Computación distribuida.

Aplicaciones de Relojes Vectoriales.

Integrantes:
Aguilera Moreno Adrian
Torres Valencia Kevin Jair
Pérez Romero Natalia Abigail

Facultad de Ciencias, UNAM

Tabla de contenido.

Los relojes vectoriales son un tipo de reloj lógico propuesto de manera independiente por *Colin J. Fidge* y *Friedemann Mattern* en 1988.

Esta técnica consiste en un mapeo entre eventos en una historia distribuida y vectores enteros.

Definiciones: Vector Tiempo.

Sistema Vectorial de Relojes. Es un mecanismo capaz de caracterizar estados locales (en adelante, eventos) en un sistema distribuido, asociando un valor vectorial a cada estado.

Tiempo Vectorial. Es la noción de tiempo capturada por los relojes vectoriales.

Caracterización Formal del Tiempo Vectorial. Sea `date(e)` la caracterización asociada a un evento `e`, de tal manera que se cumple:

1. $\forall_{e_1,e_2} : (e_1 \rightarrow e_2) \Leftrightarrow \texttt{date}(e_1) < \texttt{date}(e_2)$.
2. $\forall_{e_1,e_2} : (e_1 \,||\, e_2) \Leftrightarrow \texttt{date}(e_1) \,||\, \texttt{date}(e_2)$.

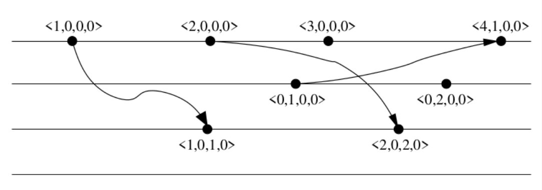


Figure: Reloj Vectorial con tiempos locales.

Reloj Vectorial. La implementación del tiempo vectorial requiere que cada proceso, en el sistema, mantenga un vector de enteros positivos $Vc_i[1, \cdots, n]$ con valores inicialmente $[0, \cdots, 0]$. Este vector debe cumplir con

1. $Vc_i[i]$ cuenta el número de eventos producidos por p_i.
2. $Vc_i[j], j \neq i$, nos dice cuántos eventos conoce p_i producido por p_j.

De manera formal, sea e un evento producido por p_i, tenemos que

$$Vc_i[k] = |\{f|(f \text{ se produjo por } p_k) \wedge (f \to e)\}| + 1(k, i).$$

Una primera aproximación. Inicialmente todos los procesos disponen de un vector con entradas igual al número total de procesos, este vector debe estar inicializado en 0 para cada entrada. A continuación se describe el algoritmo:

- Si p_i produce un evento, entonces:
  (1) $Vc_i[i] \leftarrow Vc_i[i] + 1$;
  (2) Produce un evento e caracterizado por $Vc_i[1, \cdots, n]$.
- Cuando p_i envia un mensaje a p_j, entonces:
  (3) $Vc_i[i] \leftarrow Vc_i[i] + 1$;
  (4) $\texttt{send}(\langle \texttt{msj}, Vc_i[1, \cdots, \texttt{n}] \rangle)$ a p_j.
- Cuando p_j recibe un mensaje, entonces:
  (3) $Vc_j[j] \leftarrow Vc_j[j] + 1$;
  (4) $Vc_j[1, \cdots, n] \leftarrow \forall_{k \in [1, \cdots, n]} \texttt{max}(Vc_i[k], Vc_j[k])$.

Notación: Para, cualesquiera, dos vectores Vc_1 y Vc_2 del tamaño n. Tenemos que

- $Vc_1 \leq Vc2 =_{def.} (\forall_{k \in \{1,\cdots,n\}} : Vc_1[k] \leq Vc_2[k]);$
- $Vc_1 < Vc2 =_{def.} (Vc_1[k] \leq Vc_2[k]) \wedge (Vc_1[k] \neq Vc_2[k]);$
- $Vc_1 || Vc_2 =_{def} \neg(Vc_1 \leq Vc_2) \wedge \neg(Vc_2 \leq Vc_1).$

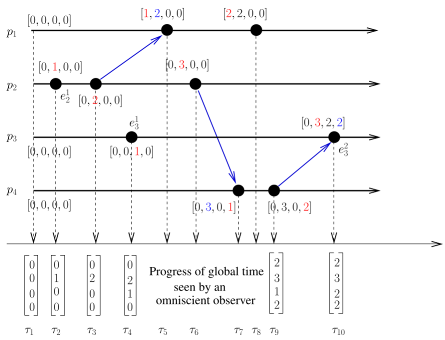


Figure: Ejemplo de propagación en un reloj Vectorial.

Def. Sea $e.Vc$ el vector asociado al evento e.

