Universidad Pública de Navarra

Trabajo Fin de Grado

Algoritmos distribuidos en Node.js

David González Portillo

Tutor:
Jose Ramón González de Menvidil

24 de julio de 2020

Índice

1. Objetivos

1.1. Sistemas distribuidos y Node.js

Entre los objetivos de este proyecto se encuentra el estudio de Node.js como lenguaje para la implementación de los algoritmos distribuidos, lo útil que puede ser frente a otros lenguajes a la hora de crear sistemas distribuidos y la estrategia a seguir para su desarrollo según las especificaciones y funcionamiento del lenguaje. Durante el proyecto se explicará el funcionamiento de Node.js, la forma en la que los algoritmos propuestos han sido adaptados a él, y cómo se ha realizado el desarrollo de su código.

1.2. Algoritmos de consenso y orden total

Todo sistema distribuido consta de una serie de nodos que trabajan simultáneamente con el fin de distribuir el trabajo de cómputo entre todos los nodos. Para que éstos funcionen de forma conjunta correctamente, es necesario que exista cierta comunicación entre ellos, de forma que se consiga garantizar el consenso y el orden total de las acciones que el sistema debe realizar como si fuese un solo nodo. A los algoritmos pensados para establecer la comunicación entre nodos y garantizar estas propiedades se los denomina protocolos de consenso y de orden total.

A lo largo del tiempo se han ideado muchos protocolos de este tipo. Entre ellos, URB y EpTO son dos que vamos a estudiar en este proyecto. Más adelante se explicarán en profunidad.

1.3. Sistema de comunicación de nodos tolerante a fallos

El principal objetivo de este proyecto es implementar una serie de protocolos de consenso y orden total que permitan desarrollar un programa simple de comunicación de nodos broadcast que sea tolerante a fallos. Aunque las pruebas se realizen en redes no muy extensas y los programas construidos encima de los protocolos realicen una comunicación muy simple, los protocolos están pensados para un uso a mayor escala y utilización en aplicaciones distribuidas mucho más complejas. Al fin y al cabo, el protocolo de comunicación es sólo la base de un sistema distribuido.

2. Introducción

2.1. Aplicaciones y servicios en sistemas distribuidos

Hoy en día el volúmen de datos y clientes es inmenso, y requiere que los sistemas de información y aplicaciones tengan cada vez más disponibilidad, más robustez y sean aún más eficientes. En definitiva, se requiere el desarrollo de sistemas distribuidos que sean capaces de soportar grandes cargas y cantidad de fallos.

2.2. Ventajas potenciales

Ventajas de sistems distribuidos, rendimiento, alta disponibilidad y fallos

2.3. Problemas y soluciones

Problemas que resuelven los sistemas distribuidos, y problemas imposibles de resolver (ejemplo consenso)

2.4. Modelos de sistemas distribuidos

Asincronía, formas de comunicación, ejemplos grandes sistemas

2.5. El problema del consenso

Importancia consenso, equivalencia con otros problemas como el broadcast de orden total. Importancia broadcast de orden total.

2.6. Organización del proyecto

Como se van a abarcar los objetivos, y como se estructura el proyecto

2.7. ¿Por qué en NodeJS?

Si hay algo por lo que se caracterizan los sistemas distribuidos, es que son asíncronos. Están preparados para funcionar a pesar de los retrasos, caídas y demás fallos. Se piensan para ser muy robustos a pesar de toda complicación. NodeJS (Javascript) es un lenguaje asíncrono, pensado para entornos reactivos, ocurrencia de eventos. Cuando se dispara un evento este se añade a la cola, donde se desapila el primero y se ejecuta el código reactivo correspondiente, el cual puede desencadenar más eventos, y así sucesivamente. Los programas terminan en cuanto no esperan ningún evento, cosa que puede no suceder nunca.

Durante el primer mes me dediqué a recordar NodeJS y ZMQ. Utilicé ambas durante la beca que realicé el año pasado, la cuál era también en sistemas distribuidos. Para recordar NodeJS utilicé el mismo libro que el año pasado [3]. El libro tiene una serie de capitulos con ejercicios básicos que explican progresivamente todo lo necesario. ZeroMQ (ØMQ) [4] es una libreria universal de código abierto de comunicación mediante mensajes disponible en

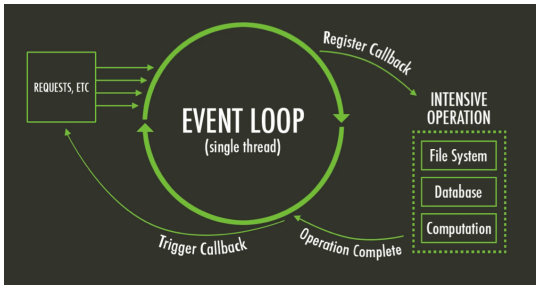

Figura 1: Ciclo NodeJS

múltiples languages, entre ellos, Node. Es la capa intermedia entre nuestro protocolo y el nivel de transporte. ZMQ se encarga de todo lo necesario a nivel socket: envío y recepción de mensajes, creación y escucha de sockets, serialización, conexión, restablecimiento de la misma... Hace que resulte muy cómoda la implementación de la parte de comunicación entre nodos. Es una librería muy reconocida y utilizada, además de que posee una amplia documentación.

Figura 2: Node.js y ZMQ

3. URB: Uniform Reliable Broadcast

3.1. Descripción

En este capítulo vamos a centrarnos en la abstracción del problema Uniform Reliable Broadcast y su implementación en un sistema de paso de mensajes asíncrono tolerante a fallos. El que sea tolerante a fallos es el centro de diseño de cualquier sistema distribuido, puesto que todos requieren la garantía en la entrega de mensajes.

3.1.1. Fiabilidad garantizada

Para entender a dónde queremos llegar, antes hay que entender lo más básico. Dado un conjunto de n procesos y un mensaje m, broadcast(m) es el proceso que envía ese mismo mensaje a todos los procesos.

En un sistema asíncrono, si el proceso que envía el mensaje es correcto, entonces una copia del mensaje es enviada a todos los procesos, y como los canales son fiables, todos los procesos que no hayan fallado recibirán una copia del mensaje. Como los canales son también asíncronos, cada uno de los procesos puede recibir la copia del mensaje en instantes de tiempo independientes. Además, si el proceso que realiza el broadcast falla durante los envíos, entonces un subconjunto arbitrario de procesos recibirán una copia del mensaje. Esto quiere decir que el broadcast por sí solo no indica de niguna forma qué procesos recibirán el mensaje y cuáles no. El objetivo de URB es encontrar la forma de diseñar un broadcast alternativo que garantize la entrega de los mensajes a todos los procesos.

3.1.2. Uniform Reliable Broadcast

URB propone dos nuevas operaciones llamadas URB-broadcast(m) y URB-deliver(). El primero permite a un proceso enviar un mensaje a todos los procesos (incluido el mismo) y el segundo permite a un proceso recibir un mensaje que ha sido previamente enviado mediante URB-broadcast por otro proceso.

La especificación de estas operaciones asume que cada mensaje enviado es único. Para implementar esto tan sólo se debe asociar un ID único a cada mensaje. Una buena forma de hacer esto es formar un ID concatenando el ID del proceso emisor y el ID de secuencia de envío de mensajes auto-incremental de ese mismo proceso.

Definición La especificación consiste en tres propiedades. Cualquier implementación debe cumplirlas para ser correcta:

- Validez: Si un proceso realiza URB-deliver de un mensaje m, entonces antes otro proceso ha realizado URB-broadcast(m).
- Integridad: Un proceso realiza URB-deliver de un mensaje m como mucho una vez.
- Terminación: Si un proceso correcto realiza URB-broadcast de un mensaje m, o si un proceso realiza URB-deliver de un mensaje m, entonces todos los procesos correctos realizan URB-deliver(m).

