'$

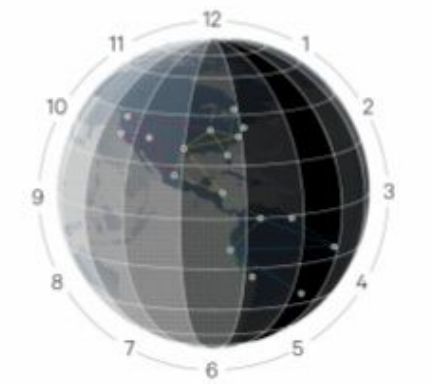
# Ejemplo




- ¿Qué eventos son concurrentes?
  - e || c, porque ni V(c) ≤ V(e), ni V(e) ≤ V(c)
  - ¿Algún otro?

# Nueva generación de sincronización: TrueTime

- TrueTime API, propuesto para BD global de Google: Spanner




***Idea:*** Accept uncertainty, keep it small and quantify (using GPS and Atomic Clocks)

Novel **API** distributing a globally synchronized **„proper time”**

| Method | Returns |
|---|---|
| TT.now() | TTinterval: [earliest, latest] |
| TT.after(t) | True if t has definitely passed |
| TT.before(t) | True if t has definitely not arrived |

**TT interval** - is guaranteed to contain the **absolute time** during which TT.now() was invoked

- Lecturas recomendadas:
  - http://research.google.com/archive/spanner.html
  - http://www.wired.com/2012/09/google-spanner/

# TrueTime API

## Google Spanner – TrueTime


- "Global wall-clock time" with bounded uncertainty
- Novel API behind Spanner's core innovation
- Leverages hardware features like GPS and Atomic Clocks
- Implemented via TrueTime API.
  - Key method being now() which not only returns current system time but also another value (ε) which tells the maximum uncertainty in the time returned
- Set of time master server per datacenters and time slave daemon per machines.
- Majority of time masters are GPS fitted and few others are atomic clock fitted (Armageddon masters).
- Daemon polls variety of masters and reaches a consensus about correct timestamp.

출처: http://www.cse.buffalo.edu/~okennedy/courses/cse704fa2012/9.2-Spanner.ppt

# Coordinación y Acuerdo

# Coordinación y Acuerdo

- **Exclusión mutua distribuida**
- **Elecciones**
- Comunicación multicast
- Consenso y problemas relacionados

# Coordinación Distribuida

- ¿Pueden los procesos coordinar sus acciones?
- ¿Pueden los procesos llegar a un acuerdo con respecto a un grupo compartido de valores?
- ¿Qué tipos de algoritmos podemos utilizar para resolver estos problemas?
- ¿Cómo afectan las fallas a estos algoritmos?

# Importancia

- Esquemas maestro/esclavo son muy rígidos
  - Tolerancia a fallos
  - Escalabilidad
- Alternativa: grupos de procesos que coordinan sus actividades

