Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación Distribuida

Tarea 5

Johann Ramón Gordillo Guzmán
418046090

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

10 de Noviembre del 2020

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

Actividades

1. (2 ptos) Considera el siguiente algoritmo distribuido para un sistema síncrono que calcula la la distancia entre la raíz de una gráfica y el nodo que se está visitando.

Algoritmo 1: Calcula distancia desde la raíz

```
neighbors_i = {Conjunto de vecinos de i};
distance_i;
inicio
    si p_s = p_i entonces
        distance_i = 0;
        para cada j en neighbors_i hacer
            send distance_i + 1 to p_j
        fin
    en otro caso
        distance_i = ∞;
    fin
fin
When: distance_j es recibido de p_j
inicio
    si distance_j < distance_i entonces
        distance_i = distance_j;
        para cada k en neighbors_i hacer
            send distance_i + 1 a p_k
        fin
    fin
fin
```

Demuestra por **inducción** que si un proceso está a distancia k de la raíz, entonces al finalizar la ronda k debe tener $distance_i = k$.
Considera los siguientes puntos para tu demostración:

- Sobre qué estás haciendo inducción.
- Cuál es tu hipótesis, base y paso inductivo.
- Argumentar correctamente en el paso inductivo.
- Al final escribir cuál es la conclusión de tu demostración.

Si alguno de los puntos anteriores no está en tu demostración se bajarán puntos.

Demostración.

Sea $G = (\Pi, C)$ un sistema distribuido síncrono.

Procedemos por inducción sobre k el número de ronda.

- **Caso Base.**

  En la ronda $k = 0$ los procesos inicializan sus valores. La raíz inicializa su atributo $distance_i = 0 = k$ y por ser la raíz, claramente está a distancia 0 de si misma.

Como el sistema distribuido G es síncrono, en cada ronda vamos avanzando una unidad de distancia, por lo que en la ronda $k = 1$, los vecinos de la raíz recibirán el mensaje $distance_0 + 1 = 1$ y como su $distance_i$ actual es ∞ y $1 < \infty$, al finalizar la ronda actualizarán su distancia a $distance_i = 1 = k$.

$\therefore$ Se cumple el caso base.

- **Hipótesis de Inducción.**

  Supongamos que si un proceso está a distancia k de la raíz, entonces al finalizar la ronda k debe tener $distance_i = k$.

- **Paso Inductivo.**

  Al finalizar la ronda k, por hipótesis los nodos a distancia k tienen ya $distance_i = k$. Luego, como el sistema distribuido es síncrono, avanzaremos una unidad de distancia y el mensaje $k+1$ llegará a los vecinos (en caso de tenerlos) de los nodos a distancia k de la raíz en la ronda $k + 1$. Ahora, como $k + 1 < \infty$, los nodos que reciben el mensaje $k + 1$, y que no han recibido un mensaje anteriormente, actualizarán su distancia a $distance_i = k + 1$. Esto es, al finalizar la ronda $k + 1$ los nodos a distancia $k + 1$ de la raíz tendrán $distance_i = k + 1$.

  $\therefore$ Los nodos a distancia $k + 1$ de la raíz tienen $distance_i = k + 1$ al finalizar la ronda $k + 1$.

$\therefore$ Un proceso a distancia k de la raíz, al finalizar la ronda k debe tener $distance_i = k$. ■

2. (3 ptos) Escribe el protocolo de consenso para sistemas asíncronos sin fallas.

   Respuesta.

   Para implementar el protocolo de consenso en sistemas asíncronos, podemos utilizar la primitiva *wait*, que nos permite esperar hasta que una condición se cumpla, para que un proceso espere a que todos los vecinos le manden mensaje para poder avanzar. Es imposible que un proceso se quede colgado esperando mensajes de un vecino, pues en el sistema no hay fallas y todos los mensajes se entregarán.

