Capítulo IV: Fundamentos teóricos de computación distribuida

Sincronismo, relojes, causalidad y consistencia de estados globales

Prof. Dr.-Ing. Raúl Monge Anwandter ♦ 2º semestre 2025

Objetivos del capítulo:

Objetivo general:

- Comprender los fundamentos teóricos de la computación distribuida, especialmente sobre aspectos de sincronismo y consistencia de un estado distribuido, como base teórica para resolver problemas complejos en el diseño de sistemas distribuidos

Objetivos de aprendizaje:

1. Comprende los diferentes modelos de sincronismo y coordinación distribuida.
2. Explica los algoritmos de sincronización de relojes de tiempo real de tipo determinísticos y probabilísticos.
3. Aplica los conceptos de relojes lógicos y vectoriales para resolver problemas de coordinación y ordenamiento de eventos de una computación distribuida.
4. Analiza correctamente la consistencia de un estado global.

Agenda

1. Coordinación y sincronismo en sistemas distribuidos
2. Sincronización de relojes de tiempo real
3. Causalidad, concurrencia y Relojes lógicos Lamport
4. Relojes vectoriales
5. Estados globales y consistencia

4.1 Coordinación y sincronismo en Sistemas Distribuidos

Coordinación en Sistemas Distribuidos

Problemas fundamentales de coordinación

Coordinar: *"Unir dos o más cosas de manera que formen una unidad o un conjunto armonioso"* — [RAE]
- Esto implica alcanzar algún tipo de acuerdo.

1. **Sincronización de relojes**. Lograr que relojes asociados a diferentes procesos compartan una base de tiempo común precisa y exacta.
2. **Simultaneidad**. Hacer que ciertos eventos o acciones ocurran al mismo tiempo en diferentes procesos del sistema.
3. **Exclusión mutua**. Garantizar que ciertas acciones no se ejecuten simultáneamente en diferentes procesos del sistema.
4. **Ordenamiento**. Ordenar la ocurrencia de ciertos eventos o acciones en diferentes procesos del sistema, de acuerdo a algún criterio o plan.
5. **Consenso**. Ponerse de acuerdo entre varios procesos sobre un mismo valor de una variable o sobre realizar alguna posible acción común.

Modelos de sincronismo

Noción de tiempo para procesos distribuidos y cooperativos

Modelo asincrónico: No hay garantía de tiempo sobre la velocidad de ejecución y/o los retardos en la comunicación; i.e., no existe noción de una medida tiempo.

- **Entrega de mensajes**: Mensajes pueden tardar un tiempo arbitrario en entregarse, pero finito.
- **Velocidad de procesamiento**: Procesos pueden tardar un tiempo arbitrario en ejecutar los pasos, pero finito.
- **Sincronización de relojes**: Relojes reales pueden desviarse arbitrariamente.

  $\therefore \nexists$ noción global de tiempo real.

Modelo sincrónico: Procesos funcionan de forma sincronizada y se garantiza límites estrictos en velocidad de ejecución y/o retardos en la comunicación:

- **Entrega de mensajes**: Se garantiza la entrega de mensajes dentro de un límite de tiempo conocido.
- **Velocidad de procesamiento**: el tiempo que tarda un proceso en ejecutar un paso está acotado.
- **Sincronización de relojes**: Relojes de procesos están sincronizados dentro de límites conocidos.

  $\therefore \exists$ noción global de tiempo real.

Comentario: Si no se conocen límites de tiempo en un sistema, se debe trabajar en un modelo asincrónico.

Ejemplo Nº1: Un problema clásico de coordinación distribuida

Suposición:

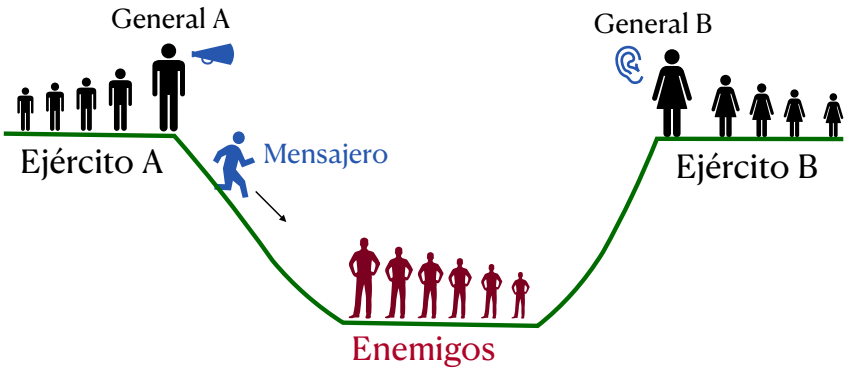
- Dos procesos A y B se comunican enviándose mensajes por un canal bidireccional.
- Ningún proceso puede fallar.
- Los canales pueden experimentar fallas transitorias, con posible pérdida de algunos mensajes, pero garantizando su integridad (no se corroen).

Objetivo:

- Encontrar un protocolo para ponerse de acuerdo sobre realizar una de dos posibles acciones u y v, tal que:
  - Ambos procesos A y B deciden realizar la misma acción ($u \veebar v$), o
  - No se hace nada (ninguna de las acciones se realiza, por falta de acuerdo).

Comentario: ¡Se trata de un problema de acuerdo entre dos procesos usando un canal de comunicación no fiable (modelo de fallas de omisión)!

La paradoja de los dos generales



Condiciones:

- Ataque de ejércitos A y B debe ser coordinar para tener éxito y derrotar al enemigo.
- Mensajeros pueden ser interceptados (comunicación no es fiable)
- Decisión de iniciar un ataque deber ser por consenso (acuerdo).

Problema: ¿Existirá algún protocolo que permita llegar a un acuerdo a ambos generales, después de intercambiar un número finito de mensajes?

Fuente: Jim Gray, "Notes on Data Base Operating Systems", *Operating Systems, An Advanced Course*. Springer, pp. 393–481, 1978.

Demostración de no existencia de tal protocolo

Demostración por el absurdo

- **Suposición inicial:**
  - Se asume que existe un protocolo con un intercambio mínimo de n mensajes.
- **Consecuencia:**
  - Entonces consenso se debe haber alcanzado en mensaje $\#(n-1)$, pues n-ésimo mensaje se pudo haber perdido.

$\therefore$ Existe una contradicción $\Longrightarrow$ ¡No existe tal protocolo!

Ejemplo Nº2: Problema de elección de un líder

Con un modelo trivial de fallas

Supongamos:

- Sea un conjunto de n procesos $P_1, P_2, \cdots, P_n$, donde se debe seleccionar a un líder.
- $\forall i : 1 \leq i \leq n : P_i$ tiene un identificador único $uid(i)$;

  $\therefore$ identificadores definen un ordenamiento total de los procesos en el sistema.
- Todos los procesos comienzan simultáneamente el algoritmo (digamos en $t = 0$).
- Los procesos y la comunicación son totalmente fiables.

Objetivo:

- Diseñar un protocolo donde todos los procesos aprenden la identidad del líder.

a) Solución en un modelo asincrónico del sistema

ALGORITMO:

- Cada proceso P_i difunde (*broadcast*) a todos los procesos ($\forall i : 1 \leq i \leq n$) un mensaje que contiene su identificador $uid(i)$.
- Cuando un proceso P_j reciba n mensajes, decide que el líder es aquel proceso P_k con menor $uid(k)$.

CORRECTITUD:

- Cada proceso va eventualmente a recibir exactamente un mensaje por proceso (total: n mensajes), dado que la comunicación es fiable y la entrega ocurre en tiempo finito.

$\therefore$ Todos los procesos tendrán el mismo conjunto de n mensajes y c/u puede independientemente elegir al mismo proceso P_k: con menor $uid(k)$; u otro criterio, acordado previamente por todos.

Observación: Nótese que n difusiones de mensajes son requeridas para una elección.
$\therefore$ Complejidad en mensajes del algoritmo: $\sim O(n^2)$

b) Solución en un modelo sincrónico del sistema

RESTRICCIONES DE TIEMPO: Supóngase que T es constante de tiempo conocida, mayor que:

- Mayor retardo para transmitir y entregar un mensaje en el sistema.
- Mayor diferencia de tiempo entre relojes de dos procesos arbitrarios (define precisión de sincronismo).
- Todos los procesos comienzan el algoritmo al mismo tiempo (digamos en $t = 0$).

ALGORITMO:

- Al iniciarse el algoritmo, todo proceso P_i parte un *timer* y espera hasta que se cumpla alguna de las siguiente condiciones:
  (i) Recibe un mensaje de difusión (*broadcast*) de otro proceso menor y detiene su *timer*.
  (ii) **Si** ha transcurrido tiempo $T \cdot uid(i)$ en su reloj (i.e., ocurre *timeout*), **entonces** procede a difundir un mensaje con su $uid(i)$.
- Se elige al primer proceso del cual se recibe un mensaje de difusión.

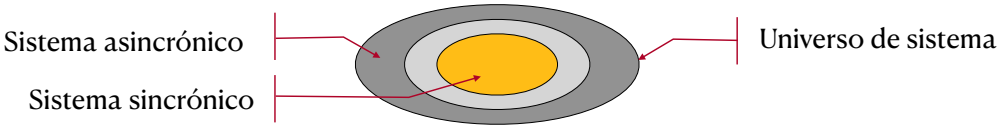
CORRECTITUD:

- En cada ventana de tiempo de tamaño T, llegará a los más un mensaje de difusión.
- Como los procesos por su identificador definen un orden total, el algoritmos terminará antes que transcurra T [s] desde que el proceso con menor identificador difundió su mensaje (que es el elegido por todos).