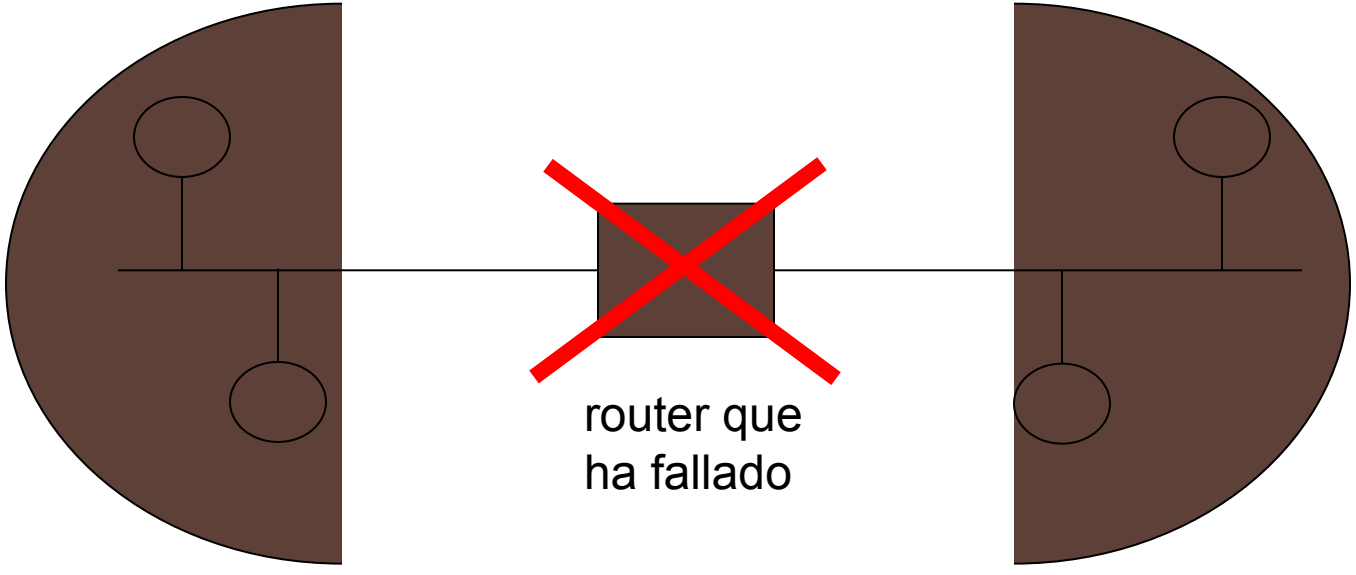
# Modelo de Fallas

- Canales confiables
  - Retransmisión y redundancia
- Fallo de un proceso no afecta comunicación de otros procesos
- Pueden haber retrasos en comunicaciones
- Retrasos en comunicaciones entre procesos no son necesariamente iguales
- Fallas de proceso son tipo *crash*
  - Proceso correcto: no falla durante ejecución

# Fallas en los Canales

- Pueden particionar la red
- Conectividad puede ser asimétrica
- Conectividad puede ser intransitiva
  - P – Q y Q – R, pero no P – R
- Puede que no todos los procesos puedan comunicarse al mismo tiempo
- Eventualmente cualquier enlace o router que haya fallado será reparado o podrá ser circunvalado

# Una Partición en la Red



# Detección de fallos

# Detección de Fallos

- Detector de fallos
  - Servicio que procesa consultas sobre si un proceso en particular ha fallado
  - Usualmente implementado por un objeto local a cada proceso que corre un algoritmo de *detección de fallos* en conjunto con su contraparte en los otros procesos
    - Parte local: detector de fallos local

# Detectores de Fallos

- No son necesariamente precisos
  - Mayoría son *detectores de fallos no confiables*
    - Dos valores
      - *No se sospecha*: hay razones para creer que proceso está en funcionamiento
      - *Se sospecha*: hay razones para creer que proceso ha fallado
        - Ejemplo: no se ha escuchado nada de un proceso X en determinado tiempo, en un sistema asíncrono
- *Detectores de fallos confiables*
  - Dos valores: *No se sospecha* o *ha fallado*

# Detectores de Fallos

- No confiables
  - Tipo más común
- Confiables
  - Solo en sistemas síncronos
- Detectores de fallos locales pueden dar respuestas diferentes a sus procesos
  - Pueden no tener información completa

# Detectores No Confiables

- Algoritmo:
  - Cada proceso *p* envía un mensaje "*p* está vivo" periódicamente a los otros procesos
  - Detector de fallos usa un estimado del tiempo máximo de transmisión de un mensaje, *D*
  - Si detector local en *q* no ha recibido "*p* está vivo" en *T* + *D* segundos, reporta a *q* que *se sospecha* de *p*
  - Si posteriormente se recibe mensaje de *p*, se reporta que *no se sospecha* de *p*

# Detectores No Confiables

- Tiempo límite
  - Si es muy pequeño, se sospechará frecuentemente de los procesos
  - Si es muy grande, hay que esperar mucho para poder sospechar de un proceso
  - Valor debe reflejar las condiciones de demora observadas en la red
    - Ej.: si detector de fallos recibe un “*p* está vivo” en 20 segundos, y no en 10 como lo esperaba, puede hacer *T* = 20 segs.

# Detectores Confiables

- Algoritmo anterior puede ser usado para obtener un detector confiable en un sistema síncrono
- *D* no es estimado, sino real (conocido)

# Uso

- Detectores confiables
  - Pocos sistemas son síncronos
- Detectores *no confiables*
  - No son precisos
- Entonces, ¿Qué tan prácticos son los detectores de fallos?
  - Detectores no confiables, bajo ciertas propiedades bien definidas, ayudan a proporcionar soluciones prácticas a problema de coordinación de procesos en presencia de fallos

# Exclusión Mutua Distribuida

# Exclusión Mutua Distribuida

- Muchos procesos quieren tener acceso a un recurso compartido
- ¿Cuál es el resultado si múltiples actualizaciones se ejecutan en el recurso compartido?
  - Inconsistencia
    - Ej.: actualizaciones en bases de datos
- Comúnmente servidores administran recursos para lograr exclusión mutua

# Exclusión Mutua

- Recurso
  - Debe poder ser usado exclusivamente
  - Necesidad de exclusión mutua para asegurar validez
- Sección crítica
  - Parte del procesamiento donde se usan los recursos compartidos

# Problema de Sección Crítica

- Se necesita ejecutar las secciones críticas bajo exclusión mutua
- Asegura validez, ya que a lo mucho un proceso a la vez puede modificar recurso
- Problema común en sistemas operativos
  - Soluciones usan variables compartidas y/o dependencia de un *kernel* local

