Tarea 5

Ian Israel García Vázquez Armando Ramírez González
Ricardo Montiel Manriquez Christopher Alejandro Escamilla Soto
Jonás García Chavelas

25 de noviembre de 2021

1. El algoritmo de consenso bizantino visto en clase, resuelve el problema del consenso bizantino para $f < \frac{n}{4}$ procesos. Para valores más grandes de f, el algoritmo podría fallar por violar una o más de las propiedades de terminación, validez o acuerdo. Para este algoritmo:

   - ¿Qué tan grande debe ser f para evitar terminación?

     La terminación nunca la vamos a poder romper porque, aunque haya "n" procesos fallidos la terminación por vacuidad se va a cumplir, porque todos los procesos correctos van a decir un valor, como no hay ningún proceso correcto, esto va a estar bien porque por vacuidad se cumple la terminación.

   - ¿Qué tan grande debe ser f para evitar validez?

     Si todos los procesos correctos inician con el mismo valor, es porque hay concenso, entonces si $f = \frac{n}{4}$ es porque el concenso no se mantiene durante toda la ejecucion porque cada proceso correcto ve la condicion $n - f > \frac{n}{2} + f$ como falso, puesto que:

     $$n - f \not> \frac{n}{2} + f$$
     $$n - \frac{n}{4} \not> \frac{n}{2} + \frac{n}{4}$$
     $$\frac{3n}{4} \not> \frac{3n}{4}$$

     $\therefore$ Se establece como propuesta al coordinador de la fase.

   - ¿Qué tan grande debe ser f para evitar acuerdo?

     Tenemos que $f = \frac{n}{4}$, porque el algoritmo se ejecuta en $f + 1$ fases y hay a lo mas f fallas, por lo que se tiene una fase en la que el coordinador es correcto y si en la primera ronda de esa fase correcta no se ha llegado al concenso, entonces el coordinador de la fase impondria su valor para que se llegue a un concenso, pero como cada proceso correcto ve la condición $n - f > \frac{n}{2} + f$ como falsa entonces el concenso se romperia en las todas las fases posteriores.

     $\therefore$ Se establece como propuesta al coordinador la fase.

2. 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 solo 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?

   El problema no se puede solucionar en un sistema asíncrono sujeto a fallas de tipo paro.

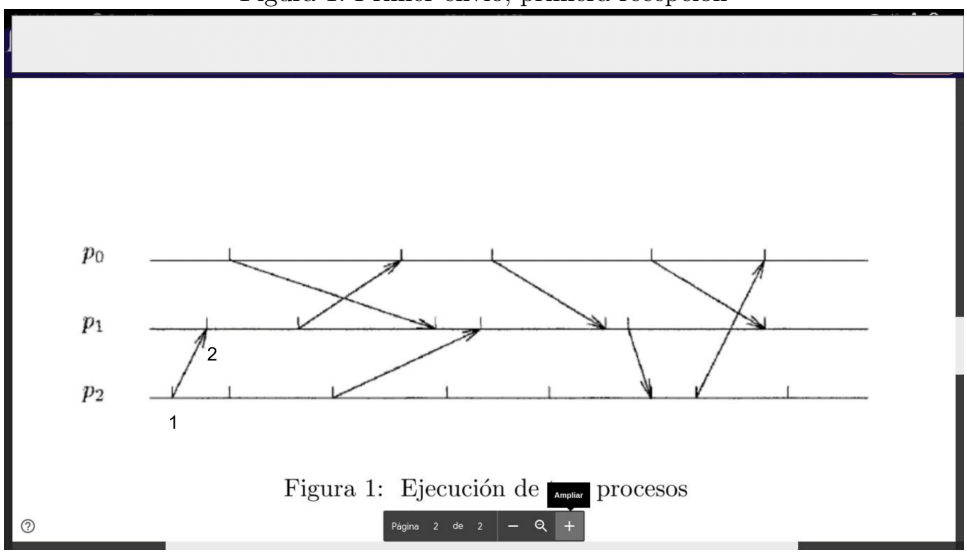
   ¿Por qué no? El principal problema es que no estamos trabajando sobre un sistema de comunicación síncrona y eso nos imposibilita llegar a un consenso porque no es posible distinguir el estado de un proceso del que no se ha recibido mensaje si este es un proceso lento o es un proceso fallido es decir como en nuestro problema de transaction commit necesitamos tener el estado de todos procesos y si existe un proceso lento la ejecución de ese sistema será tan larga como el tiempo que tarden en llegar los mensajes (sin garantía de tiempo) de cada uno de los procesos sin embargo el problema de transaction commit se vuelve imposible de resolver cuando existe un proceso fallido pues con un solo proceso fallido podemos concluir gracias al teorema FLP 85 será un sistema sin solución, Está es la razón por la que no podremos resolver este problema en un sistema asíncrono sujeto a fallas de tipo paro.