   La condición del wait nos dirá que esperemos los mensajes de todos los vecinos, es decir, de todos los vértices de la gráfica (por se una gráfica completa), para poder avanzar. Una vez que un proceso recibe todos el mensaje de todos los vecinos, podemos proceder a iterar sobre los mensajes recibidos para llenar la lista rec_from.

Mi implementación es la siguiente:

Algoritmo 1 Protocolo de consenso en sistemas asíncronos sin fallas

```
1: Function Consensus(v_i)

2: V_i ← [⊥, ..., v_i, ..., ⊥];
3: New_i ← {(v_i, i)};
4: when r = 1, 2, ..., f + 1 do
5: begin round
6:     if(New_i ≠ ∅) then
7:         foreach j ≠ i do
8:             send (New_i) to p_j                          ▷ Enviamos el mensaje
9:         end for
10:    end if
11:    wait until all messages are received                 ▷ Esperamos todos los mensajes
12:    for j ≠ i do
13:        rec_from[j] ← set received from p_j (∅ if no msg);
14:    end for
15:    New_i ← ∅;
16:    foreach j ≠ i do
17:        foreach (v, k) ∈ rec_from[j] do
18:            if then
19:                V_i[k] ← v;
20:                New_i ← New_i ∪ {(v, k)};
21:            end if
22:        end for
23:    end for
24: end round
25: let v = first-non-⊥ value of V_i;                       ▷ Elección del lider
26: return(v)
```

3. (2.5 ptos) Explica con tus propias palabras por qué el consenso no puede ser resuelto en sistemas asíncronos con al menos una falla.

Respuesta.

Es imposible alcanzar el consenso en sistemas asíncronos con al menos una falla gracias al **Resultado de Imposibilidad FLP** [3], nombrado así por los autores de la publicación donde se presenta el famoso resultado: Michael Fischer (Yale), Nancy Lynch (MIT) y Michael Paterson (Warwick).

Asumamos que los canales de comunicación son confiables, es decir, que se entrega un y solo un mensaje y que éste se entrega de manera correcta a un proceso del sistema distribuido. Además, asumamos que el sistema es completamente asíncrono, lo que implica que los delays pueden ser arbitrariamente largos o cortos.

Dado que es imposible para un proceso saber si otro proceso ha fallado o si es muy lento para enviar mensajes, puede llegar a esperar indefinidamente un mensaje y causar que la ejecución del algoritmo se quede colgada.

En el algoritmo anterior se hizo la suposición de que ningún proceso fallaría, por lo que fue posible usar la primitiva *wait* para esperar a que todos los mensajes llegaran. Pero si tenemos por lo menos una

falla, un proceso se quedará esperando indefinidamente a que llegue el mensaje del proceso que falló ya que éste desconoce si el algoritmo falló o si solamente está tardando demasiado en llegar su mensaje.

Así, la clave en la demostración es que un único proceso fallido o lento puede causar que otros procesos del sistema no lleguen a tomar una decisión sobre qué lider tomar.

La prueba se basa en el concepto de *indistinguibilidad*, el cual nos dice que una ejecución A es indistinguible de una ejecución B para algún proceso p_i si p_i observa las mismas cosas (mensajes o resultados de operaciones) en ambas ejecuciones. Otros conceptos importantes en la demostración son el de *configuración*, que no es más que un estado global del sistema, el cual puede ser visto como una colección de estados, uno para cada proceso, y también el concepto de *evento*, que consiste en el envío de mensajes, la recepción de mensajes y los cómputos locales llevados a cabo por un proceso visto como un autómata. En las demostraciones que vi se usa la bivalencia de las configuraciones, pero como no vimos eso en clase, omito mencionarlo y me quedo con la idea básica detrás de la prueba. Me gusta la explicación que da el profesor Indranil Gupta de la Universidad de Illinois en Urbana-Champaign [4].

4. (2.5 ptos) Explica con tus propias palabras por qué el problema del ataque coordinado no puede resolverse con $n \geq 2$.

Respuesta.

Dentro del problema del ataque coordinado hacemos la suposición de que el canal de comunicación no es confiable.