# Problema de Sección Crítica

- En sistemas distribuidos hay 2 alternativas:
  - Un servidor administra el recurso
    - Servidor puede proporcionar mecanismos locales (ej.: usando el S.O.) para exclusión mutua
  - No hay un servidor que administre el recurso
    - Ejemplo: servidores sin estado (ej.: bloqueo de archivos en NFS), acceso a un medio compartido (ej.: Ethernet)
    - Debe resolverse únicamente por paso de mensajes

# Algoritmos

- Conjunto $N$ de procesos $p_i$
  - No comparten variables
  - Accesan recursos comunes, pero solo en la sección crítica
- Asumimos que
  - Sistema es asíncrono
  - Procesos no fallan
  - Entrega de mensajes es confiable
    - Entrega eventual, exactamente una vez e íntegra

# Algoritmos

- Sección crítica se protege de la siguiente manera:

```
Entrar() // se bloquea entrada
Sección crítica
Salir() // otros procesos pueden entrar ahora
```

- Requerimientos de exclusión mutua:
  - Seguridad
  - Viveza

# Requerimientos

- ME1: Seguridad
  - A lo mucho un proceso puede estar en la sección crítica en cualquier momento
- ME2: Viveza
  - Requerimientos de entrar y salir de la sección crítica deben tener éxito eventualmente
- ME3: Requerimiento adicional: orden causal
  - Entradas a la sección crítica deben respetar el orden causal de los pedidos

# Propiedades

- Exclusión mutua (ME1)
- No ocurren interbloqueo (deadlocks) (ME2)
- No hay inanición (starvation) (ME2)
- Equidad (ME3)

# Evaluación del Rendimiento

- ¿Cómo comparar algoritmos de exclusión mutua?
  - Ancho de banda
    - Número de mensajes enviados en cada operación de *entrar* y *salir*
  - Demora para el cliente
    - Tiempo de espera en cada *entrada* y *salida*
  - Demora de sincronización
    - Tiempo entre *salida* y *entrada* del siguiente proceso

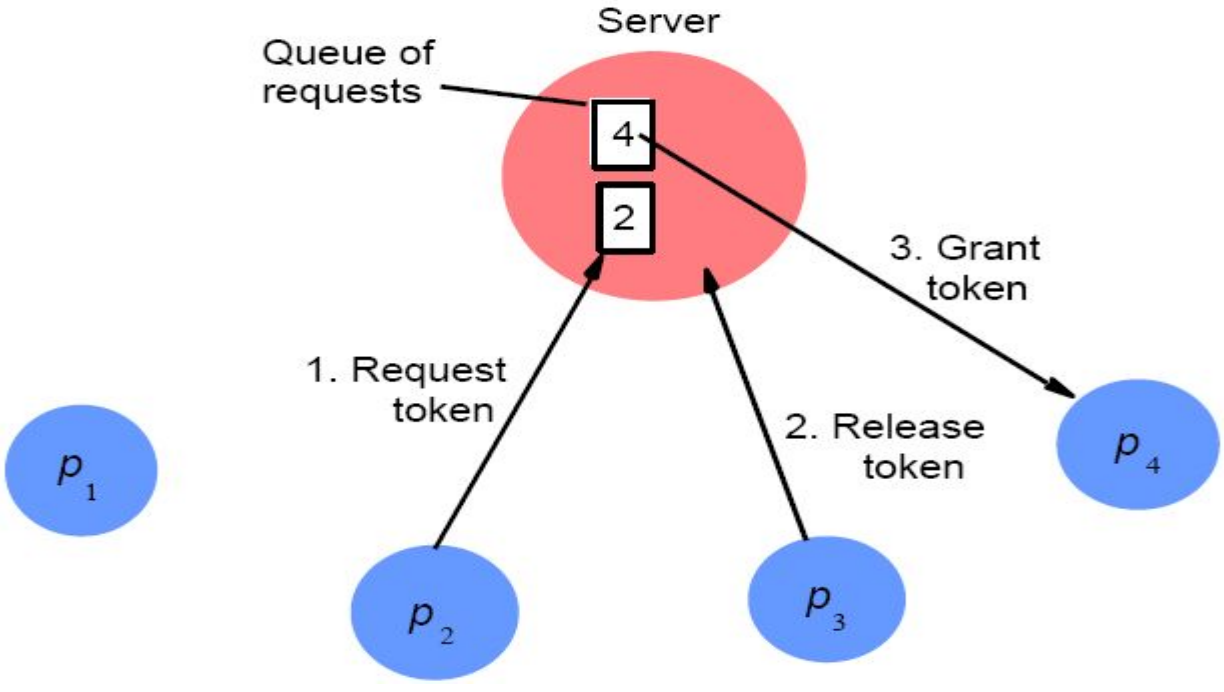
# Enfoque Centralizado

- Uno de los procesos del sistema coordina las entradas a la sección crítica
- Un proceso que quiere entrar a la sección crítica envía un pedido al coordinador
- El coordinador decide qué proceso puede entrar a la sección crítica, y le envía una *respuesta* a ese proceso
- Cuando un proceso recibe un mensaje de respuesta del coordinador, entra en la sección crítica
- Al salir de la sección crítica, proceso envía un mensaje de *soltar* al coordinador

