upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Universidad Pública de Navarra

Memoria proyecto

Beca de Colaboración

David González Portillo

Tutor:
Jose Ramón González de Menvidil

24 de febrero de 2020

Índice

1. Introducción **1**
1.1. Qué es EpTO . . . 1
1.2. ¿Para qué sirve? . . . 1
1.3. Porqué en NodeJS . . . 1

2. Objetivos **2**
2.1. Generales . . . 2
2.2. Adicionales . . . 2

3. EpTO: El proyecto **3**
3.1. Primera fase: Aprendizaje . . . 3
3.1.1. NodeJS y ZMQ . . . 3
3.1.2. Comprensión EpTO . . . 3
3.2. Segunda fase: Implementación . . . 6
3.2.1. Diseño . . . 6
3.2.2. Arquitectura . . . 7
3.2.3. Recursos utilizados . . . 8
3.3. Tercera fase: Testeo . . . 8
3.3.1. Clustering . . . 8
3.3.2. Resultados . . . 8

4. Conclusión **9**

Referencias **10**

1. Introducción

1.1. Qué es 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.

1.2. ¿Para qué sirve?

Sus utilidadades son muchas. Dado que es un protocolo, puede ser implementado por debajo de muchas aplicaciones que funcionen de forma distribuida y que tengan una gran cantidad de nodos y clientes, aplicaciones que necesiten estar siempre disponibles para usarse y que toleren gran cantidad de fallos. En definitiva, EpTO se reduce a la comunicación de los nodos de una red distribuida, de forma que la aplicación o utilidad que se le de por encima, es abstraída de su funcionamiento.

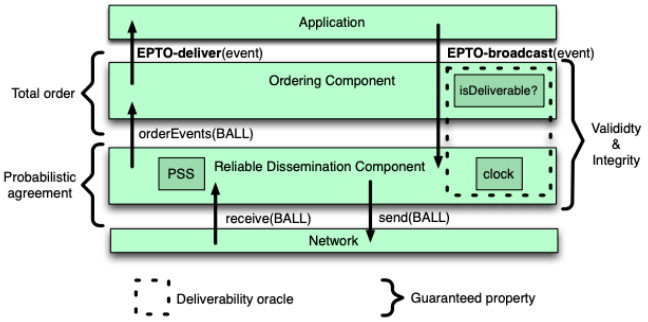

Figura 1: Arquitectura EpTO

1.3. Porqué 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.

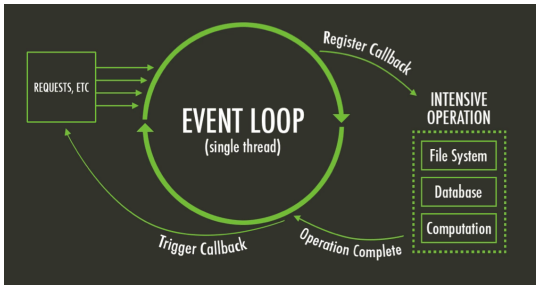

Figura 2: Ciclo NodeJS

2. Objetivos

2.1. Generales

El objetivo de este proyecto es simple: Conseguir una implementación correcta y perfectamente funcional del algoritmo EpTO en Node.js. Se han realizado implementaciones de este algoritmo en otros lenguajes pero no en este. Para ello es necesario un profundo conocimiento del lenguaje y de entornos asíncronos en general, del algoritmo en sí y de todas las librerías necesarias para su implementación. Además será necesario la implementación de un programa de testeo especializado para un sistema distribuido, que permita obtener resultados de las ejecuciones de gran escala.

2.2. Adicionales

Con el desarrollo de este proyecto también se espera la obtención de nuevas capacidades como el conocimiento y aprendizaje de diversos entornos como son Typescript, VS code, npm, ZMQ... También ganar destreza con LaTeX para la documentación de éste y de otros futuros proyectos, y, por supuesto, una buena explicación del código mediante comentarios para que este trabajo sea comprensible, y reutilizable para ser impartido en el futuro a otras personas en la universidad.

3. EpTO: El proyecto

3.1. Primera fase: Aprendizaje

3.1.1. NodeJS y ZMQ

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 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 3: Node.js y ZMQ

3.1.2. Comprensión EpTO

Llegamos a la clave del proyecto, el algoritmo. Después de recordar Node y ZMQ, dediqué las proximas semanas a estudiar el algoritmo, sirviéndome del artículo en el que había sido propuesto [1]. En éste, se explica breve y claramente cada uno de los componentes del protocolo, cómo se demuestra su correcto funcionamiento, y cómo se garanizan las cuatro propiedades propuestas: integridad, validez, orden total y entrega probabilística.

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 4: Propiedades EpTO

La idea 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 sel mensaje no ha 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 una vez por lo menos sea muy alta.

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

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

El funcionamiento es simple: cuando el proceso recibe un evento nuevo desde la aplicación, lo añade a la próxima BALL. Cuando se recibe una BALL de otro proceso, se analizan todos sus eventos de forma que si su TTL ya es muy alto se ignoran, y si no se actualizan o/e introducen en la próxima BALL para ser reenviados. Cuando comienza la próxima ronda, se aumenta en 1 el TTL de todos los eventos pertenecientes a la próxima BALL, se selecciona un subconjunto de tamaño K de todos los procesos correctos, y se les envía la BALL. Una vez hecho, se pasa la BALL al componente de ordenación y se reinicia a conjunto vacío para la próxima ronda.

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 5: Componente de difusión

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: recieved es el mapa de eventos recibidos pero no entregados, delivered es el conjunto de eventos ya entregados y lastDeliveredTs es el tiempo en el que se creo el último evento entregado. Dicho esto, el funcionamiento es el siguiente: Para empezar, todos los eventos recibidos aumentan el TTL en 1, dado que acaba de terminar la ronda. Después, para cada evento de la BALL, si no ha sido entregado ya y es posterior al último evento entregado, se actualiza y/e introduce en el mapa de recibidos.

