Universidad Nacional Autónoma de México
Facultad de Ciencias
Computación distribuida
Practica 2
Morales Chaparro Gael Antonio 320076972
Rivera Lara Sandra Valeria 320039823
Cervantes Duarte Jose Fernando 422100827
22 de septiembre de 2024

1 Funcionamiento del metodo enviar de la clase Canal Broadcast

Este método es parte de la clase CanalBroadcast la cual ya tiene definido lo siguiente:

1. **env:** El ambiente en donde se ejecutarán los procesos
2. **capacidad:** La capacidad que tendrá los canales de comunicación
3. **canales:** Una lista con los canales de comunicación que están definidos en el ambiente

Entonces, el método **enviar** simulará el envío de mensajes *one-to-many*, y para ello recibirá una lista de vecinos y un mensaje a difundir.

Inicialmente dicha lista la pre-procesará para obtener los canales de dichos vecinos, y entonces en todos los canales obtenidos les enviaremos con el método **put** el mensaje que querramos comunicar.

El método terminará su ejecución en el momento en que todos los mensajes hayan sido enviados, y esto se simulará con el método *env.all_of* de Simpy, que nos permite esperar a que completen los múltiples eventos anteriores de forma simultanea.

Notemos que con nuestra implementación, también podemos simular el envío de mensajes *one-to-one*, pues para ello la lista de vecinos que recibimos solo será de longitud uno, y contendrá el vecino o proceso al cual le queremos hacer llegar el mensaje.

2 Funcionamiento de implementación de ConoceVecinos

Este método permite que un nodo o proceso adquiera conocimiento sobre los vecinos de sus vecinos, y esto se lográ con los siguientes pasos:

1. **El proceso envía su Lista de Vecinos:**

   Al inicio de su ejecución, el nodo envía su lista de vecinos a través de su canal de salida. Esto proporciona a los vecinos del nodo la información necesaria para establecer conexiones adicionales.

2. **El proceso está en espera de mensajes:**

   Se entra en un ciclo infinito con *While true*, lo que implica que el proceso permanece en un estado de espera, listo para recibir mensajes de sus vecinos.

3. **Se recibe el mensaje y se actualizan los id's:**

   Al recibir un mensaje, que contiene identificadores de los vecinos de un vecino, el nodo procede a actualizar su conjunto de *identifiers*.

3 Funcionamiento de implementación Broadcast

El proceso que empieza la ejecución del Broadcast es el nodo distinguido 0, y lo que hará es notificar a sus vecinos con un $GO(mensaje)$, donde $mensaje$ es lo que se busca transmitir, y para ello usará el método *enviar* de la clase $CanalBroadcast$.

Para todos los demás procesos estos estarán en espera (la cual se simula con un *yield*) a que les llegue el mensaje $GO(mensaje)$, una vez que esto pase se limpiará el $GO(mensaje)$ para extraer solo el mensaje y guardarlo en el proceso actual, y este proceso hará lo mismo que antes, le notificará a todos sus vecinos con un $GO(mensaje)$, y para esto se ayudará del método *enviar* de la clase $CanalBroadcast$.

Al final de todo el procesamiento anterior, se ejecutará un $yieldenv.timeout(TICK)$ que representará una simulación del costo del tiempo ($TICK$) que nos cuesta procesar todo.

4 Funcionamiento del algoritmo para construir un árbol generador

Una cosa importante a destacar de nuestro algoritmo es que para la comunicación de mensajes entre los procesos usaremos el método **enviar** de la clase **CanalBroadcast**, ya que este nos permitirá simular el envío de mensajes *one-to-many* para aquellos casos en los que un proceso quiera comunicar el mensaje a todos sus vecinos y *one-to-one* (y para ello solo se enviará una lista de longitud uno con el vecino al cual queramos comunicar) para cuando un proceso quiera decirle al proceso del cual le llegó el mensaje que si será su hijo o no.

Inicialmente nuestro algoritmo revisa si el proceso que está ejecutando es el proceso distinguido 0, y de ser así entonces por convención definiremos que este será su propio padre, y que esperará un número de mensajes correspondientes a todos los vecinos que este tiene.

