E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Protocolo de Difusión de mensajes fiable y uniforme con garantía parcial de Orden Total

Grado en Ingeniería Informática

Trabajo Fin de Grado

David González Portillo

Dr. Jose Ramón González de Mendívil Moreno

Pamplona, 24 de Agosto de 2020

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

En este trabajo presentamos el diseño y la implementación de un algoritmo de difusión de mensajes tolerante a fallos con las propiedades de fiabilidad, uniformidad, acuerdo de terminación y orden total (débil). El sistema distribuido considerado para la corrección del algoritmo es un sistema asíncrono con canales fiables en los que se fija el número mínimo de procesos correctos. Al ser un sistema asíncrono no se garantiza la propiedad fuerte de orden total, que por el resultado de imposibilidad FLP no es posible obtener en este tipo de sistemas. El algoritmo, por tanto, realiza el mayor esfuerzo para obtener una garantía parcial del orden total en la entrega de los mensajes. No obstante, la implementación realizada en NodeJS sobre un grupo de contenedores Docker en un sistema tipo 'cluster' de máquinas asegura que un gran porcentaje de mensajes son entregados en orden total por todos los procesos, lo que hace que sea un algoritmo de utilidad en estos entornos. Además, por la uniformidad y fiabilidad, todos los mensajes enviados por procesos correctos son entregados a la capa de la aplicación. Como desventaja, el precio a pagar por obtener orden total en la mayoría de los procesos es una mayor latencia en la entrega de los mensajes.

Palabras clave: sistemas distribuidos, difusión de mensajes, protocolos de consenso y orden total, lenguajes asíncronos, tolerante a fallos.

Abstract

In this project, we present the design and implementation of a fault tolerant broadcast algorithm that guarantees reliability, uniformity, termination agreement and (weak) total order. Algorithm correctness is provided for an asynchronous system based on reliable channels in where the minimum of correct processes is fixed a priori. As the system is asynchronous it does not guarantee the strong total order property, as it is shown by the FLP impossibility result. Strong total order property cannot be obtained in this kind of systems. Therefore, the proposed algorithm does a best effort to obtain a partial guarantee of the total order property. The algorithm is implemented using NodeJS and it is deployed on a cluster using a group of Docker containers. In this environment, the algorithm delivers an high percentage of messages in total order by all the correct processes. This makes the algorithm very useful in these types of environments. By the uniformity and reliability properties, it also guarantees that every message sent by any correct process is delivered to the application layer. As a disadvantage, total order in the majority of processes has a price to pay: an higher latency on delivering messages.

Keywords: distributed systems, message dissemination, total order & consensus protocols, asynchronous languages, fault tolerant.

Índice

1. Introducción

Un gran número de aplicaciones distribuidas requieren que los procesos participantes envíen mensajes al resto de los procesos para informarles de las operaciones que se deben realizar o bien para transmitir información que debe ser compartida por todos los procesos en la computación. Esta comunicación entre los procesos se realiza por medio de una abstracción denominada difusión (broadcast). Básicamente, cuando un proceso realiza la difusión de un mensaje m invocando la primitiva $Broadcast(m)$, esta difusión transmite el mensaje a todos los particiantes en el broadcast, incluyendo al mismo proceso enviador. De la misma forma que a la comunicación punto a punto entre dos procesos se le exige ciertas propiedades es conveniente que la difusión también presente ciertas garantías. La garanía más común es la *fiabilidad*: (1) los mensajes no se crean o destruyen espontáneamente (*validez*), (2) se entregan como mucho una vez (*integridad*), y (3) en caso de que los procesos sean correctos, un mensaje enviado es eventualmente entregado por todos los procesos correctos (*acuerdo en la entrega*).

Diseñar protocolos de difusión que cumplan este tipo de propiedades es una tarea bastante simple en el caso de que los sistemas distribuidos no presenten fallos en la red de comunicaciones o en los procesos que ejecutan los protocolos. Otra cuestión diferente es diseñar estos protocolos en presencia de fallos.

Una de las características que debe disponer un servicio o aplicación distribuida es que presente un cierto grado de tolerancia a fallos, ya que en un sistema distribuido compuesto, en general, por nodos geográficamente dispersos conectados por una red de comunicaciones, es bastante probable que alguno de sus componentes falle. La técnica más común para garantizar la tolerancia a fallos consiste en la replicación de los componentes de la aplicación [1]. Idealmente, al disponer de varias copias para el servicio distribuido, el fallo de una de ellas no compromete la disponibilidad del servicio al tener otras copias disponibles y por otro lado, al disponer de varias copias, los clientes pueden distribuir sus operaciones entre cualquiera de ellas. Así que, por una parte, la replicación nos permite tener un cierto grado de tolerancia a fallos y, por otra parte, mejorar la productividad al disponer de una mayor capacidad para aceptar operaciones de los clientes concurrentemente.

La existencia de operaciones concurrentes ejecutadas en diferentes partes del servicio replicado origina la necesidad de gestionar la consistencia de las operaciones, y por tanto, de los datos que son modificados por dichas operaciones. Idealmente, si los objetos que se encuentran replicados admiten una especificación secuencial (por ejemplo, ficheros, colas, listas, pilas, tablas clave-valor, registros de lectura/escritura), es decir, su comportamiento se describe correctamente cuando las operaciones sobre el objeto se ejecutan secuencialmente una seguida de otra, entonces en el servicio replicado de dicho objeto desearíamos el mismo comportamiento. Esto se conoce como atomicidad o linearización [2]: un servicio de un objeto replicado es atómico cuando cualquier ejecución concurrente de operaciones sobre el servicio es equivalente a una ejecución sobre el objeto que se corresponde con su especificación secuencial y respetando el orden de precedencia de las operaciones en la ejecución concurrente [2][16].

La cuestión es cómo conseguimos una implementación de un servicio para un sistema replicado de objetos distribuido que respete la atomicidad y sea además tolerante a fallos. Este problema se ha estudiado y resuelto en la literatura de sistemas distribuidos ya que tiene una gran importancia práctica [1]. Un tipo de solución se basa en disponer de un servicio de difusión de mensajes fiable con garantías de entrega de mensajes en orden total [3].

El problema de disponer ciertas propiedades de ordenación en la entrega de los mensajes en sistemas de comunicación basados en la difusión de mensajes ha sido uno de los problemas fundamentales en sistemas distribuidos que ha sido ampliamente estudiado durante las últimas decadas [4]. Entre las propiedades de orden que se pueden exigir encontramos: difusión con orden FIFO, difusión con orden Causal (CO), difusión con Orden Total (TO), difusión FIFO-TO, y difusión CO-TO. Las citadas son las más comunes en la literatura. Si disponemos de un sistema de comunicación con difusión que sea fiable y satisfaga la entrega de mensajes en TO en un sistema distribuido con fallos entonces no es muy complicado construir un servicio tolerante a fallos de sistema de replicación de objetos en dicho sistema [5].

Hasta el momento hemos empleado el término sistema distribuido de una forma poco específica y llegado a este punto, es necesario fijar algunas cuestiones importantes. Cuando estamos hablando de un sistema distribuido estamos pensando en un grupo de nodos de procesamiento conectados por alguna red de comunicación (con ciertas garantías). Asumimos que el subsistema de comunicaciones que hay por debajo está formado por canales punto a punto fiables (o que el protocolo que permite enviar mensajes punto a punto entre las máquinas es fiable). Los fallos que consideramos sólo son fallos de los procesos que se ejecutan en los nodos de procesamiento (fallos fail-stop)[1]. También vamos a considerar que el sistema distribuido es en principio asíncrono, es decir, no se conocen los límites de tiempo en las latencias de los mensajes, en los procesamientos de los mismos, o en la duración de las operaciones. En otras palabras, no hay conocimiento sobre el tiempo que pueda aprovecharse para el diseño de los servicios. Utilizando la notación de Michael Raynal [16] vamos a denominar al sistema anterior por $\mathcal{AS}_{n,f}[\emptyset]$ donde n es el número de procesos, f es el número de procesos que pueden fallar, y $[\emptyset]$ indica que no hay restricciones adicionales. En este tipo de sistemas no se puede distinguir entre un nodo que ha fallado de un nodo que simplemente va muy lento o que sus mensajes se retrasan un tiempo no definido. Esto es debido a que en un sistema distribuido asíncrono no hay cotas superiores conocidas en las computaciones de los procesos o en las latencias de los canales de comunicación. Aún así podemos decir que los procesos correctos (aquellos que no fallan) progresan en la ejecución (tarde o temprano hacen 'algo').

En un protocolo de difusión que debe garantizar orden total, los procesos participantes deben llegar a un acuerdo en el orden en el que deben entregar los mensajes. Esto es debido a que justamente la propiedad de orden total establece que si un proceso entrega un mensaje m y luego m', ningún otro proceso (correcto) puede entregar m' sin haber entregado antes m. En la literatura de algoritmos distribuidos el problema del *acuedo distribuido* es un problema fundamental [6]. Además, se demuestra que cualquier algoritmo que resuelva el problema del acuerdo distribuido puede utilizarse para implementar un protocolo de difusión con TO, y viceversa, cualquier algoritmo que implemente un protocolo de difusión con TO puede utilizarse para resolver el problema del acuerdo distribuido. Ambos problemas son equivalentes [6][7]. La mayor dificultad es que el problema del acuerdo distribuido es imposible de resolver mediante un algoritmo determinista en un sistema distribuido $\mathcal{AS}_{n,f}[\emptyset]$ incluso aunque sólamente un proceso pueda fallar. Este resultado se conoce como la *imposibilidad FLP* en honor a sus autores Fischer-Lynch-Patterson [8]. Como conclusión, no podemos implementar un algoritmo para la difusión con TO en un sistema $\mathcal{AS}_{n,f}[\emptyset]$.

Las alternativas para solventar la imposibilidad-FLP son: (1) cambiar el tipo de sistema distribuido imponiendo restricciones; (2) construir algoritmos no-deterministas (basados en probabilidad) que solucionen el problema con cierta probabilidad; (3) debilitar la especificación del problema del acuerdo distribuido.

[1] Podemos admitir también fallos un poco más débiles como el fallo de crash que permite que todo el nodo de procesamiento deje de funcionar.

Chandra y Toueg [6] proponen algoritmos de acuerdo distribuido utilizando detectores de fallos no fiables que pueden implementarse en un sistema distribuido *parcialmente síncrono*: el sistema es asíncrono pero llegado un tiempo existen cotas superiores a la latencia de los mensajes aunque no son conocidas por los algoritmos. Este trabajo ha sido fundamental en el desarrollo de los algorimos distribuidos tolerantes a fallos. Así que con un detector de fallos no fiable DF formamos un sistema $\mathcal{AS}_{n,f}[DF]$ y en dicho sistema podemos obtener un algoritmo de acuerdo distribuido y por tanto, un algoritmo de difusión de TO utilizando el algoritmo de acuerdo distribuido como procedimiento auxiliar para alcanzar el acuerdo en el orden en el que deben entregarse los mensajes.

Chandra et al. [7] demostraron que el detector de fallos más simple para obtener el acuerdo distribuido es el detector Ω (*elección de líder*). El algoritmo de PAXOS propuesto por Lamport [9] y recientemente, el algoritmo de RAFT [11][2], son algoritmos que se basan en la elección de un líder que impone el orden en el que tienen que realizarse las operaciones y por tanto, el líder también puede imponer el orden en la entrega de los mensajes.

Este tipo de soluciones al problema de la difusión en TO tienen un coste adicional en el rendimiento por el mecanismo de adaptación del acuerdo. Normalmente se emplean en sistemas distribuidos del tipo *cluster* de máquinas y, una crítica a dichas soluciones es que no son escalables con el número de nodos, a partir de un cierto número de nodos el rendimiento se degrada por un aumento de la latencia en la entrega de los mensajes. Este hecho justifica que en el caso de diseñar un algoritmo de difusión de TO en un sistema distribuido de área extendida con un alto número de procesos participantes y con un grafo de comunicaciones no completo (tipo sistemas peer-to-peer), la opción elegida sea emplear un algoritmo probabilístico para la difusión en TO. Este tipo de algoritmos se basan en la propagación epidémica de mensajes.