Después se añade a una lista los eventos que son entregables por el momento (cuyo TTL es superior al límite), y se calcula el mínimo tiempo de creación de los eventos que NO lo son. A estos primeros entregables restaremos aquellos cuyo tiempo de creación sea posterior al mínimo de los que no son entregables, puesto que si un evento anterior a ellos no es entregable, éstos tampoco lo son. Finalmente, ya tenemos los eventos que son realmente entregables a la aplicación, así que se ordenan por proceso de origen y tiempo de creación y se entregan.

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 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.

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 6: Componente de ordenación

3.2. Segunda fase: Implementación

3.2.1. 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 y 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, utilización 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 desserializar 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.

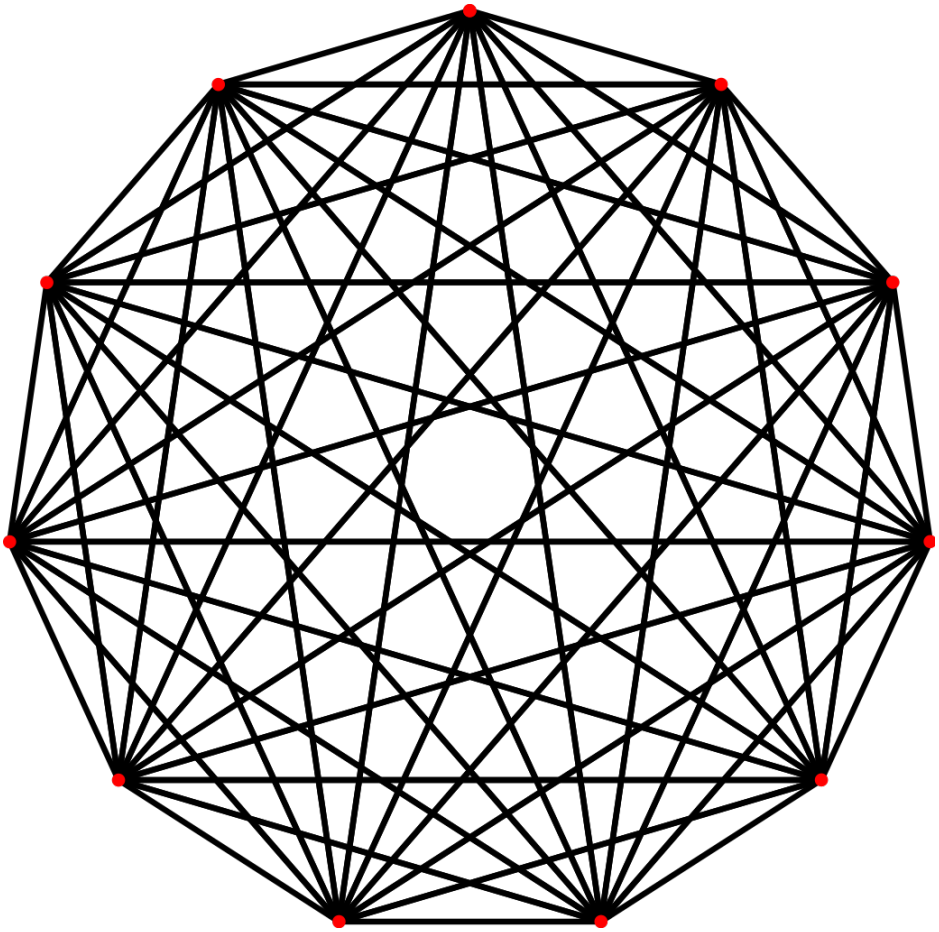

Figura 7: Arquitectura red de nodos

3.2.2. 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 llega finalmente a todos los nodos. EpTO intenta resolver éste problema mediante el reenvío contínuo de los mensajes con un máximo determinado número de saltos siempre 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 7 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).

3.2.3. 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, a diferencia del año pasado, que desarrollé en Javascript puro. Typescript ofrece la oportunidad de desarrollar de forma más clara, dado que es un lenguaje tipado, así que el código será más fácilmente interpretado.

3.3. Tercera fase: Testeo

3.3.1. Clustering

En el laboratorio de la universidad disponía de un sistema de clústeres Proxmox con siete máquinas distintas, en las cuales creé una serie 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, 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.

3.3.2. 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 %.

Hay que tener en cuenta, que dado que las pruebas se realizaron en un entorno más bien cerrado, un clúster con todas las máquinas en la misma red, con una buena comunicación entre sus máquinas, es muy improbable la pérdida de mensajes. En un sistema más grande, con más asincronía, sería de esperar la falta de algunos mensajes (cosa totalmente normal). Los resultados para este proyecto son óptimos, al menos desde punto de vista.

4. Conclusión

Una vez más, al igual que el año pasado, considero esta experiencia como una oportunidad de inversión en mí mismo, además de posibilidad de ayudar en proyectos a la universidad, en la cual he aprendido muchísimo. Esto me acerca y prepara cada vez más para la salida, ya muy cercana, de la universidad. Además, estos dos últimos años, me ha permitido aprender y desarrollarme en un círculo que no se enseña en la carrera, que es muy interesante y me puede ser de gran ayuda fuera, y se trata de sistemas distribuidos.

He de decir que he tenido la suerte de permanecer los dos años en el mismo departamento, con el mismo profesor y mismos compañeros, lo que ha hecho muy cómodo a la larga mi estancia. Gracias a esto, todo lo relacionado a este año ha ido más rápido y he tenido menos problemas, de forma que he podido abarcar un proyecto más grande y complejo.

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/`