Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación Distribuida

Tarea 7

Johann Ramón Gordillo Guzmán
418046090

Tarea presentada como parte del curso de **Computación Distribuida** impartido por la profesora **M.C Karla Rocío Vargas Godoy**.

01 de Diciembre del 2020

Link al código fuente: https://github.com/JohannGordillo/

Actividades

1. (2.5 ptos) Demuestra que solo hay un corte consistente maximal que precede a cualquier corte.

   Demostración.

   Sea r un corte no consistente y procedamos por contradicción. Supongamos que hay más de un corte consistente maximal de r. Sean entonces S_1 y S_2 cortes consistentes maximales de r.

   Tenemos los siguientes casos:

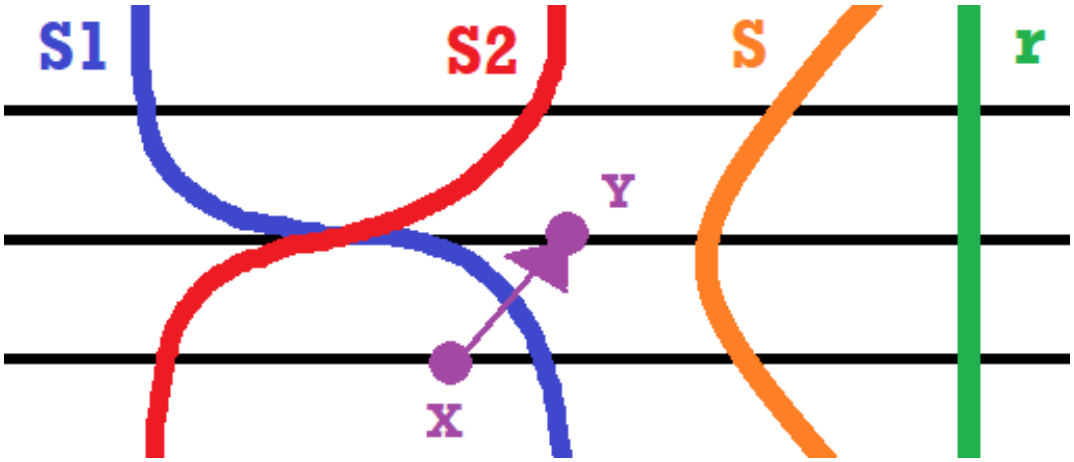
   - **Caso 1:** Los cortes no se cruzan.

     Este caso lo revisamos en clase. Como los cortes no se cruzan, S_2 es más reciente.

   - **Caso 2:** Los cortes se cruzan.

     En este caso, sin perdida de generalidad, supongamos que existe un evento X en S_1 tal que X no está en S_2 y X es el envío de un mensaje. Sea Y el evento de recepción del mensaje enviado en X. Como S_2 es consistente, necesariamente Y ocurrió en el futuro de S_2. Entonces podemos formar un corte S tal que cubra los eventos de S_2, pero que además cubra a X y a los eventos que suceden después de X, incluyendo a Y. Así, S es un corte consistente de r más grande que los cortes que definimos, pero esto es una contradicción a que S_1 y S_2 son maximales, por lo que $S_1 = S_2$.

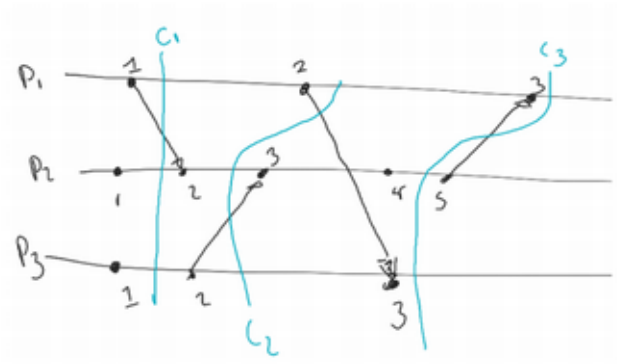
     El argumento anterior se ve como sigue:

   $\therefore$ Hay un único corte consistente maximal de r. ■

2. (2.5 ptos) Realiza lo que se pide.

   - ¿Cuáles de los siguientes cortes son consistentes? De los cortes que no fueron consistenes di cual sería su corte consistente maximal.



Respuesta.

Consideremos los cortes por separado:

- Corte C_1.

