Implementacion de un ledger distribuido en una red en Erlang

Franco Ignacio Vallejos Vigier y Facundo Emmanuel Messulam

22 de junio de 2021

1. Introducción

A partir de el algoritmo propuesto en "Formalizing and Implementing Distributed Ledger Objects" (Anta, Georgiou, Konwar and Nicolaou 2018) y el algoritmo "ISIS algorithm for total ordering of messages" damos una implementación de un ledger distribuido que soporta añadir objetos (records) y leer el estado actual (la lista de records). Este ledger posee las garantías de consistencia atómica y es resistente a fallos en parte de los servidores que lo mantienen.

2. Secuenciación

Para implementar el algoritmo del ledger distribuido (en adelante simplemente algoritmo ledger) se requiere implementar una forma de ordenamiento estricto sobre algunos de los mensajes de este, específicamente, los broadcasts (`ABroadcast`). Para lograr este ordenamiento estricto se usa un orden estricto ascendente en los mensajes. La pregunta es como se logra un algoritmo que sea capaz de generar este ordenamiento total y que problemáticas se pueden dar, el lector no debe olvidar que los mensajes son en una red, y no puede permitir que el ordenamiento se rompa con los diversos problemas que estas pueden tener.

2.1. Sequencer

Una primera implementación fue proveída por la cátedra en la forma de un Sequencer, este se clasifica como "Fixed Sequencer Algorithm" con "Unicast-Broadcast" (Défago, Schiper & Urbán 2003), esta clasificación hace referencia a que hay un solo secuenciador que hace toda la operación de numeración, es decir, la red no tiene una decisión federada sobre el numero, el numero se asigna al llegar un mensaje al secuenciador y luego se reenvía el mensaje numerado a todos.

La implementación descrita en esta sección se basa en un proceso de secuenciación, utilizando un enfoque centralizado que distingue entre un proceso (en adelante: secuenciador) y los procesos participantes.

El secuenciador recibe los mensajes de cualquier proceso y los difunde a todos los procesos, garantizando el orden total gracias a que incluye un número de secuencia en los mensajes para que todos los procesos a su vez entreguen los mensajes en el mismo orden (en nuestro caso al algoritmo ledger), es decir, un mensaje de número n sólo se entrega si el mensaje $n-1$ ya se ha entregado.

En ausencia de fallos esta implementación cumple claramente con las propiedades de la primitiva de broadcast atómico:

- Validity: si un servidor correcto envia un mensaje, el secuenciador eventualmente lo entrega
- Uniform Agreement: como son todos iguales los participantes no coordinadores, es fácil ver que si el mensaje lo entrega (al ledger) uno, todos los otros también lo van a hacer
- Uniform Integrity: es claro ver que no aparecen mensajes de la nada con este sistema. Lo que se envía al secuenciador es recibido y entregado y lo que no se envía no es ni recibido ni entregado
- Uniform Total Order: como se explico anteriormente, los mensajes están ordenados y la entrega al algoritmo ledger es en orden.

Esta implementación satisface la propiedad de atomicidad de la primitiva de difusión atómica porque un proceso sólo puede entregar un mensaje si fue

enviado por el secuenciador, y el este mantiene todos los mensajes en el almacenamiento hasta que se garantice que son mensajes entregables al algoritmo ledger. La propiedad de orden está garantizada por los números de secuencia únicos asociados a los mensajes.

Una vez que el secuenciador recibe un mensaje para ser difundido, todos los procesos participantes lo recibirán correctamente si no fallan, garantizando la propiedad de terminación; cabe destacar que esto es algo más complicado en la practica, pero es transparentemente solucionado por Erlang.

2.1.1. Problemas

Uno de los problemas más claros es que si el único secuenciador de la red se cae, no se puede transmitir ningún mensaje, al menos hasta que haya uno nuevo que lo reemplace; cabe destacar que el secuenciador de la cátedra no tiene una manera de ser automáticamente relanzado cuando se cae. Crear o elegir un nuevo secuenciador no es una tarea simple: la red puede ser muy grande pero la cantidad de secuenciadores debe siempre ser uno solo, es decir, se debe encontrar una forma de llegar a un acuerdo de cual es el nuevo secuenciador único, sin un secuenciador para mediar estos mensajes.