Teo 1. Por el algoritmo mencionado tenemos que, para cualesquiera e_1 y e_2 distintos tenemos que

a) $(e_1 \to e_2) \Leftrightarrow (e_1.Vc < e_2.Vc)$;

b) $(e_1 || e_2) \Leftrightarrow (e_1.Vc || e_2.Vc)$.

Cor 1. Dadas dos caracterizaciones a eventos (fechas), determinar si estos eventos están relacionados o no, puede requerir hasta n comparaciones de enteros.

Teo 2. Sean dos eventos e_1 y e_2 con tuplas $\langle e_1.Vc, i\rangle$ y $\langle e_2.Vc, j\rangle$ de manera respectiva y $i \neq j$. Entonces

a) $(e_1 \to e_2) \Leftrightarrow (e_1.Vc[i] \leq e_2.Vc[i])$;

b) $(e_1 || e_2) \Leftrightarrow ((e_1.Vc[i] > e_2.Vc[i]) \wedge (e_2.Vc[j] > e_1.Vc[j]))$.

Mejora en la complejidad en tiempo. Hasta el momento la complejidad en tiempo para combinar 2 eventos nos toma $\mathcal{O}(n)$, con n el número de procesos en el sistema.

Por el *Teo. 2*, sabemos que basta con verificar dos entradas para saber como es un evento respecto al otro. Así, basta comparar dos entradas para combinar la caracterización de 2 eventos esto nos toma $\mathcal{O}(1)$.

Consideremos

Relojes Vectoriales: Desventajas.

Esta mejora a los relojes lógicos de Lamport tiene un problema de implementación, que en un momento será más evidente.

Desventaja

Esta desventaja es que cada proceso tiene que cargar con espacio igual al número de procesos en el sistema y cada intercambio entre eventos es de este tamaño.

Def. Un predicado es local para p_i ***Sii*** se encuentra en variables de p_i solamente.

Def. El predicado LP_i es estable ***Sii*** en cuanto se vuelva verdadero, este permanece así siempre.

Notación: $\sigma_i \models LP_i$. Indica que el estado local σ_i de p_i satisface el predicado LP_i.

Def. Sea $\{p_1, \cdots, p_n\}$ un sistema distribuido y $LP_1, \cdots, LP_n$ n predicados locales, uno por proceso (con su respectivo proceso). Un *estado global consistente* $\sum = (\sigma_1, \cdots, \sigma_n)$ satisface el predicado global $LP_1 \wedge \cdots \wedge LP_n$ denotado por

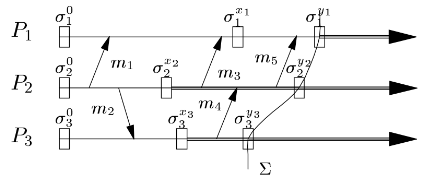
$$\sum \models \bigwedge_i LP_i$$

siempre que $\bigwedge_i (\sigma_i \models LP_i)$.

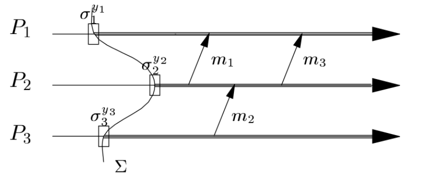
Problema.

Detectemos sobre la *historia* del sistema, y sin utilizar controles de mensajes adicionales, el primer estado global consistente Σ que satisface una conjunción de predicados locales estables.

Aplicación: Solución.



(a) Ejemplo 1.



(b) Ejemplo 2.

Figure: Detección de una conjunción local estable.

Evento relevante En algún nivel de abstracción, sólo un subconjunto de los eventos son relevantes. Por ejemplo, en algunas aplicaciones sólo la modificación de algunas variables locales, o el procesamiento de ciertos mensajes especificos son relevantes.

Dado un computo distribuido $\widehat{H} = (H, \xrightarrow{ev})$, sea $R \subset H$ los eventos relevantes.

Definición. Relación causal de Precedencia, denotada como $\xrightarrow{re}$:

$$\forall e_1, e_2 \in R : (e_1 \xrightarrow{re} e_2) \iff (e_2 \xrightarrow{ev} e_1)$$

Esta relación es la proyección de $\xrightarrow{ev}$ sobre los elementos de R.

Sean $e_1, e_2 \in R$. Si e_1 es **predecesor inmediato de** e_2 entonces:

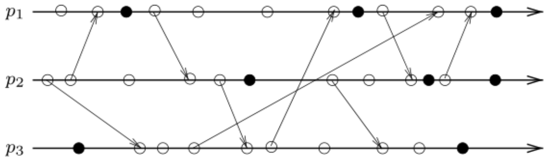
$$(e_1 \xrightarrow{re} e_2) \wedge (\nexists e \in: (e_1 \xrightarrow{re} e) \wedge (e \xrightarrow{re} e_2))$$

when producing a relevant internal event e **do**
(1) $vc_i[i] \leftarrow vc_i[i] + 1$;
(2) Produce the relevant event e. % The date of e is $vc_i[1..n]$.

when sending $\text{MSG}(m)$ **to** p_j **do**
(3) send $\text{MSG}(m, vc_i[1..n])$ to p_j.

when $\text{MSG}(m, vc)$ **is received from** p_j **do**
(4) $vc_i[1..n] \leftarrow \max(vc_i[1..n], vc[1..n])$.

