UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 6

INTEGRANTES

Torres Valencia Kevin Jair - 318331818
Aguilera Moreno Adrián - 421005200
Natalia Abigail Pérez Romero - 31814426

PROFESOR

Miguel Ángel Piña Avelino

AYUDANTE

Pablo Gerardo González López

ASIGNATURA

Computación Distribuida

24 de noviembre de 2022

1. En el problema de `transaction commit` para bases de datos distribuidas, cada uno de los n procesos forma una opinión independiente sobre realizar un `commit` o abortar una transacción distribuida. Los procesos deben tomar una decisión consistente, de modo que si la opinión de un sólo proceso es abortar, entonces, la transacción es abortada y si la opinión es realizar el `commit`, entonces, la transacción es realizada. ¿Se puede solucionar este problema en un sistema asíncrono sujeto a fallas de tipo paro? ¿Por qué sí o por qué no?

No, el problema de `transaction commit` se puede reducir al problema del consenso el cual es imposible en un sístema asíncrono (FLP85). No podemos resolver `transaction commit` dado que no podemos saber si un proceso falló o simplemente sea lento, de manera que no podemos detectar fallas, y no podemos garantizar que todos los procesos conozcan la decisión.

2. Considera la ejecución de la figura 1. Haz lo siguiente:

a) Ejecuta el algoritmo de relojes lógicos y asigna el tiempo lógico a cada evento.

b) Ejecuta el algoritmo de relojes vectoriales y asigna el vector de tiempo a cada evento.

c) Muestra dos cortes consistentes y dos inconsistentes.

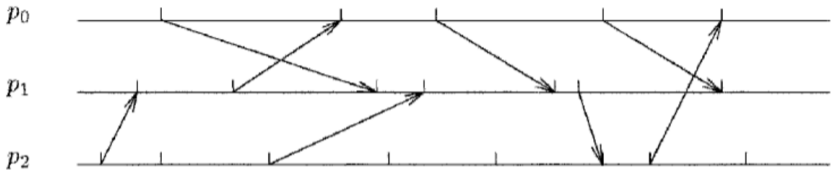
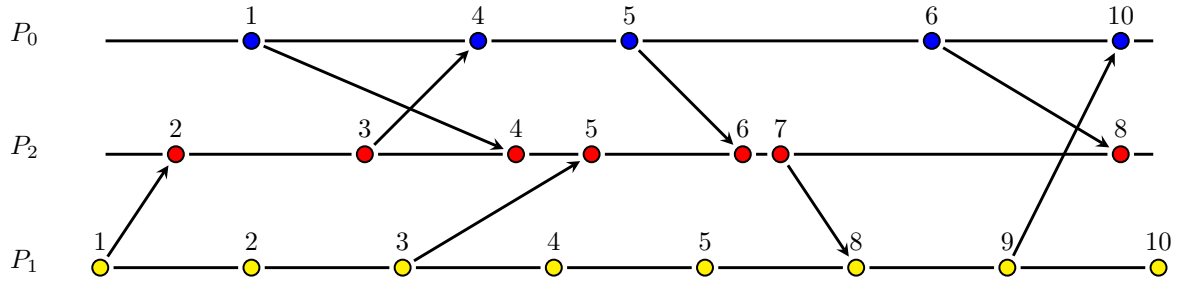
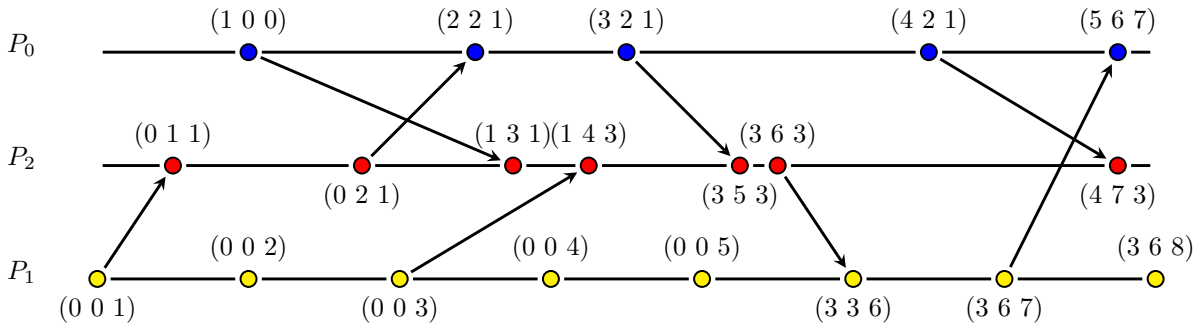


Figura 1: Ejecución de tres procesos

▷ **Solución:**
(*a*) *Relojes lógicos.* A continuación se muestra una ejecución del algoritmo con relojes lógicos:



(*b*) *Relojes Vectoriales.* A continuación se muestra una ejecución del algoritmo con relojes vectoriales:



