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.

Introduccion

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. $\forall_{e_1,e_2} : (e_1 \mid\mid e_2) \Leftrightarrow \texttt{date}(e_1) \mid\mid \texttt{date}(e_2)$.

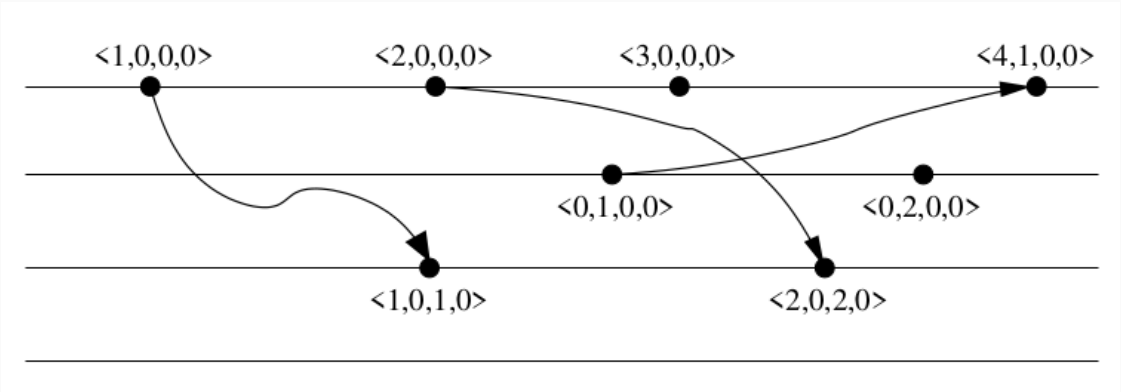


Figure: Reloj Vectorial con tiempos locales.

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

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

Reloj Vectorial: Algoritmo.

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) `send(⟨msj,` Vc_i`[1, ···, n] ⟩)` a p_j.
- Cuando p_j recibe un mensaje, entonces:
  (3) $Vc_j[j] \leftarrow Vc_j[j] + 1$;
  (4) $Vc_j[1, \cdots, n] \leftarrow \forall_{k \in [1, \ldots, n]} \texttt{max}(Vc_i[k], Vc_j[k])$.

Algoritmo: Propagación del tiempo vectorial.

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

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

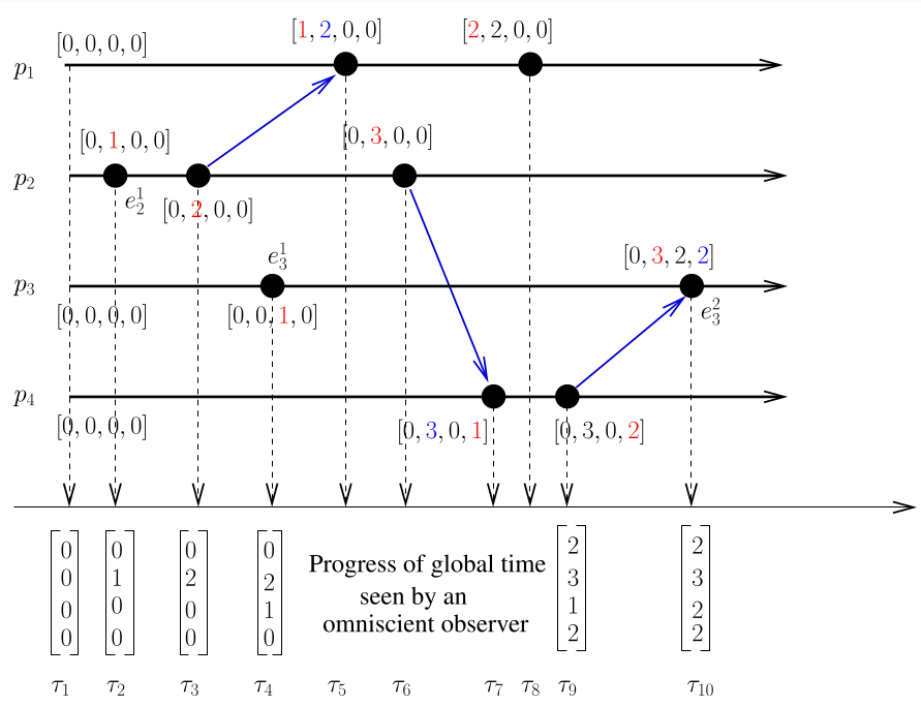


Figure: Ejemplo de propagación en un reloj Vectorial.

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

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 $\mathcal{O}(1)$.

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.


APLICACIONES

Aplicación: DynamoDB

Es un servicio de base de datos noSQL ofrecido por Amazon como parte de Amazon Web Services.

De manera general lo utilizan para poder controlar el orden de los registros de multiversión.

