Tarea 2

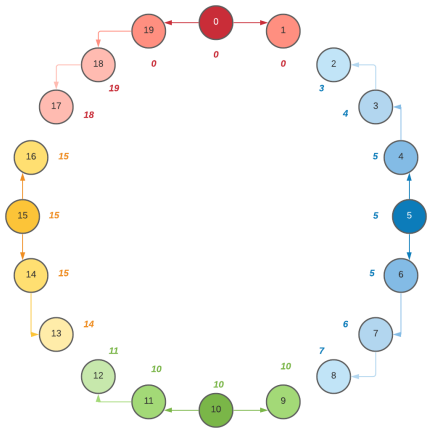
Ian Israel García Vázquez Armando Ramírez González
Ricardo Montiel Manriquez Christopher Alejandro Escamilla Soto
Jonás García Chavelas

12 de noviembre de 2021

1. Para concebir el tamaño mínimo que puede tener un árbol, considere el sistema distribuido en el que $k = 1$ y $m = 2$. Notamos que en dicho sistema es el más pequeño posible y que $n = 2$, pero como los procesos en las posiciones 0, k, $2k$, ..., $(m-1)k$ son marcados inicialmente como líderes, en éste caso los líderes son los dos nodos 0 y $(m-1)k = (1)(1) = 1$. Por lo que tenemos un sistema en el que todos los procesos son líderes, entonces el tamaño mínimo del árbol es de 1. Esto también aplica para sistemas con otros $n = mk$ procesos porque siempre existe la posibilidad de que algún líder no logre conseguir reclutas.

   Luego observamos que la existencia del árbol de tamaño máximo posible asume que un proceso líder "reclutó" a dos procesos aledaños, que a su vez reclutaron a dos y así sucesivamente hasta que dos procesos intentaron "reclutar" a dos líderes o casi fueron "reclutados" por ellos. Esto significa que el árbol máximo esta delimitado por la posición de los dos líderes a ambos lados, es decir, el número de procesos no líderes que integraran al árbol máximo será el número de procesos que hay entre dos líderes dos veces (por estar en ambos lados de un líder). Entonces el tamaño del árbol será esta cantidad más el mismo líder o raíz, por lo que el árbol máximo tiene un tamaño de $2(k-1)+1$.

   Un ejemplo de un sistema donde el algoritmo se ejecutó correctamente es en el que $k = 5$ y $m = 4$. Como se puede inferir de la imagen inferior, en este caso hipótetico los procesos $0, 5, 10, 15$ inicialmente son los líderes mientras que los demás procesos tienen `parent` $= \perp$. En este caso es posible que los mensajes entre los procesos 15 y 16, 0 y 1 y, 9 y 8 hayan tenido fallos en ciertos momentos, retrasando el crecimiento de sus respectivos árboles.



2. Realice un análisis preciso de la complejidad de tiempo y la complejidad de mensajes de:

- El algoritmo de broadcastTree.

  Complejidad en Tiempo:

  Sea T un árbol binario de profundidad n y cuya estructura es lineal.

  Entonces, en el tiempo inicial el algoritmo ejecutará una instrucción para llegar a su único nodo hijo, posteriormente el nodo hijo (que ahora es el padre) ejecutará una instrucción para llegar a su nodo hijo, así sucesivamente hasta llegar al nodo m. Por lo que el algoritmo ejecutará $m-1$ instrucciones para llegar al nodo m en el árbol de estructura lineal.

  Como sabemos que el árbol tiene una estructura lineal, además sabemos que tiene una profundidad de $m-1$ que son el número de operaciones que tarda el algoritmo para terminar su ejecución.

  Por lo tanto el algoritmo tiene una complejidad de tiempo igual a la profundidad del árbol

  Complejidad de Mensajes:

  Sea la raíz de un árbol binario, esta enviará un mensaje a cada uno de sus hijos al inicio de la ejecución del algoritmo, así sucesivamente hasta enviar el último mensaje al último nodo (que es una hoja del árbol), como se envía un mensaje por cada vértice del árbol y sabemos que un árbol binario tiene a lo más $v-1$ aristas.

  Entonces el algoritmo enviará $v-1$ mensajes antes de terminar. Por lo que su complejidad de mensajes es de $v-1$, siendo v el número de vértices del árbol y $v-1$ el número de aristas.

  Por lo tanto la complejidad de mensajes del algoritmo broadcasTree es $v-1$, con v vértices.

- El algoritmo de convergecast.