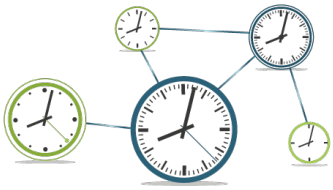
Observación: Nótese existe una única difusión de mensajes para una elección sincrónica.
$\therefore$ Complejidad en mensajes del algoritmo: $\sim O(n)$

Conclusión sobre el modelo de sincronismo

- El **modelo asincrónico** es más general, pues no hace supuestos sobre el tiempo.
  - El **modelo sincrónico** es un caso particular de un sistema (asincrónico), que impone ciertas restricciones temporales para garantizar un determinado comportamiento.
- Conocimiento sobre el tiempo permite, en general, diseñar algoritmos más eficientes (pero podrían ser más complejos de diseñar e implementar).
- Existen problemas que sólo tienen solución en un modelo sincrónico (e.g. consenso con fallas).
- Sincronismo es difícil de lograr en sistema a gran escala y con redes no fiables: en consecuencia, el modelo asincrónico se considera como un modelo más realista.
- Existe una gradualidad de modelos entre asincrónico estricto y sincrónico total.



4.2 Sincronización de relojes de tiempo real



Motivación al uso de relojes

Necesidad de disponer de marcas de tiempos precisas

APLICACIONES DEL TIEMPO: Disponer de una base de tiempo común es fundamental para resolver problemas de coordinación en sistemas distribuidos, tales como ordenamiento de eventos, sincronización de procesos y consistencia de estado. Por ejemplo:

- **Actualización de información** (e.g., saber cuál es última versión de un archivo que se actualiza frecuentemente).
- **Detección de fallas** (e.g., uso de un *timer* para reconocer que no se responde a tiempo y asumir una falla).
- **Causalidad y consistencia** (e.g., ordenar mensajes según sus marcas de tiempo, para una interpretación consistente de la relación causal entre ellos , tal como *e-mails* en el buzón de entrada).
- **Observabilidad y *debugging*** (e.g., monitorear eventos en un sistema, ordenándolos por tiempo de ocurrencia para analizar consistentemente la realidad y explicar correctamente un comportamiento observado).

NECESIDADES DE RELOJES Y MARCAS DE TIEMPO:

- Establecer con certeza cuándo sucedieron ciertos eventos en el sistema, para analizar y concluir consistentemente sobre el comportamiento observado.
- Una marca de tiempo se obtiene con la lectura de un reloj (local) para asociar un tiempo a cada evento de interés.
- Para asociar a los eventos una marca de tiempo con una medida precisa, se requiere sincronizar los relojes de los diferentes procesadores y/o computadores con un alto grado de precisión.
  - si se interactúa con sistemas externos. se requiere sincronización externa con un patrón de tiempo universal o global

Modelo de un reloj real o físico

MODELO DEL SISTEMA: Sistema con n máquinas (o procesadores) y n relojes*. Para cada máquina i, existe un único proceso P_i que controla el reloj local C_i. $\forall i : 1 \leq i \leq n$, se define las siguientes funciones:

- $C_i(t)$: Lectura de reloj C_i en el tiempo real t
- $c_i(T)$: Tiempo real cuando el reloj C_i alcanza el valor T (i.e., el tiempo $t_0 : C_i(t_0) = T$

* **NOTA**: Una simplificación para una arquitectura de computador *multicore* y/o de múltiples procesadores con memoria compartida.

Clases de relojes (físicos):

- **Reloj perfecto**: Si un reloj C_i está siempre perfectamente sincronizado, entonces se cumple:

  $\forall t \geq 0 : C_i(t) = t$

- **Reloj correcto**: En general, para que cualquier reloj C_i $(1 \leq i \leq n)$ esté funcionando correctamente, su tasa de deriva (*drift rate*) está acotada a una constante ρ: Es decir:

  $\forall t \geq 0 : \left| \frac{d}{dt} C_i(t) - 1 \right| < \rho$

- **Reloj fallado**: Un reloj que se detiene o su tasa de deriva excede ρ.

Lectura de reloj local

Ejemplo: Time Stamp Counter (TSC) de arquitectura X86 en lenguaje C

```c
#include <stdio.h>
#include <stdint.h>

// lectura del tiempo con Asembler X86 usando instrucción de máquina RDTSC
// del registro del TSC (Time Stamp Counter) del procesador asignado
static inline uint64_t rdtsc(void) {
   unsigned int lo, hi;
   __asm__ volatile ("rdtsc" : "=a" (lo), "=d" (hi));
   return ((uint64_t) hi << 32) | lo;       // retorna 64 bits de lectura de reloj
}

int main() {
   uint64_t start = rdtsc();
   // ...
   // Código que se le quiere medir tiempo de ejecución
   // ...
   uint64_t end = rdtsc();
   uint64_t cycles = end - start;
   printf("Ciclos TSC transcurridos: %llu\n", cycles);
   return 0;
}
```

OBSERVACIÓN:

- El programa retorna la cantidad de ciclos del contador TSC transcurridos, en un registro de un número entero de $64b$ sin signo.
- Si queremos saber el tiempo transcurrido T, se debe calcular:

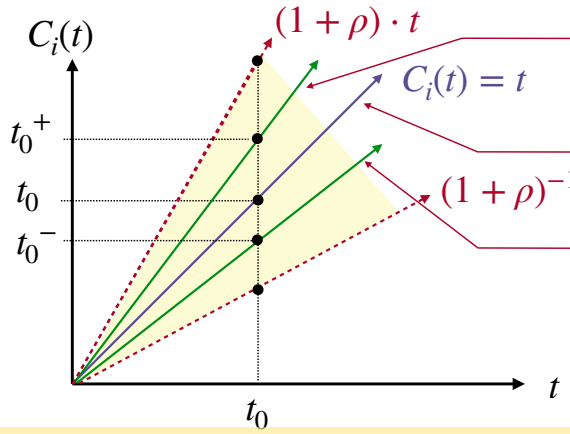
$$T\ [s] = \frac{end - start\ [\text{ciclos TSC}]}{f_{TSC}\ [Hz]}$$

f_{TSC} está determinado por la frecuencia del reloj del procesador. (e.g., $1.8[GHz]$).

Representación del tiempo en el Computador

- **Tiempo en Unix y Posix**: medido en intervalos de 1 [s] desde el 1 de enero de 1970 (época actual). Esta codificado con un entero con signo (i.e. 31 bits de magnitud: aprox. 68 años)
  - e.g., 1.000.000.000 corresponde a $09/Sep/2001, 01:46:40$
  - En 2038 habrá cambio de época (problema parecido al año 2.000)
- **FILETIME en Windows**: medido en intervalos de 100 [ns] desde el 1 de enero de 1601. Representado como entero sin signo de 64 bits.
- **Llamadas al sistema en Unix**: Analizar las diferencias de:
  - *time*() : reloj del sistema (segundos)
  - *gettimeofday*() : reloj del sistema(microsegundos)
  - *rdtsc*() : Contador de ciclos de CPU (TSC) (nanosegundos)
  - *clock_gettime*() : *Timer* de alta resolución del sistema (nanosegundos)

Deriva de un reloj (ρ)



$$(1+\rho)^{-1} \leq \frac{C_i(t) - C_i(t')}{(t-t')} \leq (1+\rho)$$

Observaciones:

- Valores típicos de tasa de deriva en relojes de cristal son del orden $\rho \approx 10^{-5}$ (más de 1 [s] de desviación diaria)
- Relojes atómicos pueden tener una tasa de deriva $\rho < 10^{-11}$ (e.g., menos de 1 [s] de desviación en 6.000 [año] para uso de isótopo Cesio-133).

Problema de sincronización de relojes

Problema básico de sincronización de relojes:

- Cada máquina (o procesador) tiene su propio reloj local, siendo la base de tiempo para sus procesos.
- Si no se hace nada, los relojes se desvían unos de otros (*clock skew*), por inevitable existencia de diferentes tasas de deriva.
- La complejidad de sincronizar relojes es mayor si existen fallas (peor caso: modelo de fallas bizantinas).

Objetivo: Lograr que relojes de diferentes máquinas se coordinen para reducir su desviación del tiempo a un máximo conocido.

Requisitos (propiedades):

- **Precisión** (*precision*): Relojes tienen tiempos cercanos entre si. Formalmente, existe una constante de tiempo pequeña π, tal que:
  $$\forall i, j : 1 \leq i, j \leq n : \left| C_i(t) - C_j(t) \right| < \pi$$
  - en sincronización interna, π corresponde a tiempo de desviación máximo entre cualquier par de relojes.
- **Exactitud** (*accuracy*): Tiempo de cada reloj es cercano al tiempo real. Formalmente, existe una constante de tiempo pequeña α, tal que:
  $$\forall i : 1 \leq i \leq n : \left| C_i(t) - t \right| < \alpha$$
  - en sincronización externa, α es desviación máxima con un reloj patrón de tiempo real (e.g., UTC: *Coordinated Universal Time*).
- **Monotonicidad**. Un reloj siempre se incrementa en el tiempo, i.e.: $\forall i : 1 \leq i \leq n : t < t' \Longrightarrow C_i(t) < C_i(t')$
  - Ajuste de relojes: no tener saltos negativos al sincronizar relojes tal como reducir velocidad por un lapso de tiempo.