Aplicación: DynamoDB

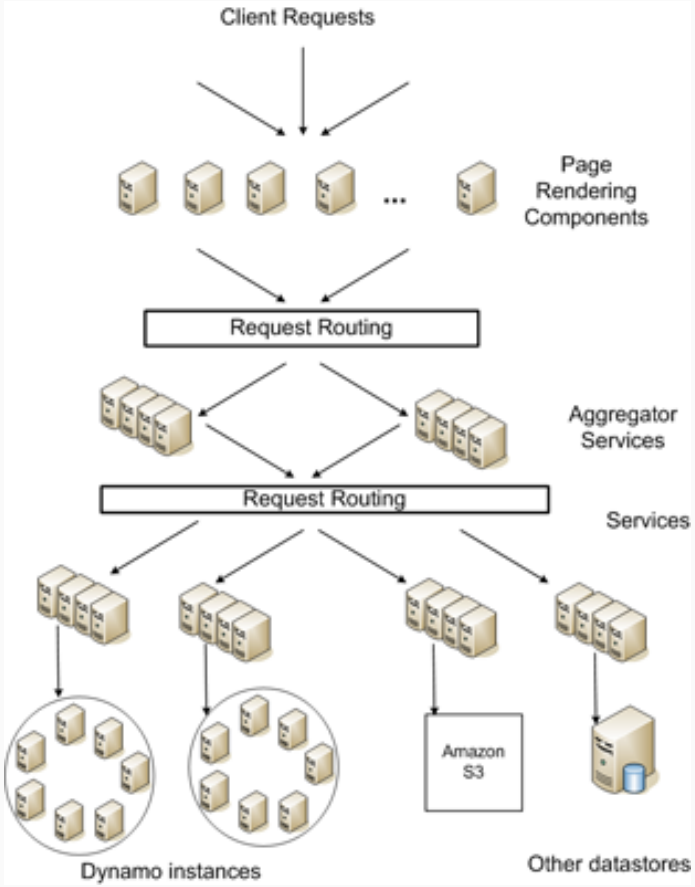
Problema: Alta disponibilidad para escrituras

Técnica: Relojes vectoriales con reconciliación durante las lecturas.

Ventaja: El tamaño de la versión está desvinculado de las tasas de actualización.

Aplicación: DynamoDB

Dynamo utiliza relojes vectoriales para mantener el control de versiones del objeto en varias réplicas. Un reloj vectorial es efectivamente una lista de pares (nodo, contador). Un reloj vectorial está asociado con cada versión de cada objeto.



Para que los utiliza:

Para capturar la causalidad entre diferentes versiones del mismo objeto.

Un reloj vectorial está asociado con cada versión de cada objeto. Uno puede determinar si dos versiones de un objeto están en ramas paralelas o tienen un orden causal, examinando sus relojes vectoriales.

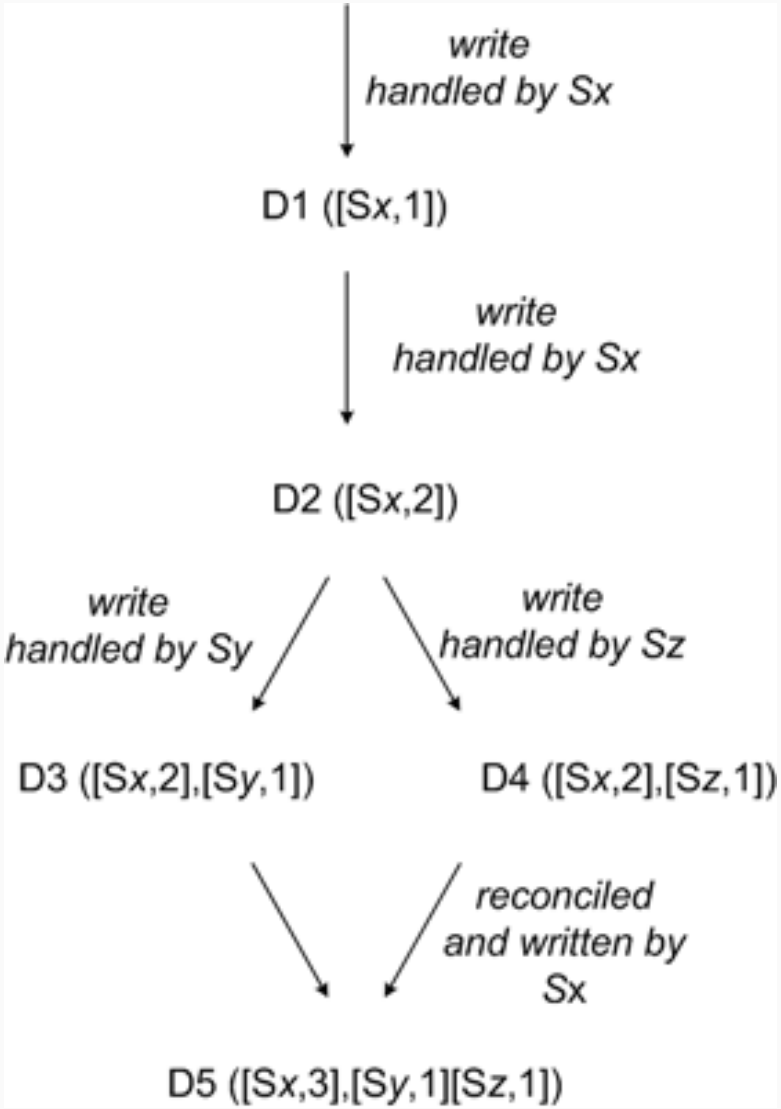
Si los contadores del reloj del primer objeto son menores o iguales que todos los nodos del segundo reloj, entonces el primero es un ancestro del segundo y puede olvidarse. De lo contrario, se considera que los dos cambios están en conflicto y requieren reconciliación.