Como ejemplo de esta clase de protocolos podemos citar EpTO [12]. EpTO es un algoritmo de diseminación epidémica de mensajes con garantías de entrega en Orden Total que ofrece garantías de fiabilidad probabilísticas. Básicamente, EpTO garantiza que los procesos eventualmente entregan los mensajes con una alta probabilidad y que los mensajes entregados, son entregados en orden total. La idea de este algoritmo se basa en combinar un componente de diseminación de mensajes ente los participantes, y un componente de ordenación de los mensajes. Los mensajes contienen la marca de tiempo (time stamp) en el que fueron creados. Gracias a las marcas de tiempo se puede decidir el orden total de entrega de los mensajes. En este trabajo adaptaremos el componente de ordenación de mensajes propuesto en EpTO.

Las soluciones comentadas en esta breve introducción sólo son algunas de las posibilidades que encontramos en la literatura. En el trabajo de Xabier Defago [17] se estudia toda una posible taxonomía de soluciones que incluyen soluciones basadas en otros paradígmas como los basados en secuenciadores fijos, secuenciadores variables, comuncación basada en historia, entre otras. En lo que sigue presentamos los objetivos concretos del presente trabajo.

1.1. Objetivos del proyecto

Siguiendo una de las tres opciones anteriormente descritas para solventar la imposibilidad FLP, en este proyecto vamos a optar por idear un algoritmo que nos garantice la entrega de mensajes así como

[2] El algoritmo de RAFT se propuso como alternativa comprensible al algoritmo complejo de PAXOS [11].

su validez e integridad pero debilitando la especificación consiguiendo una garantía aproximada de orden total. De esta forma, conseguiremos un algoritmo que construido sobre un sistema asíncrono con canales fiables entrega siempre mensajes válidos, íntegros, y, la mayoría de las veces, en orden total por todos los procesos.

El diseño ha sido inspirado en el algoritmo *Uniform Reliable Broadcast (URB)* propuesto por Raynal en [16]. Este algoritmo nos garantiza tanto la fiabilidad como la integridad, validez, y terminación en la entrega de mensajes. Además de modificar su especificación ligeramente le añadiremos una adaptación del componente de ordenación del algoritmo de EpTO [12] con el objetivo de conseguir el orden total. Para esto es necesario el estudio de ambos algoritmos en profundidad, así como el dominio de un lenguaje cuyo paradigma es asíncrono como es NodeJS para realizar la implementación.

La implementación requiere del aprendizaje de NodeJS, de su ciclo de ejecución y funcionamiento además de la librería ZMQ (ZeroMQ) para la comunicación entre nodos mediante sockets fiables. Además, se necesitan conocimientos de npm (node package manager) para la gestión del proyecto y sus dependencias, instalación de paquetes y git para el control de versiones.

El proyecto se desplegará sobre un clúster de máquinas virtuales que se ejecutarán sobre contenedores docker. Docker es una tecnología que permite la creación de máquinas virtuales (contenedores) con el kernel de Linux que corren sobre una máquina física real. No obstante, estos contenedores permiten que el sistema operativo de las máquinas virtuales funcione de forma independiente a la máquina anfitriona, quedando así totalmente aisladas unas de otras aunque estén compartiendo los recursos de la misma máquina. Esta herramienta de virtualización es muy útil para realizar pruebas de todo tipo de programas y algoritmos, dado que ofrece muchas posibilidades entre las que citamos las siguientes: es una implementación basada en imágenes (lo que hace fácil compartir y transportar contenedores, su copia, etc.), proporciona entornos seguros puesto que los contenedores están aislados, y permite disponer de más máquinas de las que se tiene realmente.

La realización de las pruebas tendrá lugar en un clúster de siete máquinas reales en el laboratorio de Lenguajes y Sistemas Informáticos de la Universidad Pública de Navarra. Para ello se dispondrá de un sistema de clúster Proxmox construido sobre las siete máquinas para la creación de tantos contenedores como se desee. En cada contenedor se desplegarán múltiples nodos para conseguir simular una red de nodos lo suficientemente extensa como para realizar las pruebas de los algoritmos. Después se recogerán los resultados de las ejecuciones de todos los nodos y se estudiarán para obtener conclusiones acerca del funcionamiento y rendimiento de las implementaciones.

1.2. Organización de la memoria

Esta memoria está organizada de la siguiente forma. En la Sección 2 presentaremos la especificación del problema de la difusión de mensajes fiable y uniforme (URB) junto con un algoritmo que resuelve dicho problema de forma sencilla con el objetivo de introducirnos en la comprensión de los aspectos más básicos sobre la difusión de mensajes, así como de la terminología utilizada a lo largo de todo el proyecto. Para esto es necesario ver también el modelo de sistema distribuido que consideramos a la hora de realizar este trabajo. También explicaremos el funcionamiento del algoritmo propiamente y el problema de la ordenación de los mensajes, y por supuesto, cómo se lleva a cabo la construcción del mismo.

En la Sección 3 se presentan diferentes especificaciones y se contrastan para comprender las diferencias de las mismas. Primero se presenta la especificación formal de la comunicación fiable y uniforme en orden total, luego se presenta una variación probabilística de ésta y finalmente, se presenta la especificación propuesta en este trabajo con una garantía parcial de orden total.

En la Sección 4 se presenta el diseño del algoritmo que debe cumplir con la especificación propuesta en la sección anterior (aTO-URB). Primero se mostrará una versión mínima del algoritmo que cumple con la especificación y posteriormente se mostrará la solución completa (URB + aTO). Se describirá en profundidad tanto los algoritmos propuestos, su código, variables, y se explicará el funcionamiento de todas las partes de éstos, incluyendo demostraciones y teoremas que mostrarán por qué y cómo funcionan.

En la Sección 5 se explicará todo lo relacionado con la implementación del proyecto. El lenguaje en el que ha sido implementado, por qué se ha escogido dicho lenguaje, cuáles son sus características, qué librerías se han utilizado, cuáles son los detalles de los sistemas en los que se ha desplegado el proyecto, todo lo necesario a saber sobre los contenedores Docker utilizados, sistemas operativos y cómo se ha llevado el control de versiones. Además se mostará cómo es la arquitectura diseñada para el sistema desde la aplicación hasta la red física de comuncaciones, y se realizará una breve descripción de los módulos que han sido implementados así como de las clases y funciones que contienen.

Finalmente, en la Secciones 6 y 7 del proyecto se exponen las pruebas realizadas, cómo han sido realizadas, los recursos utilizados para el proyecto, así como los resultados que se han obtenido y las conclusiones sacadas.

1.3. Notas sobre el desarrollo del proyecto

Antes de continuar, me gustaría explicar cómo se ha llegado al desarrollo de este proyecto y la experiencia que he conseguido con anterioridad en el campo de los sistemas distribuidos (más concretamente en protocolos de consenso y de orden total). Durante los dos últimos años he estado investigando, estudiando, e implementando algoritmos de este tipo duramente mi colaboración con la universidad como becario. En ese tiempo, llevé a cabo la implementación de los algoritmos Raft y EpTO, algoritmos ya mencionados anteriormente.

Algunas de estas implementaciones, además de la de este mismo proyecto, están a disposición pública para que se puedan consultar, descargar, y desplegar para su prueba, estudio o uso. Las implementaciones están publicadas en Github. El código de EpTO programado en NodeJS está disponible en `https://github.com/deivi98/EpTO`, y aTO-URB está disponible en `https://github.com/deivi98/ATOURB`. Ambos proyectos contienen un fichero *README* que contiene toda la información necesaria sobre la instalación y lanzamiento de los mismos, así como una breve explicación del proyecto en sí mismo, así como de su estructura en lo que a ficheros se refiere.

2. Difusión fiable y uniforme

En esta sección presentamos la especificación del problema de la difusión de mensajes fiable y uniforme (*uniform reliable broadcast*, URB), y un algoritmo muy simple que cumple con dicha especificación en un sistema asíncrono con la posibilidad de fallos de caída (*crash*). Este tipo de comunicación es importante en el diseño e implementación de sistemas distribuidos tolerantes a fallos, ya que los servicios distribuidos tolerantes a fallos requieren subsistemas de comunicación con ciertas garantías en las entregas de los mensajes. Antes de centrarnos en la especificación y la solución al problema URB detallamos el modelo de sistema distribuido que estamos considerando en este trabajo. La redacción de esta sección sigue la notación y las ideas principales dadas por Raynal en [16].

2.1. Modelo del sistema distribuido

El sistema distribuido sobre el que queremos construir los algoritmos en este trabajo está formado por n procesos secuenciales, $\Pi = \{p_1, .., p_n\}$, donde p_i indica la identidad del proceso i-ésimo. Cada proceso se comporta como un autómata de estados: dispone de un estado inicial y un algoritmo local asignado al proceso que le permite ejecutar una secuencia de estados que se determinan por la función de transición definida por su algoritmo local. Cada transición que realiza el proceso se supone que tiene un comportamiento atómico en el sentido de que o se ejecuta completamente o no se ejecuta. Cada transición llevada a cabo corresponde con la ejecución de una sentencia[3] del código, el envío de un mensaje, o la recepción de un mensaje. El conjunto formado por todos estos algoritmos locales forma el algoritmo distribuido.

Los procesos pueden fallar pero el modelo de fallo considerado es el fallo de *caída* (*crash*). Un proceso que falla deja de realizar transiciones y no se recupera. Antes de fallar cumplía de forma precisa con su algoritmo local. Un proceso que falla en una ejecución es, obviamente, un proceso *fallido* en dicha ejecución. Por tanto, un proceso que no falla en una ejecución es un proceso *correcto*. El parámetro f del modelo del sistema, $0 \leq f < n$ denota el número máximo de procesos que pueden fallar durante cualquier ejecución del sistema. Se asume que al menos un proceso es correcto. Un algoritmo distribuido es f-resiliente si se comporta correctamente cuando a lo sumo f procesos pueden fallar. Si dicho algoritmo distribuido se enfrenta a más de f fallos en el sistema, no hay garantía sobre su comportamiento.

En el sistema distribuido considerado, cada pareja de procesos $(p, q) \in \Pi$ están conectados por un canal bidireccional. Para enviar un mensaje m al proceso q, el proceso p invoca la operación *send m to q*. Esta operación no es bloqueante. Para recibir un mensaje desde q, el proceso p invoca la operación $receive()\ from\ q$. Esta operación es bloqueante, en otras palabras, el progreso en la ejecución de p se detiene hasta que se completa la recepción.[4] Cuando la recepción se completa se conoce el mensaje m recibido.

Cada canal de comunicación se considera *fiable*. Esto indica que no hay creación, modificación o pérdida de mensajes.

[3] Al ser los procesos secuenciales sólo consideramos un único hilo de ejecución.

[4] Aunque la operación se realiza de forma bloqueante en el contexto del algoritmo, en el entorno de NodeJS esto no sucede así dado que el lenguaje es completamente asíncrono y por tanto la recepción de un evento no impide la recepción de otros paralelamente. Esto se tendrá en cuenta a la hora de implementar el algoritmo.

En cuanto a las restricciones temporales, los procesos son *asíncronos*. Cada proceso se ejecuta a su propia velocidad sin relación con el progreso de los otros procesos. La única suposición es que hasta que el proceso falla (si es que falla), el proceso progresa en su ejecución y un proceso correcto tarde o temprano ejecuta la siguiente transición de su algoritmo. Al nivel de las comunicaciones también consideramos asincronía. No hay establecido ningún límite superior en la latencia de los mensajes salvo que es finita. Esto requiere una aclaración dado que los procesos utilizan los canales para enviar y recibir mensajes:

Si un proceso p completa una operación 'send m to q', y el proceso q es un proceso correcto, entonces, si el proceso q ejecuta 'receive() *from p' entonces eventualmente recibe el mensaje m (puesto que los canales son fiables y la latencia, aunque desconocida, es finita).*

El modelo distribuido considerado se denota por $\mathcal{AS}_{n,f}[\emptyset]$, donde $\mathcal{AS}$ hace referencia a que es un modelo asíncrono con canales fiables, (n, f) indica el número de procesos y el número de fallos ($0 \leq f < n$), y $[\emptyset]$ hace referencia a que no hay ninguna otra restricción adicional. En el caso de incluir alguna restricción adicional al modelo se indica en dicho término, por ejemplo, $\mathcal{AS}_{n,f}[f < \frac{n}{2}]$, es un modelo distribuido asíncrono con canales fiables, en donde una mayoría de los procesos son correctos.

