INFORME TRABAJO PRÁCTICO 2

Joaquin Arroyo - Ignacio Cainelli

Implementación de un Broadcast Atómico Distribuido:

Para comenzar, el algoritmo para solucionar el orden de los mensajes que utilizamos es el de mediante acuerdo de los destinatarios (ISIS), este algoritmo tiene como objetivo eliminar el secuencializador para así elegir entre los destinatarios el número del siguiente mensaje que se enviará.

Implementación del algoritmo ISIS:

Vamos a considerar a $g = \{p_1, p_2, \ldots, p_n\}$ como un grupo cerrado de procesos que difunden todos los mensajes al grupo entero. El algoritmo en particular, será ejecutado en cada proceso p_i en g y este se asegurará de que todos los procesos entregarán todos los mensajes en el mismo orden.

Vamos a considerar m como el mensaje a enviar, m_{id} como el identificador del mensaje m, a s como un número de secuencia sugerido, a k como el número de proceso que sugiere el número de secuencia, y $status$ como una variable que puede ser deliverable o undeliverable.

- Inicialización de un proceso i: $s_i = 0$, $contador_i = 0$

Dentro del proceso i, este procedimiento se encontrara en la función $loopBr(S, Cont, Cola, Prop, TO)$

- Difusión del mensaje m para todos los procesos en g:
  $contador_i := contador_i + 1$
  $aBroadcast(\{m, i, contador_i\})$

- Entrega de un mensaje $\{m, i, m_{id}\}$ por de parte de un proceso p_j donde $1 \leq j \leq n$:
  $$j = Id\ del\ proceso\ p_j$$
  $$s_i := s_i + 1$$
  Enviamos $\{m_{id}, s_i, i\}$ a p_j
  Colocamos $\{m, m_{id}, j, s_i, i, undeliverable\}$en cola de espera

- Recibimos una propuesta $\{m_{id}, s_j, j\}$ desde el proceso p_j donde $1 \leq j \leq n$:
  Agregamos la propuesta $\{s_j, j\}$a la lista de números de secuencia sugeridos para el mensaje m_{id}

  Si hemos recibido una secuencia de todos los procesos en g:

$\{m_{id}, i, s_k, k\}$donde s_k es el mayor número de secuencia en la lista, el cual fue sugerido por el proceso k
(en caso de varios procesos k hayan elegido el mismo número de secuencia, se elige el menor k)
$aBroadcast(\{m_{id}, i, s_k, k\})$ enviamos el consenso

- Recibimos el consenso $\{m_{id}, i, s_k, k\}$

$$s_i = max(s_i, s_k)$$

Modificamos los mensajes con el id $\{m_{id}, i\}$en la cola:
cambiar el número de secuencia propuesta a s_k
cambiar el proceso que sugiere el número de secuencia a k
cambiar undeliverable a deliverable
Si el mensaje en la cabeza de la cola es deliverable, entonces lo enviamos

Para dicha implementación contamos además con funciones secundarias que se encuentran en el archivo $funcSec.erl$ en el cual se encuentran funciones para:

- Devolver el mayor número de propuesta en una lista de propuestas
- Agregar mensajes a una cola de espera y ordenar dicha cola según el número de propuesta
- Modificar los elementos de la cola
- Buscar si una lista de propuestas recibió todas las propuestas
- ver si un mensaje es 'deliverable' o no

En nuestra implementación, suponemos que existe una red de N nodos, en la cual cada nodo inicio el servicio con la función *broadcast:start()*, la cual inicia dos procesos registrados:
node → iniciado en la función *loopBr*, en la cual está implementado el algoritmo explicado anteriormente.
deliver → iniciado en la función *aDeliver*, la cual se utiliza para mostrar un mensaje por pantalla.

Dentro de la función loopBr, representamos a *Cola* como una lista con elementos de tipo $\{m, m_{id}, j, s_i, i, etiqueta\}$, donde *etiqueta* puede ser *derivable* o *underivable*, y a *Prop* como un diccionario donde vamos a guardar una lista de propuestas para cada mensaje *i*, donde *i* va a ser la key de la lista. La variable *TO* es utilizada para eventualmente revisar si alguna de las listas anteriormente mencionadas ya recibió todas las propuestas; inicialmente esta variable tiene valor *'infinity'*, pero cuando un nodo por lo menos tiene una lista de propuestas en su diccionario, la variable va a pasar a ser un tiempo definido en la constante *TO*, para que así el nodo pueda chequear la lista cada esa cantidad de tiempo.

Luego de esto, cada nodo podrá realizar broadcast de un mensaje *' Msg'* con la función *broadcast:aBroadcast(Msg).*

Para finalizar el servicio dentro de un nodo utilizamos la función *broadcast:quit()* la cual termina los dos procesos, y los desregistra.