Como lo hace:

Cuando un cliente desea actualizar un objeto, debe especificar qué versión está actualizando. Esto se hace pasando el contexto que obtuvo de una operación de lectura anterior, que contiene la información del reloj vectorial.

Al procesar una solicitud de lectura, si Dynamo tiene acceso a varias ramas que no se pueden reconciliar sintácticamente, devolverá todos los objetos en las hojas, con la información de versión correspondiente en el contexto. Se considera que una actualización que usa este contexto ha reconciliado las versiones divergentes y las ramas se contraen en una sola versión nueva.

Ejemplo:


write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

Ejemplo:

- Se tiene un nodo Sx, que maneja la escritura de esta clave aumenta su número de secuencia y lo usa para crear el reloj vectorial de los datos.
- Se tiene un objeto D1 y su reloj asociado [(Sx, 1)]. Este cliente lo actualiza Supongamos que el mismo nodo también maneja esta solicitud.
- Ahora el sistema tendra un objeto D2 y su reloj asociado [(Sx, 2)].
- D2 desciende de D1, y por lo tanto sobrescribe D1
- Supongamos que el mismo cliente actualiza el objeto nuevamente y un servidor diferente maneja la solicitud (por ejemplo Sy).
- Se tiene datos D3 y su reloj asociado [(Sx, 2), (Sy, 1)].

Ejemplo:

Ahora supongamos que un cliente diferente lee D2 y luego intenta actualizarlo, y otro nodo realiza la escritura (por ejemplo, Sz).

- El sistema va a tener a D4 (descendiente de D2) cuya versión de reloj es [(Sx, 2), (Sz, 1)].
- Un nodo que es consciente de D3 y recibe D4 encontrará que no existe una relación causal entre ellos. Esto quiere decir que hay cambios en D3 y D4 que no se reflejan entre sí.
- Ambas versiones de los datos deben conservarse y presentarse a un cliente (después de una lectura) para la reconciliación semántica.

Ejemplo:

Ahora supongamos que algún cliente lee tanto D3 como D4 (el contexto nos dirá que la lectura encontró ambos valores)

Nota: El contexto de la lectura es un resumen de los relojes de D3 y D4, a saber, [(Sx, 2), (Sy, 1), (Sz, 1)]. Entonces

- Si el cliente realiza la reconciliación y el nodo Sx coordina la escritura, Sx actualizará su número de secuencia en el reloj.
- Finalmente el nuevo dato D5 tendrá el siguiente reloj: [(Sx, 3), (Sy, 1), (Sz, 1)].

Posible desvantaja de esto es:

Es que el tamaño de los relojes vectoriales puede crecer si muchos servidores coordinan las escrituras en un objeto.

Lo provoca:

En caso de particiones de red o fallas de múltiples servidores, las solicitudes de escritura pueden ser manejadas por nodos que no están en los primeros N nodos en la lista de preferencias

Para evitar lo anterior:

Se emplea el siguiente esquema de truncamiento de reloj: Junto con cada par (nodo, contador), Dynamo almacena una marca de tiempo que indica la última vez que el nodo actualizó el elemento de datos. Cuando el número de pares en el reloj vectorial alcanza un umbral, el par más antiguo se elimina del reloj.

Este problema no ha surgido en la producción y, por lo tanto, este problema no se ha investigado a fondo.

Algunas empresas que lo utilizan:

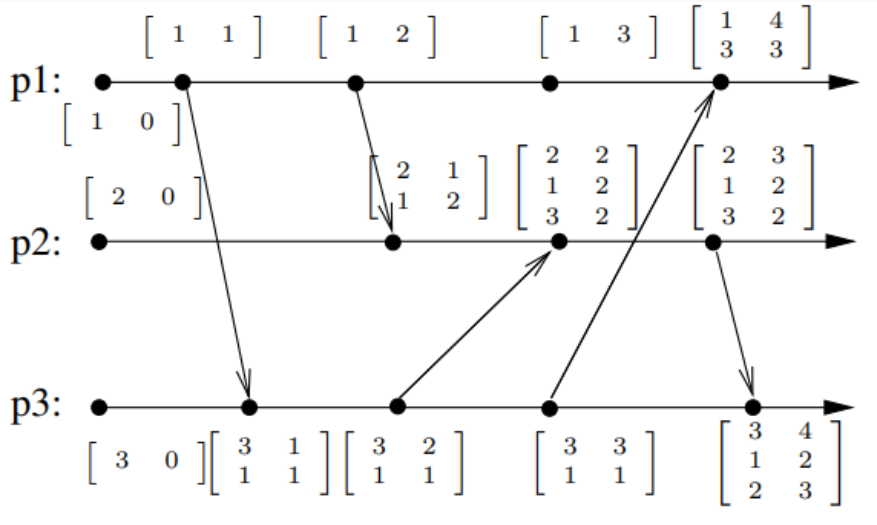
- Lyft
- Airbnb
- Redfin
- Samsung
- Toyota
- Capital One

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) Reloj vectorial dinamico.