2.2. Especificación de URB

Para entender a dónde queremos llegar, antes hay que entender lo más básico. Dado un conjunto de n procesos y un mensaje m, $broadcast(m)$ invocado por un cierto proceso es el procedimiento que envía ese mismo mensaje a todos los procesos participantes en el sistema. Su código es simplemente,

foreach $q \in \{1, .., n\}$ **do** *send m to q* **endfor**

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 diferentes. 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 ninguna forma qué procesos recibirán el mensaje y cuáles no. El objetivo de URB es encontrar la forma de diseñar un broadcast alternativo que garantice la entrega de los mensajes a todos los procesos correctos.

El interfaz para URB se compone de dos nuevas operaciones llamadas URB-broadcast(m) y URB-deliver(). La primera permite a un proceso enviar un mensaje a todos los procesos (incluido él mismo) y la segunda permite a un proceso recibir un mensaje que ha sido previamente enviado mediante URB-broadcast por otro proceso. Para evitar confusiones, cuando un proceso ejecuta URB-broadcast(m), decimos que el proceso 'URB-broadcast el mensaje m'; y cuando un proceso retoma la ejecución desde URB-deliver(m) con un mensaje m establecido por la entrega, decimos que el proceso 'URB-deliver el mensaje m'.

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 forma de hacer esto es formar un ID mediante la pareja formada por el identificador del proceso emisor y un contador auto-incremental de

envío de mensajes controlado por el proceso emisor.

Definición 1 *La especificación consiste en tres propiedades. Cualquier implementación de URB-broadcast(m) y URB-deliver() debe cumplirlas para ser correcta:*

- **Validez***: Si un proceso realiza URB-deliver de un mensaje m, entonces antes otro proceso ha realizado un URB-broadcast del mensaje m.*
- **Integridad***: Un proceso realiza URB-deliver de un mensaje m como mucho una vez.*
- **Terminación***: (1) Si un proceso correcto realiza URB-broadcast de un mensaje m, o (2) si un proceso realiza URB-deliver de un mensaje m, entonces todos los procesos correctos realizan URB-deliver del mensaje 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. Esto es debido a que son propiedades de seguridad[5]. Aquí es donde entra la tercera propiedad de progreso, **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 entregado por todos los procesos correctos.

De las anteriores propiedades se puede concluir que (1) los procesos correctos entregan exactamente el mismo conjunto de mensajes, (2) que este conjunto incluye todos los mensajes emitidos por todos los procesos correctos, y (3) que todo proceso fallido entrega un subcojunto de los mensajes entregados por algún proceso correcto. (aunque dos procesos no correctos distintos pueden entregar 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 depende de la ejecución.

En la figura 1 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 1 respeta la especificación de URB.

[5]En términos de Lamport, una *propiedad de seguridad* indica que 'algo malo no ocurre nunca', así que si no sucede nada no se incumplen las propieades de seguridad. En cambio, una *propiedad de progreso* indica que 'algo bueno tarde o temprano sucede'. Cualquier especificación debe incluir ambos tipos de propiedades.

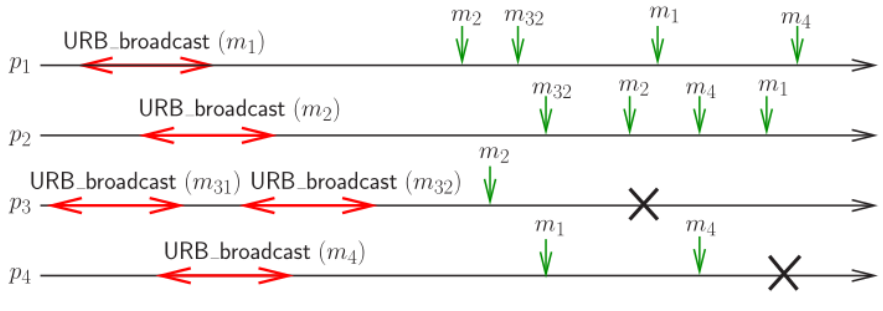

Figura 1: Un ejemplo de la garantía de entrega de mensajes [16]

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.

El término 'uniformidad' en el contexto de la disfusión fiable justamente se refiere a que todos los mensajes entregados por procesos correctos o no correctos son entregados por todos los procesos correctos también. Justamente la propiedad de **terminación** garantiza dicho comportamiento y por ese motivo, los procesos fallidos entregan un subconjunto de los mensajes que son entregados por los procesos correctos. Debilitar dicha propiedad de terminación conduce a la definición de un protocolo que simplemente es fiable. Así, la abstracción de comunicación *difusión fiable* (*reliable broadcast*, RB) es una forma debilitada de URB. Se especifica con las mismas propiedades de intregridad y validez (sin pérdida de mensajes, corrupción o duplicidad) y la siguiente propiedad fuerte de terminación:

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

Esto significa que un proceso que falle en una ejecución puede entregar mensajes que uno correcto no entregue. Esta propiedad de terminación fuerte es la misma que la propiedad de terminación de URB sin su uniformidad.

2.3. Construcción de un protocolo URB en un sistema asíncrono

Existe una construcción muy simple de la abstracción URB en el modelo de sistema $AS_{n,t}[\emptyset]$. La simplicidad de la solución se debe a que los canales punto a punto son fiables. La estructura de la construcción se muestra en la figura 2. Las operaciones de comunicación se implementan utilizando las primitivas de comunicación punto a punto que suministra la capa de red. En conjunto, la implementación

puede verse como un middleware de comunicaciones que la capa de aplicación puede utilizar para definir su funcionalidad.

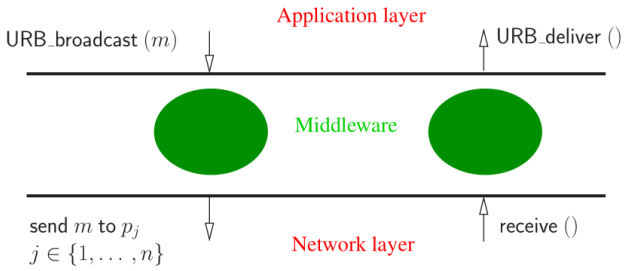

Figura 2: Arquitectura de la implementación [16].

Los códigos que implementan URB-broadcast() y URB-deliver() están descritos en la figura 3. Estos códigos se corresponden al proceso p_i. Para emitir un mensaje m, un proceso p_i envía m a sí mismo. Cuando el proceso p_i 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_i 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. Este efecto de primero reenviarlo a todos y luego entregarlo es clave para garantizar la uniformidad.

