Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Sistemas operativos I

Trabajo práctico 2

Alumnos:
Petruskevicius Ignacio - Trincheri Lucio

Junio 2021

SOI - Informe TP 2

Petruskevicius Ignacio - Trincheri Lucio, LCC UNR FCEIA 22/06/2021

1. Introducción

En este informe contaremos detalles el Ledger Distribuido implementado usando un broadcast atómico también distribuido, utilizando el algoritmo "Implementación total por acuerdo (ISIS)".

2. Broadcast atómico distribuido

Teniendo en cuenta el enunciado, nuestro broadcast debe cumplir con las siguientes propiedades:

- **Validity** Si un proceso correcto p hace un broadcast de m entonces eventualmente p envía m.
- **Uniform Agreement** Si un proceso envía un mensaje m entonces todos los procesos correctos eventualmente envían m.
- **Uniform Integrity** Para cualquier mensaje m, todo proceso envía m a lo sumo una vez, y si anteriormente $sender(m)$ hizo un broadcast.
- **Uniform Total Order** Si dos procesos p_i, $p_j \in \Theta$ envían mensajes m, m', entonces p_i envía m antes que m' si y solo si p_j envía m antes que m'.

Para asegurar estas primitivas hicimos uso del algoritmo ISIS mencionado anteriormente, el cual nos garantiza (demostrado matemáticamente) que estas propiedades se cumplen. Ahora está en nosotros que el sistema funcione correctamente y sea resistente a fallos comunes (nodos que se caen, etc).

Damos una breve descripción del algoritmo que implementamos.

Cada nodo N tiene las siguientes variables:

- A: mayor número de secuencia acordado.
- P: mayor número de secuencia propuesto.

Nombres acordados:

- Emisor (E)
- Mensaje (M)
- Identificador (I)

Implementación:

- 1) El nodo emisor (E) envía al resto un mensaje {M, I}. Este mensaje {M, I} se lo guarda para luego, poder calcular el número definitivo de prioridad (llamado D) mediante los valores P.
- 2) Cada nodo realiza lo siguiente:
  - Calcula el nuevo valor $P = max(A, P) + 1$.
  - Asigna al mensaje (M) el nuevo P.
  - Coloca el mensaje en la cola de retención, ordenada por este valor P como primer ordenamiento y por el valor I como segundo. {M, I, P, estado = prov}.
  - Luego de esto, envía el valor P actualizado a (E).
- 3) El nodo emisor (E) genera el número definitivo (llamado D) de orden para el mensaje (M). Para lograr esto, calcula el máximo de los valores P enviados por el resto de los nodos. Este número D es enviado al resto de los nodos junto con el identificador {D, I}. Ahora, cada nodo que recibe la tupla realiza lo siguiente:
  - Actualiza su valor $A = max(A, D)$.
  - Reemplaza el valor P con D en mensaje de la cola {M, I, P, estado = prov} $\Rightarrow$ {M, I, D, estado = acor}.
  - Reordena la lista según este nuevo valor D como primer ordenamiento y por el valor I como segundo.
  - Si el 1_{er} valor de la cola tiene "estado = acor", entonces es posible entregarlo si es solicitado.

Todo esto acompañado con las garantías que Erlang nos brinda sobre la conexión resuelve las propiedades pedidas.

Ahora solo falta ver como resolvimos que el sistema distribuido sea resistente a fallas, siendo estas que nodos de la red puedan desconectarse o "morir".

Como primera seguridad, por cada mensaje a enviar, se realizan checkeos sobre los nodos que deben responder. La idea se basa en que luego de haber enviado una petición de valor de orden, un agente se encargará de esperar las respuestas de todos los nodos de la red. En esta espera se aguarda un cierto tiempo y si en este no se recibieron las respuestas de todos los nodos a los que se le envió la petición, se realiza un chequeo liveness. En particular se verifica que los nodos a los que estamos aguardando estén aún en la red provista por Erlang. Si alguno de los nodos a los que se los estaba esperando no aparece como vivo, se lo borra de los mensajes a esperar. Por otra parte, si esta vivo se sigue esperando su respuesta. Esto mantendrá al sistema funcionando y ya que no tiene relación con el orden del resto de los nodos no generará problemas de prioridad en los mensajes.