Aplicación: Relojes vectoriales dinamicos

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

De manera similar podemos hacer una prueba de dependencia causal de la forma: $e_i^x \implies e_j^y$) si y solo si $\exists k, l$:

$$C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] = i$$

y

$$C_i(e_i^x)[k,2] \leq C_j(e_j^y)[l,2]$$

Y de concurrencia $e_i^x || e_j^y$ si y solo si

$$C_j(e_j^y) \not\leq C_i(e_i^x)$$

y

$$C_i(e_i^x) \not\leq C_j(e_j^y)$$

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:

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

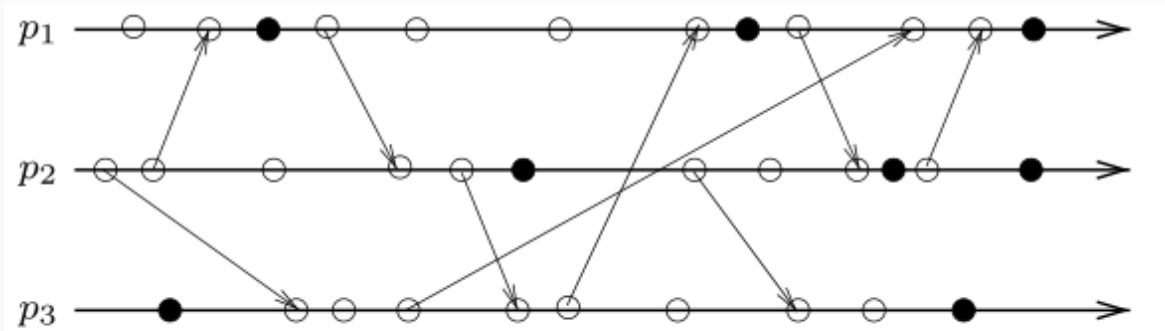
Vector Clocks in Action: Immediate Precessors

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 MSG(m) **to** p_j **do**
(3) send MSG$(m, vc_i[1..n])$ to p_j.

when 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

Vector Clocks in Action: Immediate Precessors

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 $\widehat{R} = (R, \xrightarrow{re})$.

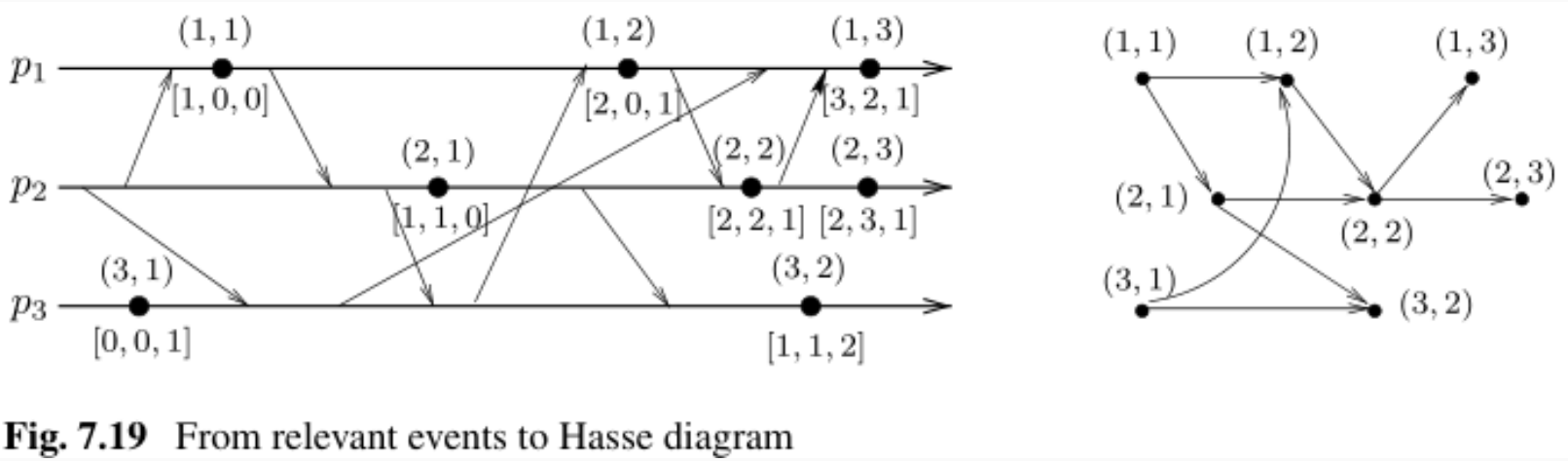


Fig. 7.19 From relevant events to Hasse diagram

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 MSG(m) **to** p_j **do**
(5) send MSG($m, vc_i[1..n], imp_i[1..n]$) to p_j.

when 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)