(*c*) *Corte consistente.*
(*d*) *Corte inconsistente.*

Como podemos notar, en (*d*) el corte inconsistente se da con base en el tercer estado local de P_0. ◁

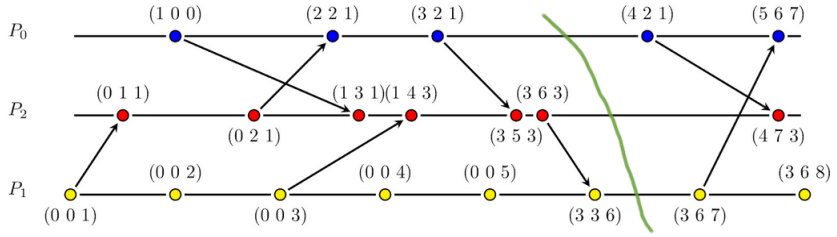


Figura 1: Corte Consistente.

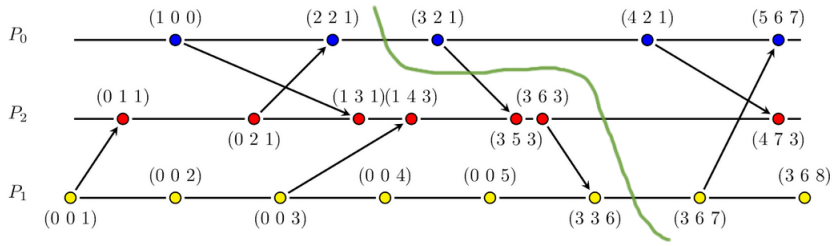


Figura 2: Corte Consistente.

3. Considera el algoritmo de relojes vectoriales y demuestra que toda ejecución se cumple que $e_1 \Rightarrow e_2 \Leftrightarrow VT(e_1) < VT(e_2)$, para cualquier par de eventos e_1, e_2. Decimos que $VT(e_1) < VT(e_2)$ si y sólo si $VT(e_1) \neq (e_2)$ y $VT(e_1) \leq VT(e_2)$, donde $\leq$ denota el orden parcial definido en clase sobre vectores n-dimensionales con entradas enteras.

$\triangleright$ Para esta prueba analicemos 2 posibles casos:

$\Rightarrow$) Notemos que el tiempo en la i-ésima posición del vector, correspondiente a p_i como estado local y aumenta a lo largo de su "vida", además cada intercambio de información resulta en la combinación del vector de llegada con el vector del proceso recepción, así

$$(e_1 \Rightarrow e_2) \Rightarrow VT(e_1) < VT(e_2).$$

$\Leftarrow$) Sabemos que solo el proceso p_i es el causante del incremento en la i-ésima entrada de cualquiera de los vectores en el sistema. Como $VT(e_1) < VT(e_2)$, en particular $e_1.Vt[i] < e_2.Vt[i]$, por lo que concluimos que

$$VT(e_1) < VT(e_2) \Rightarrow e_1 \Rightarrow e_2.$$

De lo anterior se concluye la prueba. $\triangleleft$

4. Decimos que un canal de comunicación entre dos procesos p_i y p_j es `FIFO` si los mensajes se reciben en el mismo orden en el que se envían. Entonces, si el canal `NO` es `FIFO`, puede ser que p_i envía primero M_1 y después M_2, pero p_j reciba primero M_2 y después M_1.

Dados dos procesos que comparten un canal C que no es `FIFO`, da un algoritmo que implemente un canal `FIFO` sobre C. Tu algoritmo debe tener dos secciones: una sección **send**, que recibe como entrada un mensaje M a enviarse (el cuál se envía finalmente por C), y una sección **receive**, que recibe un mensaje M de C y lo entrega. De esta forma, otro algoritmo que esté diseñado para canales `FIFO` puede usar tu algoritmo para enviar y recibir mensajes. Este esquema se puede observar en la figura 2. Argumenta que tu algoritmo es correcto. Tip: piensa en `timestamps` y considera que un mensaje que se recibe de C no tiene que entregarse inmediatamente.