Algoritmos de sincronización de relojes

Clases de algoritmos de sincronización de relojes

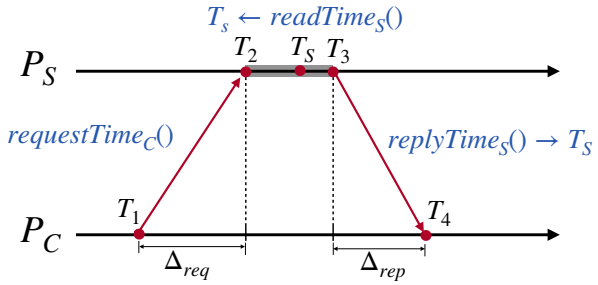
- **Determinístico:**
  - Condiciones y límites para la sincronización entre relojes están garantizados (e.g. precisión alcanzable).
  - Se requiere limitar retardo máximo de los mensajes (tb. tasa de deriva máxima entre relojes no fallados).
  - Ejemplos:
    - Algoritmo de Berkeley (1989): Sincronización en una red interna con tolerancia a fallos.
    - Algoritmo de Lynch (1988). Sincronización entre relojes, garantizando precisión entre nodos no fallados.
- **Probabilístico:**
  - Condiciones y límites para la sincronización entre relojes están garantizados sólo con cierta probabilidad.
  - No se requiere limitar retardo máximo de los mensajes, pero precisión depende de retardos estimados.
  - Ejemplos:
    - Algoritmo de Cristian (1989): Estima desfase temporal (*offset*) basado en retardo de la comunicación.
    - Protocolo NTP (versión 4: RFC 5905): Uso de filtrado estadístico para acotar el desfase temporal.

Métodos de sincronización: (1) Consultando a un servidor; o (2) cooperativos consultando a un grupo de servidores.

Problema estimación en sincronización de relojes

Un cliente P_C usa un servidor de tiempo P_S

- P_C sólo puede leer el reloj físico C_S a través de proceso controlador P_S, enviándole un mensaje y esperando una respuesta (e.g. con una invocación remota), lo que toma tiempo.
- P_C debe estimar la desviación o desfase temporal (*offset*) de su reloj C_C respecto a C_S para luego ajustar su reloj.
- Retardo de comunicación es de tiempo variable, una dificultad para establecer cuándo P_S efectivamente leyó su reloj.
- Relojes pueden fallar, dando diferentes lecturas a procesos lectores (i.e. problema de doble cara).



Posibles estimaciones en P_c:

1. Retardo de comunicación: $\Delta_{req} \approx \Delta_{rep}$.
2. Asumir que P_S lee en: $T_s \approx (T_2 + T_3)/2$ (según C_S).
3. P_c estima en T_4 el retardo medio D de la lectura de P_S como: $D \approx (T_4 - T_1)/2$ (según C_C).
4. El desfase estimado θ_{CS} de reloj C_C respecto a C_S es: $\theta_{CS} = C_C(t) - C_S(t) \approx (T_4 + T_1)/2 - T_S$

a) Algoritmo de Berkeley (Tempo)

ALGORITMO: Un grupo de n servidores, en un esquema maestro-esclavo (dinámico con elección), se sincronizan con un coordinador (o líder). Esquema cooperativo, determinístico y tolerante a fallos.

1. Coordinador (maestro) consulta a cada servidor (máquina esclava) el tiempo de su reloj; luego, estima su *offset* en base a la demora de su respuesta (*round-trip time*).
2. Coordinador descarta *offset* muy desviados, promedia los *offsets* seleccionados y calcula para cada servidor su nuevo *offset* de ajuste (incluido el suyo).
3. Coordinador le envía el *offset* a cada servidor, para que hagan su ajuste.

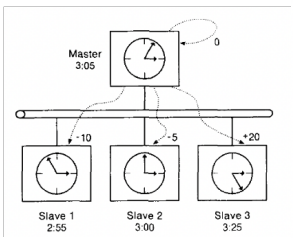
CARACTERÍSTICAS:

- Se logra típicamente una precisión del orden de varios milisegundos (sincronización interna), pero el grupo de servidores sincronizados puede derivar respecto a UTC (i.e. no se considera sincronización externa).
- Permite tolerar fallas de servidor, aislando servidores fallados con excesiva desviación de la mayoría.
- Si fallas no son bizantinas y coordinador no tiene fallas de doble cara, se cumple: $2f + 1 \leq n$

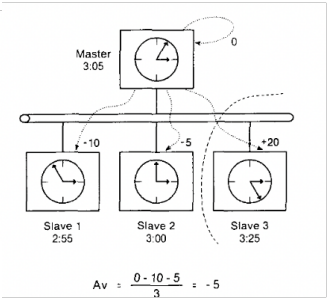
Fuente: R. Gusella & S. Zatti, "The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD," in *IEEE TOSE*, 15(7), pp. 847-853, July 1989.

Algoritmo de Berkeley

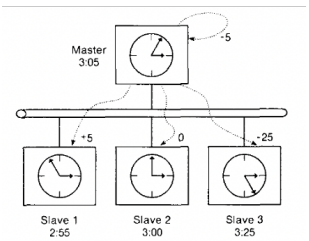
Ejemplo sobre su funcionamiento



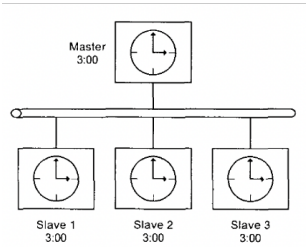
a) Medidas para estimación de *offsets*



b) Cálculo del promedio del *offset*, descartando valores fuera de rango



c) Ajuste de relojes para corregir desviación según *offset* promedio



d) Relojes se encuentran sincronizados

Algoritmo de Berkekey

Estimación del *offset*

CONSIDERACIONES. Estimación del offset θ_{AB} supone:

- $\theta_{AB} = C_a(t) - C_B(t)$
- Se asume que en corto período: $\theta(t_1) = \theta(t_3) = \theta_{AB} = cte$.

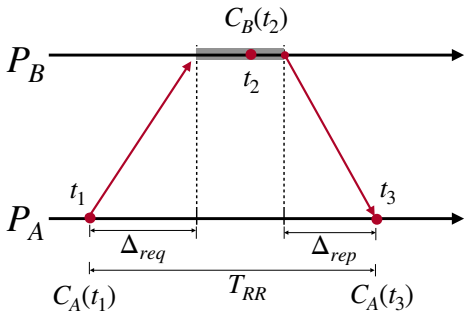
  $\therefore \quad \theta_{AB} \approx 1/2 \cdot \{C_A(t_1) + C_A(t_3)\} - C_B(t_2)$

ESTIMACIÓN DEL OFFSET:

Sea:

- T_{RR} : *round-trip time* ; i.e. $C_A(t_3) - C_A(t_1)$
- T_{min} : tiempo mínimo de comunicación para Δ_{req} y Δ_{rep}
- ϵ : error estimado para el *offset*

Se puede demostrar que error ϵ de estimación de *offset* θ_{AB} está acotado a: $\epsilon \leq 1/2 \cdot T_{RR} - T_{min}$



Observaciones:

- Error ϵ depende de tiempos de comunicación T_{RR}.
  - Si T_{RR} es muy grande, un *timer* puede forzar a repetir la medición.
- Error entre par de nodos queda acotado a 2ϵ.

b) NTP (Network Time Protocol)

Estándar RFC 5905 en Internet

- NTP is uno de los protocolos más ampliamente usados para sincronizar relojes en sistemas distribuidos y es un estándar en Internet.
- Está diseñado para sincronizar los relojes de computadores en una red, asegurando mantener consistencia y precisión.
- Ofrece tres modos de operación:
  - Cliente-Servidor: cliente se sincroniza con uno o más servidores.
  - Simétrico: Dos pares de computadores se sincronizan entre ellos; útil para redundancia.
  - Broadcast/Multicast: Permite difundir períodicamente información para sincronizar relojes en una red de área local.
- Algunas características son tolerancia a fallos con replicación de servidores, organización jerárquica para escalar, autenticación para seguridad, entre otras.

NTP: Modo básico de sincronización

Definiciones (Ambas variables se pueden indexar por #ronda):

- θ : *offset* entre los relojes de los servidores P_A y P_B es:
  $\theta(t) = C_B(t) - C_A(t) \approx \theta$ (se asume constante)
- δ : retardo medio de transmisión por mensaje
  (2δ es promedio de tiempo de entrega de un mensaje)

Cálculos intermedios: Entonces podemos calcular:

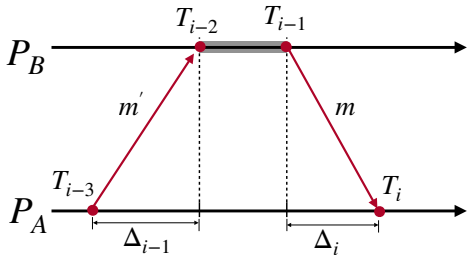
- $T_{i-2} = (T_{i-3} + \theta) + \Delta_{i-1}$ y $T_i = (T_{i-1} - \theta) + \Delta_i$
- $2\delta_i = \Delta_i + \Delta_{i-1} = (T_i - T_{i-1}) + (T_{i-2} - T_{i-3})$

Ahora, se estima el tiempo T_i de recepción de mensaje m y se obtiene *offset* estimado θ_i para ronda i:

- $T_i = (T_{i-1} - \theta_i) + \Delta_i$ o $\theta_i \approx \delta_i - (T_i - T_{i-1})$

Conclusión: Retardo promedio δ_i y *offset* θ_i estimados para ronda #i son:

$$\delta_i = \frac{(T_{i-2} - T_{t-3}) + (T_i - T_{t-1})}{2} \qquad \theta_i = \frac{(T_{i-2} - T_{t-3}) - (T_i - T_{t-1})}{2}$$



Variables para ronda #i

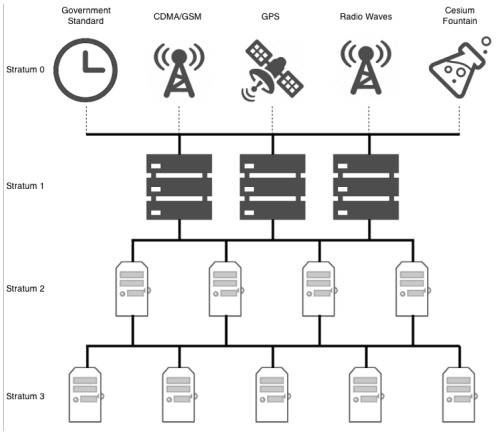
PRECISIÓN:
Se demuestra que: $\theta_i - \delta_i \leq \theta \leq \theta_i + \delta_i$;
(i.e., el error de medición depende de retardo estimado δ_i)

ALGORITMO:

- Se obtiene 8 pares (θ_i, δ_i) sucesivos de estimación (8 rondas)
- Se elige θ_i que tiene menor δ_i como estimación más confiable para θ.

