Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura
Departamento de Ciencias de la
Computación

Sistemas Operativos I

Broadcast Atómico

Marcos Cassinerio, Lautaro Cerruti

Junio de 2021

1. Introducción

El objetivo de este trabajo fue implementar un servicio de Broadcast Atómico el que luego se utilizó para la implementación de un Ledger Distribuido, haciendo uso de los algoritmos de los fragmentos 5 y 8 propuestos en el artículo.

2. Ejecución

Primero que nada, para compilar el proyecto tenemos que levantar los servidores. Para esto, todos deben compartir la misma cookie con el fin de poder conectarse entre si y estos no pueden tener el mismo nombre. Para esto debemos correr las siguiente linea en la terminal:

```
erl -sname {nombre} -setcookie {cookie}
```

Desde el segundo nodo en adelante, hay que conectarlos con algún nodo ya creado con el siguiente comando:

```
{nodo}> net_kerner:connect_node({nodoAdicional}).
```

Una vez ya iniciados y conectados todos los nodos, levantamos cada uno con las siguientes lineas:

```
{nodo}> c(aBroadcast).
{nodo}> c(dls).
{nodo}> aBroadcast:start().
{nodo}> dls:start().
```

Habiendo terminado con esta parte ya tenemos el sistema funcionando.

Para levantar los clientes, estos deben tener la misma cookie y tener la propiedad “hidden”. Estos se pueden iniciar de la siguiente manera:

```
erl -sname {nombre} -setcookie {cookie} -hidden
{nodo}> c(client).
{nodo}> client:start('{nodoServidor}').
```

Al correr el método “start”, en este se debe escribir el nombre de uno de los nodos del servidor. Al hacer esto, se nos conectara con el resto de los nodos.

3. Organización de los archivos

El programa se divide en 3 partes: dls, aBroadcast y client

Por un lado tenemos la implementación del Broadcast Atómico en el archivo `aBroadcast.erl`.

Por otro lado tenemos la del Ledger Distribuido en `dls.erl`. En el cual, también manejaremos la conexión con los clientes.

Y finalmente la implementación del cliente en `client.erl`.

4. Implementaciones

4.1. Broadcast Atómico

El Broadcast Atómico consta principalmente de 2 actores que se encuentran todo el tiempo activos, uno maneja la cola de mensajes que esta ordenada según el numero de secuencia y otro se encarga de recibir las peticiones de propuestas y de acuerdos de otros nodos del servidor. Su implementación esta basada en el algoritmo de ordenación total por acuerdo (ISIS).

Esta la hicimos de forma que cuando se desea realizar un broadcast se hace un spawn un actor en la función atomicBroadcast. Este se encarga de pedirle a la cola de mensajes el numero de secuencia para proponer. Al hacer esto, este numero de secuencia es agregado a la cola como "provicional" y con un estado que lo describe como "propuesto". Habiendo hecho esto se les pide a los otros nodos su propuesta de numero de secuencia, de estas propuestas tomamos la mayor (que es la acordada) y se la informamos tanto a nuestra cola de mensajes como a los otros servidores. Al avisarle a la cola de mensajes el numero de secuencia acordado, esta reordena la cola con el nuevo numero de secuencia y un estado que indica que este numero fue "acordado". Un mensaje es "entregado" al actor "msgsHandler" cuando este es el primero de la cola y su estado es "acordado". Luego de hacer esto se elimina dicho mensaje de la cola.

Al momento en que a nuestro nodo se le pide que realice una propuesta de un numero de secuencia, este nuevamente llama a la cola para pedirle el numero de secuencia a proponer y el comportamiento de la misma es el descripto previamente. Luego, cuando a nuestro nodo se le indica que se acordó un numero de secuencia para este mensaje, en la cola se hace lo mismo que en el párrafo anterior cuando "acordamos" un numero de secuencia.

4.2. Ledger Distribuido

Nuestra implementación del Ledger Distribuido esta basada en el algoritmo 8 del artículo mencionado en la introducción el cual es el siguiente:

Code 8 Atomic Distributed Ledger; Code for Server i

```
1: Init: S_i ← ∅; get_pending_i ← ∅; pending_i ← ∅
2: receive (c, GET) from process p
3:     ABroadcast(get, p, c)
4:     add (p, c) to get_pending_i
5: upon (ADeliver(get, p, c)) do
6:         if (p, c) ∈ get_pending_i then
7:             send response (c, GETRES, S_i) to p
8:             remove (p, c) from get_pending_i
9: receive (c, APPEND, r) from process p
10:    ABroadcast(append, r)
11:    add (c, r) to pending_i
12: upon (ADeliver(append, r)) do
13:    if r ∉ S_i then
14:        S_i ← S_i||r
15:        if ∃(c, r) ∈ pending_i then
16:            send response (c, APPENDRES, ACK) to r.p
17:            remove (c, r) from pending_i
```

Figura 1: Algoritmo 8

Cuando un cliente le pide al Dedger una operación, este utiliza el Broadcast Atómico para

distribuirla y acordar el orden de la misma. A la vez la agrega a una lista de operaciones pendientes según el tipo de operación.