  Este corte **es consistente**, pues los eventos 1 de P_2 y P_3 representan un cómputo local, y el evento 1 de P_1 representa el envío de un mensaje en el pasado del corte que es recibido en el futuro del mismo, pero esto no es un problema, ya que un corte es inconsistente solo si hay un mensaje recibido en el pasado que haya sido enviado en el futuro de un corte.

- Corte C_2.

  Este corte **es consistente**, pues los eventos de envío de mensajes ocurren en el pasado del corte y la recepción de los mismos ocurre en el futuro del corte.

- Corte C_3.

  Este corte **no es consistente**, pues en el pasado del corte hay una recepción de un mensaje enviado en el futuro. A saber, el evento 5 de P_2 es el envío de un mensaje a P_1, y este evento pertenece al futuro del corte, sin embargo, la recepción del mensaje (representada como el evento 3 de P_1) ocurre en el pasado del corte.

  El **corte consistente maximal** de C_3 es el corte $C = (2, 4, 3)$.

$\therefore$ C_1 y C_2 son consistentes, pero C_3 no.

- Ejecuta el algoritmo de snapshot en la ejecución anterior cuando el proceso P_3 recibe la instrucción de tomar un snapshot después de su evento 3. Los canales son FIFO.

  Respuesta.

  Consideremos el algoritmo de snapshot y la ejecución anterior. Inicialmente tenemos:

  $$\begin{aligned} num_1 &= 0 \\ ans_1 &= \perp \\ num_2 &= 0 \\ ans_2 &= \perp \\ num_3 &= 0 \\ ans_3 &= \perp \end{aligned}$$

  Depués del evento 3 de P_3, este proceso recibe la instrucción de tomar un snapshot. Una vez recibida esta instrucción, P_3 envía un marcador a todos los vecinos y hace $ans_3 = num_3 = 3$.

  Luego, como los canales son FIFO (First In, First Out), el proceso P_2 recibe el marcador antes de su evento 5 y el proceso P_1 recibe el marcador antes de su evento 3. Así, los procesos P_2 y P_3 actualizan sus valores:

  $$\begin{aligned} ans_1 &= 2 \\ ans_2 &= 4 \end{aligned}$$

  Por lo que el **corte** generado es $(2, 4, 3)$, que es **consistente**.

  Notemos que como $ans_i \neq \perp$ para todo proceso P_i, ans_i ya no se actualizará de nuevo y los procesos ya no se enviarán marcadores entre sí.

3. (2.5 ptos) Diseña un algoritmo para hacer broadcast de cierto mensaje M cada T unidades de tiempo. IMPORTANTE: No es necesario distinguir entre cada ola de mensajes.

Respuesta.

Que no sea necesario distinguir entre cada ola de mensajes significa que hacemos el envío del mensaje M sin distinguir de qué ronda fue, lo único que nos interesa es mandar M cada T unidades de tiempo.

Algoritmo 1 Broadcast Asíncrono

```
1:  Every process p_i executes the following
2:  clock()                                   ▷ Reloj local de p_i. Inicialmente es 0
3:  Every T units of time do
4:      if p_i = p_s then                     ▷ Si p_i es el proceso distringuido
5:          for p_j ∈ neighbors_i do
6:              send M to p_j
7:          end for
8:      end if
9:  When M is received from p_j do
10:     for p_k ∈ neighbors_i \ {p_j} do
11:         send M to p_k
12:     end for
```

4. ¿Para qué se usan los timeouts? Da ejemplos en la vida real donde sean usados.

   Respuesta.

   Los timeouts son deadlines que usamos para evitar que un procesador espere para siempre un mensaje. Son un límite de espera, que una vez cumplido, dejamos de esperar y procedemos a realizar otra acción. Por ejemplo, en la implementación del Detector de Fallos Eventualmente Perfecto que vimos en clase, usamos timeouts para que, cumplido el tiempo de espera, empecemos a considerar a un proceso, del que esperamos un mensaje, como sospechoso.

   Un ejemplo de timeouts en la vida real es en los smartphones y en las laptops. Cuando no se están usando esperan un cierto tiempo con la pantalla encendida antes de apagarse.

   Otro ejemplo es en League of Legends, y en los juegos online en general. Hay un límite de tiempo de inactividad que, de ser superado, el servidor te saca de la partida.

   Y un último ejemplo es cuando intentamos hacer ping a un servidor. Hay un cierto límite de espera de respuesta, y si no es recibida ninguna pasado este límite, se cierra la conexión.

Referencias

[1] Raynal, M. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.

[2] Aspnes, J. *Notes on Theory of Distributed Systems.* Yale University, 2017.