Organización de la Arquitectura NTP

Arquitectura jerárquica de precisión



Estrato: (*stratum*): NTP utiliza una estructura jerárquica de servidores de tiempo, organizados en niveles o estratos:

- **Stratum** 0: Relojes de referencia (e.g., relojes atómicos, GPS)
  - $<$ 50 [ns] en GPS
  - $<$ 2 [ms] en *broadcast* RF
- **Stratum** $1-3$: Servidores sincronizados directamente con dispositivos del Estrato 0 o uno de inferior precisión.

Cada estrato aumenta la posibilidad de error de sincronización.

$<$ 1 [ms] en LAN

$<$ 10 [ms] en Internet pública

$<$ 100 [ms] en ASDL pública (asimétrica)

4.3 Causalidad, concurrencia y Relojes lógicos de Lamport

Fuente: L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Comm. of the ACM* 21(7), 1978, pp. 558-565

Motivación

- Un sistema distribuido se caracteriza, entre otros, por **no tener un base de tiempo precisa y compartida**, lo que desvirtúa el concepto de simultaneidad temporal.
- Muchos problemas se resuelven en base a un razonamiento de lo que pasó antes y después, para establecer **casualidad entre eventos** (i.e., ¿cómo un evento pudo haber influido en la ocurrencia de otro?).
  - Sincronizar relojes reales tiene limitaciones y un costo de coordinación.
- Los **relojes lógicos de Lamport** tiene en sistemas distribuidos múltiples aplicaciones por ser un sistema de reloj más liviano. Ejemplos:
  - Ordenamiento de eventos y ordenamiento de transacciones
  - Seguimiento de causalidad, depuración y monitoreo distribuido
  - Exclusión mutua entre procesos, elección de un líder.

Modelo de un sistema distribuido

MODELO: Sea conjunto de n procesos $P_1, P_2, \ldots, P_n$ (secuenciales, concurrentes y asincrónicos), donde:

1. Cada proceso P_i genera localmente una secuencia de eventos E_i, definida por:

   $\forall P_i\ (1 \leq i \leq n) : E_i \stackrel{def}{=} \{e_{i,1}, e_{i,2}, e_{i,3}, \cdots\}$

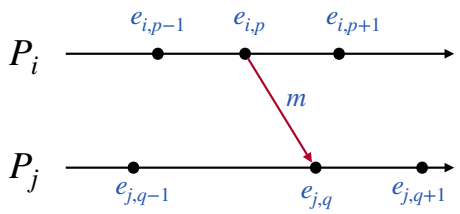
   El conjunto de todos los eventos del sistema de n procesos, se define como: $E \stackrel{def}{=} \{E_1, E_2, \cdots, E_n\}$.

2. La comunicación entre procesos es por paso de mensaje asincrónico, donde los eventos de envío y recepción usualmente ocurren en diferentes procesos.

■

Ejemplo: Si un mensaje m es enviado por P_i y recibido por P_j, se define para ambos eventos:

$e_{i,p} = \text{send}_i(m)$ y $e_{j,q} = \text{recv}_j(m)$



Relación de causalidad →

"Happen before" ("ocurrió antes que")

DEFINICIÓN: Relación "ocurrió antes que" →

La relación → captura la dependencia causal entre dos eventos; i.e., si dos eventos están causalmente relacionados. Se define → por las siguientes tres propiedades:

1) **Localidad**: $\forall a, b \in E, a \neq b : a \rightarrow b$
   si eventos a y b son eventos del mismo proceso y a ocurre en este proceso antes que b.

2) **Comunicación**: $\forall a, b \in E, a \neq b : a \rightarrow b$
   si a es el evento de envío de un mensaje m en un proceso y b es el evento de recepción de m en otro proceso.

3) **Transitividad**: $\forall a, b, c \in E$, eventos diferentes: $a \rightarrow b, b \rightarrow c \Longrightarrow a \rightarrow c$

■

Relación de concurrencia ||

DEFINICIÓN: Relación de concurrencia ||.

- Dos eventos distintos a y b son concurrentes (denotado por $a \parallel b$), si no se afectan causalmente. Formalmente:

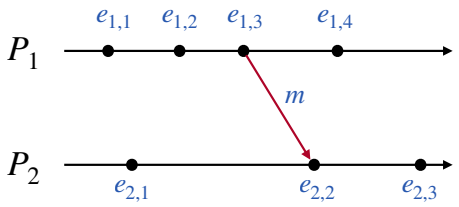
$$\forall a, b \in E, a \neq b : a \parallel b \iff \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

■

Observación: Para l n procesos $P_1, P_2, \ldots, P_n$, eventos $E = \{E_1, E_2, \cdots, E_n\}$ y la relación de causalidad $\rightarrow$, $(E, \rightarrow)$ define una computación distribuida.

Dado que eventos concurrentes en E no se pueden ordenar, se dice que $\rightarrow$ establece sólo un orden parcial sobre los eventos de E.

Ejemplos de causalidad y concurrencia



En el ejemplo, se cumple:

- $e_{1,1} \rightarrow e_{1,3}$ y $e_{1,2} \rightarrow e_{2,3}$
- $e_{1,2} \parallel e_{2,1}$ y $e_{1,4} \parallel e_{2,2}$

Comentarios:

- La relación $\rightarrow$ refleja una potencial influencia causal, pero que no necesariamente ocurre.

Algunas propiedades de relación $\rightarrow$

PROPIEDAD: Concurrencia y causalidad.

- Para dos eventos diferentes a y b cualesquiera, se cumple:

$$\forall a, b \in E, a \neq b : (a \rightarrow b) \veebar (b \rightarrow a) \veebar (a \parallel b)$$

PROPIEDAD: Tiempo y relación de causalidad.

- Sea $T(e)$ el tiempo real y absoluto en que sucedió el evento e (medido con un reloj perfecto). Entonces para dos eventos diferentes a y b cualesquiera se cumple que:

$$\forall a, b \in E, a \neq b : a \rightarrow b \Longrightarrow T(a) < T(b)$$

Observación: Si $T(a) < T(b)$, no es posible afirmar que a y b estén causalmente conectados.

Relojes Lógicos de Lamport

Suposiciones iniciales

- Se asume independencia entre relojes físicos y relojes lógicos.
- En cada proceso P_i del sistema existe un único reloj lógico LC_i (escalar en $\mathbb{N}_0$).
- Cada evento a que ocurre en el sistema recibe un tiempo lógico $LC(a)$, denominada también "marca de tiempo" (*timestamp*).
- $LC()$ puede ser vista como una función que asigna en un proceso a cada evento local a una marca de tiempo $LC(a)$.

Relojes Lógicos de Lamport

Condiciones de Reloj

Las marcas de tiempo de los relojes lógicos para los eventos pueden ser implementados, si se cumplen las siguientes dos condiciones:

[C1]: Para dos eventos a y b pertenecientes al proceso P_i :
Si $a \rightarrow b$, **entonces** $LC(a) < LC(b)$

[C2]: **Si** a es el evento de envío de un mensaje m en el proceso P_i
y b es el evento de recepción del mismo mensaje m en el proceso P_j,
entonces $LC_i(a) < LC_j(b)$

Reglas de implementación

[IR1]: Eventos locales

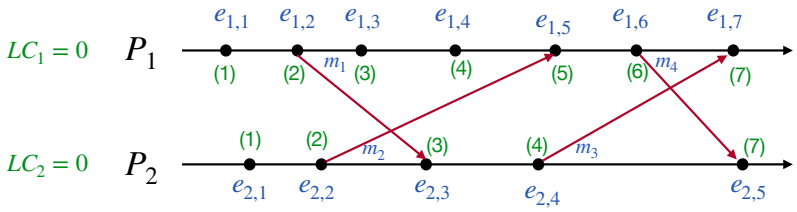
- El reloj LC_i es incrementado para dos eventos locales sucesivos en el proceso P_i, según:
  $LC_i \leftarrow {}'LC_i + d$ (con $d > 0$)
- Si a y b son dos eventos sucesivos en el proceso P_i y $a \rightarrow b$, entonces se cumple:
  $LC_i(b) = LC_i(a) + d$

[IR2]: Eventos de comunicación

- Si a es el evento de envío de un mensaje m en el proceso P_i, entonces m obtiene como marca de tiempo:
  $LC(m) \leftarrow LC_i(a)$
- Al recibirse el mismo mensaje m en el proceso P_j, su reloj LC_j es incrementado a:
  $LC_j \leftarrow max\{{}'LC_j, LC(m)\} + d$ (con $d > 0$)
- Y si b es el evento de recibo del mensaje m en P_j, entonces se cumple:
  $LC_j(b) = LC_j$ (usando valor actualizado del reloj lógico).

Ejemplo: marcado de eventos

INICIALMENTE:



Se asume siempre que incremento: $d = 1$

Ordenamiento total de eventos

Aplicación de Relojes Lógicos de Lamport

Objetivo:

- Encontrar un orden total sobre los eventos de un sistema E, para resolver problemas de decisión con control distribuido.
- Todos los procesos observan el mismo orden para todos los eventos de E.