  Complejidad en Tiempo:

  Al igual que broadcasTree, el algoritmo hará un número de instrucciones igual a la profundidad del árbol, ya que la raíz esperará una respuesta de sus hijos de los cuales uno de ellos puede tener una profundidad igual a la del árbol, es decir, puede ocurrir que el árbol únicamente tenga una altura linal en el peor de los casos.

  Entonces la complejidad en tiempo del algoritmo es igual ala profundidad del árbol.

  Por lo tanto la complejidad en tiempo del algoritmo convergecast es igual a la profundidad del árbol.

  Complejidad de Mensajes:

  Inicialmente el algoritmo enviará n mensajes siendo estos igual al número de hojas que tiene el árbol binario, posteriormente su respectivos padres esperarán hasta recibir los mensajes de sus hijos, esto ocurrirá hasta llegar a la raíz, momento en el que serán enviados los últimos mensajes.

  Entonces podemos observar que por cada dos nodos conectados, tales que uno sea el padre y otro su hijo, se enviará un mensaje, por lo que la ejecución total del algoritmo enviará $k-1$ mensajes antes de terminar, donde k es el número de vértices del árbol y por propiedades de árboles binarios sabemos que $k-1$ es igual al número de aristas del árbol.

  Por lo tanto la complejidad de mensajes del algoritmo convergeCast es de $v-1$ para v vértices del árbol.

- El algoritmo de broadConvergecastTree

  Complejidad en Tiempo:

  Como paso inicial el algoritmo ejecuta el algoritmo de BroadcastTree cuya complejidad de tiempo es igual a la profundidad del árbol T, después de ello el agoritmo hará una ejecución del algoritmo

convergeCastTree, a partir de las hojas hasta la raíz, cuya complejidad de tiempo también es igual a la profundidad del árbol.

Entonces para un árbol binario cualquiera, el algoritmo de broadConvergeCastTree, se ejecutará dos veces para volver a regresar a la raíz, entonces el algoritmo realizará $2 * profundidad(T)$ para completar su ejecución.

Por lo tanto el algoritmo broadconvergeCastTree tiene una complejidad de $2 * profundidad(T)$ en tiempo.

Complejidad de Mensajes:

Al igual que en el caso anterior, sabemos que el algoritmo ejecuta primero el algoritmo de broadcastTree, cuya complejidad de mensajes es de $v-1$ siendo v el número de vértices del árbol sobre el que se ejecuta el algoritmo. Después de ejecutar broadcasTree, el algoritmo ejecuta convegecastTree que sabemos que también envía $v-1$ mensajes hasta llegar a la raíz.

Entonces como el algoritmo ejecuta broadcasTree y después ejecuta convergeCastTree podemos afirma que el número de mensajes que envía hasta terminar su ejecución, es de $2(v-1)$.

Por lo tanto la complejidad de mensajes del algoritmo broadconvergeCastTree es de $2(v-1)$ con v los vértices del árbol binario.

3. ¿Se basan los algoritmos de BroadcastTree y Convergecast en el conocimiento acerca del número de nodos en el grupo?

   Para intentar responder esta pregunta de forma adecuada, observaremos y haremos un análisis a grosso modo de ambos algoritmos por separado.

   Primeramente, observemos que ambos algoritmos nacen bajo la necesidad de mejora del *silly algorithm* **flooding**, entonces este algoritmo, en lugar de lanzar y regresar los mensajes en bruto entre cada nodo, para el caso del algoritmo en particular, los mensajes nacen desde la raíz y luego llegan a las hojas, dejándonos comprender que la complejidad en tiempo es O(d), con d, la profundidad del arbol, es decir, al algoritmo le tomaría el mismo tiempo enviar un mensaje en un árbol de 4 nodos, ordenados secuencialmente, que un árbol binario con altura k y $2^k - 1$, así entonces, observemos que lo realmente importante para el algoritmo consiste en la altura del mismo árbol, por lo tanto los nodos no son primordiales en él.

   Por otro lado, Convergecast, tiene la peculiaridad de ser el *inverso* de BroadcastTree al comenzar la transmisión en paralelo desde las hojas hasta la raíz, cuya complejidad en tiempo y mensajes es la misma que la de BroadcastTree, tiempo O(d) con d profundidad del árbol, y de mensajes correspondiente a $|V| - 1$, observemos que en este caso como en el anterior, la complejidad de mensajes es consecuencia del envío de mensajes a través del árbol, derivado de un algoritmo que ejecuta en paralelo el envío los hijos, pero nuevamente su importancia radica en los pisos que esté árbol contenga. En la bibliografía como : *Ïstributed Computing: A Locality-Sensitive Approach¨* este se representa como $Diam(G)$. Luego podemos concluir, que el conocimiento del número de nodos para ambos algoritmos no es determinante, y tampoco necesario, sino consecuencia inherente del envío de mensajes totales a través de un árbol, que claramente podríamos ignorar el número de nodos en el árbol y ambos algoritmos continuarían funcionando perfectamente.