La propiedad de validez relaciona el envío con la entrega de mensajes. Dicho de otra forma, garantiza que no hay creación ni alteración de mensajes. La propiedad de integridad garantiza que no hay duplicidad de mensajes. Nótese que ambas propiedades de validez e integridad se cumplen incluso si ningún mensaje es entregado nunca, sin importar los mensajes que hayan sido emitidos. Aquí es donde entra la tercera propiedad de terminación: si el

proceso que realiza URB-broadcast de un mensaje es correcto, o si al menos un proceso (sea correcto o no) realiza URB-deliver de un mensaje, entonces ese mensaje es recibido por al menos todos los procesos correctos.

De las anteriores propiedades se puede concluir que (1) los procesos correctos reciben exáctamente el mismo conjunto de mensajes, (2) que este conjunto incluye todos los mensajes emitidos por todos los procesos correctos, y (3) que todo proceso no correcto recibe un subcojunto de los mensajes recibidos por un proceso correcto. (aunque dos procesos no correctos distintos pueden recibir dos subconjuntos de mensajes distintos). Es importante ver que un mensaje emitido por un proceso no correcto puede ser o no ser entregado. No es posible exigir un requerimiento a un mensaje si el emisor puede fallar antes de que el mensaje haya sido enviado a un proceso correcto. La entrega de los mensajes enviados por procesos no correctos dependen de la ejecución.

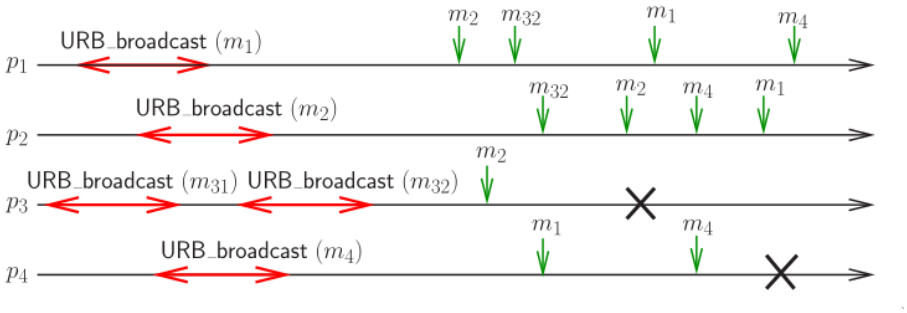

Figura 3: Un ejemplo de la garantía de entrega de mensajes

En la figura 3 se puede ver un simple ejemplo. Hay cuatro procesos que envían 5 mensajes en total. Los procesos p1 y p2 son correctos mientras que p3 y p4 fallan en cierto punto (marcado en la figura con una cruz). Las flechas verdes indican cuándo un proceso entrega un mensaje. Se puede ver que los dos procesos correctos entregan el mismo conjunto de mensajes (aunque en desorden) y en cambio, los procesos no correctos, entregan un subconjunto de éste. Nótese que el mensaje m31 enviado por p3 nunca es entregado por ningún proceso, y que incluso no entrega los mensajes m31 y m32 que él mismo envía. Por otro lado, el mensaje m32, que es enviado antes que el m31, sí que es entregado por los procesos correctos. Esto se debe a la asincronía y fallo del envío de mensajes. Es claramente visible que lo sucedido en la figura 3 respeta la especificación de URB.

El problema URB es un paradigma que captura toda una familia de problemas de coordinación distribuida. Por ejemplo, las acciones URB-broadcast() y URB-deliver() pueden interpretarse como «esto es una orden» y «ejecutando orden» respectivamente. Por lo tanto todos los procesos correctos ejecutarán las mismas acciones, incluyendo las órdenes enviadas por los procesos correctos por supuesto, y algunas enviadas por los procesos fallidos.

Nótese que URB es un problema de un disparo cada vez. Es decir, la especificación se aplica individualmente a cada mensaje que es enviado mediante URB-broadcast, independientemente de los demás.

La abstracción de comunicación reliable broadcast es una forma debilitada de URB. Se define con las mismas propiedades de intregridad y validez (sin pérdida de mensajes, corrupción o duplicidad) y la siguiente propiedad de terminación:

- Terminación: Si un proceso correcto (1) realiza URB-broadcast() de un mensaje m, o (2) realiza URB-deliver de un mensaje m, entonces todo proceso correcto realiza URB-deliver(m).

Esto significa que un proceso que tenga fallos puede entregar mensajes que uno correcto no entregue. Esta propiedad de terminación es la misma que la propiedad de terminación de URB sin su uniformidad.

Se puede ver que la propiedad de terminación de la abstracción del reliable broadcast no obliga a que los

mensajes entregados por los procesos con fallos sean un subconjunto de los enviados por los que son correctos. Así que es mucho menos restrictivo que URB, y por lo tanto, menos práctico.

3.2. Implementación

3.2.1. Construyendo URB

Existe una construcción muy simple de la abstracción URB en el modelo de sistema $AS_{n,t}[\emptyset]$. Esto es gracias a que los canales punto a punto son fiables. La estructura de la construcción se muestra en la figura 4.

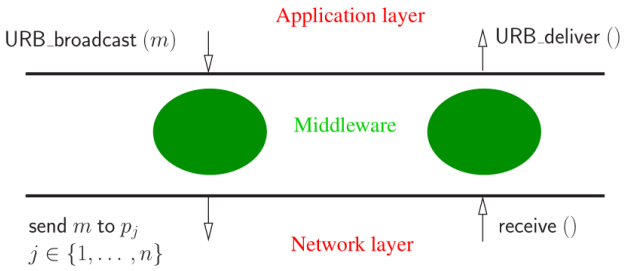

Figura 4: Arquitectura de URB

Los algoritmos que implementan URB-broadcast() y URB-deliver() están descritos en la figura 5. Para emitir un mensaje m, un proceso p envía m a sí mismo. Cuando un proceso p recibe un mensaje, lo descarta si ya ha recibido una copia del mismo. Gracias a que cada mensaje m tiene un identificador único, es fácil para p comprobar si ha recibido m anteriormente. Si lo está recibiendo por primera vez, entonces lo reenvía a todos los procesos (excepto a sí mismo y al proceso que se lo acaba de enviar), y sólo entonces lo entrega a la aplicación.

Es importante darse cuenta del hecho de que la recepción del un mensaje m no es atómica. Un proceso p puede recibir un mismo mensaje múltiples veces.

Teorema El algoritmo descrito en la figura 5 construye una abstracción de comunicación Uniform Reliable Broadcast en $AS_{n,t}[\emptyset]$.

```
operation URB_broadcast (m):
      send MSG(m) to p_i.

when MSG (m) is received from p_k:
      if (first reception of m) then
        for each j ∈ {1, ..., n} \ {i, k} do send MSG (m) to p_j end for;
         URB_deliver (m) % deliver m to the upper layer application %
      end if.
```

Figura 5: Uniform Reliable Broadcast en $AS_{n,t}[\emptyset]$.

Demostración La prueba de la propiedad de validez se extrae directamente de la parte del algoritmo en la que se reenvían únicamente los mensajes que se han recibido. La prueba de la propiedad de integridad se extrae directamente del hecho de que un mensaje m sólo es entregado cuando se recibe por primera vez.

La propiedad de terminación es una consecuencia directa de la estrategia «primero se reenvía y luego se entrega». Supongamos que un proceso correcto p realiza URB-broadcast de un mensaje m. Como p es correcto, reenvía el mensaje al resto de procesos y luego se lo entrega a sí mismo. Como los canales son fiables, eventualmente una copia del mensaje llegará a todos y cada uno de los procesos restantes, y realizarán URB-deliver(m).

Ahora vamos a considerar el caso de un proceso (con fallos o correcto) que entrega un mensaje m. Antes de haber entregado el mensaje, ha tenido que reenviar el mensaje a todos los procesos, y esto se aplica consecutivamente, lo que prueba la demostración de la propiedad de terminación.

3.2.2. Implementación en NodeJS

Para la comunicación entre nodos o procesos y uso de sockets en NodeJS utilizaremos una librería muy famosa llamada ZMQ. Esta librería proporciona una serie de herramientas para la creación y uso de sockets de una forma muy sencilla.