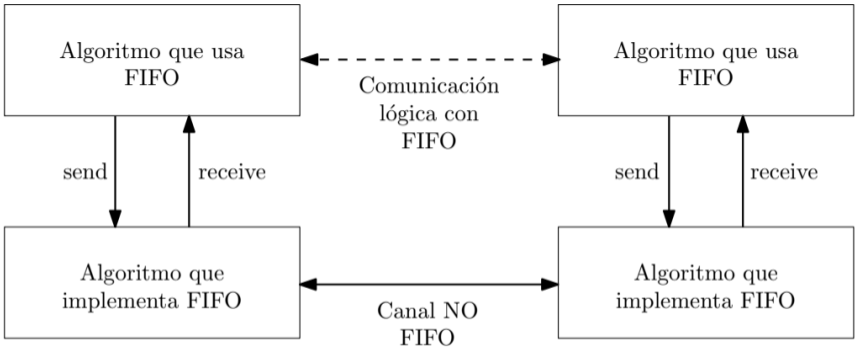


Figura 2: Esquema de la implementación de canales FIFO

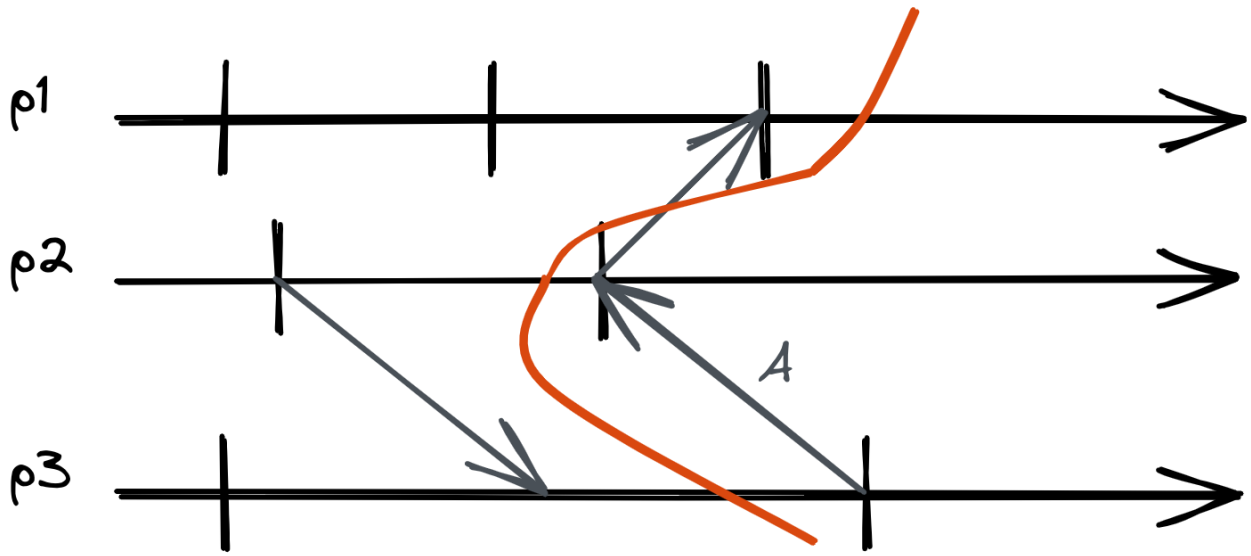
Algoritmo: No_FIFO.
1. // F. Auxiliar que realiza los registros (locales).
2. record():
3. $\sigma_i \leftarrow$ estado actual de p_i
4. //Estado global respecto a p_i
5. globalSt $\leftarrow$ rojo // *rojo* si el estado esta registrado
6. **registra** σ_i receive[c_entrada], **and** sent[c_salida]
7. **send** lo anterior a *pc* //Sea *pc* un proceso controlador.
8.
9. Al inicializarse:
10. **if** $globalSt = verde$ **then** $record()$ **end if**
11.
12. //Seccion receive
13. Al recibir un mensaje (m) $entra\_canal[j]$:
14. $receive[j] \leftarrow receive[j] \cup \{m\}$
15. **if** $(m.color = rojo) \ \wedge (globalSt = verde)$ **then** $record()$ **end if**
16. pasa el mensaje a p_i
17.
18. //Seccion send
19. Al enviar un mensaje (m) $sale\_canal[j]$:
20. $m.color \leftarrow globalSt$
21. $sent[k] \leftarrow sent[j] \cup \{m\}$

Cuando el controlador recibe cada pc_i (Nota: pc es un proceso controlador), los reúne para obtener la suma de cada uno de los estados de p_i. Y la suma de los canales C, se definirá el estado del canal de p_j p_i como: $c\_estado(j, i) = sent[i]$ *and* $receive[j]$.

El algoritmo es correcto, ya que de la definición se tiene que $sent[i]$ es el conjunto de mensajes enviados por p_j a p_i antes del estado j (σ_j) y también de que $receive[j]$ es el conjunto de mensajes recibidos de p_i a p_j antes del estado i (σ_i), que el estado de $c\_estado(j, i)$, registra los mensajes que están en translado respecto al par ordenado de los estados (σ_j, σ_i).

Como no se encuentran mensajes perdidos respecto a los estados de manera local registrados, por lo que el estado global al final será consistente.

5. Considera el algoritmo para calcular cortes consistentes visto en clase. Da una ejecución para tres procesos en la que los canales de comunicación no son `FIFO` y el corte calculado no es consistente.



La siguiente ejecución no es `FIFO` dado que el mensaje con etiqueta `A` es recibido por el proceso A antes del mensaje que indicaría el corte. Notar que el siguiente corte (indicado por la linea naranja) no es consistente dado que el proceso 1 recibe un mensaje enviado por el proceso 2 el cual es un evento no contemplado en el corte.