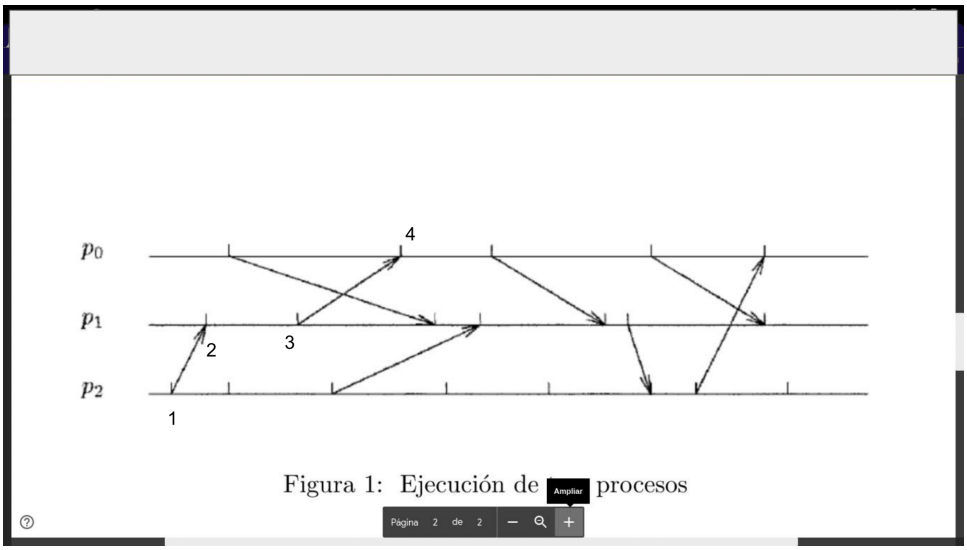
3. Considera la ejecución de la figura 1. Haz lo siguiente:

   - Corre el algoritmo de relojes lógicos y asigna el tiempo lógica a cada evento

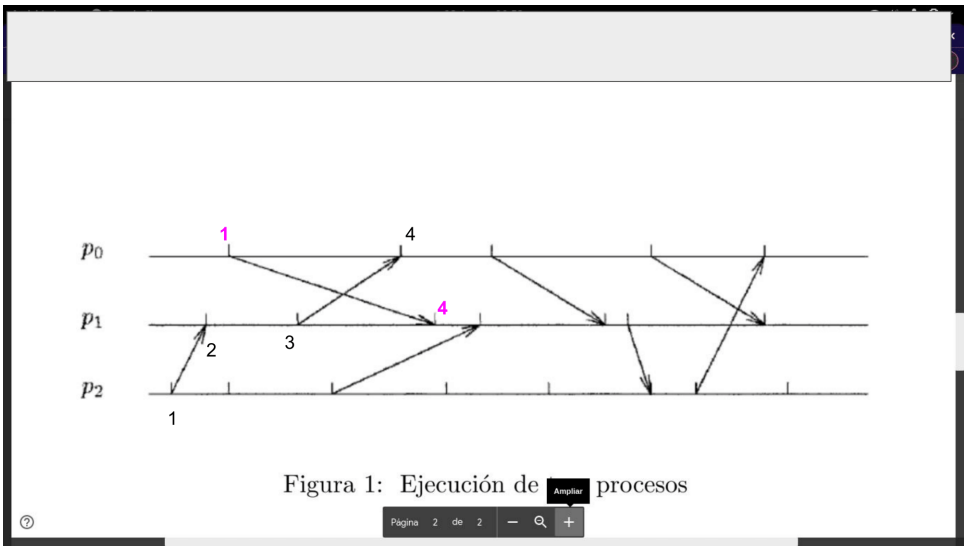
     En este primer paso, $p2$ ha comenzado su proceso, marcado con 1, enviado a $p1$ y marcando este con 2, agregamos 1 al proceso inicial.