Aplicaciones:

- Ordenar mensajes, respetando causalidad
- Transacciones distribuidas
- Observación o monitoreo de un sistema

Relación de orden total $\prec$

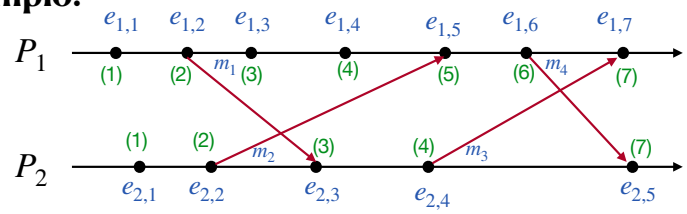
DEFINICIÓN: **Relación de orden total** $\prec$. Se asume existencia de orden total para procesos y sean a un evento en P_i y b un evento en P_j. Entonces, $\forall a \in E_i, b \in E_j, a \neq b$, se define $a \prec b$ por dos reglas:

(i) $LC_i(a) < LC_j(b)$, o

(ii) $LC_i(a) = LC_j(b)$ y $P_i < P_j$

■

Ejemplo:



Observaciones:

- Si $a \rightarrow b \Longrightarrow a \prec b$
- Eventos concurrentes se ordenan "arbitrariamente", pero en forma única.
- Cualquier observador aplicando $\prec$ generará el mismo orden para $(E, \rightarrow)$.

Si $P_1 < P_2$, entonces los eventos se ordenan así: $< e_{1,1}, e_{2,1}, e_{1,2}, e_{2,2}, e_{1,3}, e_{2,3}, e_{1,4}, e_{2,4}, e_{1,5}, e_{1,6}, e_{1,7}, e_{2,5} >$

Limitaciones de Relojes Lógicos de Lamport

Problemas:

1. No es posible determinar causalidad o concurrencia entre dos eventos cualesquiera si sólo se conocen las marcas de tiempo de un Relojes Lógico de Lamport.
2. Existe falta de densidad (no es posible saber a partir de las marcas de dos eventos si existe o no otro evento con una marca intermedia, especialmente si $d > 1$).

Observaciones:

- Relojes reales sufren también de ambos problemas.
- Además, relojes reales sólo garantizan ordenamiento total (respetando causalidad), si existe sincronismo perfecto (una imposibilidad).
- La diferencia sustancial radica en que el valor de un reloj lógico en principio no tiene relación con el tiempo real y no es una métrica de tiempo.

4.4 Relojes vectoriales

Fuente: O. Babaglou, K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms". Technical Report UBLCS-93-1. January 1993. Laboratory for Computer Science. University of Bologna. Bologna (Italy).

Motivación a relojes vectoriales

Ejemplos aplicación

- **Monitoreo y depuración distribuidas** (e.g. causalidad de eventos, análisis de concurrencia y cuellos de botella)
- **Control de consistencia de réplicas** (e.g. sistemas de archivos, bases de datos)
- **Control de versiones** (e.g. Git, edición colaborativa de documentos, copias en *cache*)
- **Orden causal de mensajes o eventos** (e.g. sistemas de mensajería, consenso distribuido)
- **Juegos multi-jugador en línea** (sincronizar eventos o acciones de jugadores entre servidores)

Historia causal

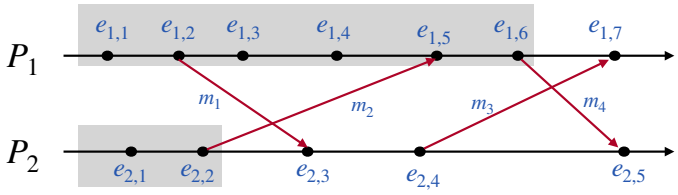
DEFINICIÓN: Historia causal de un evento. En una computación distribuida $(E, \rightarrow)$, la historia causal de un evento $e \in E$ —denotado como $h(e)$—, se define como:

$$h(e) \stackrel{def}{=} \{e' \in E \mid e' \rightarrow e\} \cup \{e\}$$

■

PREGUNTA:

¿Cuál es la historia causal $h(e_{1,6})$?



RESPUESTA: $h(e_{1,6}) = \{e_{1,1}, e_{1,2}, e_{1,3}, e_{1,4}, e_{1,5}, e_{1,6}, e_{2,1}, e_{2,2}\}$

Fuente: Schwarz, R. & Mattern, F., "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", *Distributed Computing*, Springer Verlag, 1994.

Historias causales

Propiedades

Propiedad Nº1: $\forall e, e' \in E, e \neq e' : [e' \rightarrow e] \iff [h(e') \subset h(e)]$

Propiedad Nº2: $\forall e, e' \in E, e \neq e' : [e' \parallel e] \iff [h(e') \not\subset h(e)] \wedge [h(e) \not\subset h(e')]$

COMENTARIOS:

- Las historias causales permiten detectar si existe concurrencia o causalidad entre 2 eventos.
- Una condición de reloj más fuerte es posible realizarla considerando las dos propiedades anteriores, cuya implementación denominaremos “relojes vectoriales”.
- Los relojes vectoriales fueron propuestos "concurrentemente" por Fidge y Mattern, en 1988.

Fuentes primarias:

- C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Eleventh Australian Computer Science Conference*, pp. 55–66, U. of Queensland, February 1988.
- F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pp. 215–226. North-Holland, October 1989.

Proyecciones de una historia causal

DEFINICIÓN: Proyección de una historia causal. Para un evento $e \in E$, la proyección de $h(e)$ sobre E_i, denotada por $h_i(e)$, se define como: $h_i(e) \stackrel{def}{=} h(e) \cap E_i$

Observaciones:

1. Todos las proyecciones de $h(e)$ definen una partición; i.e.: $\bigcup_{i=1}^{n} h_i(e) = h(e)$ y $\bigcap_{i=1}^{n} h_i(e) = \phi$
2. Para la historia causal de un evento e vale: $\forall e_{i,k} \in E_i : \{e_{i,k} \in h_i(e) \iff \forall j, j < k : e_{i,j} \in h_i(e)\}$

   $\therefore$ Basta un número entero único para representar $h_i(e)$, el último evento por proceso.

Ejemplo: usando la figura del ejemplo anterior.

- Se tiene dos proyecciones: $h_1(e_{1,6}) = \{e_{1,1}, e_{1,2}, e_{1,3}, e_{1,4}, e_{1,5}, e_{1,6}\},\ h_2(e_{1,6}) = \{e_{2,1}, e_{2,2}\}$
- $h_1(e_{1,6}) \cup h_2(e_{1,6}) = h(e_{1,6})$ y $h_1(e_{1,6}) \cap h_2(e_{1,6}) = \phi$

Reloj vectorial (tb. Vector de tiempo)

Definición

DEFINICIÓN: Reloj vectorial (VC). Sean n procesos; VC es un vector n-dimensional que a un evento e le asigna una marca $VC(e)$ que representa a $h(e)$, tal que:

$$\forall i, 1 \leq i \leq n : \ VC(e)[i] = k \implies e_{i,k} \in h_i(e) \wedge e_{i,k+1} \notin h_i(e)$$

Se dice que "$VC(e)$ es la marca del reloj vectorial para el evento e"

■

OBSERVACIÓN:

- $\forall e \in E : VC(e)$ representa exactamente $h(e)$.
- Considerando propiedades anteriores 1 y 2, es posible determinar con los relojes vectoriales si existe causalidad o concurrencia entre un par de eventos cualesquiera.

Reloj vectorial

Modelo de implementación

- Como en relojes lógicos de Lamport, cada proceso P_i del sistema puede mantener un reloj local VC_i que refleje el conocimiento global que tiene el este proceso sobre los eventos ocurridos en el sistema.

  $\therefore$ Existirán n relojes vectoriales en el sistema (uno por proceso), donde cada vector tendrá n dimensiones (conocimiento sobre cada proceso).

- A cada evento local y de envío de un mensaje de un proceso P_i, se le puede asociar una marca de tiempo igual al valor que tiene el reloj vectorial local VC_i .
- Inicialmente: $\forall i, j : 1 \leq i, j \leq n : VC_i[j] = 0$

Reloj vectorial

Interpretación operacional

- Cada proceso P_i asignará una marca de tiempo a cada uno de sus eventos locales o de comunicación e, tal que:
- $\forall i : 1 \leq i \leq n : \ VC_i(e)[i]$:

  Nº de eventos que han ocurrido en P_i cuando sucede evento e
- $\forall i, j : 1 \leq i, j \leq n, i \neq j : \ VC_i(e)[j]$:

  Nº de eventos que P_i conoce de P_j cuando sucede e

Reloj vectorial

Reglas de implementación

[IR1]: Evento local. Si $e_{i,k}$ es un evento interno o de envío en el proceso P_i, entonces:

1. $VC_i(e_{i,k})[i] \leftarrow VC_i(e_{i,k-1})[i] + 1$
2. $\forall j, i \neq j : VC_i(e_{i,k})[j] \leftarrow VC_i(e_{i,k-1})[j]$

En caso de ser $e_{i,k}$ evento de envío de mensaje m, entonces m llevará una marca de reloj $VC(m)$ igual a la marca del evento de envío (local); i.e. $VC(m) = VC_i(e_{i,k})$.

[IR2]: Evento de comunicación. Si en el proceso P_i el evento $e_{i,k}$ es un evento de recepción de un mensaje m con marca de tiempo $VC(m)$, entonces:

- $VC_i(e_{i,k}) := \begin{cases} i)\ VC_i(e_{i,k})[i] \leftarrow \max\{VC_i(e_{i,k-1}), VC(m)\} \\ ii)\ VC_i(e_{i,k})[i] \leftarrow VCi(e_{i,k})[i] + 1 \end{cases}$

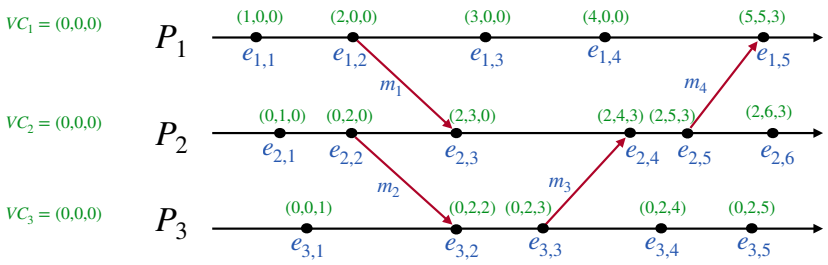
OBSERVACIÓN:

- Se ha tratado la recepción de un mensaje como un evento local más del proceso receptor P_i.
- Valor del reloj para otro proceso debe ser igual a marca del último evento conocido por el proceso receptor.