Casos para la actualización del predecesor inmediato

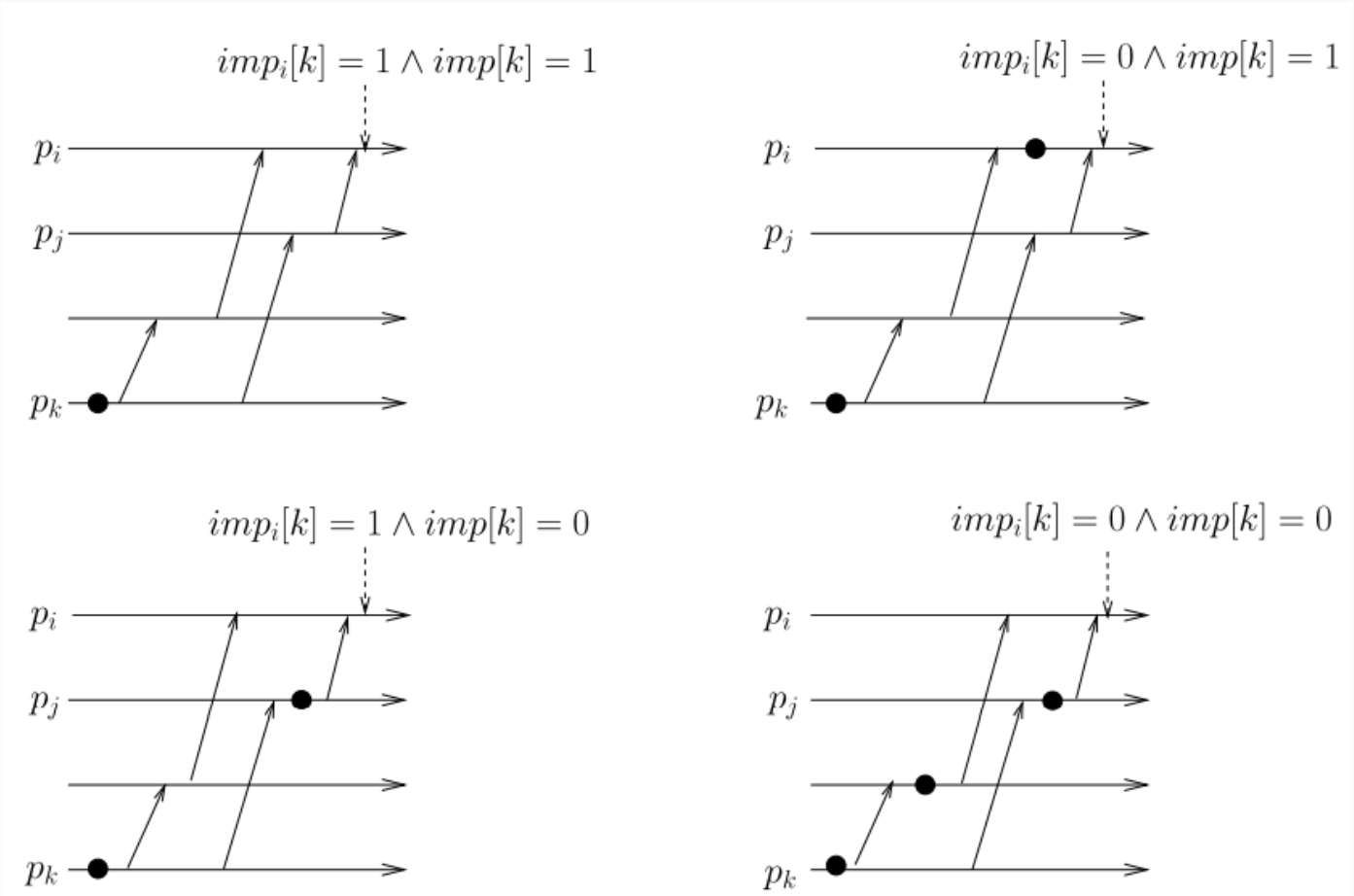


Fig. 7.21 Four possible cases when updating $imp_i[k]$, while $vc_i[k] = vc[k]$

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 σ_i de p_i satisface el predicado LP_i.

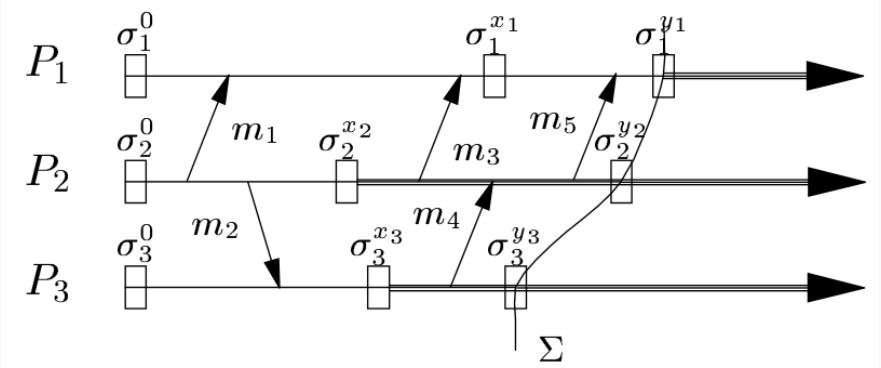
Def. Sea $\{p_1, \cdots, p_n\}$ un sistema distribuido y $LP_1, \cdots, LP_n$ n predicados locales, uno por proceso (con su respectivo proceso). Un *estado global consistente* $\sum = (\sigma_1, \cdots, \sigma_n)$ satisface el predicado global $LP_1 \wedge \cdots \wedge LP_n$ denotado por