4. **TTL (Time To Live)**

   El TTL o Time to live(Tiempo de vida) es un mecanismo que se usa para limitar la duración de la información que circula por la red. Esto evita que la información se mueva indefinidamente por internet, favoreciendo la privacidad y el rendimiento.

Por otro lado, cuando se realiza un cambio de zonas DNS o de DNS, la actualización de información tardará más o menos tiempo en ser efectiva en función del valor del parámetro TTL. Este valor nos indica el tiempo que se mantiene almacenada en el servidor DNS, por lo que hasta que transcurra no hablaremos de propagación.[2]
De la misma forma, el TTL como concepto permite indicar y limitar por cuantos nodos puede pasar un paquete antes de ser descartado.

Implementación

El valor se inicializa en el emisor, el nodo líder, tiene la función de ir descontando una unidad según viaje de un nodo a otro. Si el contador llega a cero el paquete es descartado y reenviado a su origen.
Consideremos lo siguiente[1]:

- El campo TTL tiene dos funciones, limitar el tiempo de vida de los segementos TCP y terminar con las rutinas que generan loops interminables.
- A pesar de que el TTL es un atributo expresado en segundos, también contiene algunos atributos de cuenta de hop's.[3]
- Un valor fijo debe ser al menos lo suficientemente grande para el diámetro del internet, es decir, el camino más largo posible.
- Un valor razonable es al menos dos veces más grandes que el diámetro para permitir su crecimiento
- No puede ser enviado un paquete cuyo valor de TTL sea igual a cero.
- Un host no debe descartar un paquete solo porque este tenga un valor de TTL menor a dos.

```
1   flooding_TTL(ID, LIDER, M, d):
2    Ejecutar inicialmente
3    flag=false
4    muerte=false
5    if ID == Lider entonces:
6       flag=true
7       muerte=true
8       if d>0:
9           send(<M>, d)
10      else:
11          end succesfully
12   Enviar el mensaje por todos los puertos si d>0
13
14  Al recibir un mensaje <M> por algun puerto
15
16  if not muerte:
17      d=d-1
18      if not flag and  d>0:
19          flag = true
20          muerte =true
21          send(<M>,d)
22  Tras haber cubierto el diametro correcto, solo los nodos que hayan
23  presenciado la muerte del paquete podran reenviarlo para llegar al nodo de origen.
24
25  if muerte and (d<=0 or [Nodo es hoja]):
26      if Recibe varias "d":
27      d=Math.min([di...dj])
28      send(<M>, d)
29      else:
30      send(<M>,d)
31
```

```
    Finalmente cuando el mensaje regresa al lider, recibe por todos los puertos, si la
    d recibida es mayor a cero, indicara la existencia de alguna rama con profundidad
    menor a d

    if ID==Lider y muerte:
        receive(<M>, d) ->
        if d>0:
            warning("There is a branch smaller than d")
        end(succesfully)
```

Para verificar que es correcto, observemos que el inicio es trivial, entonces si d, que será nuestro valor de TTL, es mayor a cero, el mensaje será enviado.
Ahora ya que el mensaje ha sido enviado, dentro cada nodo pregunta si es que el paquete ha muerto en él, es decir, el paquete ya pasado por el nodo y su $d > 0$ cuando esto ocurrió, ahora si esto es verdadero, continuará desperdigando el mensaje hasta que el contador d==0. Por otro lado en caso contrario, simplemente el nodo no recibirá nada y no tomará en cuenta este mensaje.
Finalmente, ya que el contador ha llegado a cero o el nodo es una hoja y $d \geq 0$, comenzará a retornar el mensaje con dirección al primer emisor, nodo líder, para ello, por cada nodo preguntará si la bandera muerte ya fue levantada, y continuará el envío hasta llegar al nodo líder quien terminará el proceso.

Complejidades

Dado que los mensajes son enviados en forma paralela, con límite en diámetro d y deben recorrerlo y volver al origen, tomará T= $2 \cdot d$.
Y los mensajes al ser enviados por cada una de las aristas en un diametro d, entonces la complejidad de los mensajes será $|D|$ tal que:

$$D = \{e \in E | E \text{ aristas } \in Diam(d)\}$$

Es decir, D es equivalente a todas las aristas acotadas por el diámetro d, por lo tanto $|D| \leq |E|$. Entonces $M = |D| \leq E$.