Ejemplo: Relojes vectoriales

Marcado de eventos con relojes vectoriales

Inicialmente:



Condición fuerte de reloj

Relación de orden de marcas de relojes vectoriales

DEFINICIÓN: Relación de igualdad y de orden para relojes vectoriales.

a) $V = V' \equiv \{\forall k : V[k] = V'[k]\}$

b) $V < V' \equiv \{V \neq V'\} \wedge \{\forall k : V[k] \leq V'[k]\}$

■

Nótese que $\neg(V < V')$ no es equivalente a decir: $(V' < V) \vee (V' = V)$

PROPIEDAD: Condición fuerte de reloj.

a) $e \rightarrow e' \equiv VC(e) < VC(e')$

b) $e \parallel e' \equiv \neg\{VC(e) < VC(e')\} \wedge \neg\{VC(e') < VC(e)\}$

■

Propiedades de Relojes Vectoriales

PROPIEDAD: Número de eventos precedentes. Sean $e_i \in E_i$ y marca vectorial del evento $VC(e_i)$. Entonces el el número total de eventos e' que preceden a e_i (i.e., cumplen que $e' \rightarrow e_i$), está dado por:

$$\left(\sum_{j}^{n} VC(e_i)[j] \right) - 1$$

PROPIEDAD: Detección de apertura. Sean $e_i \in E_i$ y $e_j \in E_j$ dos eventos conocidas con $i \neq j$ (i.e., ambos eventos suceden en diferentes procesos). Entonces:

$$VC(e_i)[i] < VC(e_j)[i] \implies \exists e' \in E_i : e_i \rightarrow e' \rightarrow e_j$$

OBSERVACIÓN: Esta última propiedad permite a un proceso P_j detectar que existe un evento e' que ocurre en otro proceso P_i después que un evento e_i ya conocido por P_j.

Evaluación de Relojes vectoriales

Capacidades:

- **Detección de causalidad:** Permite establecer dependencia causal entre eventos.
- **Detección de concurrencia**: Permite establecer que eventos no tienen relación causal.
- **Coordinación centralizada**: Un mecanismo de marcas de tiempo en sistemas (relativamente) centralizados.
- **Tolerancia a fallos**: Al ser descentralizado, funciona bien en sistemas sin un único punto de falla.

Limitaciones:

- **Overhead:** Tamaño de vectores consumen memoria y ancho de banda; por lo mismo, no escala bien para grandes sistemas (con aumento de Nº de procesos).
- **Complejidad**: Operaciones son más complejas que marcas escalares, tal como en relojes de tiempo real y relojes lógicos de Lamport.
- **Sistemas dinámicos**. Es difícil manejar los vectores si proceso entran o salen del sistema, requiriendo redimensionar su tamaño.
- **Modelo de fallas**. No tolera fallos bizantinos.

Caso: Monitoreo de una computación distribuida y observaciones válidas

Monitoreo de una computación distribuida

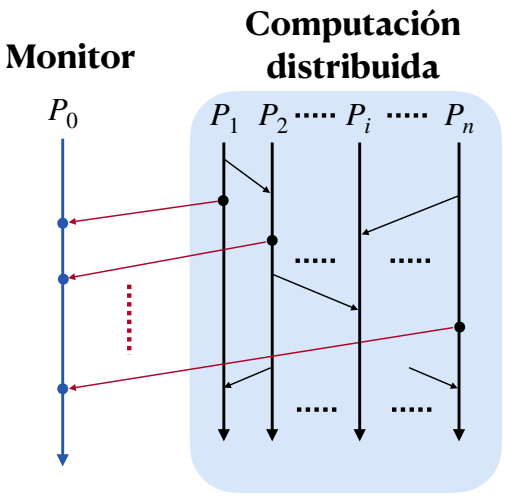
Concepto de observación válida

OBSERVACIÓN: Una observación de una computación distribuida$(E, \rightarrow)$, en general, impone un cierto orden arbitrario a los eventos, que no necesariamente respeta el orden parcial establecido por la relación $\rightarrow$. Denotaremos por $e_i \prec e_j$ para expresar que "e_i se observa antes que e_j".

DEFINICIÓN: Observación válida. Para que una observación sea válida sobre $(E, \rightarrow)$, debe cumplirse que se respete en toda observación el orden parcial de los eventos sobre. Es decir :

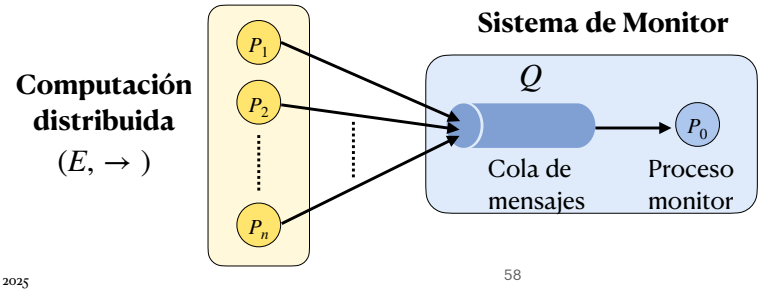
$$\forall e_i, e_j \in E, i \neq j : e_i \rightarrow e_j \implies e_i \prec e_j$$

$\therefore$ no se impone orden a eventos concurrentes.



Modelo de implementación

- Todos los procesos P_i $(1 \leq i \leq n)$ de $(E, \rightarrow)$ envían mensajes a un monitor P_0, notificándole sobre los eventos observables (de interés para monitorear).
- Cada mensaje m enviado por un proceso P_i al monitor P_o lleva una marca $VC_i(m)$.
- P_0 mantiene una cola de mensajes Q, conjunto de mensajes recibidos y no entregados.
- Cola Q está inicialmente vacía.



Condiciones de entrega causal

Concepto de recepción y ordenamiento en entrega de mensajes

CONCEPTO: Sea $\text{send}(m) \in E_j$ (mensaje m enviado desde proceso P_j) y $m \in Q$.
Entonces, para asegurar Observación válida es seguro entregar m en monitor P_0, ssi:

$$\nexists m' : (m' \in Q) \vee (m' \text{en tránsito en la red}) : \text{send}(m') \rightarrow \text{send}(m)$$

Para cumplir lo anterior, es suficiente verificar antes de entregar m a P_0 que se cumplan dos condiciones:

1. **Condición FIFO** (para mensajes enviados por un mismo proceso P_j)
2. **Condición causal** (mensajes enviados desde cualquier otro proceso $P_k, k \neq j$)

Condiciones de entrega causal

Formalización de las condiciones necesarias y suficientes

Para la entrega correcta (i.e., en orden causal) en P_o del mensaje $m \in Q$ proveniente de P_j (i.e., $\text{send}_j(m) \in E_j$), se deben cumplir dos condiciones:

1. **Condición FIFO (mismo proceso origen)**:
   $$\nexists m' \in E_j : \{\text{send}_j(m') \rightarrow \text{send}_j(m)\} \wedge \{(m' \text{ no ha sido entregado en } P_0\}$$
2. **Condición causal (diferentes procesos de origen)**:
   Si m' es último mensaje entregado de P_k en P_o (i.e. $\text{send}_k(m') \in E_j$ y m' entregado), con $k \neq j$, entonces para entregar m de P_j en P_0, se debe cumplir para procesos diferentes a P_j:
   $$\forall k, k \neq j : \nexists m'', \text{send}_k(m'') \in E_k :$$
   $$\{\text{send}_k(m') \rightarrow \text{send}_k(m'') \rightarrow \text{send}_j(m)\} \wedge \{(m'' \text{ no ha sido entregado en } P_0\}$$

Verificación de condiciones de entrega

Aplicando relojes vectoriales para entregar m proveniente de P_j

Condición FIFO: Fácil de verificar con relojes vectoriales, pues se debe cumplir que:

$VC(m)[j] - 1$: mensajes han sido entregados en P_0 para P_j

Condición de entrega causal: Se realiza aplicando Propiedad de Detección de Apertura (PDA), haciendo $i = k$ y considerando los siguientes eventos:

$e_i = \text{send}_k(m'),\ e' = \text{send}_k(m'')$ y $e_j = \text{send}_j(m)$

Y dado que $e_i, e' \in E_k$ (eventos de proceso P_k diferente a P_j), PDA se puede reescribir como:

$$\exists k, k \neq j : \{VC(m')[k] < VC(m)[k]\} \implies \exists m'', \text{send}_k(m'') \in E_k : \{\text{send}_k(m') \rightarrow \text{send}_k(m'') \rightarrow \text{send}_j(m)\}$$

Luego, para que no exista mensaje m'', se debe negar la expresión anterior, quedando:

$$\forall k, k \neq j : \{VC(m')[k] \geq VC(m)[k]\}$$

Algoritmo de entrega causal (1/2)

Estado de entrega de mensajes en el Monitor (P_0)

DEFINICIÓN: Contador de entrega de mensajes. Usar en monitor P_0 un vector n-dimensional D, que representa a n contadores de mensajes entregados para cada proceso P_j $(1 \leq j \leq n)$.
Es decir:

$D[j]$: número de mensajes entregados para proceso P_j

■

Inicialmente:

$D \leftarrow 0$ (i.e., $\forall i, 1 \leq i \leq n\ .\ : D[i] \leftarrow 0$; no se ha entregado ningún mensaje)

Entrega de un mensaje: Si último mensaje entregado para proceso P_j es el mensaje m, entonces:

$D[j] = VC(m)[j]$

Algoritmo de entrega causal (2/2)

Reglas de entrega de mensajes

Regla Nº1: Entregar en monitor P_o mensaje m desde P_j tan pronto se cumplan las siguientes dos condiciones:

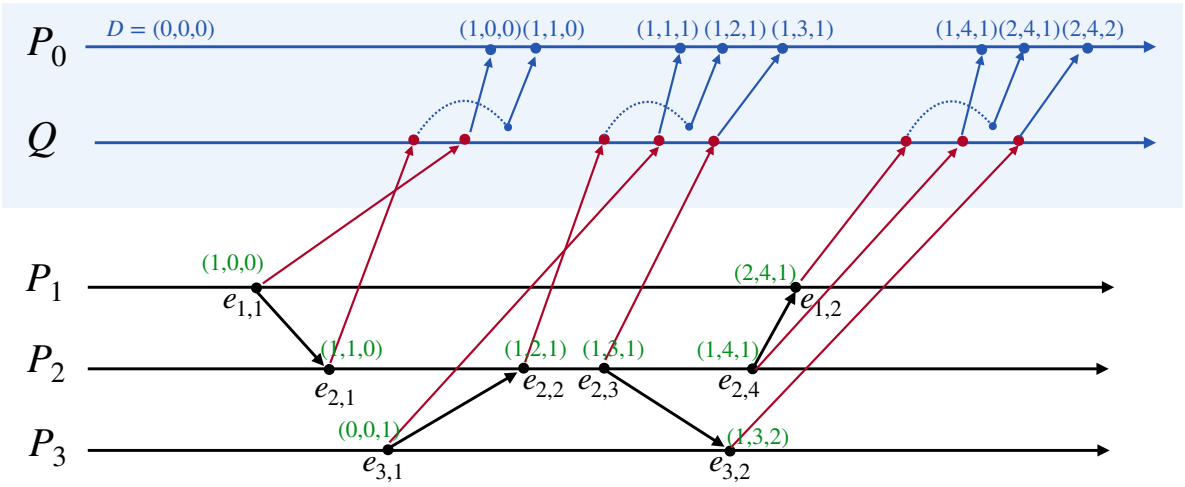
1. $D[j] = VC(m)[j] - 1$ (condición FIFO)
2. $\forall k, k \neq j : \ D[k] \geq VC(m)[k]$ (condición causal)

Regla Nº2: Si se ha cumplido regla anterior y se ha entrega en P_o mensaje m proveniente de proceso P_j, entonces actualizar contador de entrega D:

- $D[j] \leftarrow VC(m)[j]$ (Contador de entrega de mensajes de P_j)

Ejemplo de Monitor con Entrega causal

Generación de una observación válida



4.5 Estados globales y consistencia

Estado Global y Corte

MODELO: Sea un conjunto de procesos $P_1, P_2, \cdots, P_n$ secuenciales y concurrentes, que representan una computación distribuida $(E, \rightarrow)$. Entonces se define:

DEFINICIÓN: Corte. Un corte C de una computación distribuida $(E, \rightarrow)$ es un subconjunto finito de eventos de E, tal que:

$$\forall e, e' : \ (e, e' \in E_i) \wedge (e \in C) \wedge (e' \rightarrow e) \implies e' \in C$$

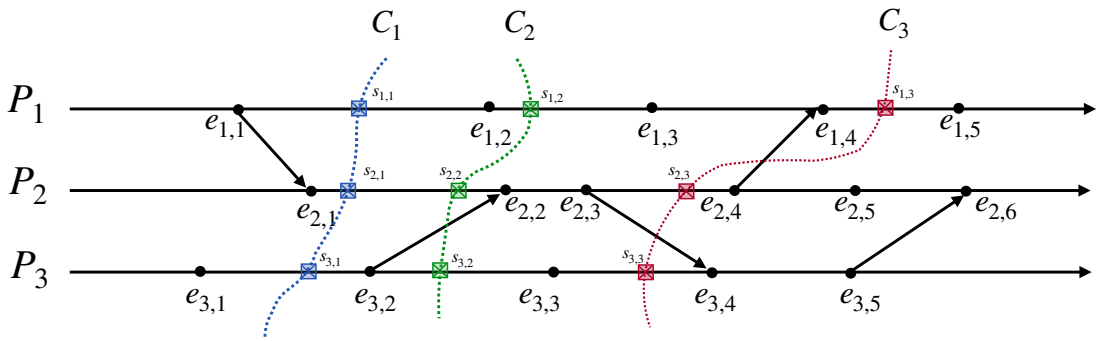
■

DEFINICIÓN: Estado global. El estado global $S = \{s_1, s_2, s_3, \cdots, s_n\}$ asociado al corte C es un conjunto de estados locales (uno por proceso), tal que para cada evento $e_i \in C$:

El efecto del evento e_i está reflejado en el estado s_i de $(E, \rightarrow)$.

■

Ejemplo de Estados globales



$C_1 = \{e_{1,1}, e_{2,1}, e_{3,1}\}$ $\quad S_1 = \{s_{1,1}, s_{2,1}, s_{3,1}\}$

$C_2 = \{e_{1,1}, e_{1,2}, e_{2,1}, e_{3,1}, e_{3,2}\}$ $\quad S_2 = \{s_{1,2}, s_{2,2}, s_{3,2}\}$

$C_3 = \{e_{1,1}, e_{1,2}, e_{1,3}, e_{1,4}, e_{2,1}, e_{2,2}, e_{2,3}, , e_{3,1}, e_{3,2}, e_{3,3}\}$ $\quad S_3 = \{s_{1,3}, s_{2,3}, s_{3,3}\}$

Estados globales consistentes

DEFINICIÓN: Corte consistente. Un corte C, tal que $C \subseteq E$ de una computación distribuida $(E, \rightarrow)$, es consistente ssi:

$$\forall e, e' \in E : \ (e \in C) \wedge (e' \rightarrow e) \implies e' \in C$$

■

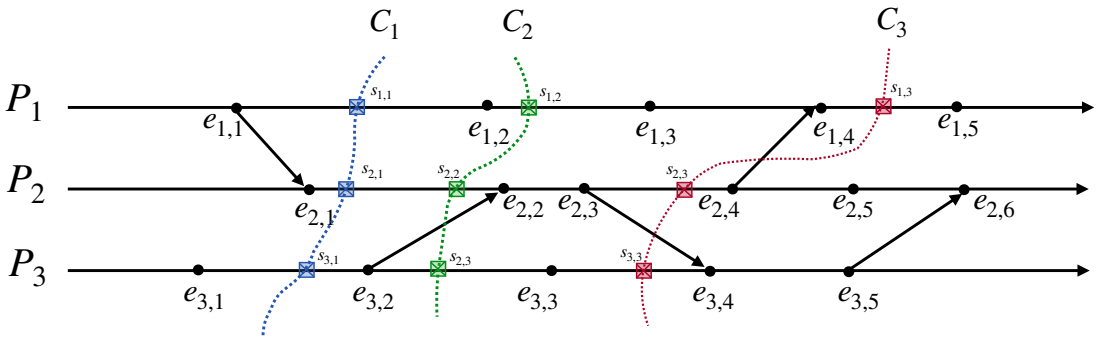
DEFINICIÓN: Estado global consistente. Un estado global S es consistente, ssi S está asociado a un corte consistente.

■

Interpretación operacional de consistencia: Un estado global $S = \{s_1, s_2, s_3, \cdots, s_n\}$ para n procesos es consistente, ssi para cada par de procesos P_i y P_j $(i \neq j)$ se cumple que:

- $\nexists$ mensaje m enviado por P_i después de tomar su estado s_i y recibido por P_j antes de tomar su estado s_j

Ejemplo de consistencia



¡C_1 y C_2 son consistentes!

¡C_3 no es consistente!

Algoritmo de Chandy-Lamport (1/2)

Objetivo: Registrar un estado global consistente de una computación distribuida*, sin detener la ejecución de la computación en curso.

* modelo asincrónico y sin fallas

Suposiciones:

- Procesos se comunican por paso de mensajes unidireccional, usando canales FIFO.
- Topología conocida, que corresponde a un Grafo Dirigido Conectado.
- Alguien comienza el algoritmo (e.g. un proceso controlador o iniciador).
- Cada uno de los procesos es capaz de registrar su estado local.
- Cada proceso es capaz de registrar el estado de todos sus canales de entrada.
- Una vez terminado el algoritmo, el estado de todos los procesos y canales puede ser recolectado para obtener el estado global del sistema (e.g. por el controlador).

Fuente: Chandy, K. M., & Lamport, L., "Distributed snapshots: Determining global states of distributed systems". *ACM Transactions on Computer Systems*, 3(1), pp. 63-75, Feb. 1985.

Algoritmo de Chandy-Lamport (2/2)

- **Partida**: Inicialmente un único proceso cualquiera recibe una marca (e.g. de un proceso controlador o autoenvío).
- **Ejecución:** Cada proceso P_i que recibe una marca de partida o desde un proceso cualquiera P_j, por medio algún canal de entrada denotado como $C_{j \to i}$, ejecuta el siguiente algoritmo:

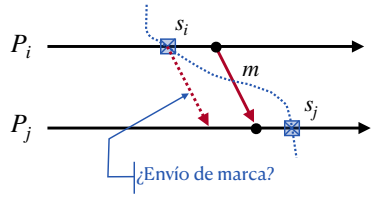
```
if (Pi recibe por primera vez una marca) then {      /* realiza atómicamente */
    · Pi registra su estado local si;
    · Pi registra el estado de todos sus canales de entrada como vacíos;
    · Pi envía una marca a todos sus vecinos (usando sus canales de salida);

} else {   /* Pi recibe otra vez una marca; en este caso, por canal de entrada Cj→i */
    · Pi registra el canal de entrada Cj→i igual a secuencia de mensaje
      recibidos por este canal desde que Pi registró su estado si;
}
```


Demostración (1/2)

a) Algoritmo registra un estado consistente

Demostración por el absurdo: Suponga que algoritmo registra un estado inconsistente. Entonces existe un par de procesos P_i, P_j y un mensaje m, tal que se dé el siguiente escenario:



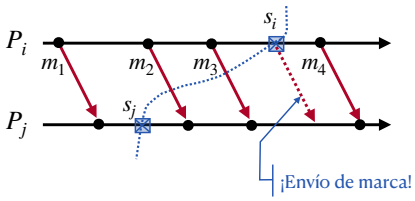
En el corte de este estado global se observa que P_j ha recibido m, sin embargo m no ha sido enviado por proceso P_i .