Un problema mucho más grande que este no está en que alguno de los nodos no responda con un valor de prioridad, sino que el nodo que solicitó el valor muera durante el proceso. Esto se puede clasificar en las siguientes 3 situaciones:

- 1) El emisor comienza a solicitar el número provisorio y muere mientras esta mandando los pedidos (no a todos los nodos de la red le llega una solicitud). En este caso algunos de los nodos tienen el mensaje con valor provisorio y otros ni siquiera poseen el mensaje.
- 2) El emisor termina de solicitar el número provisorio pero muere antes de poder responderle a alguno de los nodos de la red. En este caso todos los nodos se quedan con un mensaje con un número provisorio.
- 3) El emisor comienza a responder con el número de orden acordado y muere mientras esta mandando las respuestas (no a todos los nodos de la red le llega un valor acordado para el mensaje). En este caso algunos de los nodos tienen el mensaje con valor acordado y otros van a quedar con un valor provisorio.

Para solucionar estos inconvenientes, se trabajo sobre la función **pop** de la **pqueue**. Si el primer mensaje de la lista (ordenada por prioridad) tiene estado provisorio, se procede a comprobar el estado de liveness del emisor. Si el emisor esta vivo, se sigue esperando la confirmación del mensaje. En caso contrario procedemos a preguntarle a la red si alguno de los nodos le llego el valor acordado para el mensaje por parte del emisor. *Sabemos que esta situación de inconsistencia no era necesario intentar resolverla pero pensamos que la solución a la que llegamos no es demasiado engorrosa y provee una mayor seguridad al programa.* Siguiendo, luego de este proceso si alguien si recibió un valor acordado, el resto de los nodos actualizan su mensaje con ese valor. En caso que todos los nodos respondan negativamente al pedido, se sabe que todos tienen solamente un valor provisorio del mensaje. En este caso, se procede a borrar el mismo ya que no afectará a la red.

Cabe aclarar que esta solución no es completa, pues siguen existiendo al menos un caso que conocemos en el cual se pierde la consistencia. El mismo es el siguiente: Un nodo emisor n_1 envía una solicitud de orden a los nodos de la red. Una vez conformada la prioridad acordada gracias a las respuestas de los nodos se comienza a reenviar esta prioridad final a los nodos involucrados.

El nodo emisor muere durante este proceso. Una cantidad **P** de nodos poseen respuesta con valor acordado. El resto de nodos (cantidad **Q**) no poseen este valor. Estos **Q** nodos entran al proceso anteriormente mencionado, donde solicitan al resto de los nodos de la red que le provean el valor acordado del mensaje si lo poseen. Acá se puede dar el caso extremadamente improbable que TODOS los **P** nodos que poseían el valor acordado crasheen mientras responden a los **Q** nodos. Esto llegaría a una situación en la cual una porción **V** de los **Q** nodos ahora posean el mensaje con un valor acordado mientras que los **Q - V** nodos restantes no poseen la respuesta. Esto decanta en un estado inconsistente en el sistema.

Dejando de lado este caso improbable, esta idea para ser resistente a fallos nos permite que el sistema distribuido continuará su ejecución por más de que nodos se retiren de la red (el ISIS intentará mantener el orden y la totalidad de los mensajes).

3. Ledger distribuido

Recordemos antes de continuar el objetivo pedido: implementar un objeto ledger de forma distribuida que sea tolerante a fallas utilizando el servicio de broadcast atómico implementado en la sección anterior.

Algunas asunciones que tuvimos son: al comienzo de la ejecución del ledger todos los nodos que van a ser parte de la ejecución están conectados a la red. El sistema no es capaz de asegurar que si un nodo se une a la red más tarde no obtendrá la lista completa de mensajes (a menos que se una antes de que se envíe el primer mensaje). Por otro lado, el sistema continuara su funcionamiento hasta un punto de parada establecido, el cual es que estén vivos la mitad de los nodos iniciales menos uno o quede solo 1 vivo (lo que suceda primero). Es decir que aunque un nodo deje la red, si sigue habiendo la cantidad necesario de nodos, el ledger continuará su funcionamiento.