Esta librería de código abierto está disponible para multitud de lenguajes, entre ellos NodeJS. Ofrece la posibilidad de enviar mensajes a través de sockets de forma atómica a través de diferentes capas de transporte como son IPC, TCP, y Multicast. También ofrece diferentes patrones de comunicación, como petición-respuesta, suscriptor-publicador y distribución de tareas. Está preparada mayormente para sistemas concurrentes y sobretodo está pensada para sistemas distribuidos y paradigmas asíncronos, como es nuestro caso.

Nosotros utilizaremos un esquema de conexión peer-to-peer en el cual cada nodo o proceso está conectado a todos los demás individualmente. El algoritmo funcionará sobre la capa de transporte TCP sobre canales fiables.

La implementación en NodeJS, al igual que el pseudocódigo, resulta muy sencilla. Consiste en construir un módulo que tendrá cinco partes fundamentalmente:

- Conexión con el resto de procesos: se encargará de crear los sockets necesarios para la comunicación con cada proceso de forma que los canales que se tengan sean fiables.
- Escuchar mensajes de otros procesos: capturará los eventos de recepción de mensajes y los tratará como se indica en el pseudocódigo.
- Enviar mensajes al resto de procesos: preparará el contenido de los mensajes a enviar y los enviará a través de los sockets correspondientes.
- Reloj: controlará el tiempo de los mensajes mediante un reloj lógico o global.
- Aplicación: es la parte que se encargará de enviar los mensajes a la capa superior o aplicación para que el cliente pueda recibirlos.

Conexión con el resto de procesos Al iniciarse un proceso, inmediatamente crea un conjunto de sockets a partir de las direcciones del resto de procesos. Para esto se utilizan los módulos *router* y *dealer* de ZMQ. Cada proceso dispone de su componente *router* para recibir mensajes del resto de procesos, y también posée un conjunto de componentes *dealer* (uno para cada proceso). Cada uno de estos *dealers* se conecta con un router del resto de procesos, formando así una red de sockets que conecta cada proceso con los demás.

Por ejemplo, si tuvieramos un sistema de 20 nodos, cada nodo tendría 1 *router* y 19 *dealers*, puesto que utilizaría su *router* para recibir mensajes de los *dealers* de otros nodos y usaría cada uno de sus *dealers* para conectarse a un *router* de otro proceso.

Al final se requiere una conexión *router-dealer* para la comunicación entre dos nodos distintos.

Escuchar mensajes de otros procesos Una vez que los sockets han sido creados e inicializados, nos disponemos a preparar la escucha de mensajes. Para esto se le asigna a cada nodo o proceso un identificador único (que también se asigna a su *router*) de forma que otros procesos puedan identificar de dónde proceden los mensajes que reciben. Una vez hecho esto, cada proceso realiza el *bind* en su dirección IP y entonces puede comenzar a recibir mensajes.

Cuando el proceso recibe un mensaje por primera vez, lo reenvía al resto de nodos y inmediatamente lo entrega a la capa superior de la aplicación.

Enviar mensajes al resto de procesos Cuando nuestro proceso recibe un mensaje de capa superior o aplicación, entonces es cuando debe preparar el mensaje con la información necesaria (número de secuencia de mensaje, identificador único de proceso emisor, contenido del mensaje, tiempo de creación...) para enviarlo al resto de procesos realizando un *broadcast*. Para garantizar las propiedades del algoritmo que hemos visto anteriormente, el proceso primero debe enviar una copia del mensaje a través de cada uno de los sockets con el resto de procesos, y una vez hecho esto, ya puede entregar el mensaje de vuelta a la aplicación para que ésta pueda indicar al cliente que el mensaje ha sido entregado.

Reloj El reloj es un simple módulo que obtiene el tiempo actual para poder marcar el tiempo en el que los mensajes han sido creados y así poder ordenarlos según cuándo han sido enviados.

3.3. Pruebas y rendimiento

Las pruebas se realizaron en un clúster de máquinas del laboratorio de la Universidad Pública de Navarra, en el cual se disponía de 7 máquinas y un sistema de clúster Proxmox sobre el sistema operativo Linux, y docker para el uso de contenedores. Un contenedor es muy similiar a una máquina virtual, realiza todas las funciones de una máquina independiente real, o más bien, simula, de forma que es ideal para hacer pruebas de sistemas que luego realmente sí van a estar en máquinas totalmente independientes. Los contenedores están completamente aislados entre sí, permitiendome así testar el algoritmo como si de una red real se tratase.

Así pues en cada una de las 7 máquinas se realizaron diferentes pruebas con 1, 3, 5 y 10 contenedores, consiguiendo redes de 7, 21, 35 y hasta 70 nodos distintos. Se programaron una serie de tests en los que cada nodo enviaba mensajes simulados automatizados aleatoriamente dentro de un rango de tiempo determinado. Al mismo tiempo, cada nodo recogía los mensajes que recibía y los imprimía en registros que luego servirían para diferenciar los mensajes recibidos entre nodos diferentes.

Al ser una red tan aislada y local, fue imposible comprobar como reaccionaría el sistema si alguno de los nodos fallase. Dado que ninguno fallaba, se programó también varios tests en los que aleatoriamente algunos de los nodos simulaban una caída. Los resultados, tanto en la primera serie de tests como en las siguientes, fueron los esperados. Según dicta la teoría, aquellos nodos que eran correctos siempre entregaban el mismo conjunto de mensajes, y aquellos que eran fallidos entregaban un subconjunto de éste.

4. EpTO

4.1. Descripción

En este capítulo vamos a ver en qué consiste el algoritmo EpTO y de cómo ha sido implementado en NodeJS. Al igual que URB, es un algoritmo diseñado para sistemas distribuidos, esta vez a gran escala, cuyo objetivo gira en torno a la difusión de mensajes broadcast. Sin embargo, el objetivo de este algoritmo es algo más ambicioso.

Integrity: For any event e, every process EpTO-delivers e at most once, and only if e was previously EpTO-broadcast.
Validity: If a correct process EpTO-broadcasts an event e, then it eventually EpTO-delivers e.
Total Order: If processes p and q both EpTO-deliver events e and e', then p EpTO-delivers e before e' if and only if q EpTO-delivers e before e'.
Probabilistic Agreement: If a process EpTO-delivers an event e, then with high probability all correct processes eventually EpTO-deliver e.

Figura 6: Propiedades EpTO

Epidemic Total Order Algorithm for Large-Scale Distributed Systems (EpTO) es un protocolo de orden total o protocolo de consenso diseñado para sistemas distribuidos altamente escalables. Se trata de un algoritmo epidémico que nos garantiza el orden total, la validez e integridad y, con una gran probabilidad, la entrega de los mensajes. La idea del protocolo es un simple broadcast: los nodos se envían mensajes entre sí, cada nodo a todos los demás. Mensajes que reciben de la capa superior (la aplicación). El objetivo es que todos los nodos reciban los mismos mensajes exáctamente en el mismo orden. Para ello garantiza cuatro propiedades: integridad, validez, orden total y entrega probabilística.

4.1.1. Comprensión EpTO

La idea detrás de EpTO para la entrega probabilística es el problema de los hoyos y las canicas. Los nodos son pensados como un conjunto de hoyos y cada mensaje como un conjunto de canicas. Los hoyos a los que no caen canicas representan los nodos a los que los mensajes no han llegado debido a un fallo. El objetivo es enviar el número de canicas por mensaje necesario para garantizar que la probabilidad de que todos los hoyos hayan recibido el mensaje al menos una vez sea muy alta.

El protocolo consta de dos componentes principales: el componente de difusión y el componente de ordenación.

