Términos útiles

Reloj lógico:
Conceptualización de una función matemática que asigna números a eventos.
Mecanismo para capturar relaciones cronológicas y casuales en un sistema distribuido.
Permite un orden global de los eventos que ocurren en distintos procesos de un sistema distribuido.
Su primera implementación fue en Lamport Timestamps.

Clock drift: Cuando un reloj no corre a la misma velocidad que un reloj de referencia, lo que provoca que el reloj se desincroniza gradualmente del otro reloj.

Datos que no salen en la presentación, pero se explicaron.

NTP: Modos de sincronización

Modo simétrico activo: Un host manda mensajes periódicos independiente del estado de alcance (reachability) o estrato de sus pares (peers).

Modo simétrico pasivo: Se crea cuando un sistema recibe un mensaje de un par que opera en el modo simétrico activo siempre y cuando (el par) esté al alcance y esté en el mismo o un estrato por debajo del host. Este es un modo donde el host comunica que está dispuesto a sincronizarse y ser sincronizado por el par.
Este modo ofrece la mayor precisión por lo que es usado en servidores maestros.
(Un par de servidores intercambian mensajes con data temporal, esta data es retenida para mejorar la precisión en la sincronización a medida que pasa el tiempo).

Modo de procedimiento de llamada: Es similar al Cristian's algorithm, un cliente indica que está dispuesto a ser sincronizado por el servidor (pero no a sincronizar al servidor).

Modo multicast: Destinado a LANs de alta velocidad. Tiene relativamente baja precisión y menos seguro ya no es tan útil.

SNTP: Modos de operación

Unicast: El cliente manda un request a un servidor designado. Una vez recibida la respuesta el cliente determina el tiempo, offset del reloj y roundtrip delay con respecto al reloj de referencia.

Multicast: Un servidor periódicamente manda mensaje multicast sin esperar requests de clientes. (Usualmente los clientes no mandan requests por la interrupción de servicio causada por servidores multicast desconocidos y poco confiables. Esto se puede evitar con mecanismo de control de acceso que permite al cliente seleccionar un servidor que conoce o confía).

Anycast: El cliente manda un request a un broadcast local o a una dirección multicast y

toma la primera respuesta recibida de los servidores que respondan. (Desde ahí el protocolo pasa a ser unicast) .

Vector Clocks

Los vector clocks funcionan a partir de 4 reglas:

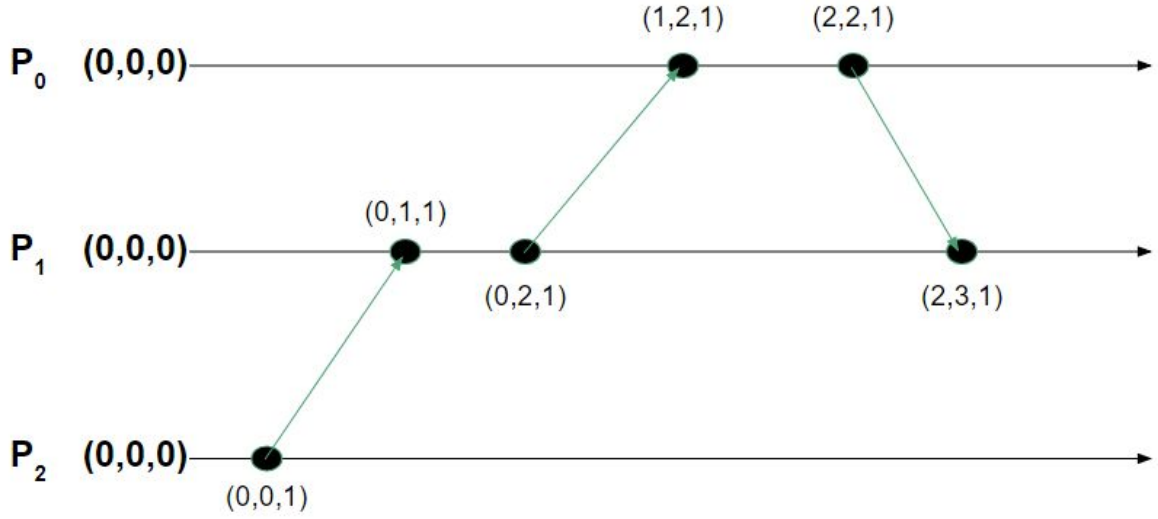
- Para cada proceso, el vector se inicializa en cero.
- Cuando un proceso experimenta un evento aumenta en uno la posición correspondiente a él mismo en su vector local.
- Cuando un proceso envía un mensaje incrementa su contador en uno y añade su vector al mensaje,
- Cada vez que un proceso recibe un mensaje, actualiza el valor de cada elemento de su vector tomando el máximo valor entre el vector local y el vector recibido por cada posición de éste (compara elemento a elemento) y luego incrementa su vector en uno.