Otro de los problemas es de congestión, todos los mensajes (a secuenciar) deben llegar al secuenciador, y todos los mensajes (secuenciados) deben salir del secuenciador, dependiendo que algoritmo estén ejecutando los nodos, esto puede significar una cantidad enorme de mensajes entrante, y más enorme aun saliente, si bien la congestión se puede solucionar con una configuración de entrada y salida (en la conexiones) que permita el volumen, este es un problema que debe tenerse en cuenta.

Un problema que puede darse en ciertas conexiones es que se pierdan paquetes, esto es de consideración aquí porque los mensajes tienen una sola copia cuando se envía, es decir, si se pierde un mensaje en la ida al secuenciador, no va a ser secuenciado y repartido. Erlang soluciona este problema garantizado eventual llegada de mensajes al destinatario, es decir, en un futuro, si no se cae el emisor o el destinatario, el mensaje se recibe (una nota: también se garantiza que M enviado antes que N de p_i, llegan en orden).

Aunque esta implementación ofrece un buen rendimiento, la desventaja es que se basa en que el secuenciador debe ser un proceso correcto y sin fallos, es decir, perfecto y eterno, condición muy difícil de cumplir (Mesbahi, Rahmani and Hosseinzadeh 2018) y no esperada para un único nodo en este trabajo.

2.2. ISIS

La cátedra propone “ISIS algorithm for total ordering of messages” (a partir de ahora simplemente ISIS) como una solución a los problemas planteados en la sección anterior. Analizaremos esto y daremos nuevos posibles problemas.

Es claro que con una implementación de el secuenciador que no recide en un solo nodo, la secuenciación no se puede simplemente ‘caer’, esto requiere una definición más detallada de la condición de caída. Para simplificar la explicación

simplemente vamos a tomar el numero dado por el paper del ledger a su palabra y establecemos una condición fuerte en el código, esto es, cuando la cantidad de nodos activos es menos que $2f + 1$ donde f es los nodos no vivos, decimos que la red esta caída, tanto el ISIS como el algoritmo ledger.

También se soluciona el problema de la congestión, a pesar de que hay mas mensajes en la red en total (el algoritmo requiere al menos dos broadcasts) estos viajan de p_i a p_j donde ni i ni j es constante entre mensajes.

Se puede decir además, que ISIS envía siempre muchas copias del mensaje, entonces, a pesar de que se podría perder alguno, el mensaje en si es recibido y ordenado correctamente por la mayoría, a menos que el nodo haya sido desconectado completamente de la red. Sin embargo, esto no es un problema en Erlang simplemente debido a las garantías de Erlang.

Una forma sencilla de conseguir un ordenamiento coherente de los mensajes en diferentes nodos es asignar prioridades a los mismos, y luego entregar los mensajes en el orden de prioridad creciente, es decir, el mensaje de menor prioridad se entrega primero. A diferencia del secuenciador no hay un solo proceso coordinador en la red capaz de asignarle un orden a todos los mensajes, en este protocolo todos los diferentes nodos asignan prioridades a los mensajes, usando un mecanismo de 'recomendación'.

El protocolo ISIS hace que todos los nodos acuerden explícitamente la prioridad de un mensaje y, a continuación, sólo se asignan prioridades más altas a los mensajes posteriores.

Podemos resumir el algoritmo en tres fases:

Fase I. El emisor transmite el mensaje m con una prioridad p a todos los nodos.

Fase II. Cada receptor añade el mensaje a su cola y lo etiqueta como "undeliverable". A continuación, asigna a este mensaje una prioridad superior a la de cualquier otro mensaje que se haya en su cola de mensajes. Después, asigna a este mensaje una prioridad superior a la de cualquier de cualquier mensaje que se haya colocado en el búfer.

Fase III. El emisor recoge las respuestas de todos los nodos que no han fallado. Luego, calcula el valor máximo de todas las prioridades que ha recibido y envía este valor a los receptores. Siendo este valor de prioridad el consensuado entre los nodos.

Cada receptor cambia la prioridad del mensaje a la prioridad recibida del remitente, y etiqueta el mensaje como "deliverable". Clasifica la cola en el orden de la prioridad de los mensajes. Empezando a entregar los mensajes con la etiqueta "deliverable" desde aquellos con menor prioridad. La entrega de los mensajes se detiene en el primer encuentro de un mensaje etiquetado como "undeliverable". Este protocolo garantiza la primitiva de un atomic broadcast pero sin el problema de un nodo centralizado "infalible".