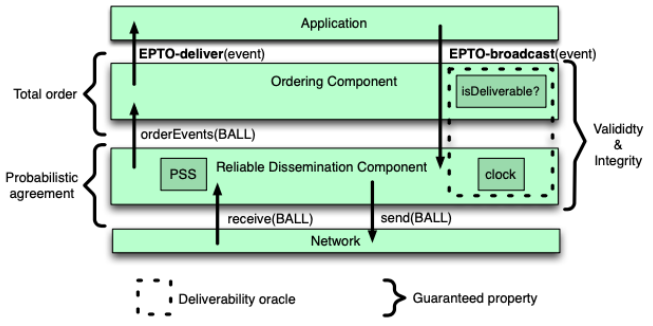

Figura 7: Arquitectura EpTO

4.1.2. Componente de difusión

Algorithm 1: Dissemination component (process p)

```
1  initially
2      view ← ... // system parameter: set of uniformly random
       correct peers
3      K ← ...                                  // system parameter: fanout
4      TTL ← ...  // system parameter: nb times events need to
       be relayed
5      nextBall ← ∅     // set of events to be relayed in the next
       round
6  procedure EpTO-broadcast(event)
7      event.ts ← getClock()
8      event.ttl ← 0
9      event.sourceId ← p.id
10     nextBall ← nextBall ∪ (event.id, event)
11 upon receive BALL(ball)
12     foreach event ∈ ball do
13         if event.ttl < TTL then
14             if event.id ∈ nextBall then
15                 if nextBall[event.id].ttl < event.ttl then
16                     nextBall[event.id].ttl ← event.ttl
                       // update TTL
17             else
18                 nextBall ← nextBall ∪ (event.id, event)
19         updateClock(event.ts)        // only needed with logical
           time
20 task every δ time units
21     foreach event ∈ nextBall do
22         event.ttl ← event.ttl + 1
23     if nextBall ≠ ∅ then
24         peers ← Random (view,K)
25         foreach q ∈ peers do
26             send BALL(nextBall) to q
27     orderEvents(nextBall)
28     nextBall ← ∅
```

Figura 8: Componente de difusión

Este componente se encarga de la comunicación y difusión de mensajes, los cuales se denominan eventos. Cada evento es un mensaje que se desea enviar a todos los nodos. El componente empaqueta múltiples eventos en lo que llama BALL (lo que antes hemos entendido como canica), y la envía al resto de nodos. Antes de comprender el funcionamiento completo necesitamos entender las variables utilizadas:

- View: conjunto de todos los procesos correctos.
- K: tamaño de la muestra de procesos a los que va a ser enviada la próxima BALL.
- TTL (Time To Live): número máximo de saltos que un evento puede realizar (mismo funcionamiento que los paquetes TCP-IP).
- Delta: duración de cada ronda.
- nextBall: próxima BALL o próximo conjunto de eventos a enviar.

El funcionamiento es simple:

1. El proceso recibe un evento nuevo desde la capa de la aplicación.
2. Lo añade a la próxima BALL a enviar.
3. Cuando se recibe una BALL de otro proceso, se analizan todos sus eventos.
4. Si su TTL ya es muy alto se ignoran, y si no se actualizan o/e introducen en la próxima BALL para ser reenviados.
5. Cuando comienza la próxima ronda, se aumenta en 1 el TTL de todos los eventos pertenecientes a la próxima BALL.
6. Justo después se selecciona un subconjunto de tamaño K de todos los procesos correctos (view) y se les envía la BALL.
7. Se pasa la BALL al componente de ordenación y se reinicia a conjunto vacío para la siguiente ronda.

Nótese que a diferencia de URB éste no es un funcionamiento púramente reactivo, sino que funciona por rondas con un cierta duración. Los procesos reenvían los mensajes de forma redundante, tantas veces como el TTL de los mensajes les permitan. Esto se debe a que EpTO está pensado para garantizar una alta probabilidad de entrega de mensajes para redes realmente extensas aunque los canales no sean totalmente fiables. Es por esto que los procesos reciben y reenvían los mismos mensajes muchas veces, asegurandose con gran probabilidad de que todos los procesos hayan recibido al menos una vez los mensajes, justo como se explica en el problema de los hoyos y las canicas.

Las constantes TTL, delta y K escogidos para el algoritmo dependen del tamaño y las distancias entre nodos de las redes en sí para conseguir un funcionamiento óptimo. Normalmente el número de saltos TTL suele ser igual al número de nodos existentes en la red, la duración de las rondas delta varía entre los 150 y 300 milisegundos y el tamaño de la muestra aleatoria de procesos K debe ser estrictamente mayor que la mitad de los nodos existentes.

4.1.3. Componente de ordenación

Algorithm 2: Ordering component (process p)

```
1  initially
2      received ← ∅   // map of received but not delivered events
3      delivered ← ∅                          // set of delivered events
4      lastDeliveredTs ← 0  // maximum timestamp of delivered
       events
5  procedure ORDEREVENTS(ball)
       // update TTL of received events
6      foreach event ∈ received do
7          received[event.id].ttl ← received[event.id].ttl + 1
       // update set of received events with events in the ball
8      foreach event ∈ ball do
9          if event.id ∉ delivered ∧ event.ts ≥ lastDeliveredTs
           then
10             if event.id ∈ received then
11                 if received[event.id].ttl < event.ttl then
12                     received[event.id].ttl ← event.ttl
13             else
14                 received ← received + (event.id, event)
       // collect deliverable events and determine smallest
          timestamp of non deliverable events
15     minQueuedTs ← ∞
16     deliverableEvents ← ∅
17     foreach event ∈ received do
18         if ISDELIVERABLE(event) then
19             deliverableEvents ← deliverableEvents ∪ event
20         else if minQueuedTs > event.ts then
21             minQueuedTs ← event.ts
22     foreach event ∈ deliverableEvents do
23         if event.ts > minQueuedTs then
               // ignore deliverable events with timestamp
                  greater than all non-deliverable events
24             deliverableEvents ← deliverableEvents \ event
25         else
               // event can be delivered, remove from received
                  events
26             received ← received − (event.id, event)
27     foreach event ∈ deliverableEvents sorted by (ts, srcId)
       do
28         delivered ← delivered ∪ event
29         lastDeliveredTs ← event.ts
30         DELIVER(event)        // deliver event to the application
```

Figura 9: Componente de ordenación

Una vez que los eventos llegan al componente de ordenación se preparan y ordenan para entregarlos cuando sea posible a la aplicación en orden total. Tenemos tres sencillas variables:

- received: mapa de eventos recibidos pero no entregados.
- delivered: conjunto de eventos ya entregados.
- lastDeliveredTs: tiempo en el que se creó el último evento entregado a la capa de la aplicación.

Dicho esto, el funcionamiento es el siguiente:

1. Al terminar la ronda, todos los eventos recibidos aumentan el TTL en 1.
2. Para cada evento de la BALL, se comprueba si ha sido entregado anteriormente.
3. Si no ha sido entregado y es posterior al último evento entregado, se actualiza o/e introduce en el mapa de recibidos.
4. Después se añade a una lista los eventos que son entregables por el momento (cuyo TTL es superior al límite).
5. Luego se calcula el mínimo tiempo de creación de los eventos que NO son entregables.
6. A los que inicialmente hemos seleccionado como entregables restaremos aquellos cuyo tiempo de creación sea posterior al del evento no entregable de tiempo mínimo o más antiguo, puesto si un evento anterior a ellos no es entregable, éstos tampoco lo son.
7. Finalmente, al tener los eventos que son realmente entregables, se ordenan por proceso de origen y tiempo de creación y se entregan a la capa de la aplicación.

Reloj y sincronización La ordenación de los eventos o mensajes de éste algoritmo depende en gran medida del tiempo en el que se generan. Para ello el artículo que explica el algoritmo asume que utilizando un reloj global el orden se garantiza siempre, aunque dice también que puede funcionar con un reloj lógico perfectamente, y explica cómo.

Es preciso ver la importancia del componente de ordenación y cómo funciona. El algoritmo se apoya firmemente en el hecho de poder ordenar y entregar a la aplicación los mensajes una vez que su TTL llega al límite, es decir, que ha pasado el tiempo suficiente como para que el mensaje haya llegado a todos los procesos con una gran probabilidad. Así pues, únicamente ordena y entrega aquellos mensajes que hayan llegado a dicho límite y que no sean precedidos por otros mensajes que no cumplan con las mismas condiciones. De este modo se garantiza la imposibilidad de que dos procesos diferentes entregen los mismos mensajes en distinto orden a pesar de la asincronía.