$$\sum \models \bigwedge_i LP_i$$

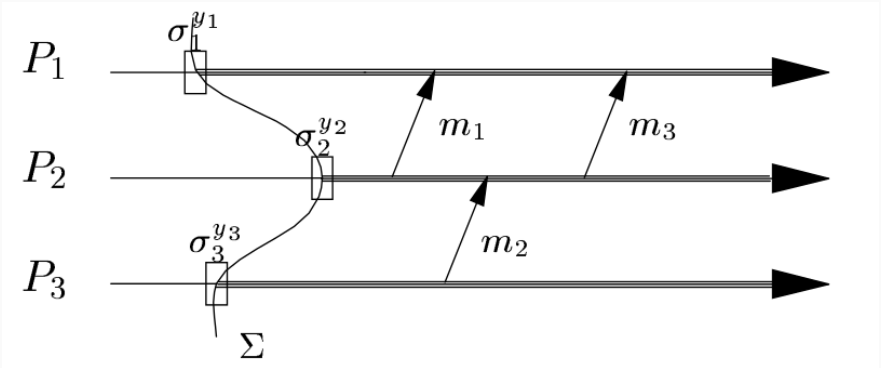
siempre que $\bigwedge_i (\sigma_i \models LP_i)$.

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.



(a) Ejemplo 1.



(b) Ejemplo 2.

Figure: Detección de una conjunción local estable.

Conjuntos Imposibles: Problema.

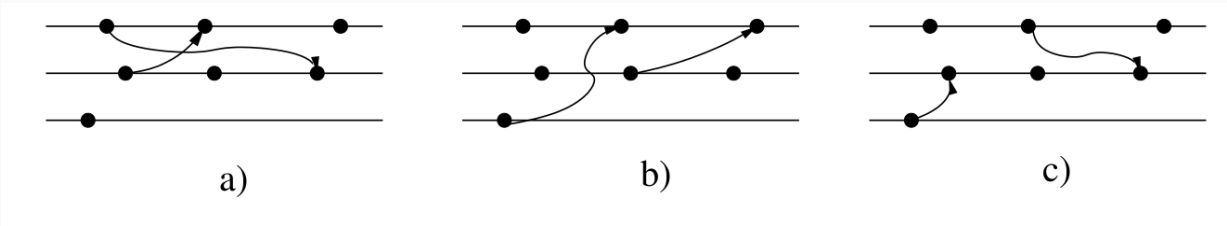
El problema de los *conjuntos imposibles* establece:

Dado un conjunto con n etiquetas vectoriales de k entradas cada una, decidir si es posible o no.

Ejemplo:

$$\mathcal{A} = \{<2,3,1>,<3,0,0>\}$$
$$\mathcal{B} = \{<1,1,1>\}$$

(a) Conjuntos.



(b) Historias.

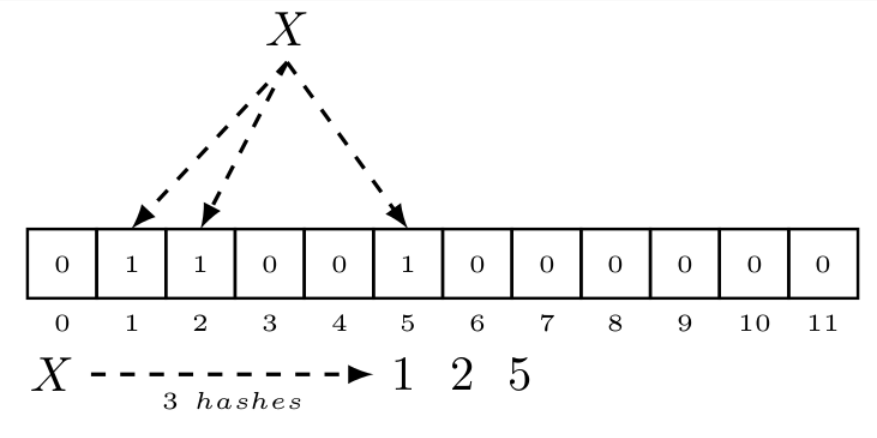
Figure: Detectando conjuntos.

Relojes de Bloom

Def. El *filtro Bloom* es una estructura de datos, probabilística, eficiente en espacio diseñada para verificar rápidamente si un elemento esta contenido en conjunto.

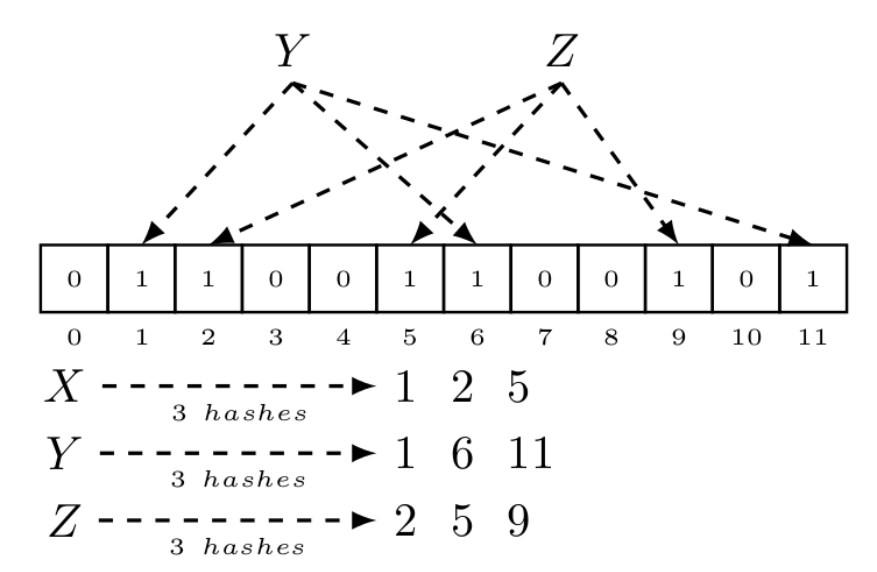
Algoritmo.