Cabe destacar que la prioridad no solo involucra al numero de secuencia, sino también al envíate, para deshacer empates, que de hecho son posibles.

Mostrar como el ISIS cumple las propiedades necesarias de un atomic broadcast no es simple. Se puede encontrar una descripción mas detallada del algoritmo en "Reilable Communication in the Presence of Failures" (Birman and Joseph 1987).

3. Ledger

En este trabajo implementamos un ledger distribuido, sabemos que el ledger va a estar presente en todos los nodos de nuestra red, brindándole atomic consistency a cada uno de nuestros clientes. Una vez que algún cliente realiza un broadcast todos los demás nodos actualizan su copia del ledger consistentemente (teniendo todos los clientes la misma lista de records). Destaca el hecho de que en el ledger distribuido se elimina la necesidad de que una autoridad central controle la manipulación y actualización de datos.

Cabe destacar que para que el ledger funcione necesitamos que la mayoría de nodos estén activos ($f < 2*n+1$, donde n es la cantidad de nodos al inicio), esto deriva de que todo ledger distribuido necesita el algoritmo de consenso (ISIS) entre sus nodos para realizar una actualización en cada uno.

Ahora se nos presenta un nuevo problema, ya que nuestro objetivo es implementar un ledger distribuido con servicio de broadcast atómico debemos garantizar en nuestro ledger una consistencia atómica entre los request de todos nuestros nodos. Esto es para que después de cada instrucción (que puede ser un `get` o un `append`) todos los ledgers sean actualizados correctamente. Para ellos necesitamos lo denominado consistencia atómica entre los n nodos de nuestra red. Guiados por el paper de la cátedra combinamos nuestra implementación de los códigos 5 y 8 para obtener un atomic distributed ledger, en el cuál el ledger de cada nodo es actualizado luego de que cualquiera de los mismos haga una llamada a `append`.

3.1. Ledgers y su relación con las blockchain

El paper hace referencia al uso de sistemas descentralizados que almacenan información de forma que cada elemento subsiguiente esta intrínsecamente extendiendo la cadena. Esto es, cada bloque posee una identificación criptográficamente difícil de crear por lo tanto es muy improbable que entradas anteriores sean revertidas.

La forma de nuestro ledger hace referencia a este mecanismo, sin embargo no posee los requisitos de dificultad de creación de una desde cero. Pero, muy importantemente, posee características que son muy deseadas de las blockchain, esto es consistencia atómica y garantías de funcionamiento, a pesar que estas garantías, como fueron implementadas en el trabajo no son resistentes a clientes malintencionados, es claro ver como si un desarrollador necesita un punto de partida, este trabajo es extremadamente útil.

A partir de las garantías creadas no solo por Erlang, y también por los algoritmos implementados, un desarrollador puede ver la utilidad de este trabajo.

4. Ejecución del código

Se compila con:

```
erl -compile isis ledger
```

Luego hay que levantar varias instancias de consola:

```
erl -setcookie wermer -sname <nombre>
```

Una vez que levantamos eso, podemos hacer `ping` para conectar:

```
net_adm:ping('<nombre1>@<host1>').
net_adm:ping('<nombre2>@<host2>').
net_adm:ping('<nombre3>@<host3>').
...
```

Levantamos el ledger en cada consola:

```
ledger:start().
```

Luego podemos hacer:

```
ledger:get().
```

Para obtener una copia del ledger.
Y:

```
ledger:append({IdUnico, self(), Mensaje}).
```

Para agregar un elemento al ledger.

Referencias

[BT87] K. Birman y Joseph T. *Reliable Communication in the Presence of Failures*. ACM Transactions on Computer Systems 5, 1987.

[DUS04] Xavier Défago, Peter Urban y André Schiper. "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey". En: *ACM Computing Surveys* 36 (dic. de 2004). DOI: 10.1145/1041680.1041682.

[Ant+18] Antonio Fernández Anta y col. *Formalizing and Implementing Distributed Ledger Objects*. 2018. arXiv: 1802.07817 [cs.DC].

[MRH18] Mohammad Reza Mesbahi, Amir Masoud Rahmani y Mehdi Hosseinzadeh. "Reliability and high availability in cloud computing environments: a reference roadmap". En: *Human-centric Computing and Information Sciences* 8.1 (jul. de 2018). DOI: 10.1186/s13673-018-0143-8. URL: https://doi.org/10.1186/s13673-018-0143-8.