Figura 1: Primer envío, primera recepción





Ahora en el local de $p1$ este ha aumentado 1, obteniendo un valor de 3 y es enviado por $p1$ a $p0$, escogiendo el máximo entre el, aún no señalado, 1 local de $p0$



Señalamos el evento 1 del proceso $p0$, enviandolo a $p1$, para seleccionar $max\{1, 3\}$ y sumar 1 a la selección, propia del caso 3, para que esta terminé luciendo como 4

Finalizamos el algoritmo ejecutandola hasta finalmente obtener 10 en $p0$ y 8 en $p1$

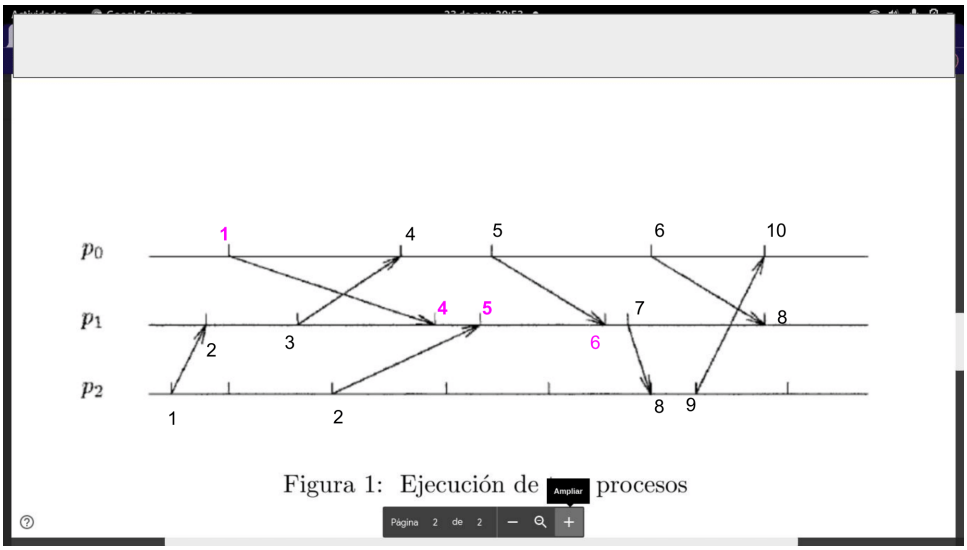


Figura 1: Ejecución de procesos

- Corre el algoritmo de relojes vectoriales y asigna el vector de tiempo a cada evento

En esta parte del proceso, hemos inciado con vectores que se encontraban en $0's$, el primer evento en ser señalado es el correspondiente a $p2$, este se visualiza como $(0, 0, 1)$, luego al ser enviado a $p1$, el componente correspondiente a $p1$ es inicializado, ahora el vector se visualiza como $(0, 1, 1)$, ahora en lo local de $p1$, se aumenta el mismo componente, visualizandose ahora como $(0, 2, 1)$; por otro lado, en $p0$ se ha inicializado localmente un proceso $p0$, este se mira como $(1, 0, 0)$, luego es enviado a $p1$, seleccionando el máximo de los componentes correspondientes, para verse $(1, 3, 1)$, asimismo el vector $(0, 2, 1)$ de $p1$ es enviado a $p0$ y siguiendo el proceso de selección de los máximos en los componentes observamos $(2, 2, 1)$ como vector resultante, hasta esta fase.

