

# Computación distribuida.

Aplicaciones de Relojes Vectoriales.

---

Integrantes:

Aguilera Moreno Adrian

Torres Valencia Kevin Jair

Pérez Romero Natalia Abigail

Facultad de Ciencias, UNAM



**1**

## **Introducción.**

---

- Tiempo Vectorial.
- Relojes Vectoriales.
- Algoritmo.
- Desventajas.

**2**

## **Aplicaciones.**

---

- El caso DynamoDB.
- Un problema de conjuntos.
- Determinando Propiedades Globales.
- Implementación en sistemas dinámicos.
- Detección de una conjunción de predicados locales estables.

**3**

## **Determinación sobre la marcha de los predecesores inmediatos**

---

- Introducción.

**4**

## **Relojes vectoriales dinámicos**

---

- Introducción.

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  $\text{date}(e)$  la caracterización asociada a un evento  $e$ , de tal manera que se cumple:

1.  $\forall e_1, e_2 : (e_1 \rightarrow e_2) \Leftrightarrow \text{date}(e_1) < \text{date}(e_2)$ .
2.  $\forall e_1, e_2 : (e_1 \parallel e_2) \Leftrightarrow \text{date}(e_1) \parallel \text{date}(e_2)$ .

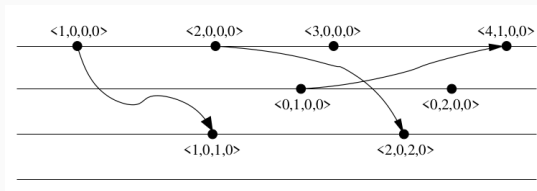


Figure: Reloj Vectorial con tiempos locales.



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

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



**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  $Vc_i[1, \dots, n]$ .
- ▶ Cuando  $p_i$  envia un mensaje a  $p_j$ , entonces:
  - (3)  $Vc_i[i] \leftarrow Vc_i[i] + 1$ ;
  - (4)  $\text{send}(\langle \text{msj}, Vc_i[1, \dots, n] \rangle)$  a  $p_j$ .
- ▶ Cuando  $p_j$  recibe un mensaje, entonces:
  - (3)  $Vc_j[j] \leftarrow Vc_j[j] + 1$ ;
  - (4)  $Vc_j[1, \dots, n] \leftarrow \forall_{k \in [1, \dots, n]} \max(Vc_i[k], Vc_j[k])$ .

# Algoritmo: Propagación del tiempo vectorial.

**Notación:** Para, cualesquiera, dos vectores  $V_{c1}$  y  $V_{c2}$  del tamaño  $n$ . Tenemos que

- ▶  $V_{c1} \leq V_{c2} =_{\text{def.}} (\forall_{k \in \{1, \dots, n\}} : V_{c1}[k] \leq V_{c2}[k]);$
- ▶  $V_{c1} < V_{c2} =_{\text{def.}} (V_{c1}[k] \leq V_{c2}[k]) \wedge (V_{c1}[k] \neq V_{c2}[k]);$
- ▶  $V_{c1} || V_{c2} =_{\text{def.}} \neg(V_{c1} \leq V_{c2}) \wedge \neg(V_{c2} \leq V_{c1}).$

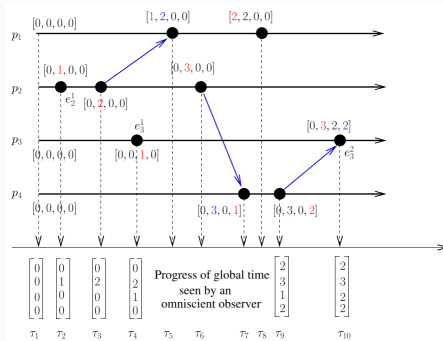


Figure: Ejemplo de propagación en un reloj Vectorial.



**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

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

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

**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

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

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





## Algoritmo: Reducción de costo en la comparación de dos vectores.

**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  $\mathcal{O}(1)$ .



# Algoritmo: Relación del tiempo vectorial y estados globales.

Consideremos



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.



## Aplicación: Detección de conjunciones de predicados estables.

**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  $\sigma_i$  de  $p_i$  satisface el predicado  $LP_i$ .

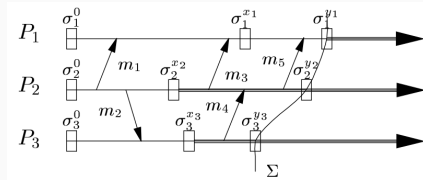
**Def.** Sea  $\{p_1, \dots, p_n\}$  un sistema distribuido y  $LP_1, \dots, LP_n$   $n$  predicados locales, uno por proceso (con su respectivo proceso). Un estado global consistente  $\Sigma = (\sigma_1, \dots, \sigma_n)$  satisface el predicado global  $LP_1 \wedge \dots \wedge LP_n$  denotado por

$$\Sigma \models \bigwedge_i LP_i$$

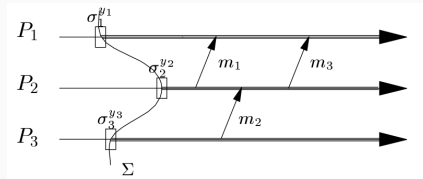
siempre que  $\bigwedge_i (\sigma_i \models LP_i)$ .

### Problema.

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



(a) Ejemplo 1.



(b) Ejemplo 2.

Figure: Detección de una conjunción local estable.



En proceso

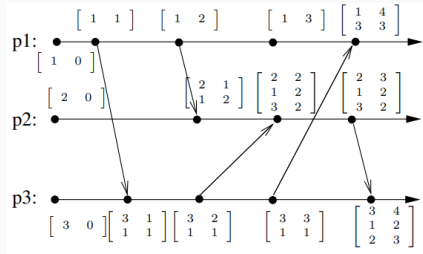


# Aplicación: Relojes vectoriales dinamicos

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) Ejemplo

Figure: 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  $e_j^x$ , 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 envío  $m$  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:  
 $C_i(e_i^x)[1, 2] = C_i(e_i^{x-1})[1, 2] + 1$