1. Definir *k* funciones *HASH* independientes.
2. Definir un arreglo de `bits` con *m* `bits`, todos inicialmente igual a 0.
3. Para agregar un elemento al *filtro bloom*, hacer
   - 3.1 Pasamos el elemento, digamos *x*, por las *k funciones hash*.
   - 3.2 Cada hash apunta a un índice en el arreglo de `bits`
   - 3.3 La posición a la que se apunte cambia de 0 a 1.



(a) Inserción al filtro.

¿Será cierto qué todos los elementos para los cuales las *hash-funciones* nos regresen valores que en el *bloom-filtro* tienen asigando 1, son valores que han sido introducidos al filtro?



(a) Falsos Positivos.

¿Qué podemos hacer? ...

¿Qué podemos hacer? ...
Podemos controlar la tasa de falsos positivos ajustando:

1. El tamaño del *bloom-filtro* (m).
2. El número de elementos insertados en el filtro (n).
3. El número de *hash-funciones* (k) utilizadas en la codificación.

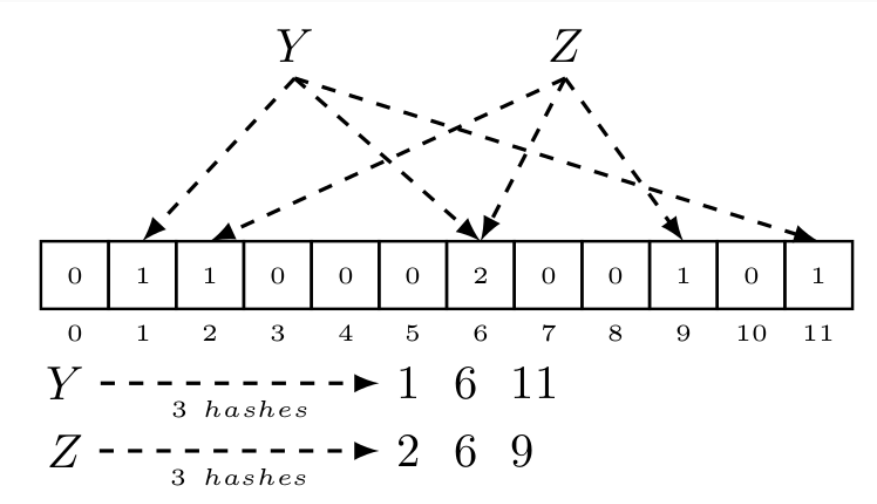
y por medio de la ecuación:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

Idea Intuitiva:

- $1 - \frac{1}{m}$ es la probabilidad de tener un `bit` en el filtro que no este establecido como 1 por alguna determinada función.
- $\left(1 - \frac{1}{m}\right)^{k}$ es la probabilidad de que un bit en particular no se establezca en uno, dado que k hashes pueden señalarlo potencialmente.
- $\left(1 - \frac{1}{m}\right)^{kn}$ es la probabilidad de que un bit en particular siga siendo cero después de n elementos.
- $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$ es la probabilidad de que k índices sean 1 después de insertar n elementos, también conocido como tasa de falsos positivos.

¿Qué podemos hacer? ...
Podemos usar una variante de *bloom-filtros* con conteo de entradas, como se muestra en el siguiente ejemplo:



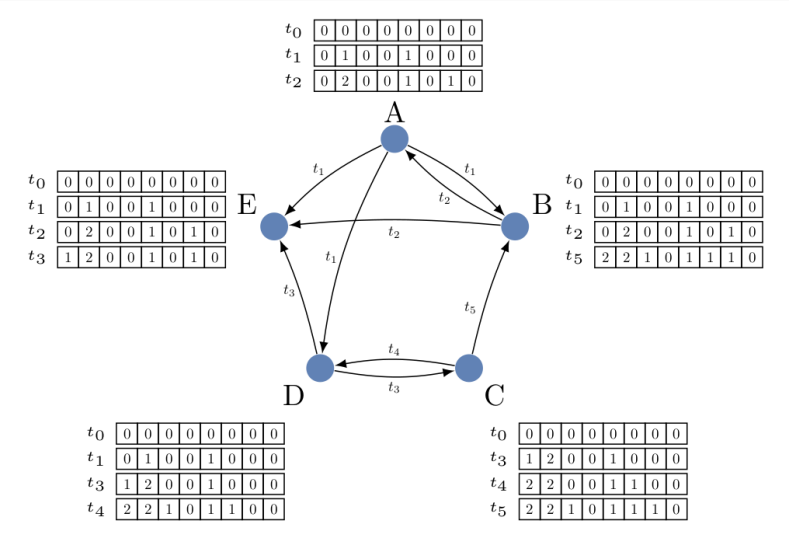
(a) Filtro de Bloom tipo contador.