Vector clock nos asegura para dos eventos a y b cualquiera:

- Si **a pasa antes que b entonces V(a) <= V(b)** (cada integral en el vector a es menor o igual al integral en la misma posición del vector b)
- Si **V(a) <= V(b) entonces a pasa antes que b.**

La desventaja de utilizarlo es la necesidad de mayor almacenaje y mayor tamaño del mensaje ya que se debe manipular un vector en vez de un solo número entero.

Ejemplo:



En el ejemplo tenemos un sistema con 3 procesos, por lo que cada proceso maneja un vector de 3 enteros de manera local.

¿Qué ocurre en la imagen?:

Versión corta:

1. P_0 => (0,0,0) |P_1 => (0,0,0) | P_2 => (0,0,0)
2. P_0 => (0,0,0) |P_1 => (0,0,0) | P_2 => (0,0,1)
3. P_0 => (0,0,0) |P_1 => (0,1,1) | P_2 => (0,0,1)
4. P_0 => (0,0,0) |P_1 => (0,2,1) | P_2 => (0,0,1)
5. P_0 => (1,2,1) |P_1 => (0,2,1) | P_2 => (0,0,1)
6. P_0 => (2,2,1) |P_1 => (0,2,1) | P_2 => (0,0,1)
7. P_0 => (2,2,1) |P_1 => (2,3,1) | P_2 => (0,0,1)

Versión explicada paso a paso:

1. Cada proceso inicializa su vector con todos los enteros en 0.
2. P_2 va a enviar un mensaje por lo que aumenta el entero de su vector correspondiente a él, su vector queda como **(0,0,1)**. P_2 envía el mensaje a P_1 y adjunta su vector al mensaje.
3. P_1 recibe el mensaje y compara su vector con el vector recibido dejando los valores máximos para cada entero por lo que ahora el vector de P_1 es (0,0,1), como recibe el mensaje aumenta el entero correspondiente a él, por lo que su vector queda como **(0,1,1)**.
4. P_1 va a enviar un mensaje por lo que aumenta el entero de su vector correspondiente a él, el vector de P_1 queda como **(0,2,1)**. P_1 envía el mensaje a P_0 y adjunta su vector al mensaje.
5. P_0 recibe el mensaje, compara los vectores por lo que ahora el suyo es (0,2,1) y como recibe el mensaje, aumenta el integral por lo que el vector queda como **(1,2,1)**.
6. P_0 va a enviar mensaje por lo que su vector queda como **(2,2,1)**. P_0 envía el mensaje a P_1 y adjunta su vector al mensaje.
7. P_0 recibe el mensaje, compara los vectores por lo que ahora el suyo es (2,2,1) y como recibe el mensaje, aumenta el integral por lo que el vector queda como **(2,3,1)**.

Si se miran los vectores después de los eventos (sin mirar cómo pasaron) se puede decir que:

- Como $V(P_0) < V(P_1)$ se puede asegurar que el evento de P_0 pasó antes que el de P_1.
- Como $V(P_2) < V(P_1)$ y $V(P_2) < V(P_0)$ se puede asegurar que el evento de P_2 pasó antes que los eventos de P_1 y P_0.

Utilizando Laport Timestamps no hubiese sido posible concluir lo anterior.