5. Generaliza el algoritmo convergecast para recolectar toda la información del sistema. Esto es, cuando el algoritmo termine, la raíz debería de tener todas las entradas de todos los procesos. Analiza la complejidad de bits, es decir, el total de bits que son enviados sobre los canales de comunicación. (hint: Cada mensaje de información puede tomar k bits).

Algoritmo Convergecast(ID):

```

    PADRE, HIJOS, soyRaiz = soyRaiz, #recibidos = 0
        list [] informacion

        informacion=[ID]

        Ejecutar inicialmente: // tiempo cero
            Si |HIJOS| == 0 entonces:
            send(<ok,informacion>) a PADRE

        Al recibir <ok,info> de algun puerto en HIJOS:
            #recibidos++
            informacion = f(informacion,info)
            Si #recibidos == |HIJOS| entonces:
```

```
15                  send(<ok,informacion>) a PADRE
16
```

donde f = función de concatenación de listas:

$f(l1, l2)$ que regresa una lista, resultado de la concatenación de dos listas en el mismo orden,ejemplo: $f([1, 2].[3, 4]) = [1, 2, 3, 4]$

Al final del algoritmo la raiz tendra una lista con los ID de todos los elementos icluida ella, por lo tanto dicha lista contendra todos la informacion: Todos los vertices del arbol T.

Podemos ver que se envían mensajes desde las hojas hacia la raíz todos los nodos envían un mensaje a su padre menos la raíz pues el no tiene padre, entonces la cantidad de mensajes que se envían es de $|v| - 1$ y si suponemos que cada mensaje toma kbits en enviarse, eso quiere decir que $|v|k - k$ seria la cantidad de mensajes si tomaran k-bits.

6. *a*) Da un algoritmo distribuido para contar el número de vértices en un árbol enraízado T, iniciando en la raíz.

Para resolver este inciso, nos basamos en el cómputo por agregación y en el algoritmo de Broad-ConvergeCastTree que como revisamos en clase, es un algoritmo que nos ayuda a recorrer el árbol iniciando desde la raíz hasta las hojas, posteriormente confirmar la información estando desde las hojas recorrer todo el árbol nuevamente pero comenzando desde las hojas y terminando en la raíz, que termina recibiendo la información de que el recorrido se ha completado además agregaremos a este algoritmo una función de agregación f(x,y) que es la funcion que regresa como resultado x+y, Asi:

Algoritmo broadConvergecastTree para la pregunta 6a:z

```
1       BroadConvergecastTree(ID, soyRaiz):
2
3       PADRE, HIJOS, #noVecientos = 0, acc = 1
4
5       Ejecutar inicialmente: // tiempo cero
6           Si soyRaiz entonces:
7               send(<START>) a todos en HIJOS
8
9           Al recibir <START> de PADRE:
10              Si |HIJOS| ≠ 0 entonces:
11                  send(<START>) a todos en HIJOS
12              Sino
13                  send(<ok, acc>) a PADRE
14
15      Al recibir <ok, accum> de algun puerto en HIJOS:
16          #noVecinos++
17          acc  = f(acc, accum)
18          Si #noVecinos == |HIJOS| entonces:
19              Si soyRaiz entonces:
20                  return acc
21              Sino
22                  send(<ok, acc>) a PADRE
23
```

Al recibir <ok,accum>notemos que la variable accum que se pasa como argumento, es un dato de

tipo Int que posteriormente podemos ver que llevará guardado el número de los vértices que se van acumulando de abajo hacia arriba, al final la raíz tendrá la cuenta de los vértices que tiene como hijos y se agregara a él mismo a la cuenta para posteriormente terminar el algoritmo regresando un número entero que representa la cantidad de vértices en el árbol.

b) Extiende tu algoritmo para una gr áfica arbitraria G.

Para resolver este inciso, agregamos únicamente un algoritmo antes, que nos asegure la existencia de un árbol con raíz, para posteriormente ejecutar el mismo algoritmo anterior y resolver para una gráfica arbitraria G, entonces ejecutamos el algoritmo:

```
BuildSpanningTree(ID, root, M):
Parent = null

Ejecutar inicialmente:
    Si ID == root entonces: // Soy el lider
        Parent = root
    send(<M>) por todos los puertos

Al recibir algun mensaje <M> por algun puerto P:
    Si Parent == ⊥ entonces:
        Parent = P
    send(<M>) por todos los puertos