El sistema del Broadcast Atómico hace un deliver de la acción a procesar a msgsHandler el cual dependiendo de la operación (get o append) la envía al actor correspondiente para que las procese. Estos actores chequean si esta acción a realizar es una acción pendiente (si fue requerida a este nodo) o simplemente es una acción informada por otro nodo de la red. En el caso de ser necesario se realiza la operación y si estaba como pendiente se elimina de esta lista.

4.3. Cliente

Nuestra implementación del Cliente esta basado en el algoritmo 5 del artículo mencionado en la introducción el cual es el siguiente:

Code 5 External Interface of a Distributed Ledger Object $\mathcal{L}$ Executed by a Process p

1: $c \leftarrow 0$
2: Let $L \subseteq \mathcal{S} : |L| \geq f + 1$
3: **function** $\mathcal{L}$.get( )
4: $c \leftarrow c + 1$
5: **send** request (c, GET) to the servers in L
6: **wait** response (c, GETRES, V) from some $i \in L$
7: **return** V
8: **function** $\mathcal{L}$.append(r)
9: $c \leftarrow c + 1$
10: **send** request (c, APPEND, r) to the servers in L
11: **wait** response (c, APPENDRES, res) from some $i \in L$
12: **return** res

Figura 2: Algoritmo 5

Como se puede ver, este pseudocódigo envía las peticiones a una cantidad mayor o igual a `f+1` nodos siendo `f` el máximo numero de nodos que pueden llegar a fallar. Al no conocer este numero, se decide enviar la operación a todos los nodos que constituyen el Ledger Distribuido, y se toma la primer respuesta recibida.

Para conectarnos solo necesitamos saber el nombre de un nodo, pero la implementación se encarga de pedirle a este nodo los nombres de los demás y los almacena para poder conectarse con todos.

5. Desarrollo y complicaciones

Al comenzar a desarrollar lo hicimos sin haber leído el artículo y interpretamos erróneamente como funcionaba el Algoritmo ISIS de ordenación total, específicamente lo que tratamos de implementar fue que los nodos acordaran el orden de los elementos dentro del Ledger y no el orden de los mensajes/operaciones.

Luego de haber leído el artículo entendimos mejor como debía ser todo mas modularizado y como hacer las implementaciones.

Uno de los problemas que se nos presentó fue la elección del identificador para cada mensaje. Estábamos usando como identificador una tupla que consistía en el Pid del usuario y el numero de operación de este cliente. Lo que pasó fue que como el cliente enviaba cada operación a

realizar a todos los nodos, este Id estaba repetido una vez por cada nodo en las colas de mensajes. Esto causaba que cuando queríamos cambiar un mensaje a estado "acordado" se cambiara un mensaje que no era el que se había acordado, por lo que redefinimos el Id como una tupla que tenia una tupla con el Identificador de la operación (el Pid del cliente y el numero de operación) y el nombre del nodo al cual le llegó esta operación y realizo el broadcast. Otro problema fue que en una parte del código habíamos hecho un receive dentro de otro receive esto causaba que en el segundo recibiéramos mensajes equivocados que eran en realidad para el primero.

El ultimo día de desarrollo un compañero realizo una pregunta por el foro sobre el comportamiento del programa cuando un nodo se caía luego de haber pedido que le propongan números de secuencia. El problema especifico era que quedaban en la cola mensajes de los nodos, mensajes que nunca iban a recibir el valor acordado del número secuencia por lo que bloqueaban el programa. La solución implementada fue tener un actor monitoreando los nodos conectados y en caso de que alguno se caiga, borra de la cola de mensajes todos los propuestos por este nodo.

6. Mejoras

Siguiendo con lo último mencionado, hay un caso mas que puede llevar a estados inconsistentes que es cuando un nodo se cae después de haber enviado a algunos pero no a todos los nodos el valor acordado. Una idea que tuvimos para arreglar/mitigar este caso es que cuando un nodo recibe un mensaje de que se acordó un numero de secuencia, lo retransmita a todos los otros nodos. Esto haría que el acuerdo le llegue a todos los nodos a pesar de que el nodo que lo acordó se cayó. Esta mejora no fue implementada por falta de tiempo.