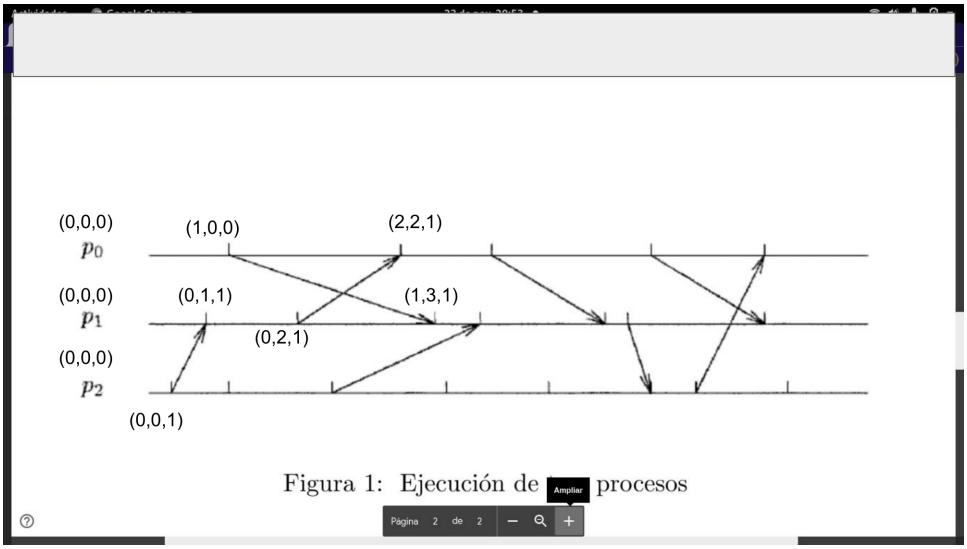
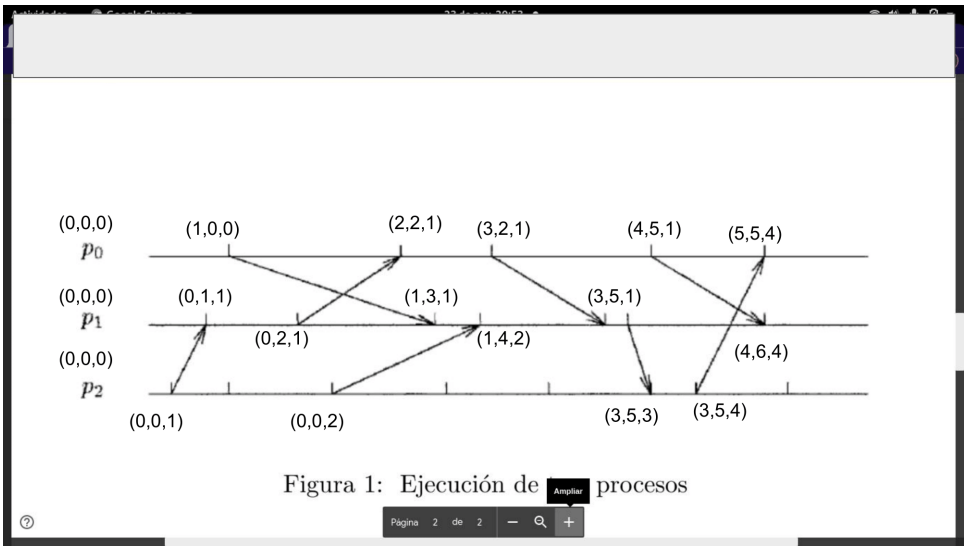
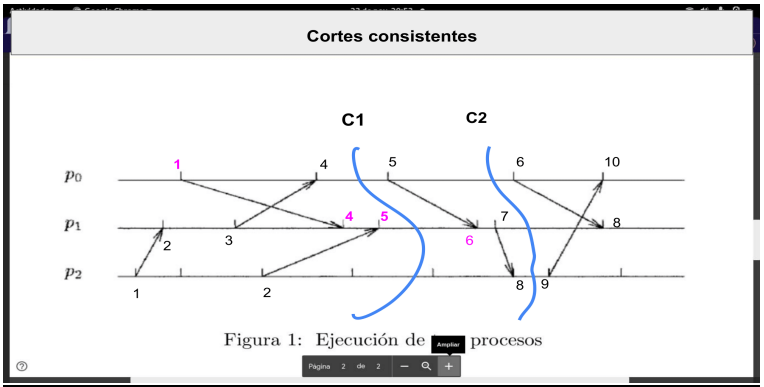


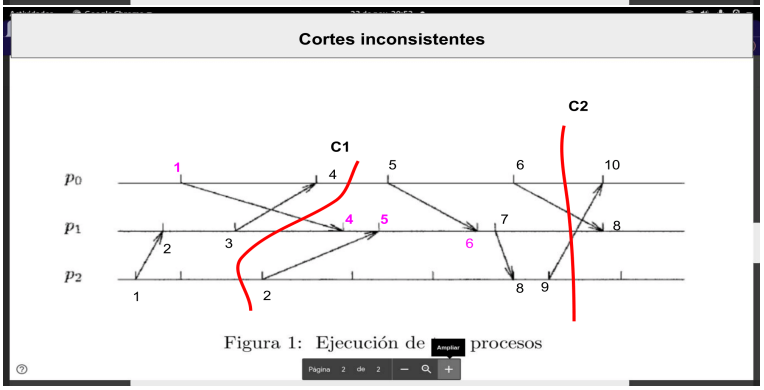
Figura 1: Ejecución de procesos

Finalizamos el algoritmo de relojes vectoriales, siguiendo y repitiendo lo anterior, hasta los eventos finales de los procesos $p0$ y $p1$ con $(5, 5, 4)$ y $(4, 6, 4)$ respectivamente