En caso de que supongamos que sí tiene solución, llegaremos a una contradicción de la validez, la cual se cumple solamente sí: todos los procesos tienen la misma entrada x y deciden x, o bien, si tienen entradas distintas o uno o más mensajes se pierden pero se alcanza el acuerdo.

Para entender mejor en qué consiste el problema, básicamente podemos suponer dos generales Bob y Alice comunicados entre sí por un canal no confiable y que quieren ponerse de acuerdo entre atacar a la armada enemiga o retirarse. Sin pérdida de generalidad, supongamos que Alice decide atacar y envía a Bob un mensaje escrito *Attack*. El problema es que el canal no es confiable y Alice no sabe si el mensaje fue recibido por Bob o si fue interceptado por el enemigo, por lo que no puede decidirse por atacar sin el consentimiento de Bob, es decir, sin llegar a un acuerdo. Podría pensarse que una manera de resolver este problema es por medio de mensajes de reconocimiento donde un general avise a otro que ha recibido su mensaje, pero este mensaje de reconocimiento también pudo haber sido interceptado; por ejemplo, si Bob recibió el mensaje *Attack* de Alice y éste envía un mensaje de reconomiento a Alice para hacerle saber que sí recibió su mensaje, su mensaje puede ser interceptado y Bob no podrá tomar una decisión sin la posibilidad de que Alice tome la decisión opuesta. Podemos continuar así esté ciclo, pero siempre llegaremos a que habrá incertidumbre de la recepción de los mensajes, y de esto se sigue la imposibilidad de la solución al problema.

La demostración formal de la imposibilidad de una solución al problema se basa en el concepto de *indistinguibilidad*, y consiste en suponer que el problema tiene solución y posteriormente tomarnos una cadenita de ejecuciones indistinguibles de tal manera que podamos llegar a una contradicción de la validez.

Construimos la cadena primero tomando $k + 1$ ejecuciones:

$$A_0 A_1 ... A_{k-1} A_k$$

en las que la entrada es 1 y para la primera todos los mensajes se entregan, para la segunda todos menos un mensaje se entregan, y así sucesivamente hasta que en la última de las ejecuciones ningún mensaje se entrega.

A la subcadena de ejecuciones anterior le agregamos otra subcadena de n elementos:

$$A_{k+1}A_{k+2}...A_{k+n-1}A_{k+n}$$

donde invertimos a entrada para algún proceso. Es decir, va cambiando el input de un proceso en cada ejecución.

Finalmente, agregamos otra subcadena de ejecuciones:

$$A_{k+n+1}A_{k+n+2}...A_{2k+n}$$

que funcione como una inversión de la primera subcadena. Esto es, que en la primera ejecución de esta nueva subcadena de ejecuciones se entregue un único mensaje, en la segunda dos mensajes, y así sucesivamente hasta llegar a la ejecución A_{2k+n} en la que se entreguen todos los mensajes. Bajo esta suposición, en la ejecución A_{2k+n} con entrada 0 la salida será 0.

Si el acuerdo se satisface, la indistinguibilidad de ejecuciones adyacentes para algún proceso significa que la misma entrada en A_0 es la misma que en A_{2k+n}, pues suponemos que la salida es la misma para todas las ejecuciones en la misma cadena de ejecuciones. Sin embargo, la validez requiere que A_0 y A_{2k+n} tengan la misma salida, lo que nos lleva a una contradicción de la validez.

Referencias

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

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

[3] Fischer, M., Lynch, N. & Paterson, M. *Impossibility of Distributed Consensus with One Faulty Process*. Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374-382.

[4] Gupta, I. *Cloud Computing Concepts I - The FLP proof*. Coursera & University of Illinois at Urbana-Champaign, s.f.

[5] Whittaker, M. (s.f.). *Two Generals and Time Machines*. University of California at Berkeley. Recuperado el 8 de Noviembre del 2020 de https://mwhittaker.github.io/blog/two_generals_and_time_machines/