4.2. Implementación

4.2.1. Diseño

El diseño está planteado por módulos, cada uno con una función específica. La implementación ha resultado mucho más sencilla y ha permitido que el control de versiones y la evolución del proyecto hayan sido mucho más limpios y eficientes. Distinguimos los siguientes módulos con una breve descripción:

Client (Cliente) Implementa las funcionalidades básicas de un cliente. Conectarse con otros clientes, enviar y recibir mensajes. Es el punto intermedio entre el protocolo y la capa de la aplicación. Es decir, la aplicación tendrá uso de un cliente en cada máquina en la que se instale.

Process (Proceso) Implementa las funcionalidades más próximas al nivel de transporte. Conexiones con otros procesos, uso de ZMQ, recepción de mensajes serializados... También posee un identificador único y está directamente ligado al cliente. En otras palabras, un cliente siempre tiene un proceso, y un proceso siempre pertenece sólo y exclusivamente a un cliente.

Message (Mensaje) Objeto que almacena la información a nivel de la aplicación que contiene un mensaje. Implementa también las funciones básicas necesarias para serializar y deserializar los datos de forma que sean transferibles a través de la red.

Event (Evento) Objeto que almacena un mensaje y toda la información relevante del evento, como el tiempo de creación, el proceso o cliente que lo ha creado, su TTL y su identificador único.

Ball (Canica) Objeto que representa un conjunto de eventos listos para ser enviados. También es totalmente serializable.

Clock (Reloj) Simple módulo que simula el funcionamiento del reloj y nos proporciona el tiempo. Puede estar implementado de forma que sea un reloj global o uno lógico.

PSS (Peer Sample Service) Módulo que dado un conjunto de conexiones a procesos correctos y un tamaño de muestra, nos devuelve un subconjunto aleatorio de las mismas.

Dissemination Component (Componente de difusión) Implementa todas las funcionalidades de dicho componente. Recepción y envío de canicas, rondas..

Ordering Component (Componente de ordenación) Implementa todas las funcionalidades del componente de ordenación. Tan solo consta de dos funciones, ordenar y entregar los eventos posibles, y comprobar si un evento es entregable.

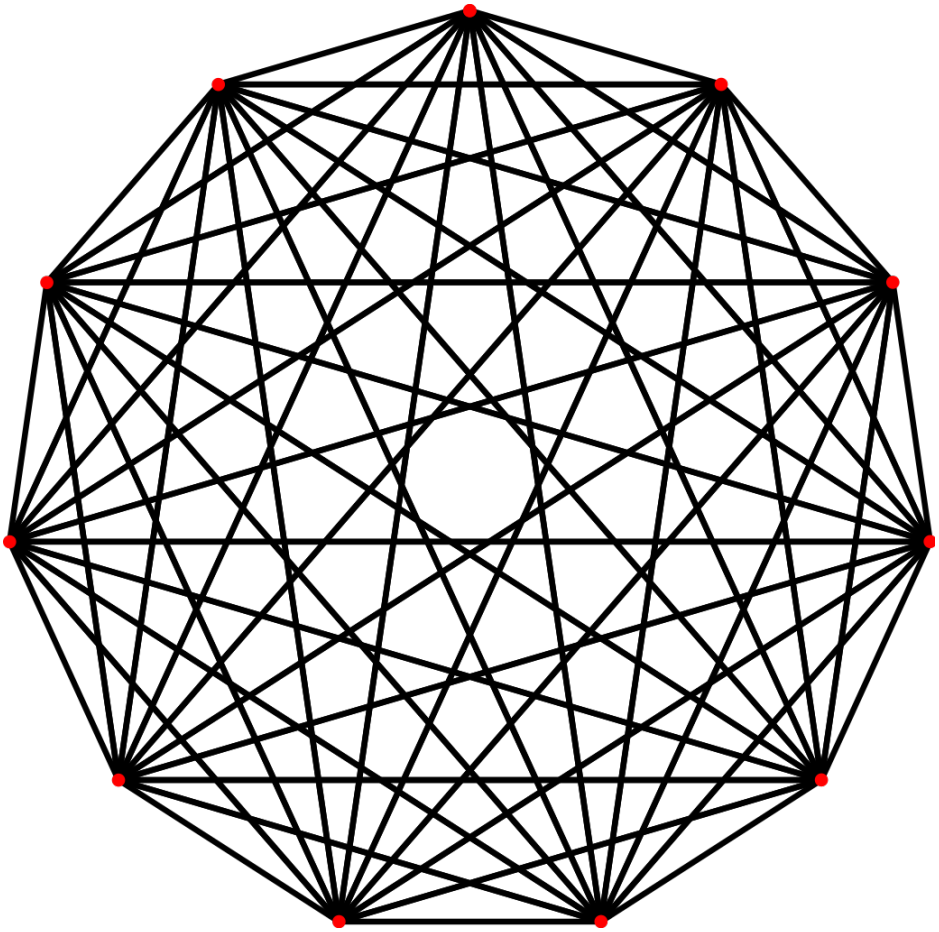

Figura 10: Arquitectura red de nodos

4.2.2. Arquitectura

La mayoría de los protocolos de orden total tienen grandes limitaciones, y la primera de ellas suele ser la escabilidad. EpTO trata de resolver este problema, ofreciendo una solución funcional para redes distribuidas con un gran número de nodos. El problema cuando existen tantos nodos es que llega un punto en el que es computacionalmente contraproducente enviar cada mensaje al resto de nodos, dado que la cantidad de estos suele ser abrumadora. Y si no se envían a todos es difícil garantizar que el mensaje llegue finalmente a todos los nodos. EpTO intenta resolver éste problema mediante el reenvío contínuo de los mensajes (con un máximo determinado número de saltos) a un subconjunto (mucho más reducido, y no computacionalmente contraproducente) aleatorio de procesos. Los procesos mismos se encargan de propagar, no mejor dicho, el mensaje a toda la red.

En la figura 10 podemos ver un pequeño ejemplo de esto. Aunque todos los nodos saben de la existencia del resto, cada vez que éstos reciban un mensaje que deban reenviar, lo harán sólo a un subconjunto mínimo de ellos, hasta que finalmente y con las rondas necesarias, el mensaje habrá sido recibido por todos los nodos (justo como explica el problema de los hoyos y las canicas).

4.3. Pruebas y rendimiento

4.3.1. Clúster

Las pruebas realizadas para éste algoritmo fueron muy parecidas a las realizadas con URB. En este caso las pruebas resultaron incluso más difíciles de realizar dado que el algoritmo está pensado para redes extensas y en el laboratorio disponía de un equipo de clústers relativamente pequeño. Una vez más, tuve que realizar una serie de tests que simulaban las condiciones ideales (número de nodos, canales no fiables, caídas aleatorias...) de forma que los resultados fueran los esperados. Así pues, realicé múltiples tests y pruebas, desde redes con tamaño pequeño (7 nodos) hasta redes de gran tamaño (70 nodos). Por supuesto, inicialmente tuve que lidiar con problemas que iban surgiendo, sobretodo según aumentaba el tamaño de la red y la frecuencia con que se enviaban los mensajes entre sus nodos. Progresivamente fui corrigiendo los errores en el código y resolviendo los problemas generados en las pruebas, hasta que obtuve una versión perfectamente funcional.

4.3.2. Resultados

Como se ha descrito inicialmente, el objetivo del algoritmo es conseguir que todos los nodos de la red reciban los mismos mensajes exáctamente en el mismo orden. Para comprobar esto, implementé una serie de tests que imprimían en ficheros por cada cliente los mensajes según estos los recibían, y posteriormente comparaba los registros de los diferentes clientes. En todas las pruebas realizadas, todos los registros coinciden al 100 %.