- Muestra dos cortes consistentes y dos cortes inconsistentes





4. Considera el algoritmo de relojes vectoriales y demuestra que toda ejecución se cumple que $e1 \Rightarrow e_2 \Leftrightarrow VT(e_1) < VT(e_2)$, para cualquier par de eventos e_1, e_2. Decimos que $VT(e_1) < VT(e_2)$ si y sólo si $VT(e_1) \neq VT(e_2)$ y $VT(e_1) \leq VT(e_2)$, donde $\leq$ denota el orden parcial definido en clase sobre vectores $n - dimensionales$ con entradas enteras.

   Demostraremos primero $e1 \Rightarrow e_2 \Rightarrow VT(e_1) < VT(e_2)$, por lo que afirmamos $e1 \Rightarrow e_2 \Rightarrow VT(e_1) <$

$VT(e_2)$, debemos demostrar:

i Sean e_1 y e_2 dos eventos consecutivos de un mismo proceso.
Por el algoritmo de relojes vectoriales, al ser un evento local tenemos que $VT(e_1) < VT(e_2)$, por lo tanto se cumple.

ii Sea e_1 un envío de un mensaje M y sea e_2 una recepción del mensaje M de e_1
Como e_1 es un envío de mensaje, e_2 recibe el valor de $e_1 + 1$ por el algoritmo de relojes vectoriales, por lo tanto se cumple que $VT(e_1) < VT(e_2)$.

iii **Transitividad** PD. Que para el conjunto $F = \{f_0, \ldots, F_k\}$ ocurre que $e_1 \Rightarrow f_0 \Rightarrow \cdots \Rightarrow f_k \Rightarrow e_2$.
Por el algoritmo tenemos que para cualquier pareja de eventos $f_i, f_j \in \{e_1, e_2\} \cup F$ ocurre que:
1. Si son dos eventos locales de un mismo proceso, ocurre que $VT(f_i) < VT(f_j)$.
2. Si f_i es un envío de mensajes y f_j una recepción de mensajes por el inciso *ii* tenemos que $VT(f_i) < VT(f_j)$.
Entonces por los incisos 1 y 2 tenemos que $VT(f_i) < VT(f_j)$ y como $f_i, f_j \in \{e_1, e_2\} \cup F$ ocurre que $VT(e_1) < VT(f_j) \ldots < VT(f_j) < VT(e_2)$.
Por lo tanto se cumple que $VT(e_1) > VT(e_2)$.

$$\text{Por lo tanto } e1 \Rightarrow e_2 \Rightarrow VT(e_1) < VT(e_2).$$

Por lo que se cumple que $e1 \Rightarrow e_2 \Rightarrow VT(e_1) < VT(e_2)$.

$\diamond$ Ahora demostraremos $e1 \Rightarrow e_2 \Leftarrow VT(e_1) < VT(e_2)$ **Por contrapositiva**

Supongamos que $e1 \not\Rightarrow e_2$. Por demostrar que $VT(e_1) \geq VT(e_2)$:

Como $e_1 \not\Rightarrow e_2$ tenemos que no cumple con las propiedades *i* y *ii* del paso anterior, por lo que se cumple que $VT(e_1) \geq VT(e_2)$.

Además tenemos que para el conjunto $F = \{f_0, \ldots, F_k\}$, ocurre que para cualquier pareja $f_i, f_j \in \{e_1, e_2\} \cup F$ tal que $VT(f_i) \geq VT(f_j)$ tenemos que $VT(e_1) \not\Rightarrow VT(f_j) \not\Rightarrow \cdots \not\Rightarrow VT(f_j) \not\Rightarrow VT(e_2)$, es decir, $VT(e_1) \geq VT(f_j) \geq \cdots \geq VT(f_j) \geq VT(e_2)$.

$$\text{Por lo tanto se cumple } e1 \not\Rightarrow e_2 \Rightarrow VT(e_1) \geq VT(e_2).$$

$$\text{Por lo tanto se cumple } e1 \Rightarrow e_2 \Leftarrow VT(e_1) < VT(e_2).$$

$$\text{Por lo tanto se cumple la proposición } e1 \Rightarrow e_2 \Leftrightarrow VT(e_1) < VT(e_2)$$

5. Decimos que un canal de comunicación entre dos procesos pi y pj 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 pi envíe primero M1 y después M2, pero pj reciba primero M2 y después M1.