La parte del código que maneja la recepción de un mensaje m no requiere que su ejecución sea 'excluyente' con la ejecución que maneja la recepción de otro mensaje m'. En otras palabras, el proceso p_i puede entrelazar la ejecución de varias de estas sentencias y una implementación concurrente del código es posible. No obstante, consideramos que cada proceso p_i es secuencial, formado por un único hilo de ejecución, de manera que no hay ejecución concurrente. Cada sentencia del tipo '**when**...' y el código asociado a dicha sentencia se ejecuta sin interrupción. Las sucesivas recepciones se ejecutan secuencialmente.

```
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 3: Protocolo de difusión fiable y uniforme en un sistema $AS_{n,t}[\emptyset]$ [16].

Teorema 1 *El algoritmo descrito en la figura 3 satisface la especificación de la comunicación URB 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 y al hecho de que los canales punto a punto son fiables.

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».

(1) Supongamos que un proceso correcto p realiza URB-broadcast de un mensaje m. Como p es correcto y se lo envía primero a sí mismo lo recibe y como consecuencia reenvía el mensaje m 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 correctos restantes. Al ser los receptores correctos necesariamente realizarán URB-deliver(m) la primera vez que lo reciban. (2) Ahora vamos a considerar el caso de un proceso (con fallos o correcto) que entrega un mensaje m, ha ejecutado URB-deliver(m). Antes de entregar el mensaje, ha tenido que reenviar el mensaje a todos los procesos, y por tanto, por la fiabilidad de los canales punto a punto, los procesos correctos recibirán dicho mensaje. De nuevo, al ser los receptores correctos necesariamente realizarán URB-deliver(m) la primera vez que lo reciban. Esto prueba la demostración de la propiedad de **terminación**. □

2.4. El problema de la garantía en Orden Total

Como hemos visto en la subsección anterior URB permite que si un proceso correcto entrega un conjunto de mensajes R hasta un punto de la ejecución en la que ya no hay más envíos el resto de procesos correctos tarde o temprano también entregarán dicho conjunto R, y los procesos fallidos sólo entregan un subconjunto $R' \subseteq R$. En algunas aplicaciones se requiere alguna condición más fuerte de entrega que implica cierta ordenación en la entrega de mensajes. El caso más simple de aumentar la especificación de URB es añadiendo la condición de entrega FIFO. Los mensajes entregados por los procesos correctos respetan el orden FIFO de la creación de mensajes por parte de los emisores de los mismos. Esta propiedad permite aumentar la calidad de servicio de URB. Es posible implementar un protocolo FIFO-URB en un sistema distribuido asíncrono. Básicamente, un proceso receptor p, antes de entregar un mensaje generado por un emisor q, comprueba si se respeta el orden FIFO de q para lo cual, analiza el número de orden generado por q en el mensaje con respecto al que el proceso p espera entregar como siguiente entrega para el emisor q. La implementación sólo requiere variables de control locales al proceso (ejemplo, 'el valor del siguiente número de orden por cada emisor') y de información que se guarda en el propio mensaje (ejemplo, 'el contador generado por el orden FIFO del proceso emisor').

De una manera similar también se puede implementar un protocolo CO-URB que ofrezca ganrantías de respetar la causalidad entre mensajes: si un mensaje m se difunde como causa de la recepción de otro mensaje m' ($m' \rightarrow m$) o m se difunde después de la difusión de m' ($m' \rightarrow m$) en el mismo proceso entonces ningún proceso correcto puede entregar m sin antes haber entregado m'. De esta forma se respeta la relación causal '$\rightarrow$' «ocurrir antes que».

Como hemos indicado en la introducción (sección 1) las aplicaciones para la replicación de objetos

tolerante a fallos con garantías fuertes de consistencia requieren la propiedad de entrega de mensajes en orden total. En ese caso la propiedad de orden total (TO) requiere que cualquier pareja de mensajes tiene que ser entregada en el mismo orden en cualquier proceso incluso si la difusión de los mensajes no es ni FIFO ni CO. Por ejemplo, considera que el proceso p envía $m1$ como su primer mensaje y el proceso q envía $m2$ como su primer mesaje. Ambas difusiones no tienen ninguna relación ente ellas. La entrega en orden total de ambos mensajes no podrá basarse en alguna información almacenada en los mensajes que son difundidos por la aplicación. Necesariamente, ambos procesos necesitan cooperar entre ellos, posiblemente enviando mensajes de control entre los procesos, para establecer un orden de entrega común. Ambos procesos tienen que 'acordar' este orden de forma que si p entrega entrega primero $m1$, entonces q no puede entregar $m2$ si antes no ha entregado $m1$. Claramente tiene que resolverse un problema de acuerdo distribuido en un entorno de fallos, y si el modelo es asíncrono $\mathcal{AS}_{n,f}[\emptyset]$, será imposible disponer de un protocolo de difusión en orden total por el resultado de imposibilidad-FLP.

En la siguiente sección damos la especificación formal de la comunicación fiable y uniforme en orden total y presentamos una variante probabilística de dicha especificación. Comentaremos ambas especificaciones y realizaremos una nueva propuesta de especificación que es la que vamos a tratar de resolver en este trabajo.

3. Especificación de la difusión con orden total

En la definición 1 presentamos la especificación de la abstracción URB, a continuación presentamos la especificación de la abstracción TO-URB que incluye la garantía de entrega en orden total. Cambiamos ligeramente la notación y nos referimos indistintamente sobre mensajes o sobre *eventos*. Un evento es un poco más general que un mensaje de la aplicación ya que incluye al mensaje de la aplicación pero también puede incluir otras informaciones relativas a la implementación del protocolo que debe cumplir la especificación. Por orden total (TO) se entiende que los procesos entregan los eventos en el mismo 'orden' sin importar cómo se forma dicho orden. En todas las especificaciones dadas los eventos son únicos en el sistema puesto que cada mensaje creado por la aplicación es único en el sistema. También asumimos que el sistema es $\mathcal{AS}_{n,f}[\emptyset]$

Definición 2 *Las operaciones TO-broadcast() y TO-deliver() constituyen el interfaz de comunicación. Cualquier implementación de TO-URB debe satisfacer las siguientes propiedades:*

- **Validez**. *Si un proceso TO-deliver un evento e, entonces e ha sido TO-broadcast por algún proceso.*
- **Integridad**. *Un proceso TO-deliver un evento e como mucho una vez.*
- **Entrega en Orden Total**. *Si un proceso TO-deliver un evento e y posteriormente TO-deliver un evento e', entonces ningún proceso TO-deliver el evento e' a menos que antes haya realizado TO-deliver del evento e.*
- **Terminación**. *(1) Si un proceso correcto realiza TO-broadcast de un evento e, o (2) si un proceso realiza TO-deliver de un evento e, entonces todos los procesos correctos realizan TO-deliver del evento e.*

Hay que observar que la propiedad de **Entrega en Orden Total** se tiene que cumplir para todos los procesos sean estos correctos o fallidos. La especificación anterior es la especificación más fuerte que podemos encontrarnos en la literatura puesto que asegura, que para cualesquiera dos procesos, p y q, si $hist(p)$ y $hist(q)$ representan, respectivamente para p y q, la historia de eventos entregados en cualquier punto de la ejecución del sistema entonces, o bien se cumple $hist(p) \preceq hist(q)$ o $hist(q) \preceq hist(p)$, donde $\preceq$ indica la relación 'prefijo de'. Los procesos fallidos no pueden *contaminar* a los procesos correctos (en la figura 4 se muestra el efecto de la 'contaminación' de procesos correctos por procesos fallidos). Esto es una gran ventaja en el caso en el que tuviésemos que 'recuperar' la ejecución de un proceso fallido puesto que su estado antes de fallar (en cuanto a la entrega de eventos) es consistente con los estados de un proceso correcto.

Otra observación importante es que la abstración TO-URB es justamente la especificación dada para URB en la definición 1 y la propiedad de orden total, TO-URB= URB + TO.

Por lo indicado en la sección 2, URB es implementable en $\mathcal{AS}_{n,f}[\emptyset]$ pero, por desgracia, no podemos implementar TO-URB en $\mathcal{AS}_{n,f}[\emptyset]$. Su implementación requiere sincronía parcial o bien la existencia de un detector de fallos (o la solución del acuerdo distribuido): TO-URB es implementable en $\mathcal{AS}_{n,f}[\Omega]$.

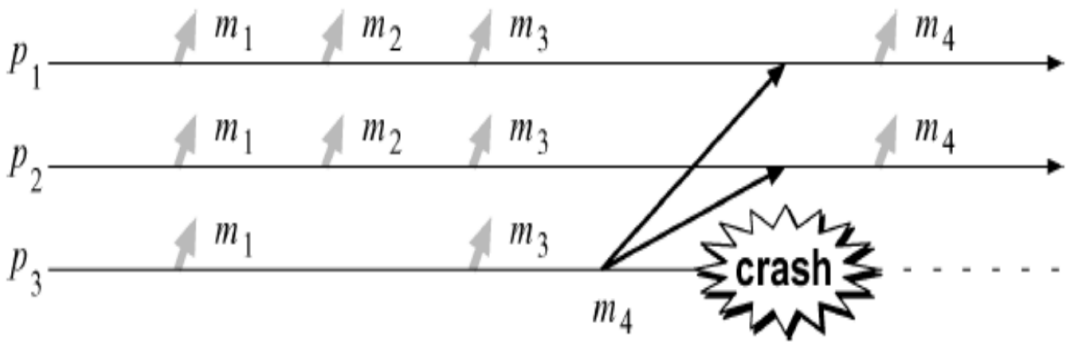

Figura 4: Contaminación de procesos correctos ($p1, p2$), por un message ($m4$), basado en un estado inconsistente ($p3$ entrega $m3$ pero no entrega $m2$).

Si no queremos cambiar el tipo de sistema distribuido y admitimos la posibilidad de una solución probabilística podemos proponer otra especificación del problema que admita este tipo de soluciones. Esta es la propuesta que hace Matos et. al. en [12] [13]. La especificación es la siguiente:

Definición 3 *Las operaciones EpTO-broadcast() y EpTO-deliver() constituyen el interfaz de comunicación. Cualquier implementación debe satisfacer las siguientes propiedades:*

- **Validez**. *Si un proceso EpTO-deliver un evento e, entonces e ha sido EpTO-broadcast por algún proceso.*
- **Integridad**. *Un proceso EpTO-deliver un evento e como mucho una vez.*
- **Orden Total**. *Si dos procesos p y q EpTO-deliver los eventos e y e′, entonces, p EpTO-deliver el evento e antes que el evento e′* **si y sólo si** *q EpTO-deliver el evento e antes que el evento e′.*
- **Terminación de procesos correctos**. *Si un proceso correcto realiza EpTO-broadcast de un evento e, entonces dicho proceso realizará EpTO-deliver del evento e.*
- **Terminación probabilística**. *Si un proceso realiza EpTO-deliver un evento e, entonces* **con alta probabilidad** *todos los procesos correctos realizan EpTO-deliver del evento e.*

En la especificación anterior las propiedades de **validez** y de **integridad** son las mismas que en el resto de especificaciones dadas anteriormente. La propiedad de **Orden Total** de EpTO es más débil que la presentada en TO-URB y no previene la posibilidad de contaminación por procesos fallidos. La **Terminación de procesos correctos** asegura que los procesos correctos siempre entregan los eventos que difunden. Hay que observar que la **Terminación probabilística** puede producir agujeros en las historias de eventos entregados por los procesos. Cualquier implementación debería producir estos vacíos con muy poca (o casi nula) probabilidad. Es posible entonces que las historias de entregas puedan diferir, pero la propiedad de **Orden Total** asegura que los eventos están en todas las historias con el mismo

orden (entre cada pareja de eventos). En la figura 5 se puede ver la diferencia entre la especificación de la abstracción TO-URB y EpTO. En el caso de TO-URB los tres procesos entregan exactamente los mismos mensajes en el mismo orden. En el caso de EpTO, la especificación permite que haya huecos en la entrega ya que la terminación es probabilísitca. Observe además que si los tres procesos son correctos y los eventos e y e' han sido generados por p y el evento e'' ha sido generado por el proceso r después de haber entregado e' y de haber entregado e'' entonces, aun siendo r correcto está produciendo un efecto de contaminación ya que ha difundido e'' en un estado inconsistente con la historia de entregas de p y de q. A pesar de este comportamiento, el protocolo EpTO ha sido implementado en un sistema distribuido $\mathcal{AS}_{n,f}[\emptyset]$ de área extendida [12].

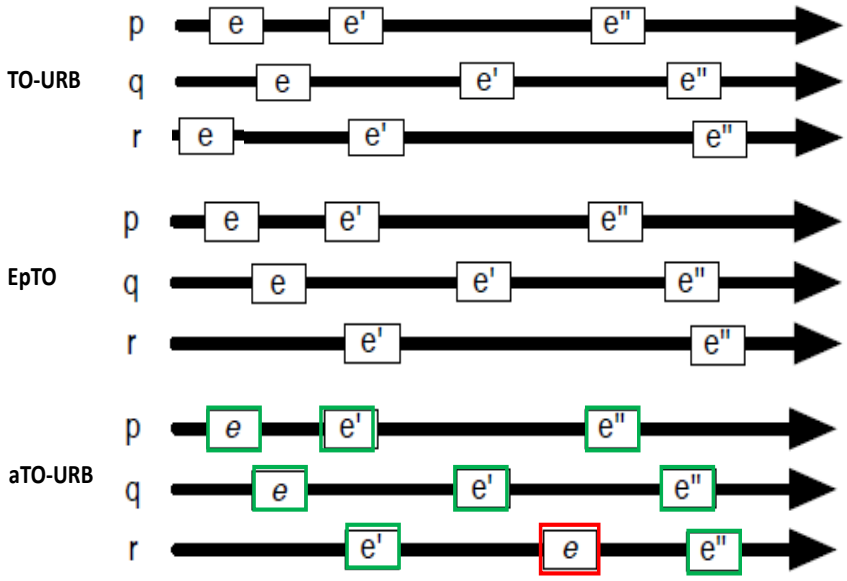

Figura 5: Diferencias entre las tres especificaciones dadas en esta sección.

3.1. Propuesta de especificación de difusión con garantía parcial de orden total

Nuestra propuesta de especificación se basa en la idea que se muestra en la figura 5. Sabemos que no podemos conseguir definitivamente URB+TO (cualquiera de las dos versiones fuerte o débil) en un sistema distribuido $\mathcal{AS}_{n,f}[\emptyset]$, entonces lo que proponemos es URB+aTO, es decir, difusión uniforme y fiable con una aproximación al orden total (aTO), o como se indica en el título del trabajo con una garantía parcial de orden total. En nuestra especificación diferenciamos dos tipos distintos de entrega de eventos: TO-deliver() y U-deliver(). En la figura los colores verde y rojo diferencian TO-deliver() y U-deliver() respectivamente. Cuando un proceso realiza TO-deliver de un evento e se asegura una entrega en orden total (en el sentido débil dado por la especificación) con respecto a todos los procesos que realizan un TO-deliver de e. En cambio cuando un proceso realiza un U-deliver de un evento e puede asegurar que se ha entregado fuera del orden total. Todos los procesos correctos entregan el mismo conjunto de mensajes por ser URB; con lo que una entrega fuera del orden total es detectada, y dependiendo de la aplicación podría ser corregida posteriormente. Por tanto, con nuestra propuesta podríamos aliviar la contaminación por parte de procesos correctos: podemos tener un orden total aproximado porque no todos los eventos

participan en el orden sin embargo en los procesos correctos tenemos a todos los eventos entregados, el mismo conjunto de eventos. Por la propiedad que estamos barajando de orden total débil no podemos evitar la contaminación por parte de los procesos fallidos.

En realidad, lo anterior es una intención puesto que en una implementación de una solución a la especificación aTO-URB que estamos proponiendo, sobre un sistema $\mathcal{AS}_{n,f}[\emptyset]$ sigue siendo un protocolo URB que hace el 'mayor esfuerzo' por mantener un orden total. La cuestión es cuándo la garantía parcial de orden total se convierte prácticamente una garantía completa de orden total. Veremos que si implementamos una solución aTO-URB en un sistema tipo cluster de máquinas con una mayoría de procesos correctos, $\mathcal{AS}_{n,f}[f < \frac{n}{2}]$ obtenemos en la práctica un protocolo TO-URB. A continuación presentamos los detalles de la especificación.

Definición 4 *(***Especificación aTO-URB***) Las operaciones U-deliver(), TO-deliver(), y Broadcast(), constituyen el interfaz de comunicaciones. En lo que sigue, X-deliver() representa a cualquiera de las operaciones U-deliver() o TO-deliver(). Una implementación de aTO-URB debe satisfacer las condiciones siguientes:*

- **Validez**. *Si un proceso X-deliver un evento e, entonces e ha sido Broadcast por algún proceso.*
- **Integridad**. *Un proceso X-deliver un evento e como mucho una vez.*
- **Orden Total**. *Si dos procesos p y q TO-deliver los eventos e y e', entonces, p TO-deliver el evento e antes que el evento e'* **si y sólo si** *q TO-deliver el evento e antes que el evento e'.*
- **Terminación**. *(1) Si un proceso correcto realiza Broadcast de un evento e, o (2) si un proceso realiza X-deliver de un evento e, entonces todos los procesos correctos realizan X-deliver del evento e.*

4. Difusión con garantía parcial de orden total

Llegamos al verdadero objetivo de este proyecto que consiste en el diseño de un algoritmo que cumpla con la especificación dada en la definición 4. Denomimanos a este algoritmo aTO-URB y cuyo fin es proporcionar uniformidad, fiabilidad y una garantía parcial (o aproximada) al orden total. Introduciremos el algoritmo final de forma progresiva. Primero presentaremos un algoritmo aTO-URB bastante simple que cumple con la especificación y a continuación presentaremos la solución completa. Esta solución final se basa en la estructura URB a la que se le añade un componente de ordenación más complejo que el dado en la solución trivial y que está inspirado en la ordenación propuesta por Matos en el protocolo EpTO [12].

4.1. Algoritmo aTO-URB simple

El algoritmo aTO-URB simple se basa en modificar el protocolo URB dado en la figura 3 incluyendo en cada evento una marca de tiempo (timestamp ts) que permite ordenar totalmente a los eventos.

Sea $\Pi = \{p_1, .., p_n\}$ el conjunto de n procesos en el sistema. Cada p_i representa el identificador único del proceso, un número entero distinto de cero, $p_i \neq 0$. El sistema es asíncrono con canales fiables $\mathcal{AS}_{n,f}[\emptyset]$ con $0 \leq f < n$, es decir, al menos un proceso es correcto.

Asumimos que cada proceso mantiene un reloj lógico en el sentido dado por Lamport [10]. Cada proceso dispone de una variable lc (logical clock) de tipo entero iniciada a valor cero. Una acción que supone el envío de un mensaje m por un canal añade el valor actual de lc al mensaje MSG(m, lc), lo envía, e incrementa el valor de lc, $lc \leftarrow lc + 1$. Cuando el proceso p recibe un mensaje que incluye un valor de tiempo MSG(m, u) se actualiza su reloj local, $lc \leftarrow \max(lc, u)+1$. Las acciones internas también actualizan el reloj lógico pero para nuestros propósitos no es necesario considerarlas. Los relojes lógicos cumplen una propiedad global en el sistema: si dos estados s y r de dos procesos son causalmente dependientes $s \rightarrow r$ (r depende causalmente de s) entonces $s.lc < r.lc$, es decir, el valor del reloj lógico en el estado r es mayor que el del estado s. La propiedad contraria no es siempre cierta y requiere, por ejemplo, el uso de vectores lógicos.

Gracias a los relojes lógicos podemos proporcionar un orden total sobre el conjunto de eventos en el sistema utilizando la noción de marca de tiempo. Un time stamp ts está formado por la pareja $ts = (lc, p)$, el valor del reloj lógico y el identificador de proceso. Cada evento e producido por un proceso se etiqueta con su ts en el momento de su creación. Así, dos eventos e y e' cumplen $e < e'$ si y sólo si $e.ts(lc) < e'.ts(lc) \vee (e.ts(lc) = e'.ts(lc) \wedge e.ts(p) < e'.ts(p))$. Se observa que dos eventos distintos no pueden tener marcas de tiempo iguales.

El algoritmo 1, a diferencia del algoritmo URB (figura 3), distingue la entrega TO-deliver() de la entrega U-deliver(). Como las líneas 17 a 20 corresponden a una alternativa exclusiva el comportamiento de este algortimo simple satisface el Teorema 1, es decir cumple con la especificación URB. Además, el algoritmo 1 cumple con la propiedad **Orden Total**.

Algorithm 1 aTO-URB Simple Algorithm