4.3.3. El problema de los eventos con el mismo tiempo

No obstante, al realizar las pruebas con caídas de nodos aleatorias y aumentando la frecuencia de envío de mensajes, los resultados parecían no ser los ideales. El porcentaje de coincidencia entre registros de mensajes de diferentes nodos cayó en picado. Todos los nodos recibían todos los mensajes, pero, extrañamente algunos pares mensajes que eran enviados en los mismos instantes de tiempo eran entregados en diferente orden por algunos nodos. A medida que se aumentaban las caídas y la frecuencia los resultados eran peores.

Finalmente mi tutor y yo descubrimos una anomalía en el algoritmo original del artículo. Nos dimos cuenta de que los autores no habían considerado la posibilidad de que dos mensajes distintos creados exáctamente en el mismo instante de tiempo podían ser ordenados de una forma en un nodo antes de ser entregados, y de otra diferente por otro nodo debido a la asincronía.

En otras palabras, nuestro problema sucedía cuando un nodo había recibido dos eventos creados en el mismo instante de tiempo, pero sólo uno de ellos había alcanzado el límite de saltos y por tanto era entregable. Al mismo tiempo en otro nodo y los mismos eventos, sucedía lo contrario. Al tener los dos eventos el mismo tiempo de creación, aunque un evento fuera entregable y el otro no (y como se ha explicado en el algoritmo de ordenación), el entregable no era precedido por ningún otro que no fuera entregable (puesto que un tiempo no es menor que sí mismo), así que cada nodo entregaba los mensajes en diferente orden.

Después de un tiempo buscando múltiples soluciones a este problema dimos con una solución sorprendentemente simple: teóriamente, si dos eventos son creados en el mismo instante de tiempo y uno de ellos ya ha alcanzado el límite de saltos (TTL) y por tanto es entregable, es decir, si ya ha pasado suficiente tiempo como para garantizar probabilísticamente que el evento ha llegado todos los nodos, entonces el otro evento también ha debido tener tiempo suficiente.

Después de realizar esta corrección, los resultados obtenidos fueron óptimos.

5. URBTO: URB Total Order Algorithm

5.1. Descripción

Llegamos al verdadero objetivo de este proyecto. URBTO (Uniform Reliable Broadcast Total Order Algorithm) es un algoritmo ideado con el objetivo conseguir una comunicación broadcast entre nodos de forma que se garantize la fiabilidad y la entrega de los mensajes al mismo tiempo que el orden total.

Para esto reúne diferentes partes de los algoritmos anteriormente explicados: el algoritmo mayormente se basa en la estructura de URB, construyendo una red sobre canales fiables TCP garantizando así la entrega de los mensajes. A esto se le añade una adaptación del componente de ordenación especialmente diseñado para este propósito, consiguiendo así el orden total en la entrega de mensajes.

Una vez más, se trata de un algoritmo de consenso o protocolo de orden total diseñado para sistemas distribuidos. Esta vez está pensado para redes no muy extensas, dado que la garantía de la entrega de mensajes se basa en que los canales son fiables.

5.1.1. Propiedades

Definición La especificación consiste en cuatro propiedades. Cualquier implementación debe cumplirlas para ser correcta:

- Validez: Si un proceso realiza URBTO-deliver de un mensaje m, entonces antes otro proceso ha realizado URBTO-broadcast(m).
- Integridad: Un proceso realiza URBTO-deliver de un mensaje m como mucho una vez.
- Terminación: Si un proceso correcto realiza URBTO-broadcast de un mensaje m, o si un proceso realiza URBTO-deliver de un mensaje m, entonces todos los procesos correctos realizan URBTO-deliver(m).
- Orden Total: Para dos mensajes a y b, si un proceso correcto realiza URBTO-deliver de a y posteriormente URBTO-deliver de b, entonces todos los procesos correctos lo entregan en el mismo orden.

La especificación es muy similar a la vista en URB. La propiedad de validez relaciona el envío con la entrega de mensajes (garantiza la no alteración de los mensajes), la de integridad garantiza la no duplicidad, y la de terminación garantiza que un mensaje enviado por un proceso correcto es al menos recibido por todos los procesos correctos. A esto se le añade la cuarta propiedad: el orden total garantiza que todos los procesos correctos entregen los mensajes exáctamente en el mismo orden.

Una vez más, de las anteriores propiedades se puede concluir que (1) los procesos correctos reciben exáctamente el mismo conjunto de mensajes, (2) que este conjunto incluye todos los mensajes emitidos por todos los procesos correctos, (3) que todo proceso no correcto recibe un subcojunto de los mensajes recibidos por un proceso correcto. (aunque dos procesos no correctos distintos pueden recibir dos subconjuntos de mensajes distintos), y (4) que los mensajes recibidos se entregan exáctamente en el mismo orden por todos los nodos correctos. Es importante ver que un mensaje emitido por un proceso no correcto puede ser o no ser entregado. No es posible exigir un requerimiento a un mensaje si el emisor puede fallar antes de que el mensaje haya sido enviado a un proceso correcto. La entrega de los mensajes enviados por procesos no correctos dependen de la ejecución.

5.1.2. Mensajes en desorden

Debido a una extrema asincronía y aunque es muy poco probable, es posible que algunos mensajes no tengan el tiempo suficiente para ser ordenados y entregados en orden total. No obstante, estos mensajes que llegan en «desorden» quedan marcados y se conoce sus circunstancias antes de ser entregados, y por tanto pueden ser tratados de forma distinta o simplemente no ser entregados. Es decir, no alteran el orden total del resto de mensajes: todos los nodos entregarán el mismo conjunto de mensajes exáctamente en el mismo orden.

5.2. Implementación

5.2.1. Algoritmo

Algorithm 1 URBTO

```
1:  initially
2:      p ← ...
3:      peers ← ...
4:      received ← ∅
5:      lastDeliveredTs ← 0
6:      lastDeliveredProcessEvents ← ∅
7:      lastDisorderDeliveredProcessEvents ← ∅
8:      N ← ...
9:      F ← ...
10:
11: procedure URBTOBROADCAST(event)
12:     event.id ← p.id + NEWID()
13:     event.ts ← GETCLOCK()
14:     event.sourceId ← p.id
15:     RECIEVEHANDLER(event, p.id)
16:
17: upon receive EVENT(event, senderId)
18:     RECIEVEHANDLER(event, senderId)
19:
20: procedure RECEIVEHANDLER(event, senderId)
21:     if event ∈ received then
22:         received[event.id].nor++
23:         if ISDELIVERABLE(received[event.id]) then
24:             ORDERANDDELIVEREVENTS()
25:     else if event ∉ lastDeliveredProcessEvents and event ∉ lastDisorderDeliveredProcessEvents then
26:         received ← received ∪ (event.id, event)
27:         received[event.id].nor++
28:         foreach q ∈ peers if p ≠ q do
29:             q.send(event)
30:
31:     UPDATECLOCK(event)                          ▷ Sólo necesario si se usa reloj lógico
```

```
32: procedure ORDERANDDELIVEREVENTS( )
33:     minTsOfNonDeliverable ← ∞
34:     deliverableEvents ← ∅
35:     maxNor ← Ø
36:
37:     foreach (a, b) ∈ received if a ≠ b do
38:         if a.ts = b.ts then
39:             maxNor ← MAX(a.nor, b.nor)
40:             a.nor ← maxNor
41:             b.nor ← maxNor
42:
43:     foreach event ∈ received do
44:         if event.ts < lastDeliveredTs then
45:             received ← received \ event
46:             lastDisorderDeliveredProcessEvents[event.sourceId] ← event
47:             DISORDERDELIVER(event)
48:         else if ISDELIVERABLE(event) then
49:             deliverableEvents ← deliverableEvents ∪ event
50:         else if event.ts < minTsOfNonDeliverable then
51:             minTsOfNonDeliverable ← event.ts
52:
53:     foreach event ∈ deliverableEvents do
54:         if event.ts > minTsOfNonDeliverable then
55:             deliverableEvents ← deliverableEvents \ event
56:         else
57:             received ← received \ event
58:
59:     foreach event ∈ deliverableEvents sorted by (ts, sourceId) do
60:         lastDeliveredProcessEvents[event.sourceId] ← event
61:         lastDeliveredTs ← event.ts
62:         DELIVER(event)
63:
64: function ISDELIVERABLE(event)
65:     return event.nor ≥ N − F
```

Variables globales El algoritmo utiliza un total de ocho variables globales para su fucionamiento. Esta es una breve descripción de cada una:

- p (proceso): esta es la variable que asocia un proceso con el módulo del algoritmo. Da acceso a todas las funciones de más bajo nivel (sockets, variables de nodejs...).
- peers (conexiones): conjunto de sockets o conexiones a todos los nodos. En NodeJS son un conjunto de componentes Dealer.
- received (recibidos): es un mapa (id evento, evento) que guarda el conjunto de eventos recibidos por un nodo.
- lastDeliveredTs: guarda el instante de tiempo en el que se entregó el último mensaje a la capa de la aplicación.
- lastDeliveredProcessEvents: es un mapa (id proceso, evento) que guarda el último evento entregado a la aplicación por cada proceso de origen.
- lastDisorderDeliverProcessEvents: igual que el anterior, aunque para los eventos entregados en desorden.
- N: número de nodos existentes en la red.
- F: número de fallos o nodos caídos permitidos para que el algoritmo funcione correctamente.