   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 cual 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

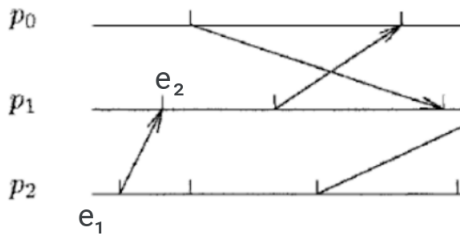
tu algoritmo para enviar y recibir mensajes. Este esquema se puede ver en la figura 2. Argumenta que tu algoritmo es correcto. Tip: Piensa en timestamps y considera que un mensaje que se recibe de C no tiene que entregarse inmediatamente.

Algoritmo5():

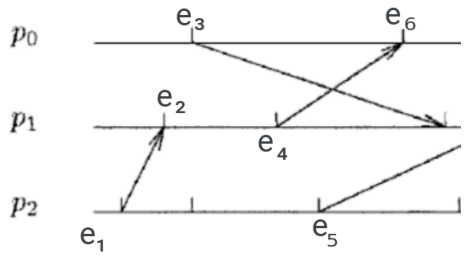
```
1
2       t = 0
3
4       SEND (M);
5       Al ejecutar un envio (e):
6       t = t + 1
7       LT(e) = t
8       send(<M, LT(e)>)
9
10      Al recibir un mensaje <M, LT(e)> (e'):
11      if(LT(e) == t++) then
12          t = max(t, LT(e)) + 1
13          LT(e') = t
14          enviar al canal C(M)
15      else
16          Al recibir un mensaje <M, LT(e)> (e'):
17
```

En el algoritmo tenemos dos secciones, una que se encarga de enviar mensajes donde a cada mensaje se le etiqueta con un valor que representa el tiempo en el que fue enviado. Y la otra sección se encarga de recibir el mensaje con su etiqueta de tiempo, solo bastará hacer una comparación para saber si el mensaje que recibimos es el primero lo mandamos al canal y si no lo es nos quedamos esperando a que llegue. Así aseguramos que estamos enviando los procesos en el orden correcto.

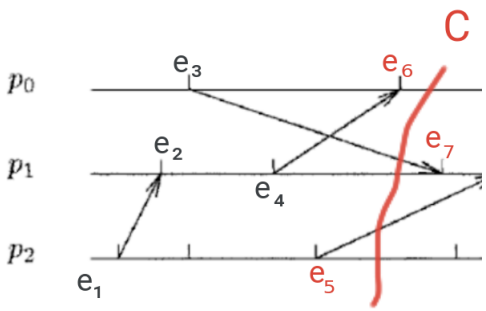
6. Dados tres procesos p_0, p_1 y p_2, suponiendo que no tenemos canales FIFO con los que contar, entonces no podemos asegurar que los mensajes se reciben en el mismo orden que se envían, por lo que habrá algún evento e tal que $e \Rightarrow q_i$ para algún q_i en el corte $\langle q_1, q_2, \dots q_n \rangle$ con $i \leq n$.
Específicamente, si recordamos la ejecución de la Figura 1 podemos observar que p_2 tendrá un evento $e_0 \Rightarrow e_1$ donde e_1 ocurre localmente para p_1, hasta aquí todo ocurre como si contaramos con canales FIFO trivialmente porque todavía no hemos enviado más que un mensaje:



Luego tenemos que para el proceso p_0 ocurre el evento e_2, el cual envía un mensaje a p_1, sin embargo antes de que sea recibido se da el evento e_4 para el proceso p_1, el cual envía un mensaje a p_0. Además ha ocurrido el evento e_5 en p_2, después del cual el mensaje enviado por p_1 causa el evento e_6 en p_0.



Supongamos que a partir de e_6 se decide realizar un corte, entonces obtendremos el corte $C = \langle e_5, e_6, e_7 \rangle$, sin embargo tenemos el evento $e_3 \Rightarrow e_7$, por lo que el corte es no es consistente:



Por dicho ejemplo no podemos asegurar que el algoritmo de cortes consistentes funcione si los canales de comunicación no son FIFO.