Vector clock system for relevant events (code for process p_i)



7 Relevant events in a distributed computation

El problema del seguimiento del predecesor inmediato (Immediate predecessor tracking IPT) Consiste en asociar a cada evento relevante el conjunto de eventos relevantes que son sus predecesores inmediatos.

Además, esto se ha hecho sobre la marcha y sin añadir mensajes de control.

La determinación de los predecesores inmediatos consiste en calcular la reducción transitiva (o diagrama de Hasse) del orden parcial $\widehat{R} = (R, \xrightarrow{re})$.

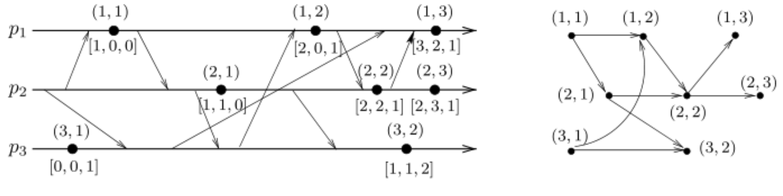


Fig. 7.19 From relevant events to Hasse diagram

Un algoritmo que resuelve el problema IPT: variables locales

when producing a relevant internal event e **do**
(1) $vc_i[i] \leftarrow vc_i[i]+1$;
(2) Produce the relevant event e; **let** $e.imp = \{(k, vc_i[k]) \mid imp_i[k]=1\}$;
(3) $imp_i[i] \leftarrow 1$;
(4) **for each** $j \in \{1,\ldots,n\} \setminus \{i\}$ **do** $imp_i[j] \leftarrow 0$ **end for**.

when sending $\text{MSG}(m)$ **to** p_j **do**
(5) send $\text{MSG}(m, vc_i[1..n], imp_i[1..n])$ to p_j.

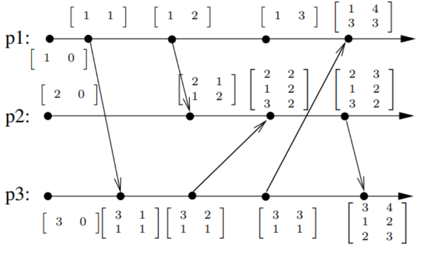
when $\text{MSG}(m, vc, imp)$ **is received do**
(6) **for each** $k \in \{1,\ldots,n\}$ **do**
(7) **case** $vc_i[k] < vc[k]$ **then** $vc_i[k] \leftarrow vc[k]$; $imp_i[k] \leftarrow imp[k]$
(8) $vc_i[k] = vc[k]$ **then** $imp_i[k] \leftarrow \min(imp_i[k], imp[k])$
(9) $vc_i[k] > vc[k]$ **then** skip
(10) **end case**
(11) **end for**.

Determination of the immediate predecessors (code for process p_i)

A menudo el número de procesos participando en un computo distribuido no es constante, por lo que los relojes debe ser capaces de crecer.

El tamaño del vector está limitado por dos veces el número de procesos de los que el proceso de mantenimiento ha recibido (directa o indirectamente) valores de reloj.

Cada proceso p_i mantiene su propio reloj logico C_i el cual es una matriz de dos columnas, variable en su número de filas, donde $C_i(e_i^x)[k, 2]$, en e_i almacena el valor del último reloj vectorial conocido por p_i del proceso cuyo ID está en $C_i(e_i^x)[k, 1]$



(a) Reloj vectorial dinamico.

Inicialmente, el reloj consiste de una sola fila con $C_i(e_i^0)[1,1] = i$ y $C_i(e_i^0)[1,2] = 0$

Las reglas para actualizar son las siguientes:

1. Al recibir el m de e_i^x, entonces $\forall l$ que no exceda el número de filas en $C_j(e_j^y)$: $C_j(e_j^y) = max\{C_i(e_i^{x-1})[k,2], C_j(e_j^y)[1,2]\}$, donde $C_j(e_j^y)$ la matriz de reloj del evento enviado en el proceso J que envio m al proceso i, y $C_j(e_j^y)[l,1] = C_i(e_i^x)[k,1]$
2. Si no existe k tal que $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$ para cualquier l, entonces una nueva fila es añadida a $C_i(e_i^x)$ con $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$ y $C_i(e_i^x)[k,2] = C_j(e_j^y)[l,2]$
3. Si e_i^x es cualquier evento, entonces el proceso incrementa su reloj vectorial: $C_i(e_i^x)[1,2] = C_i(e_i^{x-1})[1,2] + 1$