Vale aclara que la función **ledgerGet**, devuelve el historial ordenado de izquierda a derecha, es decir, los mensajes mas recientes se encuentran a la izquierda de la lista.

Para realizarlo seguimos el algoritmo propuesto por el articulo Formalizing and Implementing Distributed Ledger Objects. En particular, cada usuario a su vez ejecuta un nodo del ledger, es decir que para poder hacer uso del ledger distribuido es necesario ejecutar un nodo del mismo. A su vez este hará uso del broadcast atómico distribuido y por lo tanto ejecutara también un nodo del sistema broadcast. Se hablará de mejoras posibles sobre esto en una sección posterior.

4. Compilación y ejecución

La compilación se realiza desde una maquina de Erlang simplemente ejecutando las siguientes lineas de código

```
c(isis).
c(ledger).
```

Por otra parte, antes de comenzar la ejecución del **ledger** con la función **ledger:start()**, se debe procurar que todos los nodos de la red estén conectados, es decir que todos tengan

corriendo el proceso **net_kernel** y a su vez todos los nodos se hayan comunicado mediante la función **net_adm:ping('nombre del nodo')**.

Para finalizar la ejecucion del sistema, se puede utilizar la funcion **ledger:stop()**.

5. Desarrollo del trabajo práctico

Nos gustaría comentar ciertos inconvenientes que surgieron durante la realización del trabajo.

Sobre ISIS, luego de finalizar la implementación del algoritmo que tradujimos de lo explicado en Wikipedia, comenzamos a desarrollar el código del ledger. Acá surgió el primer problema, en el cual si dos nodos realizaban una petición de numero de orden de mensaje al mismo tiempo (llamando al append de ledger), podía llegar a suceder la situación que si el P era el mismo, ambos responderían con P+1. Esto causaría que el mensaje final tenga valor de prioridad P+1, por lo que dos mensajes tendrían el mismo valor y el orden final no sería consistente entre nodos. Para solucionarlo, volvimos a trabajar sobre el código de ISIS, en donde pensamos como primer idea el agregar al número de orden de mensaje una parte flotante representada por el PId del remitente. Esto termino creando dos problemas: el primero era que entre la idea de pasar el PId a un número y luego hacerlo como parte decimal (0.PId) se llegó a un código engorroso y difícil de entender. El segundo problema y aún mayor era que, luego de investigar, vimos que los PId cambian dependiendo del nodo desde donde se observen (ya que es un valor local). Cada nodo contiene una referencia distinta para cada otro nodo de la red. Esto hacia que nuestro orden sea inconsistente (dos nodos distintos pueden darle a los dos mensajes el mismo P final) y por lo tanto deje de ser un orden válido. Replanteando el problema, y con ayuda de la cátedra, propusimos como orden de mensajes el siguiente: primero se ordena por numero de prioridad de mensaje y, en caso de que este coincida, se ordena por nombre de nodo en la red (sabemos que cada nombre es único). De esta forma logramos ordenar de manera válida los mensajes.

6. Mejoras

Sobre el ledger, proponemos como posibles mejoras las siguientes:

- Ingreso tardío a la red: se podría implementar que si un nodo ingresa luego de que se agregó información al historial del ledger, este no quede en un estado inconsistente con respecto al resto. Es decir que este pida a los nodos de la red la cantidad inicial de nodos, el historial actual y la lista provisora de mensajes (del ISIS).
- Usuarios independientes al ledger: posibilidad de que los usuarios no tengan que ejecutar un nodo del ledger para poder acceder a la información de la red. Esto se haría utilizando nodos invisibles que se comuniquen con el ledger.

7. Bibliografía

- https://es.wikipedia.org/wiki/Multidifusión
- Formalizing and Implementing Distributed Ledger Objects