```
1:  Code for process p
2:
3:  variables
4:      peers ← Π                                   ▷ Set of process identifiers
5:      lastTS ← (0, 0)                             ▷ Last time stamp for a TO-deliver event
6:
7:  procedure BROADCAST(event)
8:      event.id ← p + NEWID()                      ▷ Unique event identifier
9:      event.ts ← GETTS()                          ▷ Time stamp for the event
10:     send event to p                             ▷ send to itself
11:
12: upon receive EVENT(event, from q)
13:     if (first reception of event) then
14:         foreach j ∈ peers if j ≠ q ∧ j ≠ p do
15:             send event to j
16:
17:         if lastTS < event.ts then
18:             lastTS ← event.ts
19:             TO-DELIVER(event)
20:         else
21:             U-DELIVER(event)
22:
```

Nota 1 *En el texto del algoritmo 1 no hemos incluido el código para la gestión de los relojes lógicos. Asuminos que la gestión de los relojes lógicos se realiza correctamente en los envíos y recepciones de mensajes. El método* GETTS(()) *devuelve la marca de tiempo en función del reloj lógico en el momento de su ejecución. Una vez asignada al evento event.ts en la línea 9, su marca no cambia en la ejecución.*

Lema 1 *El algoritmo aTO-URB simple cumple la propiedad:*

Orden Total *Si dos procesos p y q TO-deliver los eventos e y e′, entonces, p TO-deliver el evento e antes que el evento e′* **si y sólo si** *q TO-deliver el evento e antes que el evento e′.*

Demostración: Consideremos que p y q ambos TO-deliver los eventos e y e'. Supongamos que en p el evento e se entrega antes que el evento e'. En cualquier estado de ejecución S del proceso p posterior a la emisión de TO-deliver(e) se cumple necesariamente (por la línea 18) que $e.ts \leq S.lastTS$. El proceso p no falla antes de emitir TO-deliver(e'). En el estado de ejecución S' cuando se produce TO-deliver(e'), por la línea 17, se cumple $S'.lastTS < e'.ts$. Como S' es un estado posterior a la emisión de TO-deliver(e), entonces se cumple que $e.ts < e'.ts$ (las marcas de tiempo de los eventos no cambian y son distintas). Por contradicción: Supongamos que en el proceso q el evento e' se entrega antes que el evento e. Por la misma argumentación dada anteriormente para el proceso q se cumple que $e'.ts < e.ts$ pero esto es una contradicción ya que las marcas de tiempo de los eventos son únicas y no cambian. Por tanto, la propiedad de **Orden Total** se cumple. □

En conclusión el algoritmo aTO-URB simple cumple la especificación de aTO-URB dada en la definición 4. Claramente, es muy poco lo que aporta este algoritmo simple puesto que no hay ningún esfuerzo para aumentar el número de eventos que se entregan en orden total. La propiedad **Orden Total** es una propiedad de seguridad, así que se cumple trivialmente si ningún proceso realiza una entrega TO-deliver de algún evento. Además como la entrega en orden total se basa en la variable local $lastTS$ no tiene por qué haber ninguna relación entre la entrega en orden total de un proceso p con respecto a otro proceso distinto q.

4.2. Diseño del algoritmo aTO-URB

La cuestión que nos planteamos es cómo aumentar la posibilidad de que los procesos realicen un mayor número de entregas en orden total. Por una parte, si nos fijamos en el código del algoritmo 1, un mismo proceso p puede recibir varias veces el mismo evento puesto que si varios procesos correctos retransmiten el evento (ver línea 14), este proceso p tarde o temprano recibirá varias copias del evento. Podemos aprovechar esta característica y retrasar el tratamiento de un evento hasta que se haya recibido varias veces dicho evento por parte de otros procesos. De esta manera, aseguramos que el evento ha sido visto por otros procesos y así aumentar la posibilidad de que dicho grupo de procesos realice la entrega en orden total. Así que proponemos una implementación en la que asumimos que una mayoría de procesos son correctos, o de forma más precisa, fijamos el número máximo de posibles fallos $f = F$ que pueden suceder en el sistema. De esta manera trataremos los eventos cuando el número de recepciones sea mayor o igual a $n - F$, el número mínimo de procesos correctos. El precio que pagamos por aumentar las entregas en orden total es un mayor retraso en la entrega de los mensajes. No tenemos muchas más opciones puesto que seguimos considerando que el sistema es asíncrono y no disponemos de un algoritmo de consenso.

El código del algoritmo aTO-URB se muestra a continuación (Algoritmo 2).

Algorithm 2 aTO-URB Algorithm

```
1:  Code for process p
2:
3:  variables
4:      peers ← Π                                          ▷ Set of process identifiers
5:      received ← ∅                                       ▷ Map for storing received events
6:      lastDeliveredTS ← (0, 0)
7:      DeliveredEvents ← ∅                                ▷ Set of delivered events
8:      N                                                  ▷ Constant: number of processes
9:      F                                                  ▷ Constant: maximum number of faulty processes
10:
11: procedure BROADCAST(event)
12:     event.id ← p.id + NEWID()                          ▷ event unique identifier
13:     event.ts ← GETTS()                                 ▷ Get current time stamp for the event
14:     event.sourceId ← p
15:     RECEIVEHANDLER(event, p)                           ▷ send event to itself
16:
17: upon receive EVENT(event, from senderId)
18:     RECEIVEHANDLER(event, senderId)
19:
20: procedure RECEIVEHANDLER(event, senderId)
21:     if event ∈ received then
22:         received[event.id].nor++
23:         if ISDELIVERABLE(event) then
24:             ORDERANDDELIVEREVENTS()
25:     else if event ∉ DeliveredEvents then
26:         foreach q ∈ peers if p ≠ q do
27:             send event to q
28:                                                        ▷ first send, then store
29:         received ← received ∪ {(event.id, event)}
30:         received[event.id].nor ← 1                     ▷ first reception of event
```