# Comportamiento del Servidor

- Sin pedidos pendientes
  - Esperar por pedidos
- Con pedidos pendientes
  - Encolados en orden FIFO
  - Si el token está prestado
    - Esperar hasta recibir el token
  - Si el token no está prestado
    - Remover la cabeza de la cola y entregarle el token

# Algoritmo de Servidor de Tokens

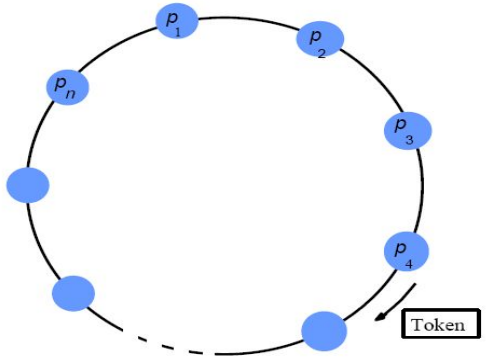


# Caracterización del Algoritmo de Servidor de Tokens

- ME1
  - # de procesos en SC limitado por número de tokens (1)
- ME2
  - Cola FIFO
  - No hay fallas
  - Procesos que entran a la cola serán atendidos eventualmente
- ME3
  - Servidor ignora relación →
  - Procesos atendidos en orden en que mensajes son recibidos
  - Se puede violar condición ME3

# Enfoques totalmente distribuidos

- Algoritmo del anillo
  - Un token circula en el anillo
  - Para entrar a sección crítica: Entrar cuando se recibe el token
  - Para salir de sección crítica: Pasar el token
  - ¿Se cumplen las 3 condiciones (M1-M3)?
- Algoritmo de Maekawa
  - N procesos compañeros usando multicast y relojes lógicos
  - Para entrar a sección crítica:
    - Multicast el pedido
    - Entrar únicamente cuando se ha recibido respuesta de todos los procesos
  - Condiciones de respuesta diseñadas para asegurar que ME1, ME2 y ME3 se cumplan



# Elecciones

## (Leader Elections)

# Elección

- Cómo *elegir* a un componente para que realice un rol particular
- Ejemplo: falla el reloj externo
  - ¿Qué otro componente en el sistema puede realizar la función?

# Función

- $P_i$ llama a una elección, e inicia el algoritmo
  - Un proceso solo llama una elección al mismo tiempo, pero *N* procesos pueden llamar *N* elecciones (la misma)
  - Un proceso participa o no en una elección
  - Proceso elegido necesita ser único aún si muchos procesos llaman a elección
    - Ej.: coordinador falla y dos procesos llaman a elección
- Criterios de selección del proceso
  - Orden total tiene que estar definido en el criterio de selección
    - Proceso con el mayor identificación
    - Proceso con la carga más baja

# Función (cont.)

- Cada $P_i$ tiene una variable $elegido_i$ que contiene un ID del proceso elegido
  - Al inicio $P_i$ participa en una elección, y $elegido_i = \perp$
  - Al final de la elección, $elegido_i = \text{ID}(P_{elegido})$
  - Si un proceso aún no está participando en la elección, $elegido_j = \text{ID}(P_{elegido\_anterior})$

# ¿Quién debe ganar la elección?

- Decisión de qué proceso debe ganar elección debe ser única
- Alternativas
  - Prioridad
    - Basado en carga < 1/carga, *i* >
    - Basado en ancho de banda
    - Basado en alguna función específica
  - ID asignado mayor
  - Dirección IP mayor
  - Proceso con versión más actualizado de estado del SD (Zookeeper)

# Coordinadores (Leaders)

- Coordinador: proceso que realiza alguna función crítica en el sistema
  - Detección de interbloqueos (deadlocks)
  - Mantenía el token en un anillo
  - Encargado de asegurar exclusión mutua
  - Servidor de tiempo
  - Primario en un sistema replicado
  - ...