Preparación y envío de eventos Cuando la capa de la aplicación (el cliente) recibe un mensaje, ésta la envía al algoritmo. En concreto la función URBTO-broadcast se encarga de esto. El mensaje es encapusulado en un evento junto con otros atributos que serán necesarios para la comunicación entre nodos y desenlace del algoritmo. En la función se le asigna un ID único al evento formado a partir del ID del proceso (también único) y un número autoincremental. Después de esto, se establece el tiempo en el que el evento ha sido creado (momento en el que se encapsula el mensaje) y el proceso donde se ha originado.

Una vez hecho esto el propio proceso se lo envía a sí mismo, para recibirlo por primera vez, reenviarlo, procesarlo y posteriormente entregarlo.

Recepción y tratado de eventos Cuando un proceso captura un evento, lo captura y lo envía al manejador: RECEIVE-HANDLER. Éste funciona de la siguiente forma: Primero, se comprueba si el evento ya ha sido recibido por el proceso con anterioridad (si existe en el mapa de recibidos). Si ya ha sido recibido, entonces se aumenta su número de recepciones y se comprueba si ha sido recibido suficientes veces como para ser entregable.

Por otro lado, si es la primera vez que se recibe el evento en cuestión (no está en el mapa de recibidos), y no es el último evento entregado con el mismo origen que el del mismo evento [1], entonces se añade al mapa de recibidos, se aumenta su número de recepciones en uno y se reenvía a todos los nodos.

[1] El único próposito de los mapas que guardan los últimos eventos entregados para cada proceso de origen es descartar las recepciones de eventos que ya habían alcanzado suficientes recepciones para ser entregables y que por tanto se entregaron y borraron de la lista de recibidos. Gracias a esto, no necesitamos almacenar indefinidamente los eventos en recibidos para que no se produzcan duplicados sino que los últimos eventos entregados van reemplazando los anteriores en los mapas descritos.

Ordenación y entrega de eventos La funcion OrderAndDeliverEvents se encarga de esta parte. Ésta es la adaptación del componente de ordenación de EpTO de la que hemos hablado anteriormente. Para funcionar necesitaremos las siguientes variables locales:

- minTsOfNonDeliverable: es el instante de tiempo del evento más antiguo en el tiempo que NO es entregable.
- deliverableEvents: conjunto de eventos que son entregables. Aquí se preparan los eventos antes de ser ordenados y entregados.
- maxNor: variable local que se usa para calcular el máximo número de recepciones entre dos eventos.

Dicho esto, el funcionamiento es el siguiente:

1. Primero, y al igual que en la corrección realizada a EpTO, se considera que si dos eventos son creados en el mismo instante de tiempo aunque uno es entregable y el otro no, los dos han tenido el tiempo suficiente para haber sido recibido al menos una vez por todos los nodos y por tanto se les asigna el mismo número de recepciones a todos los pares diferentes de eventos que compartan el mismo instante de tiempo de creación.
2. Después se comprueba para cada evento recibido si es anterior al último evento entregado. Si es así, el evento ha llegado en desorden debido a la asincronía y por tanto es entregado en desorden.
3. En cambio, si es posterior al último entregado y es entregable se añade al conjunto de eventos entregables.
4. Si no es entregable y si su tiempo es anterior al del evento no entregable más antiguo encontrado se actualiza el mínimo, buscando así el evento no entregable más antiguo.
5. Una vez que tenemos el conjunto de eventos que primeramente habíamos considerado como «entregables», tenemos que restar a estos aquellos que sean posteriores al evento de tiempo mínimo no entregable, puesto que si un evento anterior que es en el tiempo a otro NO es entregable, entonces éste último tampoco lo es todavía.
6. Finalmente y al tener el conjunto de eventos realmente entregables, los ordenamos por tiempo y proceso de origen y se entregan a la capa de la aplicación.

Por qué funciona URBTO Llegados a este punto, vamos a reflexionar sobre el funcionamiento de los anteriores algoritmos para poder ver las diferencias y entender cómo funciona realmente y por qué. Por un lado, URB es un algoritmo que reenvía los mensajes según los recibe, e inmediatamente después entrega los mensajes a la capa de la aplicación dado que no exige ningún tipo de orden. Tiene un funcionamiento púramente reactivo y garantiza la entrega de mensajes gracias a que trabaja sobre canales que son fiables. EpTO en cambio, sacrifica la garantía de entrega de mensajes por un rendimiento altamente escalable (redes muy extensas) pero con garantía de entrega de mensajes probabilística. Además su funcionamiento no es púramente reactivo sino que consta de rondas que permiten que pase el tiempo suficiente para que los mensajes sean reenviados y extendidos por la red el tiempo suficiente como para que los mensajes puedan ser recibidos por todos los nodos y entregados en orden total. Para esto EpTO utiliza el TTL para contar el número de saltos de los eventos (igual que los paquetes TCP-IP) como forma de evaluar cúando un evento puede ser entregable.

En URBTO, al estar construido sobre canales fiables, no existe la necesidad de utilizar rondas ni TTL para diferenciar cúando un evento es entregable o no. En cambio, se utiliza el NOR (número de veces que un proceso ha recibido un evento). Como la red no es muy extensa y los eventos se reenvían a todos los nodos, cada nodo puede recibir el mismo evento hasta un total de N-1 veces. Esto quiere decir que si un proceso recibe un evento al menos N - F veces entonces una mayoría de nodos correctos también lo han recibido y por tanto es entregable. De esta forma, podemos afirmar que URBTO no sólo garantiza la entrega de mensajes sino que también garantiza el orden total de los mismos.

La excepción de los mensajes en desorden Como se ha dicho antes, aunque todos los nodos correctos entregen los mismos mensajes exáctamente en el mismo orden, es posible que algunos mensajes simplemente no se entregen en orden como los demás sino que se traten de forma distinta. Esto es debido a la incontrolable asincronía, y es altamente improbable pero inevitable. Depende totalmente de la ejecución en cuestión. Sin embargo, el algoritmo captura estos eventos y da la oportunidad de tratarlos como la aplicación desee, indicando que son entregados en desorden.

Se ha estudiado en profundidad las condiciones que dan lugar a la entrega de mensajes en desorden, y se ha llegado a las siguientes conclusiones:

Teorema 1 Si un mismo proceso correcto p envía un mensaje a y posteriormente un mensaje b, entonces es imposible que los mensajes no se entregen en orden.

Demostración Supongamos que tenemos un proceso p que realiza URBTO-broadcast de un mensaje a. Entonces el proceso p prepara el evento, se lo envía a sí mismo e inmediatamente lo reenvía al resto de nodos. Después realiza URBTO-broadcast de un mensaje b y sucede lo mismo. Dado que los canales por los que se comunican los nodos son fiables y utilizan el protocolo TCP, y como el mensaje b se enviará por los mismos canales por los que se envió a, cada proceso recibirá primero a y después b, y lo reenviarán en el mismo orden al resto de nodos puesto que TCP es FIFO. Siguiendo la propiedad de terminación, es imposible b sea entregado antes que a.