```

Al finalizar este algoritmo, tendremos un árbol generador formado por una raíz y vértices que tienen asignados a sus respectivos padres, lo que convierte G a un gráfica de tipo árbol con raíz y que ahora si es candidato para ejecuta el algoritmo anterior:

```
BroadConvergecastTree(ID, soyRaiz):

PADRE, HIJOS, #noVecientos = 0, acc = 1

Ejecutar inicialmente: // tiempo cero
    Si soyRaiz entonces:
    send(<START>) a todos en HIJOS

    Al recibir <START> de PADRE:
        Si |HIJOS| ≠ 0 entonces:
            send(<START>) a todos en HIJOS
        Sino
            send(<ok, acc>) a PADRE

    Al recibir <ok, accum> de algun puerto en HIJOS:
        #noVecinos++
        acc  = f(acc, accum)
        Si #noVecinos == |HIJOS| entonces:
            Si soyRaiz entonces:
                return acc
            Sino
                send(<ok, acc>) a PADRE


```

7. Da un algoritmo distribuido para contar el número de vértices en cada capa de un árbol enraizado T de forma separada. Analiza la complejidad de tiempo y la complejidad de mensajes de tu algoritmo.

Proponemos el siguiente algoritmo:

Algoritmo para contar los nodos en cada subcapa de un árbol binario:

```
Algoritmo nodos(ID, soyLider):
   padre = null, raiz  = null, capa = 0

   Inicialmente hacer:
     Si (raiz == null) entonces:
        raiz = ID
        capa ++
        send(<Inicio, capa>) a todos su hijos

   Al recibir <Inicio, j> de Padre:
     Si |Hijos| != 0
        capa = j
        j++
        send(<Inicio, j>) a sus hijos

     Sino
        A = A[j+1]
        A[j] += 1
        send(<A>) a su padre

   Al recibir <A> y <B> de sus puertos de hijos:
        sum = Suma(A, B)
        sum[capa] += 1
        send(<sum>) al padre

        Si soyRaiz
           Reportar teminacion

   Funcion Suma(A, B)
       Si A == null
          return B

       Si B == null
          return A

       Si len(A) > len(B)
         D = A
         for (i = 0, i < len(B), i++)
            D[i] = A[i] + B[i]
         return D

      sino
         D = B
         for (i = 0, i < len(A), i++)
            D[i] = A[i] + B[i]
         return D

```

Al finalizar el algoritmo devolverá un arreglo de longitud n con el número de nodos que hay en cada capa del árbol, donde la posición i representa el número de nodos que hay en la capa i del árbol binario.

Análisis de complejidad:

- Complejidad de tiempo:
  El algoritmo ejecutará primero broadcasTree para llegar hasta las hojas, por lo que, el algoritmo tardará $v-1$ instrucciones para llegar a la hojas. Después de ello se ejecutará un número constante de instrucciones que consisten en manipular los arreglos que, en el caso de que el árbol solo este compuesto de vértices amcomodados en forma lineal, el arreglo solo se regresará y se sumará el vértice actual del proceso en el que está.

Entonces el algoritmo necesitará de $v-1$ instrucciones para llegar a la raíz, además realizará v instrucciones en cada vértice del árbol a excepción del vértice raíz, esto en el peor de los casos serán $v+(v-1)-1$ instrucciones o $2v-2$ más las $v-1$ instrucciones que se ejecutarán para llegar a las hojas del árbol, lo que no da $2v-2+v-1=3v-3=3(v-1)$.

Por lo que el algoritmo ejecutará $3(v-1)$ instrucciones para terminar.

Por lo tanto la complejidad en tiempo del algoritmo será de $3(v-1)$ con v los vértices del árbol binario.

- Complejidad de mensajes:

  El algoritmo para llegar a las hojas, en el peor de los casos, tendrá que enviar $v-1$ mensajes, es decir, $profundidad(T)$. De igual forma, para llegar a la raíz y poder finalizar, el algoritmo deberá de enviar, en el peor de los casos, $v-1$ mensajes.

  Por lo que tenemos que el algoritmo al igual que broadconvergeTree, enviará $2(v-1)$ mensajes antes de terminar su ejecución.

  Por lo tanto el algoritmo tiene una complejidad de mensajes de $2(v-1)$ con v los vértices del árbol.

Referencias

[1] R. Braden, *Requirements for internet hosts. Communication layers RFC -1123*, `https://datatracker.ietf.org/doc/html/rfc1122#page-34`, 1989, Accessed: 2021-10-12.

[2] Dinahosting, *¿Qué es el TTL?*, `https://dinahosting.com/ayuda/que-es-el-ttl/`, 2018, Accessed: 2021-10-12.

[3] Stormwind Studios, *DNS and time to live (TTL)*, `https://youtu.be/jK5Q9ImdW5U`, 2011, Accessed: 2021-10-12.