```
31: procedure orderAndDeliverEvents( )
32:     minTsOfNonDeliverable ← ∞
33:     deliverableEvents ← ∅                          ▷ set of possible deliverable events
34:
35:     foreach event ∈ received do
36:         if event.ts < lastDeliveredTS then
37:             received ← received \ event
38:             DeliveredEvents ← DeliveredEvents ∪ {event}
39:             U-DELIVER(event)
40:         else if isDeliverable(event) then
41:             deliverableEvents ← deliverableEvents ∪ {event}
42:         else if event.ts < minTsOfNonDeliverable then
43:             minTsOfNonDeliverable ← event.ts
44:
45:     foreach event ∈ deliverableEvents do
46:         if event.ts > minTsOfNonDeliverable then
47:             deliverableEvents ← deliverableEvents \ event
48:         else
49:             received ← received \ event
50:
51:     foreach event ∈ deliverableEvents sorted by (event.ts) do
52:         DeliveredEvents ← DeliveredEvents ∪ {event}
53:         lastDeliveredTS ← event.ts
54:         TO-DELIVER(event)
55:
56: function isDeliverable(event)
57:     return received[event.id].nor ≥ (N − F) − 1
```

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

- *peers*: representa el conjunto de los identificadores de los procesos en el sistema que permiten en la implementación definir el conjunto de conexiones a todos los nodos.
- *received*: es un mapa (identificador de evento, evento) que guarda el conjunto de eventos recibidos por un proceso. Además, a cada evento en *received* se le asigna una variable local entera *nor* (número de recepciones). *received*[*event.id*].*nor* cuenta el número de veces que se ha recibido el evento en un proceso.
- *lastDeliveredTS*: guarda la última marca de tiempo de los eventos entregados en orden total a la capa de la aplicación.
- *DeliveredEvents*: es un conjunto que guarda los eventos entregados a la capa de la aplicación.
- N: constante que define el número de procesos existentes en el sistema.

- F: constante que define el número de fallos máximo.

La diferencia $N - F$ establece el número mínimo de procesos correctos en el sistema. Si el número de fallos supera el máximo establecido no se asegura la propiedad de **Terminación**. Por ese motivo, asumimos que el sistema es $\mathcal{AS}_{N,f}[f \leq F]$.

Preparación y envío de eventos Cuando la capa de la aplicación (el cliente) recibe un mensaje, ésta la envía al algoritmo. En concreto la operación Broadcast() se encarga de esto. El mensaje es encapsulado 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 identificador del proceso (también único) y un número autoincremental. Después de esto, se establece la marca de 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 lo trata directamente, para recibirlo por primera vez, reenviarlo, procesarlo y posteriormente, si fuese el caso, entregarlo.

Recepción y tratado de eventos Cuando un proceso captura una acción de recepción de un evento, lo captura y lo envía al manejador: ReceiveHandler(). El manejador realiza los siguientes pasos:

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 recibido suficientes veces como para ser entregable. Recordemos que los canales son fiables, así, si un proceso completa las líneas 26 a 27 entonces todos los correctos lo recibirán. El evento se reenvía a todos los procesos menos a él mismo y por tanto, al menos $(N - F) - 1$ procesos (correctos) lo recibirán. Al ser estos procesos correctos retransmitirán el mismo evento por lo que un proceso se considerará entregable si $received[event.id].nor \geq (N - F) - 1$.

Por otro lado, si es la primera vez que se recibe el evento en cuestión (no está en el mapa de recibidos), y no ha sido ya entregado a la capa de aplicación [6], entonces se añade al mapa de recibidos, se inicia su número de recepciones en uno y se reenvía a todos los procesos menos al receptor. Observe que primero se envía y luego se almacena. Esto imita el comportamiento del protocolo URB para garantizar uniformidad en la entrega.

Ordenación y entrega de eventos El procedimiento OrderAndDeliverEvents() se encarga de esta parte. La idea del diseño de esta parte del algoritmo está inspirada en el componente de ordenación de EpTO [12]. Para su funcionamiento usamos las siguientes variables locales:

- *minTsOfNonDeliverable*: es el instante de marca de tiempo del evento más antiguo que NO es entregable.

[6] La variable *DeliveredEvents* puede modificarse para no tener que almacenar eventos ya entregados de manera indefinida. Esto es posible si se emplean canales FIFO. Basta guardar el último evento entregado a la aplicación por cada fuente. No incluimos en el código esta forma de recolección de basura por simplificar.

- *deliverableEvents*: conjunto de eventos que son entregables. Aquí se preparan los eventos antes de ser ordenados y entregados.

Dicho esto, el funcionamiento es el siguiente:

1. Se comprueba para cada evento recibido si es anterior al último evento entregado en orden total. Si es así, el evento ha llegado en desorden debido a la asincronía y por tanto, es entregado en desorden.
2. En cambio, si es posterior al último entregado y es entregable se añade al conjunto de eventos entregables.
3. 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.
4. 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.
5. Finalmente y al tener el conjunto de eventos realmente entregables, los ordenamos por sus marcas de tiempo y se entregan a la capa de la aplicación en orden total.

El comentario anterior es básicamente una lectura del código del procedimiento. En lo que sigue damos una visión un poco más precisa del funcionamiento del procedimiento OrderAndDeliverEvents() que será útil a la hora de presentar la corrección del algoritmo.

Consideremos un proceso p y un estado de ejecución de dicho proceso S justamente anterior a la ejecución del código OrderAndDeliverEvents() por parte del proceso p. En ese estado, $S.received$ son todos los eventos recibidos por p. El procedimiento OrderAndDeliverEvents() en realidad construye implícitamente cuatro conjuntos disjuntos. En lo que sigue α denota el valor $(N - F) - 1$, y $lastTS$ es una simplificación de $lastDeliveredTS$.

Definición 5 *Sea S el estado de ejecución de un proceso p justamente antes del comienzo de la ejecución del código OrderAndDeliverEvents(). Se definen los siguientes subconjuntos (disjuntos) en $S.received$:*

- *$S.U$ (Unordered): $u \in S.U$ si y sólo si $u.ts < S.lastTS$*
- *$S.ND$ (Non-Deliverable): $nd \in S.ND$ si y sólo si $S.lastTS < nd.ts \ \wedge \ S.received[nd].nor < \alpha$.*
  Sea, min el evento $min \in S.ND$ con la marca de tiempo más antigua, $S.lastTS < min.ts < nd.ts$ para todo $nd \in S.ND$.
- *$S.TOD$ (Total Order Deliverable): $e \in S.TOD$ si y sólo si $S.lastTS < e.ts < min.ts \wedge S.received[e].nor \geq \alpha$*
- *$S.AD$ (Await Deliver): $ae \in S.AD$ si y sólo si $S.lastTS < min.ts < ae.ts \ \wedge \ S.received[e].nor \geq \alpha$.*

Por las definiciones dadas, se cumple que el conjunto de mensajes recibidos es la unión de los conjuntos definidos anteriormente. Justamente, al final de la ejecución de OrderAndDeliverEvents(), el estado alcanzado S' en el proceso p cumple la siguiente propiedad.

Propiedad 1 *Sea S OrderAndDeliverEvents() S' la ejecución del procedimiento comenzando en el estado S del proceso p. Entonces, por la definición 5 y el código OrderAndDeliverEvents() del algoritmo 2 se cumple:*

1. $S.received = S.U \ \cup \ S.ND \ \cup \ S.TOD \ \cup \ S.AD$
2. $S'.received = S.received \setminus (S.U \ \cup \ S.TOD)$
3. $S'.DeliveredEvents = S.U \ \cup \ S.TOD$
4. $S.lastTS < S'.lastTS < min.ts$

La propiedad es muy simple de comprobar utilizando la definición 5 y siguiendo las líneas de código del procedimiento OrderAndDeliverEvents(). Debe observarse que $min.ts$ puede ser diferente en futuras ejecuciones del procedimiento, su valor depende del estado S. Todos los eventos en $S.U \ \cup \ S.TOD$ son finalmente entregados a la capa de la aplicación, los pertenecientes a $S.U$ por medio de U-Deliver() y los pertenecientes a $S.TOD$ se entregan en orden por medio de TO-Deliver().

Comentario 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 [16] 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 puramente reactivo y garantiza la entrega de mensajes gracias a que trabaja sobre canales que son fiables. EpTO [12] 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 puramente 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 cuándo un evento puede ser entregable.

En aTO-URB (algoritmo 2), al estar construido sobre canales fiables, no existe la necesidad de utilizar rondas ni TTL para diferenciar cuándo 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-F)-1 veces. Esto quiere decir que si un proceso recibe un evento al menos (N-F)-1 veces entonces una cantidad fija de nodos correctos también lo han recibido y por tanto es entregable. De esta forma, podemos afirmar que aTO-URB no sólo garantiza la entrega de mensajes sino que también garantiza que un gran número de mensajes sean entregados en orden total.

Como se ha dicho antes, aunque todos los nodos correctos entreguen los mismos mensajes prácticamente en el mismo orden, es posible que algunos mensajes simplemente no se entreguen 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.

4.3. Demostración de corrección

Consideremos el algoritmo aTO-URB (algoritmo 2). En dicho algoritmo consideramos que la operación Broadcast() se ejecuta atómicamente, y también consideramos que la ejecución de **upon receive** EVENT() se realiza de manera atómica. Esta última acción incluye la ejecución del código ReceiveHandler() que en su caso puede invocar (o no) el código de OrderAndDeliverEvents(), encargado de la entrega de mensajes a la capa de la aplicación por medio de los eventos U-Deliver() o TO-Deliver(). Si consideramos un proceso cualquiera $p \in \Pi$, a partir del estado inicial S_0 definido por las variables y sus valores iniciales (ver algoritmo 2 **variables**) una ejecución de dicho proceso será una secuencia $S_0\pi_1 S_1\pi_2$.... donde cada acción π_i es una de las indicadas anteriormente. La ejecución puede ser finita si el proceso falla, y en caso de que el proceso sea correcto, será una ejecución, en general, infinita si consideramos que la capa de la aplicación está enviando continuamente mensajes. El funcionamiento del algoritmo es reactivo y su implementación puede realizarse en lenguajes de este tipo como NodeJS (sección 5).

El sistema considerado tiene N procesos, todos conectados entre sí por medio de canales fiables. El sistema es asíncrono y se fija el número máximo de procesos que pueden fallar, F, con $F < N$. Por tanto, estamos considerando un sistema $\mathcal{AS}_{N,f}[f \leq F]$. El parámetro $\alpha = (N - F) - 1$ determina cuándo un evento es potencialmente entregable (ver líneas 56 y 57 en el algoritmo 2). En el resto de este trabajo consideramos $\alpha \geq 2$.

Para indicar que un estado S de un proceso conduce a la ejecución de un determinado código del proceso o que se alcanza un determinado estado posterior lo indicamos con el indicador $\rightrightarrows$, así $S \ \rightrightarrows \ S'$ determina que existe un fragmento finito de ejecución $S \ \pi_1 S_1 ... S_n \pi_{n+1} \ S'$ que comenzando en S termina en S'. En lo que sigue todas las propiedades y demostraciones hacen referencia al código del Algoritmo 2, aTO-URB.

Propiedad 2 *Sea p un proceso correcto y sea S un estado de p.*

$e \in S.received \rightrightarrows e \in S'.DeliveredEvents \vee$ *OrderAndDeliverEvents() es ejecutada por e (línea 24).*

Demostración: Si en un estado posterior $e \in S'.DeliveredEvents$ entonces $e \notin S'.received$. Todo evento entregado no puede volver a formar parte de *received* por la línea 25. Por tanto, consideremos que en todo estado posterior $e \notin S'.DeliveredEvents$. Como $e \in S.received$ entonces $e \notin S.DeliveredEvents$. Como el proceso p es correcto entonces en algún estado anterior se ha ejecutado las líneas 29 y 30, y previamente e ha sido enviado a α procesos correctos, ya que se envía a todos los procesos menos a sí mismo. Como los α procesos correctos recibirán dicho evento e al menos una primera vez (los canales son fiables) y de nuevo retransmiten el evento (líneas 26 y 27), el proceso p recibirá al menos α copias del evento, ejecutando α veces la acción **upon receive** EVENT(e, **from** senderId). Como hemos asumido