Teorema 2 Si un proceso correcto p envía un mensaje a y posteriormente un proceso distinto q envía un mensaje b, entonces a se entregará en desorden sí y sólo si ningún proceso correcto recibe a directamente desde p antes de que b llegue a una mayoría de nodos correctos y sea entregado.

Demostración DE ESTO NO ESTOY SEGURO. COMPROBAR

5.2.2. Arquitectura

La arquitectura de la red es exáctamente la misma que la de EpTO. Se puede apreciar en la figura 10. Sin embargo, la figura se ajusta más a la red de URBTO dado que se trata de una red más bien pequeña. A diferencia de EpTO, los mensajes son reenviados a todos los nodos, así que es importante que todos los nodos estén conectados entre sí, es decir, que contemos con un grafo completo.

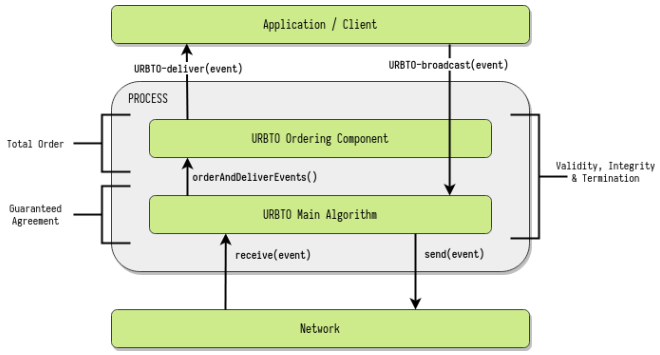

Figura 11: Arquitectura URBTO

En cuanto a la arquitectura del sistema, está diseñada por capas. Desde la aplicación o cliente hasta la propia red de transporte, justo como la figura 11 muestra. El ciclo de vida del sistema es el siguiente: Primero, el cliente o aplicación recibe un mensaje. El mensaje es tratado por el algoritmo principal, preparado y encapsulado como evento, luego se serializa y se envía a los nodos correspondientes a través de la red (sockets TCP-IP).

Cuando llega a los otros extremos de los sockets son deserializados y tratados por el algoritmo principal de nuevo, reconociendo así su origen, mensaje, etc. Posteriormente los eventos se disponen a ser ordenados por el componente de ordenación, y una vez terminado, son desencapsulados en orden los eventos en mensajes que se entregan a las aplicaciones de otros clientes.

5.2.3. Diseño

Una vez más, el diseño es muy parecido al de los algoritmos anteriores. Se distinguen dos partes, la interna y la externa. La externa consta de la aplicación o cliente y el mensaje, y es la parte que interesa modificar dependiendo del propósito de la aplicación que se decida construir por encima de este protocolo de orden total. La interna por otro lado consta de un conjunto de módulos de más bajo nivel centrados en URBTO y su implementación en NodeJS.

Client (Cliente) Implementa las funcionalidades básicas de un cliente. Conectarse con otros clientes, enviar y recibir mensajes. Es el punto intermedio entre el protocolo y la capa de la aplicación. Es decir, la aplicación tendrá uso de un cliente en cada máquina en la que se instale.

Process (Proceso) Implementa las funcionalidades más próximas al nivel de transporte. Conexiones con otros procesos, uso de ZMQ, recepción de mensajes serializados... También posee un identificador único y está directamente ligado al cliente. En otras palabras, un cliente siempre tiene un proceso, y un proceso siempre pertenece sólo y exclusivamente a un cliente.

Message (Mensaje) Objeto que almacena la información a nivel de la aplicación que contiene un mensaje. Implementa también las funciones básicas necesarias para serializar y deserializar los datos de forma que sean transferibles a través de la red.

Event (Evento) Objeto que almacena un mensaje y toda la información relevante del evento, como el tiempo de creación, el proceso o cliente que lo ha creado, su NOR y su identificador único.

Clock (Reloj) Simple módulo que simula el funcionamiento del reloj y nos proporciona el tiempo. Puede estar implementado de forma que sea un reloj global o uno lógico.

URBTO Módulo principal. Implementa la lógica de todo el algoritmo, desde la encapsulación y preparación de los eventos a enviar hasta la recepción y tratado de los mismos. Las variables globales, funciones handler y de ordenación todas tienen lugar en éste módulo.

5.3. Pruebas y rendimiento

5.3.1. Clúster

Las pruebas se realizaron en el mismo clúster de máquinas y mismas condiciones que con los algoritmos anteriores. Se realizaron pruebas de diferentes tamaños (7, 35, 70 nodos). Esta vez las pruebas se realizaron en las condiciones ideales para el algoritmo, dado que está pensado para redes no muy extensas sobre canales fiables. A diferencia del resto de pruebas, estas son las que más problemas dieron con diferencia. A menudo existian errores en el código que daban lugar a bucles de envíos de mensajes realmente bloqueantes y que con mucha frecuencia acaban tirando las máquinas del clúster. Después de mucho tiempo realizando pruebas por fin dí con una version perfectamente funcional.

5.3.2. Resultados

Hay que recordar que en este caso el objetivo es garantizar la entrega de mensajes al mismo tiempo que el orden total. Para comprobar esto tan solo se debía comparar los registros de entrega de mensajes de los diferentes nodos, los cuales eran guardados en ficheros.

Se implementaron una serie de tests en los que los nodos de enviaban mensajes generados automáticamente de forma aleatoria cada cierto tiempo. En todas las pruebas realizadas, todos los registros coinciden al 100 %. No obstante, descubrimos que según se disminuye el tiempo entre mensajes aleatorios llega un momento en el que el envío de mensajes es más rapido de lo que el sistema puede procesar y entregar, y por tanto, el sistema acaba colapsando. Este límite a la hora de escoger la frecuencia de envío de mensajes es algo difuso, puesto que depende tanto del tamaño de la red como de las propias máquinas en sí.

5.3.3. El problema de los eventos entregados en «desorden»

Aunque los resultados fueron los esperados, sabíamos que teóricamente la llegada de algunos mensajes en desorden era posible. Dado que las pruebas se realizaron de forma local y las máquinas estaban en muy buenas condiciones era imposible provocar una situación en lo que esto pudiera suceder, así que se se implementaron una serie de tests en los que se simulaba una extrema asincronía entre varios nodos del sistema para ver las reacciones de éste.

El resultado fue que, efectivamente, algunos mensajes en condiciones excepcionales no llegaban a tiempo para ser entregados en orden total como los demás. Creemos que esto no se puede remediar, aunque todavía falta más investigación al respecto.

6. Conclusiones

Bueno pues nada sabes

6.0.1. Recursos utilizados

Para el desarrollo del proyecto he utilizado diversos recursos. Para el control versiones he utilizado git, y para la implementación Visual Studio Code. Ambos entornos relativamente nuevos para mi. También he utilizado npm (node package manager) para el despliegue del proyecto y he aprendido LaTeX para redactar esta misma memoria. Además el proyecto entero está programado en Typescript. Este lenguaje ofrece la oportunidad de desarrollar de forma más clara, dado que es un lenguaje tipado, así que el código será más fácilmente interpretado.

Referencias

[1] Miguel Matos, Hugues Mercier, Pascal Felber, Rui Oliveira, José Pereira. EpTO: An Epidemic Total Order Algorithm for Large-Scale Distributed Systems. *2014*.
`https://haslab.uminho.pt/mmatos/files/p100-matos.pdf`

[2] Miguel Ângelo Marques de Matos. Epidemic Algorithms for Large Scale Data Dissemination. Programa de Doutoramento em Informática das Universidades do Minho, de Aveiro e do Porto. *Julio de 2013*.
`https://haslab.uminho.pt/mmatos/files/miguel_angelo_marques_de_matos.pdf`

[3] Jim. R Wilson. Node.js the Right Way. Practical, Server-Side Javascript That Scales.

[4] ZMQ. An open-source universal messaging library.
`https://zeromq.org/`