Contradicción: Este escenario no es posible, pues el algoritmo exige enviar inmediatamente la marca a sus vecinos cuando registra su estado local y como los canales son FIFO, P_j debe haber tomado su estado s_j antes de la llegada de mensaje m.

Demostración (2/2)

b) Algoritmo registra correctamente estado de los canales

Demostración por inducción: Suponga un par de procesos P_i y P_j donde existe un canal $< P_i, P_j >$. Entonces, cuando P_i registra su estado local s_i, debe enviar una marca a P_j.



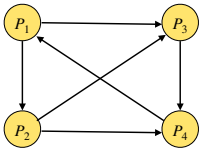
Argumentación:

- Cuando P_j registró su estado local s_j, también registró como vacío su canal de entrada $C_{i,j}$.
- Cuando P_j recibe la marca por canal de entrada $C_{i,j}$, ha registrado los mensajes $\{m_2, m_3\}$ como el estado de este canal: los dos mensajes en tránsito asociado al corte.

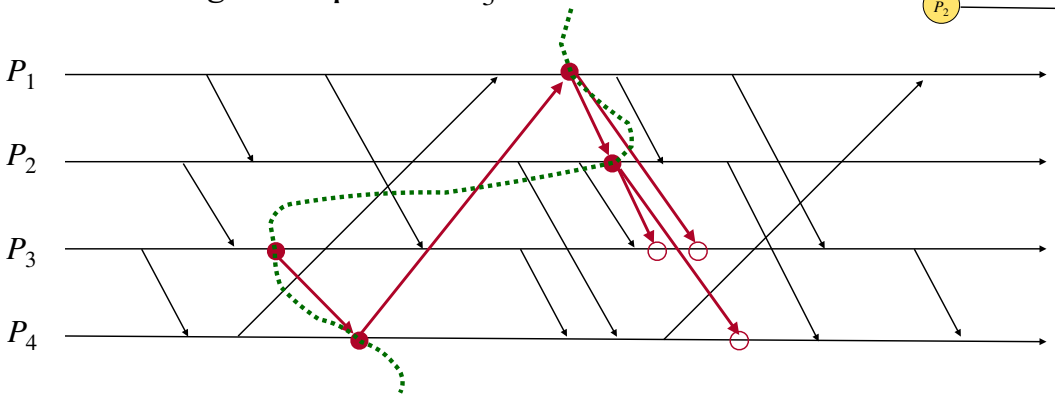
$\therefore$ Existencia de canales FIFO asegura P_j haya registrado exactamente los mensajes en tránsito (Q.E.D.)

Ejemplo de ejecución de Algoritmo de Ch-L (1/2)

Registro de estado global

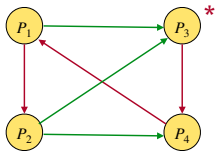


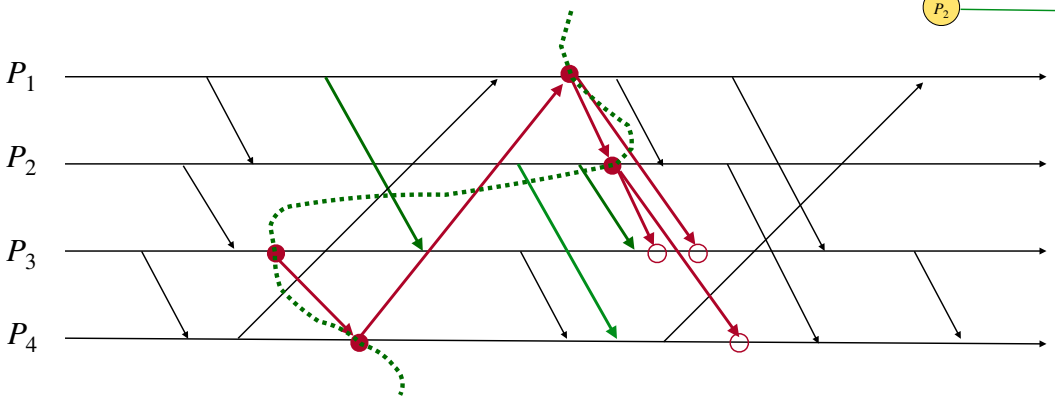
Algoritmo parte en P_3



Ejemplo de ejecución de Algoritmo de Ch-L (2/2)

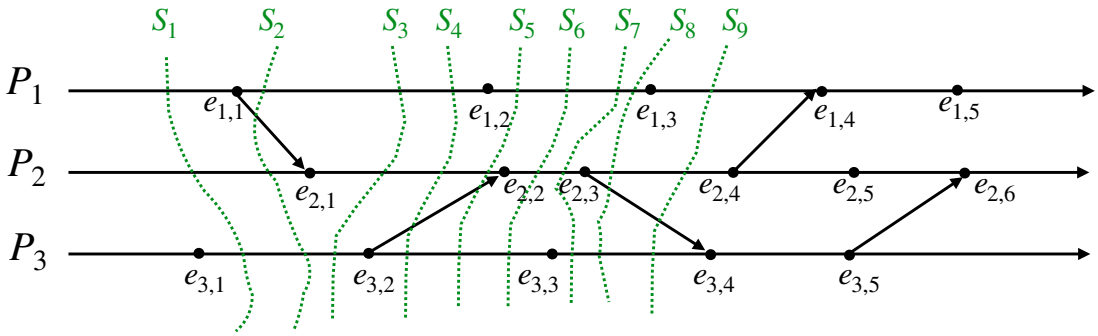
Análisis del corte





Visión de una computación distribuida

Secuencia de cambios de estados globales



Notación: Los cambios de estados se denotan como:

$$S_1 \rightsquigarrow S_2 \rightsquigarrow S_3 \rightsquigarrow S_4 \rightsquigarrow S_5 \rightsquigarrow S_6 \rightsquigarrow S_7 \rightsquigarrow S_8 \rightsquigarrow S_9$$

Y se dice que S_9 es alcanzable desde S_1, denotado por: $S_1 \overset{*}{\rightsquigarrow} S_9$

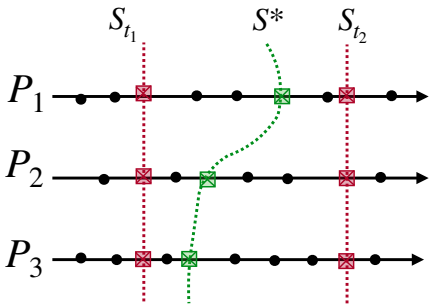
Estados estables

DEFINICIÓN: Estado estable. Un sistema está en un estado estable si existe una propiedad estable $\mathscr{P}$, tal que: $\forall S, S' : \mathscr{P}(S) \wedge (S \overset{*}{\rightsquigarrow} S') \implies \mathscr{P}(S')$

■

Ejemplos de estados estables:

- El sistema está en *deadlock*.
- El algoritmo terminó.
- Al menos 20 mensajes han sido enviados.



En la figura se cumple: $S_{t_1} \overset{*}{\rightsquigarrow} S^* \overset{*}{\rightsquigarrow} S_{t_2}$. Luego:

$\mathscr{P}(S^*) \implies \mathscr{P}(S_{t_2})$ y $\neg\mathscr{P}(S^*) \implies \neg\mathscr{P}(S_{t_1})$

4.6 Conclusiones

Lecciones principales

1. Se revisaron diferentes tipos de coordinación en sistemas distribuidos, el concepto de tiempo y uso de relojes para marcar eventos en una computación distribuida.
2. Se han estudiado los límites de los algoritmos de sincronización de relojes reales en cuanto a precisión y exactitud, eventualmente en escenarios de fallas.
3. Se ha formalizado el concepto de dependencia causal y concurrencia para establecer un ordenamiento parcial de los eventos de un sistema.
4. Se han estudiado dos sistema de relojes lógicos para sistemas asincrónicos: Relojes de Lamport y Relojes vectoriales (con algunas de sus aplicaciones).
5. Se ha formalizado el concepto de un estado global consistente y se ha visto el algoritmo de Chandy-Lamport para obtener una instantánea distribuida y consistente.

Material de estudio complementario

TEXTOS GUÍAS:

- Van Steen, et al. (2023). Cap. 5, secciones 5.1 y 5.2, pp. 247-271.
- Coulouris, et al. (2012). Cap. 14, secciones 14.1-14.5, pp. 595-619.

LECTURAS COMPLEMENTARIAS:

- L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Comm. of the ACM*, 21(7), pp. 558-565, 1978.
- M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computing Systems*, 3(1) pp. 63-75, Feb. 1985.
- O. Babaglou, K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms". Technical Report UBLCS-93-1. January 1993. Laboratory for Computer Science. University of Bologna. Bologna (Italy).
- Schwarz, R. & Mattern, F., "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", in *Distributed Computing*, Springer Verlag, 1994.

Capítulo IV: Fundamentos teóricos de computación distribuida

Sincronismo, relojes, causalidad y consistencia de estados globales

Prof. Dr.-Ing. Raúl Monge Anwandter ♦ 2º semestre 2025