# Requerimientos

- E1: Seguridad (Safety)
  - Un proceso participante $P_i$ tiene $elegido_i = \perp$ o $elegido_i = P$, donde P es un proceso elegido que no ha fallado (crash), con el identificador más grande
- E2: Viveza (Liveness)
  - Todos los procesos $P_i$ participan y eventualmente asignan $elegido_i \neq \perp$ o han fallado (crash)

# Eficiencia

- Se mide por
  - Consumo total de ancho de banda de la red
    - Proporcional al número de mensajes enviados
  - Tiempo de terminación
    - Tiempos de la transmisión de mensajes serializados entre el inicio y fin de una sola ronda
      - Medido en número de mensajes

# Algoritmo Basado en Anillo

- Organización
  - Colección de $P_i$ organizados lógicamente en un anillo
  - Sistema es asíncrono
  - Meta: elegir un único proceso con el ID más grande como el coordinador
  - No tolera fallas

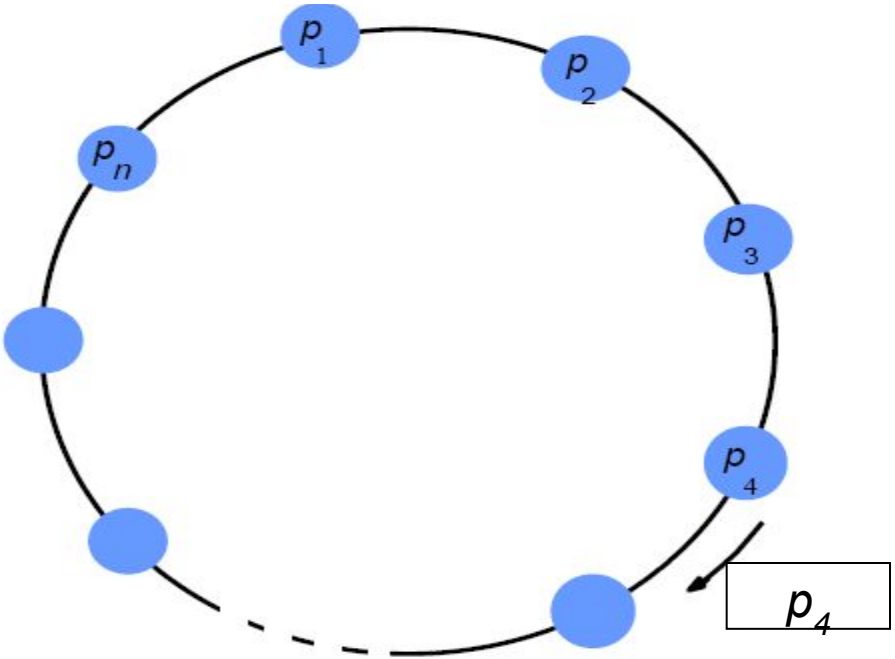
# Algoritmo Basado en Anillo

- Inicialmente: cada proceso no está participando en la elección
- Cualquier proceso $P_i$ puede iniciar la elección
  - Se marca como *participante*
  - Coloca su ID en el mensaje de elección
  - Envía mensaje a vecino (sentido del reloj)

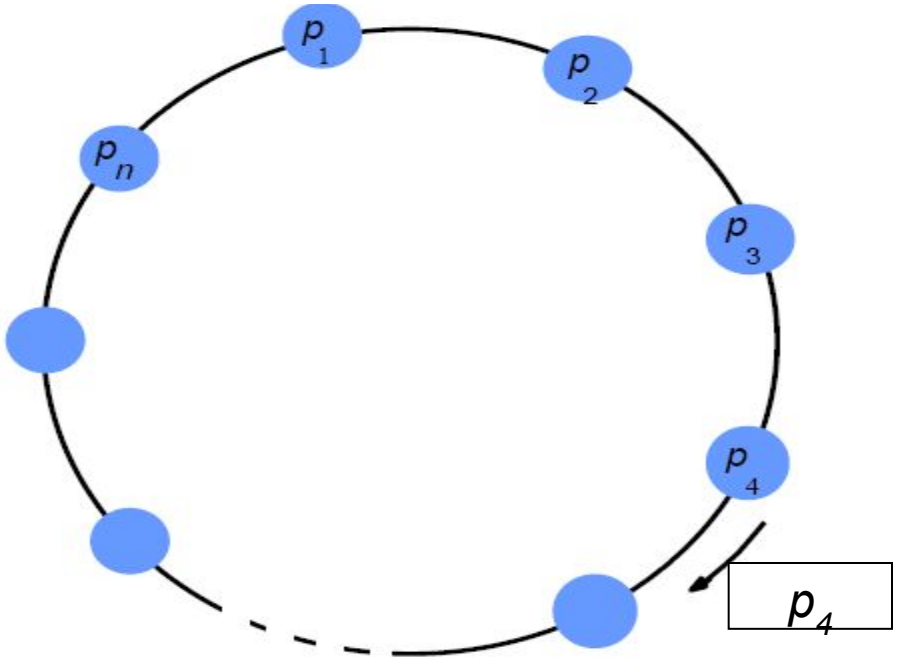
# Algoritmo Basado en Anillo

- Si proceso recibe mensaje de elección, compara el ID en mensaje con propio
  - Si ID recibido > propio, pasa el mensaje y se convierte en *participante*
  - Si ID recibido < propio y no es *participante*, lo reemplaza con el propio y pasa el mensaje
  - Si ID recibido = propio, ahora es el coordinador, deja de participar y envía un mensaje de *elegido*
  - Cualquier proceso $P_j$ que recibe un mensaje *elegido*, deja de participar y setea $elegido_j$ = ID($P_i$)