Relojes Bloom: Algoritmo.

A continuación se describe el algoritmo usado en la dispersión de tiempos:

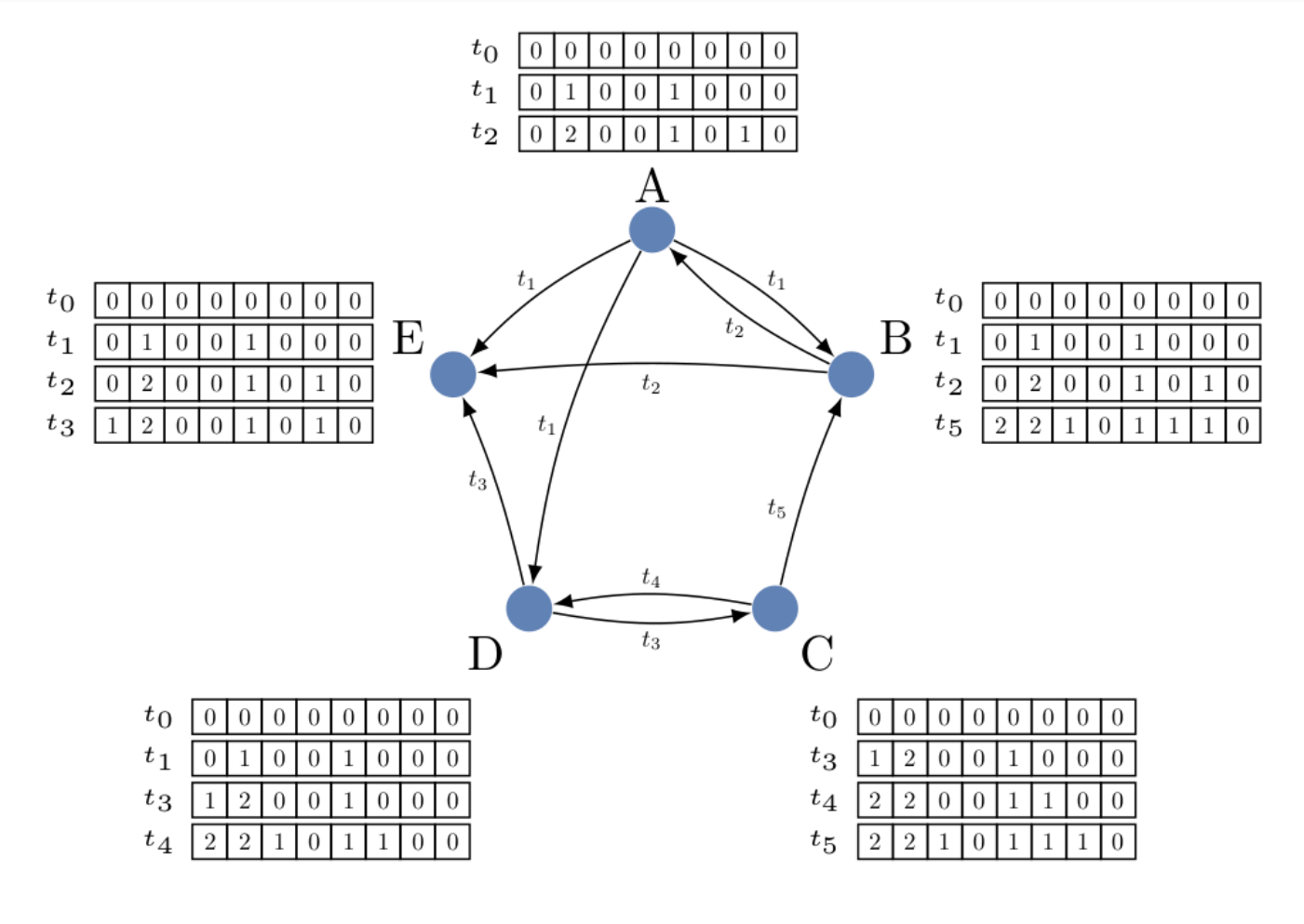
1. Definimos el tamaño del *bloom-filtro-conteo m* y las *k hash-funciones*.
2. Cada vez que un nodo tiene un evento interno, procesa ese evento con *k hash-funciones* e incrementa su filtro. Luego envía ese filtro de floración a todos los demás nodos.
3. Cada vez que un nodo recibe un evento, actualiza su filtro tomando el valor máximo de su filtro y el filtro receptor.

Ejemplo:



(a) Reloj de Bloom.

Ejemplo:



(a) Reloj de Bloom.

¡¡AL FIN!!


MUCHAS GRACIAS
POR PONER ATENCIÓN
ME CONFORMO CON UN 100


FIN DE LA
PRESENTACION
PARA PREGUNTAS CONSULTEN
GOOGLE O WIKIPEDIA, GRACIAS.
memegenerator.es