Además, definimos tres constantes:
NODES → indica la cantidad de nodos de la red.
F → indica la cantidad de nodos que se pueden caer.
Con esta información, cada uno podrá utilizar el servicio con la cantidad de nodos que le sea útil.
TO → indica la cantidad de tiempo que espera el Loop antes de ingresar en la sección after.

(Para utilizar el servicio, anteriormente se debe haber compilado el archivo *funcSec.erl*)

Implementación del Ledger Distribuido:

Luego de implementar el Broadcast Atómico con el algoritmo ISIS para el común acuerdo de los destinatarios, implementamos el Ledger, el cual tuvimos que adaptar al broadcast.
El cliente le manda un mensaje al proceso del broadcast, para que este haga broadcast (Multidifusión) del mensaje, una vez que se pueda entregar, le manda el mensaje al proceso del server y este hace la operación.

- **SERVIDOR:** En nuestra implementación, cada nodo va a iniciar el servidor con la función *server:start(),* la cual inicia tres procesos registrados:
  node → iniciado en la función *loopBr*, la cual es una modificación de la función mencionada en el broadcast atómico. La única modificación de esta función es que en el caso de que no haya nodos suficientes para operar, envía un mensaje de error al proceso *cliente*.
  server → iniciado en *loopServer*, la cual recibe las operaciones y las realiza.
  deliver → iniciado en la función *aDeliver*, la cual se utiliza para enviar el mensaje a server.

  Dentro de *loopBr*, las variables son las mismas que las mencionadas en la sección del broadcast atómico.
  Dentro de *loopServer*, representaremos a la secuencia como una lista ordenada.

  Para finalizar el servicio, cada nodo podrá utilizar la función *server:quit()* la cual termina los tres procesos y los desregistra.

  Además, tenemos definidas las tres mismas constantes que fueron mencionadas en la sección del broadcast atómico.

- **CLIENTE:** En nuestra implementación, el cliente podrá conectarse al ledger a través de un nodo 'hidden' el cual tiene que estar conectado como mínimo a un nodo de la red, o desde un nodo de la red, en el cual haya un server corriendo.
  Cada cliente iniciará su servicio con la función *client:start()*, la cual inicia un proceso registrado:
  cliente → iniciado en la función *client*, la cual recibe los pedidos de las operaciones y envía la petición , o recibe los resultados de dichas operaciones y los muestra por pantalla.

Dentro de *client*, las variables C y K representan contadores, el contador C se envía junto a la petición de la operación, y luego cuando se recibe el resultado, recibimos el mismo C y lo comparamos con K para revisar si el resultado ya fue mostrado o no; esto ya que vamos a recibir una respuesta de la operación por server.

Dentro de la implementación, tenemos las operaciones:
get() → Pedimos la secuencia a los servers. Esta operación la realizaremos con la función *client:get().*
El cliente recibirá la secuencia como respuesta y la mostrará en pantalla.
append(X) → Pedimos a los servers que agreguen X a la secuencia. Esta operación la realizaremos con la función *client:append(X).*
El cliente recibirá 'ack' en caso de que X se haya agregado a la secuencia, y 'nack' en caso de que X no se haya agregado(esto sucede si X ya pertenece a la secuencia).

Para terminar el servicio del cliente, utilizamos la función *client:quit()*, la cual termina el proceso *cliente*, y lo desregistra.

(Para utilizar el servicio, anteriormente se debe haber compilado el archivo *funcSec.erl*)

Estructura 'send' y 'msg':
send → contiene dos campos, 'msg' el cual va a contener el mensaje, y 'sender' el cual va a contener quien envió el mensaje.
msg → contiene tres campos, 'msg' el cual va a contener el mensaje, 'sender' el cual va a contener quien envió el mensaje, y 'sn' el cual contiene un número de secuencia.
'send' es utilizado para los mensajes enviados desde el cliente hacia los servidores, y 'msg' es utilizado internamente en el broadcast atómico.

Complicaciones:
Al momento de realizar la red y probar el servicio del ledger, tuvimos la complicación de que el nodo desde el cual se conectaba el cliente sumaba uno a la lista *nodes(),* por lo tanto, los servidores que estaban realizando broadcast de un mensaje, se quedaban esperando una propuesta que nunca iba a llegar. Por lo cual decidimos que el cliente tenga que conectarse a la red a través de un nodo 'hidden' o utilizar el servicio dentro de un nodo en el cual se esté corriendo un server.

Bibliografía:

- https://studylib.net/doc/7830646/isis-algorithm-for-total-ordering-of-messages#:~:text=ISIS%20algorithm%20for%20total%20ordering%20of%20messages%20Let%20g%3D%7B,messages%20in%20the%20same%20order (algoritmo ISIS)

- https://cse.buffalo.edu/~stevko/courses/cse486/spring19/lectures/12-multicast2.pdf (algoritmo ISIS)
- https://es.wikipedia.org/wiki/Multidifusión (algoritmo ISIS)
- https://arxiv.org/abs/1802.07817 (implementacion Ledger Distribuido)