# Algoritmo Basado en Anillo

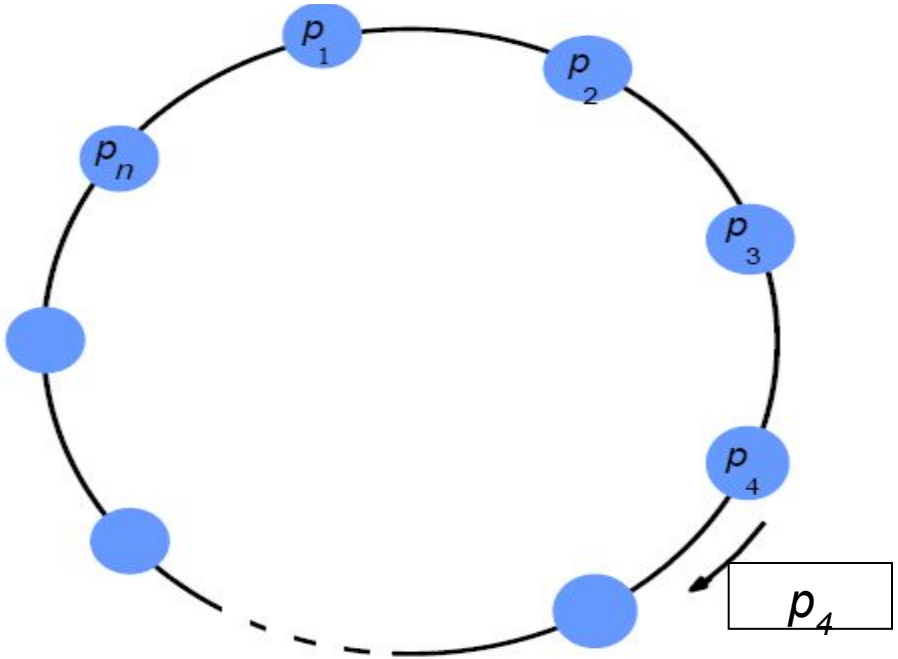


# Algoritmo Basado en Anillo




- ¿Cumple con propiedades E1 y E2?
- ¿Tiempo de terminación?
  - (medido en número de mensajes enviados en total)

# Algoritmo Basado en Anillo




- Cumple con E1: Seguridad
- Cumple con E2: Viveza
- Tiempo de terminación
  - Mínimo: $2N$
  - Máximo: $3N - 1$

# Algoritmo Abusivo

- Nombre en inglés: Algoritmo del “Bully”
- Tolera fallas (crash) de los procesos
- Asume comunicación confiable
- Asume que c/ proceso conoce a los otros, sus IDs y se puede comunicar con ellos
- Sistemas síncronos (usa tiempos de expiración)
  - Usa detector de fallos confiable ($T = 2T_{trans} + T_{procesamiento}$)

# Mensajes

- Tres tipos
  - Elección
    - Para anunciar una elección
  - Respuesta
    - En respuesta a un mensaje de elección
  - Coordinador
    - Para anunciar la identidad del coordinador elegido

# Algoritmo Abusivo

- Si un proceso $P_i$ envía un pedido que no es contestado por el coordinador dentro de un intervalo *T*, asume que coordinador falló y trata de elegirse a sí mismo como el nuevo coordinador
- $P_i$ envía mensaje de *elección* a todos los otros procesos con ID mayor
- $P_i$ espera que estos procesos *respondan* en un tiempo *T*

# Algoritmo Abusivo

- Si no hay respuesta en dentro de *T*, asume que procesos con ID mayores han fallado; $P_i$ se elige a sí mismo como el nuevo coordinador
- Si se recibe una respuesta, $P_i$ inicia intervalo *T'*, y espera recibir mensaje que que proceso con ID mayor ha sido elegido
- Si no llegan mensaje dentro de *T'*, asume que el proceso con ID mayor ha fallado e re-inicia algoritmo