Después, como este proceso distinguido es el que comienza la ejecución este tendrá que enviar un mensaje $GO(\{0\})$ a todos sus vecinos, donde $\{0\}$ corresponde al conjunto con el $id = 0$ del proceso distinguido. Hacemos esto para que al momento de que a un nodo le llegue el mensaje este pueda saber desde que proceso le llego.

Para todos los demás nodos estos tendrán que esperar a que les llegue un mensaje, (para esto usamos la palabra reservada *yield*), y si estamos en el proceso i, pueden pasar dos cosas:

1. El mensaje recibido es del tipo $GO(\{j\})$

   En este caso pueden pasar otras dos cosas:

   (a) El proceso i aún no tiene a nadie definido como su padre, esto es $padre_i = None$:

   En este caso como el mensaje vino de $\{j\}$, definiremos al padre de i como j y ahora este proceso esperará un mensaje menos, pues uno de sus vecinos (j) ya le envió un mensaje.

   Si se da el caso de que j era su único vecino, entonces i le envía un mensaje $BACK(\{i\})$ a j para indicarle a j que i será su hijo.

   En otro caso entonces el proceso i enviará un mensaje $GO(\{i\})$ a todos sus vecinos para tratar de que estos lo definan como su padre.

2. El mensaje recibido es del tipo $BACK(\{set\})$

   En este caso set hace referencia a un conjunto que contiene a aquellos procesos que podemos definir como nuestros hijos.

   Si el conjunto set es vacio o si no lo es, dará lo mismo unirlo con el conjunto set_i que ya teníamos definido, por lo tanto en ambos casos hacer esto nos asegura que estamos definiendo de forma valida a nuevos hijos para el proceso i.

   Después solo queda ver si ya no hay ningún otro mensaje por esperar y si no somos el proceso distinguido 0, pues entonces deberemos indicarle a nuestro padre con un mensaje de la forma $BACK(\{i\})$ que efectivamente seremos su hijo.

Notemos que es seguro hacer lo anterior pues para el momento en que recibimos un mensaje $BACK(\{j\})$ el proceso i ya tiene un padre definido. Esto para todo proceso excepto el proceso distinguido solo se recibe un mensaje $BACK()$ si antes se recibió un mensaje $GO()$, y al momento de recibir el $GO()$ o ya se tenía padre definido o se definió en ese momento.

5 Funcionamiento de Convergecast

Al NodoConvergecast se le agregaron los atributos de *padre* para saber a quien mandarle el mensaje, *datos_recibidos* para almacenar el la información del nodo y los datos mandados por sus hijos, *propio_mandado* para saber si el nodo ya mandó al padre su propio dato y así no repetir información.

En el algoritmo convergecast se van a mandar mensajes entre nodos dentro de $BACK()$ y a través de CanalConvergecast. Primero las hojas guardan su propio mensaje asociado a su id en una cadena de la forma *[id, mensaje]*, para después enviar eso a su padre. Ahora, cuando un nodo que no es hoja reciba datos de sus hijos, este los guarda y los manda a su padre. Asimismo, guarda una vez su propio mensaje asociado a su id y lo manda una sola vez a su padre. De esta forma, cada nodo realiza convergecast correctamente, guardándose las duplas que debería de tener cada uno en la cadena *datos_recibidos*.

Ejecución de Convergecast

Además de los datos que se piden en NodoBroadcast, vamos a solicitar el padre de cada nodo en el método constructor. Nótese que para el nodo distinguido, su padre es él mismo.

Tras ejecutar convergecast en los nodos, se guardarán los datos que debería tener un nodo en su atributo *datos_recibidos*. Se agregaron dos pruebas, la primera verifica que la raíz recibió la dupla *[id, mensaje]* de todos los nodos del árbol propuesto para los tests anteriores. El segundo test verifica con un árbol diferente que cada nodo guardó correctamente todas las duplas que debería de tener al ejecutar convergecast.