que $\alpha \geq 2$, desde el punto de vista de p, en algún estado posterior $S'.received[e].nor \geq \alpha$ y por la línea 23 se ejecutará OrderAndDeliverEvents() en nombre de dicho evento e. La propiedad se cumple. □

Propiedad 3 *Sea p un proceso correcto y sea S un estado de p.*

$e \in S.ND \rightrightarrows e \in S'.DeliveredEvents \ \vee \ e \in S'.AD$

Demostración: Si $e \in S.ND$ entonces $e \in S.received$. Por la misma argumentación a la dada en la Propiedad 2, consideremos que en todo estado posterior $e \notin S'.DeliveredEvents$. Por la demostración de la Propiedad 2, en algún estado posterior $S''.received[e].nor \geq \alpha$ y necesariamente se ejecutará OrderAndDeliverEvents() en nombre de dicho evento e. Como $e \notin S'.DeliveredEvents$ en todo estado posterior a S, entonces $e \notin (S''.U \cup S''.TOD)$, además $e \notin S''.ND$ ya que $S''.received[e].nor \geq \alpha$. Como por la definición 5 de los conjuntos dados forman una partición de $S''.received$, y $S''.lastTS < e.ts$ ($e \notin S''.U$), entonces necesariamente $e \in S''.AD$, y la propiedad se cumple. □

Lema 2 *Sea p un proceso correcto y sea S un estado de p.*

$ae \in S.AD \rightrightarrows ae \in S'.DeliveredEvents$

Demostración: Supongamos que la propiedad es falsa en las ejecuciones que realiza el proceso p a partir del estado S. El evento $ae \in S.AD$ y por tanto, $ae \in S.received$. Por la suposición realizada, para todo estado de ejecución S', posterior a S, $ae \notin S'.DeliveredEvents$ y $ae \in S'.AD$. El hecho de que $ae \notin S'.DeliveredEvents$ impide que $ae \in (S'.U \cup S'.TOD)$ y por las condiciones de $S.AD$ no es posible tampoco $ae \in S'.ND$. Entonces, se cumple lo siguiente: existe un evento $f_i \in S.ND$ con $S.lastTS < f_i.ts < ae.ts$ y $S.received[f_i] < \alpha$. Justamente, la existencia de este evento f_i produce que $ae \in S.AD$.

Paso de la demostración para el evento f_i: Dado que $f_i \in S.ND$, obviamente $f_i \in S.received$. Por la Propiedad 3, o bien existe un estado posterior S_1 tal que (A) $f_i \in S_1.DeliveredEvents$ o (B) $f_i \in S_1.AD$

Caso (A): $f_i \in S_1.DeliveredEvents$ entonces $S.lastTS \leq S_1.lastTS < f_{i+1}.ts < ae.ts$ con $f_{i+1} \in S_1.ND$ y $ae. \in S_1.AD$. El hecho de que después de entregarse f_i se mantenga $ae. \in S_1.AD$ obliga la existencia de un evento distinto a f_i con las propieades anteriores, $f_{i+1} \in S_1.ND$.

Caso (B): $f_i \in S_1.AD$ entonces, dado que $f_i.ts < ae.ts$, necesariamente existe otro evento f_{i+1} con las condiciones $f_{i+1} \in S_1.ND$ y que verifica $S.lastTS \leq S_1.lastTS < f_{i+1}.ts < f_i.ts < ae.ts$. La existencia de este evento mantiene a $f_i \in S_1.AD$ y a $ae \in S_1.AD$.

En ambos casos (A) y (B), el nuevo evento considerado f_{i+1} cumple las mismas condiciones que el evento f_i en el «paso de la demostración para el evento f_i». La iteración continuada de la demostración por «paso de la demostración para el evento f_i» por las sucesivas sustituciones de f_i conducen a que en el intervalo $[S.lastTS, ae.ts]$ aparezcan un número infinito de marcas de tiempo correspondientes a la sucesión de eventos distintos $f_i, f_{i+1}, f_{i+2}, ...$ Como entre dos marcas de tiempo distintas sólo puede

haber un número finito de marcas de tiempo creadas en el sistema se produce una contradicción. Por lo tanto, en contra de la hipótesis de partida, el lema es cierto. □

Lema 3 *Sea p un proceso correcto y sea S un estado de ejecución de p. Se cumplen las siguiente propiedades de progreso:*

$$e \in S.received \rightrightarrows e \in S'.DeliveredEvents \quad (1)$$

$$e \in S.received \wedge S.lastTS < e.ts \rightrightarrows e \in S'.TOD \quad (2)$$

$$e \in S.received \wedge S.lastTS < e.ts \rightrightarrows \ TO\text{-}Deliver(e) \ es\ ejecutado \quad (3)$$

Demostración: 1.- Para demostrar la propiedad de progreso (1), asumimos que $e \in S.received$ y para todo estado posterior S', $e \notin S'.DeliveredEvents$. Por la Propiedad 2, se ejecuta S_1 OrderAndDeliverEvents() S_2 para el evento e en un estado S_1, luego $e \in S_1.U$ o $e \in S_1.TOD$ y por su efecto, $e \in S_2.DeliveredEvents$. Esto no es posible por la suposición inicial. Luego, $e \in S_1.ND$ o $e \in S_1.AD$. En el primer caso, por la Propiedad 3, se cumple que en un estado S_3, $e \in S_3.AD$. Por el Lema 2, $e \in S_4.DeliveredEvents$ en contra de la suposición inicial. En el segundo caso sucede lo mismo, por el Lema 2, $e \in S_4.DeliveredEvents$ en un estado posterior a S. En todos los casos llegamos a una contradicción con la suposición inicial luego (1) se cumple. Todo evento recibido por un proceso correcto será entregado a la aplicación.

2.- Para demostrar la propiedad (2), asumimos que $e \in S.received \wedge S.lastTS < e.ts$ y para todo estado posterior S', $e \notin S'.TOD$. Por (1), $e \in S'.DeliveredEvents$ por lo que existe un estado S_2 tal que $e \in S_2.U$, ya que si no se entrega en orden total se entrega en desorden por U-Deliver(e). Existe un primer estado S_1 donde por la ejecución de S_1 OrderAndDeliverEvents() S_2, justamente produce que $e.ts < S_2.lastTS$ siendo $S_1.lastTS < e.ts$. El último evento toe que fija el valor $S_2.lastTS = toe.ts$ en la línea 53 del algoritmo 2 no puede verificar $toe \in S_1.TOD$ ya que por la condiciones de e se debe cumplir $e \in S_1.ND$. Supongamos que $toe \in S_1.TOD$ entonces $S_1.lastTS < e.ts < toe.ts$. Como $e.ts \notin S_1.TOD$ por hipótesis en la demostración, necesariamente $e \in S_1.ND$. Por lo que, $toe.ts \in S_1.AD$.

En conclusión, $e \in S'.DeliveredEvents$ y no existe un estado posterior con $e \in S'.U$. Luego, existe un estado posterior que verifica $e \in S'.TOD$, y (2) se cumple.

3.- La propiedad (3) es consecuencia de (1) y (2). □

El Lema 3 establece el progreso en la entrega de los procesos correctos. Por una parte (1) establece que todo evento recibido por un proceso correcto será tarde o temprano entregado y (3) establece que un evento recibido con una marca de tiempo posterior a la marca de tiempo del último evento entregado por el proceso en orden total también será entregado en orden total. Como la marca de tiempo inicial se establece en $(0, 0)$ siempre hay un primer evento que se entrega en orden total ya que cualquier marca de tiempo generada en el sistema es mayor que $(0, 0)$. Los eventos posteriores a este evento entregado en orden total también serán entregados en orden total. Finalmente, demostramos el teorema de corrección.

Teorema 2 *Sea un sistema distribuido asíncrono $\mathcal{AS}_{N,f}[f < F]$ con un máximo de F procesos que pueden fallar y sea $\alpha = (N - F) - 1$. El algoritmo 2, aTO-URB, satisface las siguientes propiedades dadas en la Definición 4:*

- **Validez**. *Si un proceso X-deliver un evento e, entonces e ha sido Broadcast por algún proceso.*
- **Integridad**. *Un proceso X-deliver un evento e como mucho una vez.*
- **Orden Total**. *Si dos procesos p y q TO-deliver los eventos e y e', entonces, p TO-deliver el evento e antes que el evento e'* **si y sólo si** *q TO-deliver el evento e antes que el evento e'.*
- **Terminación**. *(a) Si un proceso correcto realiza Broadcast de un evento e, o (b) si un proceso realiza X-deliver de un evento e, entonces todos los procesos correctos realizan X-deliver del evento e.*

  siendo X-deliver una de las dos posibilidades de entrega, U-deliver o TO-deliver.

Demostración: Las propiedad de **Validez** es trivial en cuanto que sólo se entregan eventos enviados previamente por algún Broadcast() ya que los canales son fiables y no hay creación espontánea de eventos. La propiedad de **Integridad** se cumple debido a que para que un evento e sea X-deliver(e), requiere que $e \in received$. Una vez entregado $e \in DeliveredEvents$ y por la condición en la línea 25 del algoritmo aTO-URB no vuelve a formar parte de $received$ en ningún estado posterior. En resumen los eventos se entregan como mucho una sola vez.

La propiedad de **Orden Total** tiene una demostración muy similar a la dada en la demostración del Lema 1. Si un proceso p realiza TO-deliver de primero, el evento e, y luego e' necesariamente $e.ts < e'.ts$. Si q entrega también e y e' por medio de TO-deliver, en diferente orden entonces $e'.ts < e.ts$ pero esto es una contradicción ya que las marcas de tiempo una vez establecidas en los eventos no cambian. La propiedad de **Orden Total** se cumple.

En cuanto a la propiedad de **Terminación**, si se da el caso (a) o (b) por la ejecución de las líneas 26 y 27 (que se han tenido que ejecutar (a) porque el proceso es correcto, (b) porque el evento ha sido entregado y pertenecía a $received$) entonces todos los procesos correctos recibirán dicho evento por el hecho de que los canales son fiables. Por tanto, existe, para cada proceso correcto p, un estado Sp tal que $e \in Sp.received$. Por el Lema 3, (1) se cumple que existe un estado posterior Sp' para cada proceso correcto, tal que $e \in Sp'.DeliveredEvents$, con lo que el evento e ha sido entregado por todos los procesos correctos. □

5. Implementación en NodeJS

5.1. Por qué en NodeJS

Dado el algoritmo, la elección del lenguaje no ha sido difícil. Nos encontramos en un entorno puramente reactivo, lleno de eventos y de envío y recepción de mensajes continuos, donde todo lo que sucede es causa y consecuencia, donde todo es acción-reacción. Se trata de un paradigma muy distinto a lo habitual, no se trata de una aplicación con el objetivo de realizar una operación simple y bloqueante sino de una aplicación distribuida que funciona de forma asíncrona y que trata de poner de acuerdo a múltiples nodos interconectados entre sí, comunicándose mediante mensajes que se envían de forma concurrente.

NodeJS es un lenguaje asíncrono, con un paradigma que gira en torno a los eventos dado que está diseñado para una arquitectura de cliente-servidor, en donde NodeJS actúa con el papel de servidor esperando y respondiendo a las múltiples y simultáneas consultas de los clientes. En este lenguaje toda acción es no bloqueante y asíncrona, excepto en algunos casos en los que es posible y deliberado realizar acciones bloqueantes. Es un lenguaje dinámico, interpretado, ligero, muy rápido y eficiente, y es por eso por lo que ha sido escogido para la implementación de este proyecto.

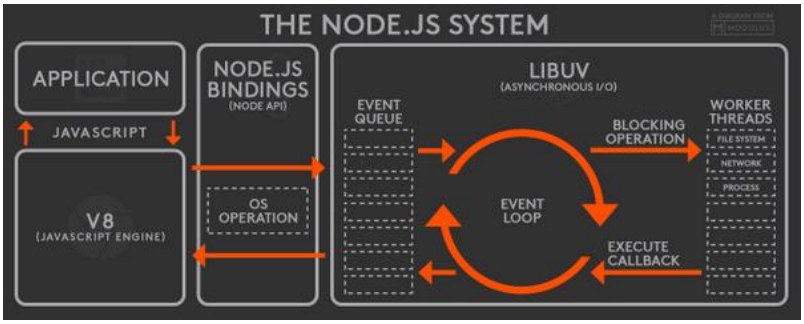

Figura 6: NodeJS Event Loop

En la figura 6 se puede ver el ciclo de ejecución de NodeJS. La ilustración muestra cómo se trata de una cola de eventos no bloqueante en la cual se realiza continuamente lo siguiente: primero se recibe un evento de la capa superior o aplicación, luego este evento se añade a la cola hasta que es desencolado y tratado de forma no bloqueante (con tratado nos referimos a que se ejecuta su código correspondiente) desencadenando o no nuevos eventos que serán añadidos a la cola en orden para que sean tratados en un futuro. A esto se le llama function - callback.

Esto quiere decir que la ideología de las funciones de este lenguaje consiste en programarlas de forma que ejecuten un código callback cuando hayan terminado de ejecutarse. Es importante ver la diferencia con otros lenguajes que son síncronos: en vez de ejecutar una función, terminar y ejecutar la siguiente de forma secuencial, NodeJS activa simultáneamente múltiples funciones, preparando nuevas funciones a ejecutar como consecuencia de las ejecuciones de las anteriores.

5.2. La librería ZeroMQ

Aunque el lenguaje es imprescindible, también lo es la comunicación entre los nodos a través de la red, y para esto son necesarios los sockets. ZeroMQ es una librería que se especializa en el envío de mensajes a través de la red y ofrece muchas facilidades a la hora de llevarlo a la implementación. La librería, además de estar disponible en múltiples lenguajes, ofrece la posibilidad de crear sockets que permiten el envío atómico de mensajes gracias a que la librería lo gestiona internamente y garantiza que todo mensaje es entregado o no entregado, sin intermedios.