# Algoritmo Abusivo

- Si $P_i$ no es el coordinador, en cualquier momento durante la ejecución, $P_i$ recibirá uno de los dos siguientes mensajes de $P_j$:
  - $P_j$ es el nuevo coordinador ($j > i$); entonces, $P_j$ registra esta información
  - $P_j$ ha iniciado una elección ($i > j$); entonces, $P_i$ envía un mensaje a $P_j$ e inicia su propio algoritmo de elección
- Después de que un proceso que ha fallado se recupera, inmediatamente inicia algoritmo
- Si no hay procesos activos con ID mayor, proceso recuperado forza todos que le permitan ser el nuevo coordinador, aún cuando ya haya un nuevo coordinador

# Algoritmo Abusivo

- Si un proceso ya sabe que tiene el ID mayor, se salta la elección y se anuncia directamente como *coordinador*

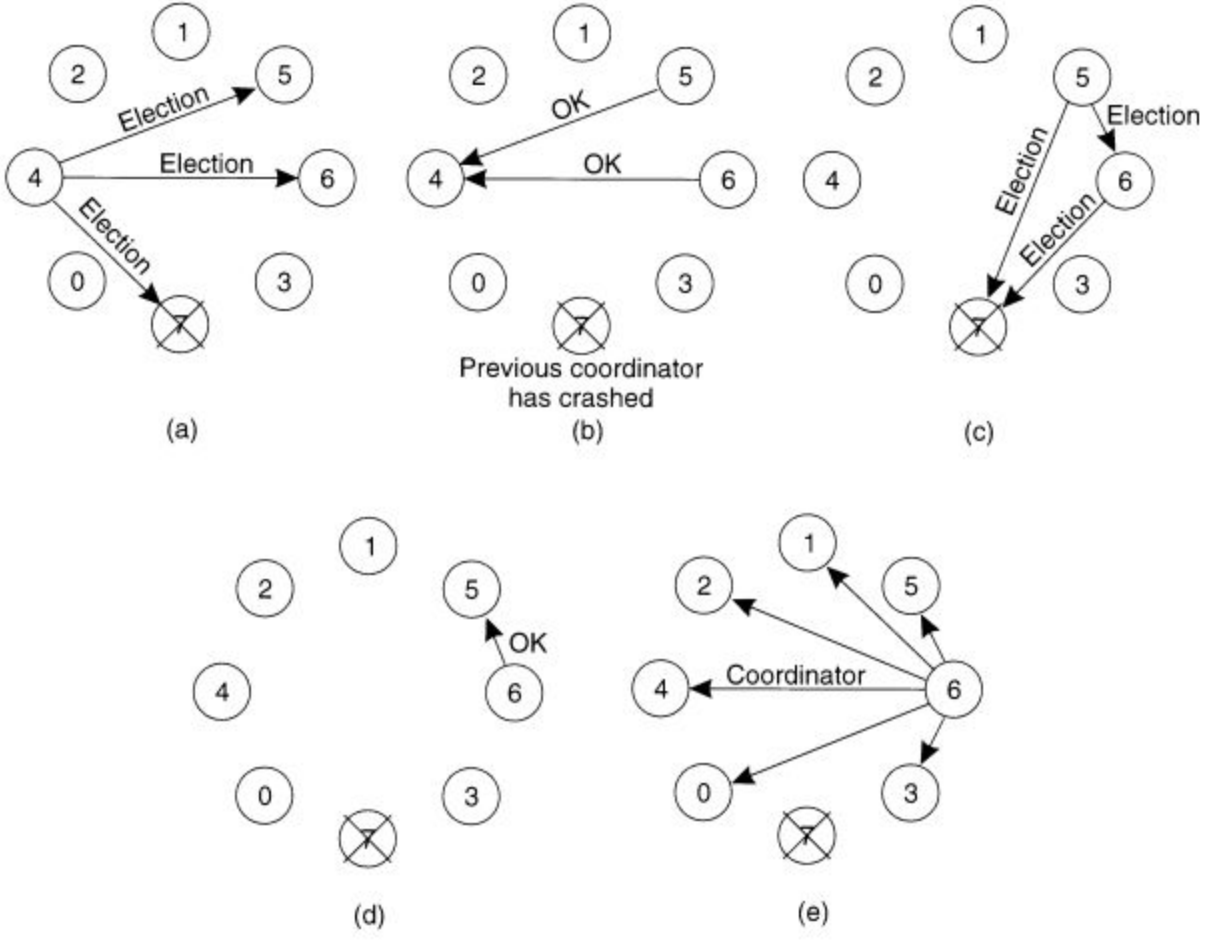



**Figure 5-11.** The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

# Requerimientos

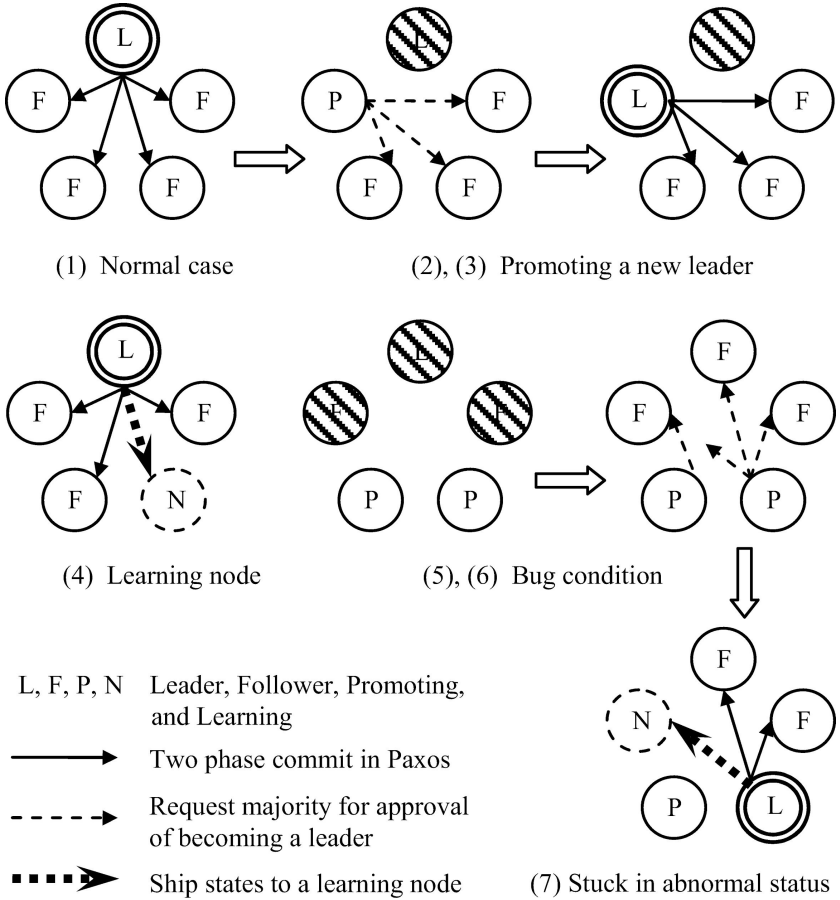
- E1: Seguridad
  - Solo puede haber un coordinador a la vez (ID mayor)
  - No se cumple si *T* está mal estimado
    - Detector de fallos no confiable
- E2: Viveza
  - Si, porque canales son confiables

# Paxos

- Inventado por Leslie Lamport, padre de los sistemas distribuidos
  - Lo inventó en los 80s, pero nadie lo entendía, así que no lo pudo publicar hasta 1998
  - http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos
  - Lamport ganó el Turing Award del 2013 por sus contribuciones: exclusión mutua distribuida, relojes lógicos, acuerdo bizantino, Paxos, TLA, LaTeX, entre otras
    - http://amturing.acm.org/award_winners/lamport_1205376.cfm
- Paxos es una familia de protocolos de consenso
  - Quorum, commits, elecciones, etc.
- ¿Quieren tratar de entender Paxos?
  - No es recomendable, muy complicado :-(
  - Mucho más fácil es entender Raft (https://raft.github.io/)
    - Hay decenas de implementaciones open-source: https://raft.github.io/#implementations
  - Implementación famosa de Paxos: Chubby (de Google, no es open source)
  - Apache Zookeeper tiene su propio protocolo (Zab), muy similar a Paxos

Otras referencias: https://www.quora.com/Distributed-Systems-What-is-a-good-open-source-Paxos-implementation

# Leader election in Paxos



# Discusión y recomendaciones

- No implementar algoritmos de SD a menos que no exista ninguna versión ya implementada
- Mejores prácticas: Usar middlewares que implementen estos algoritmos
  - Ejemplo: Zookeeper para coordinación y acuerdo, quorums, etc.
  - O, una buena implementación de Raft
- ¿Cómo elige Zookeper a su coordinador?
  - https://www.quora.com/How-is-a-leader-elected-in-Apache-ZooKeeper
  - http://stackoverflow.com/questions/27558708/whats-the-benefit-of-advanced-master-election-algorithms-over-bully-algorithm
- **OJO:** Aunque no lo crean, muchos SD no implementan bien los algoritmos de consenso y por lo tanto no son realmente tan tolerante a fallos (ej.: MongoDB)
  - Problema potencialmente grave: https://aphyr.com/posts/284-call-me-maybe-mongodb
  - Zookeeper y Raft sí funcionan bien (hay pruebas matemáticas, no empíricas)
  - https://www.quora.com/If-Raft-is-as-good-as-the-papers-claim-why-do-Zookeeper-and-others-implement-other-consensus-algorithms-Why-not-use-Raft