Además nos da la posibilidad de comunicar sockets a través del mismo proceso, entre procesos, a través de procesos de diferentes máquinas mediante TCP y multicast. Para esto nos proporciona múltiples patrones de comunicación como son request-reply, fan-out, publisher-subscriber, y task distribution. El más comúnmente usado es el patrón request-reply por los servidores, entorno en el que NodeJS está cada vez más presente hoy en día.

Con respecto a nuestro proyecto, utilizaremos dos sockets de ZMQ llamados *router* y *dealer*. Estos sockets nos permitirán interconectar todos los nodos de nuestra red entre sí mediante canales *router-dealer* bidireccionales. Cada nodo dispondrá de un router para recibir comunicación de los demás, y de tantos dealers como nodos a los que necesite comunicarse. Con esto conseguimos una única comunicación bidireccional por cada par de nodos distintos existentes en la red, lo que hemos denominado en repetidas ocasiones como canales fiables.

En resumen, ZeroMQ es una librería muy utilizada hoy en día que está pensada directamente para su uso en comunicación de aplicaciones distribuidas y que está construida sobre un núcleo puramente asíncrono.

5.3. Despliege de la implementación

Como ya se ha explicado en la introducción, el despligue se llevará a cabo en contenedores docker (Linux) para simular la comunicación entre máquinas totalmente aisladas entre sí. Docker es un software extremadamente utilizado en lo que a aplicaciones se refiere, y es increíblemente cómodo para realizar pruebas sin correr riesgos innecesarios.

El despliegue de la aplicación se realizará de la siguiente forma: primero se arrancarán todos los contenedores previamente preparados (habiendo instalado el programa de la aplicación e instalado las librerías necesarias para su ejecución). Después se configurará el programa instalado en cada contenedor según su dirección IP, la cantidad de nodos que se desee lanzar en cada contenedor y cada uno de sus puertos, así como de las direcciones IP del resto de contenedores donde se lanzarán el resto de nodos. Una vez hecho esto, se inician los programas en todos los contenedores e inmediatamente todos los nodos crearán los sockets necesarios y se conectarán entre sí.

Finalmente, ya podemos enviar y recibir mensajes, o realizar pruebas con envío de mensajes aleatorios según las opciones que ofrece el programa al ejecutarse. Su uso es relativamente simple, y tanto el código como el programa en sí está debidamente explicado y documentado en el GIT que se ha utilizado para el control de versiones y que se ha mencionado en la introducción del proyecto.

5.4. Arquitectura

La arquitectura de la red es tan simple como un grafo completo. Todos los nodos están directamente conectados al resto, mediante los canales bidireccionales que hemos explicado con anterioridad.

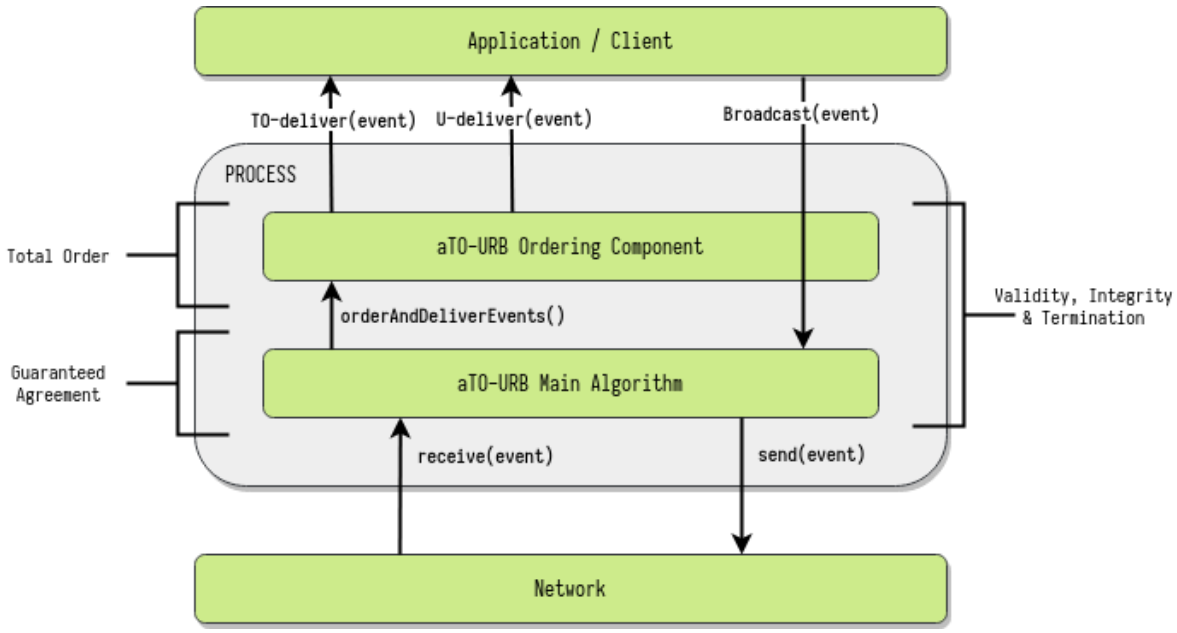

Figura 7: Arquitectura aTO-URB

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 7 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.

5.5. Diseño del servicio

En el diseño 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 aTO-URB y su implementación en NodeJS.

A continuación se presenta una breve explicación de cada módulo del programa y de las funciones que lo componen:

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.

- Función **init**: escucha los eventos entregados en orden y en desorden emitidos por el proceso.
- Función **connect**: indica al proceso la dirección IP y puerto de otro cliente.
- Función **close**: llama a la función close interna del proceso.
- Función **broadcast**: llama a la función broadcast interna del proceso.

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.

- Función **init**: bindea la dirección del router para empezar a escuchar a otros procesos.
- Función **connect**: crea e inicializa un dealer y lo conecta con otro proceso.
- Función **close**: cierra los sockets del router y todos los dealers del proceso.
- Función **broadcast**: encapsula el mensaje en un evento y se lo pasa al algoritmo aTO-URB.
- Función **newEventId**: lleva un contador autoincremental y devuelve un nuevo ID.

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.

- Función **copy**: devuelve una copia del objeto con el mismo mensaje.
- Función **serialize**: serializa el objeto recursivamente y devuelve un JSON.
- Función **deserialize**: deserializa un JSON y lo convierte en un objeto mensaje.

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.

- Función **copy**: devuelve una copia del objeto con el mismo evento.
- Función **serialize**: serializa el objeto recursivamente y devuelve un JSON.
- Función **deserialize**: deserializa un JSON y lo convierte en un objeto evento.

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.

- Función **getTime**: devuelve el tiempo actual (dependiendo de si se usa el reloj lógico o global).
- Función **updateClock**: actualiza el tiempo actual (sólo cuando se usa reloj lógico).

ATOURB 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 este módulo.

- Función **broadcast**: asigna los valores indicados al evento y se lo envía a sí mismo mediante el manejador.
- Función **isLastDeliveredOfAnyProcess**: comprueba si el evento recibido ha sido el último entregado anteriormente con cualquier proceso origen.
- Función **receiveHandler**: implementa el hanlder visto en el algoritmo que maneja un evento cuando es recibido.
- Función **orderAndDeliverEvents**: implementa el componente de ordenación del algoritmo.
- Función **isDeliverable**: comprueba si un evento ha sido recibido suficientes veces como para ser entregable.

Index Programa principal. Este módulo es el ejecutable del programa, se trata de un test configurable de número de nodos variable diseñado para realizar pruebas. Posee unas variables globales que ayudan a configurar el programa y su funcionamiento, como son el tiempo entre envío de mensajes aleatorios, la cantidad de nodos a desplegar, la configuración de la network mediante un fichero JSON, los comandos a recibir por consola, y alguna herramienta más.

NOTA: No obstante, la clase Cliente también posee una parte ejecutable para poder realizar pruebas más especificas, pudiendo así lanzar clientes individualmente y conectándolos entre ellos mediante comandos en consola durante la ejecución.

6. Pruebas y resultados de rendimiento

En esta sección se expone la forma en la que las pruebas han sido realizadas, los pasos llevados a cabo y los resultados obtenidos.

6.1. Clúster y pruebas

Las pruebas se realizaron en el clúster de máquinas de la siguiente forma: primero, se crearon tantos contenedores como máquinas (uno por cada máquina), después se fijó el número de nodos a desplegar en cada contenedor (variable según las pruebas) y se ajustaba la configuración del programa a lanzar según dicho número y las direcciones y puertos de los nodos del resto de contenedores. Una vez hecho esto, se iniciaba el programa en cada contenedor de forma simultánea, de forma que los clientes se conectaban todos entre sí y se podía comenzar el envío de mensajes automatizado.

Este envío de mensajes sucedía de la siguiente manera: cada cierto tiempo Δ definido, los programas lanzados en los contenedores seleccionaban un cliente aleatorio y le enviaban una señal para que generase y enviase un mensaje automáticamente.

Hay que tener en cuenta que estas no son las condiciones ideales para realizar las pruebas del algoritmo, puesto que el clúster está conectado de forma que todas las máquinas están en la misma red local, lo que hace que la conexión sea demasiado rápida y estable como para que se produzcan asincronías que puedan llegar a afectar significativamente al envío de mensajes. No obstante, se realizaron pruebas a diferentes escalas (10, 20, 30 nodos) y se forzaron los canales (disminuyendo la frecuencia del envío de mensajes lo máximo posible) para intentar compensar esto.

A diferencia del resto de pruebas, estas son las que más problemas dieron con diferencia. A menudo existían 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 di con una version perfectamente funcional. A continuación veremos los resultados obtenidos.

6.2. Resultados

Recordamos que el objetivo es garantizar la entrega de mensajes al mismo tiempo que maximizar el número de mensajes entregados en orden total (garantía parcial). 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 cuyo número de procesos correctos es mayor al de fallos admitidos, 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.

En las siguientes tablas se muestran los resultados de las pruebas realizadas. En las filas se pueden diferenciar las ejecuciones según el número de procesos fallidos FP (faulty processes) durante la ejecución, y en las columnas se pueden diferenciar las ejecuciones según el Δ utilizado (tiempo transcurrido entre envíos de mensajes aleatorios automatizados).

Naturalmente N y F se refieren a el número de nodos en la red y al número de fallos admitidos respectivamente. Como resultados obtenemos tanto el porcentaje de mensajes entregados en el mismo orden por todos los nodos correctos y la media de la latencia de los mismos.

N = 10, F = 4

FP / Δ	500 ms	300 ms	150 ms	75ms	30 ms
0/10 FP	100 % / 3.54 ms	100 % / 3.28 ms	100 % / 2.80 ms	100 % / 2.42 ms	100 % / 2.21 ms
1/10 FP	100 % / 3.34 ms	100 % / 2.98 ms	100 % / 2.45 ms	100 % / 2.12 ms	100 % / 2.12 ms
2/10 FP	100 % / 3.16 ms	100 % / 3.03 ms	100 % / 2.97 ms	100 % / 2.15 ms	100 % / 2.01 ms
3/10 FP	100 % / 3.16 ms	100 % / 2.95 ms	100 % / 2.57 ms	100 % / 2.07 ms	100 % / 1.91 ms
4/10 FP	100 % / 2.92 ms	100 % / 2.54 ms	100 % / 2.29 ms	100 % / 2.07 ms	100 % / 1.82 ms

N = 20, F = 9

FP / Δ	500 ms	300 ms	150 ms	75ms	30 ms
0/20 FP	100 % / 8.86 ms	100 % / 8.30 ms	100 % / 9.05 ms	100 % / 8.41 ms	100 % / 7.78 ms
2/20 FP	100 % / 9.70 ms	100 % / 7.79 ms	100 % / 8.25 ms	100 % / 7.90 ms	100 % / 7.12 ms
4/20 FP	100 % / 8.18 ms	100 % / 7.78 ms	100 % / 6.93 ms	100 % / 7.13 ms	100 % / 6.57 ms
6/20 FP	100 % / 7.65 ms	100 % / 7.53 ms	100 % / 6.73 ms	100 % / 6.84 ms	100 % / 6.21 ms
8/20 FP	100 % / 6.84 ms	100 % / 6.64 ms	100 % / 6.11 ms	100 % / 6.07 ms	100 % / 5.86 ms

N = 30, F = 14

FP / Δ	500 ms	300 ms	150 ms	75ms	30 ms
0/30 FP	100 % / 19.38 ms	100 % / 20.65 ms	100 % / 19.45 ms	100 % / 18.32 ms	99.6 % / 12.02 ms
3/30 FP	100 % / 16.77 ms	100 % / 19.37 ms	100 % / 17.50 ms	100 % / 17.04 ms	99.7 % / 13.35 ms
6/30 FP	100 % / 16.71 ms	100 % / 16.96 ms	100 % / 16.64 ms	100 % / 15.93 ms	99.8 % / 13.35 ms
9/30 FP	100 % / 15.17 ms	100 % / 14.19 ms	100 % / 15.46 ms	100 % / 14.38 ms	99.9 % / 12.45 ms
12/30 FP	100 % / 14.40 ms	100 % / 13.43 ms	100 % / 13.68 ms	100 % / 13.45 ms	100 % / 11.58 ms

Se puede observar que la latencia disminuye según aumenta el número de procesos fallidos puesto que el tráfico en la red disminuye. Esto sucede porque la entrega de mensajes se realiza antes, dado que los mensajes no necesitan llegar a todos los nodos para ser entregados.

6.3. 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 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. Esto es inevitable debido a la incontrolable asincronía. Respecto al algoritmo, se ha de tener en cuenta que aunque no garantiza el orden total, una alta cantidad de mensajes son entregados en orden total.

6.4. Recursos utilizados

Para el desarrollo del proyecto he utilizado diversos recursos. Para el control de versiones he utilizado git, y para la implementación Visual Studio Code. Ambos entornos relativamente nuevos para mí. 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.

7. Conclusiones

Inicialmente este proyecto no tenía un objetivo tan específico, pero conforme fue avanzando tanto mi tutor como yo acabamos por llegar hasta el desarrollo de este interesante algoritmo. Durante el proceso he aprendido infinidad de cosas, desde la teoría de los algoritmos distribuidos, protocolos de consenso y orden total en profundidad hasta dominar un lenguaje asíncrono. He implementado muchas versiones diferentes de los algoritmos estudiados y realizado muchas pruebas hasta conseguir una versión funcional y estable.

La implementación de URB en sí misma resultó sorprendentemente sencilla. La implementación de EpTO en cambio me dió muchas más complicaciones hasta que conseguí buenos resultados, además de que tuvimos que corregir un error presente en el artículo original del algoritmo. Las librerías que utilicé para el desarrollo del proyecto fueron muy sencillas de aprender debido a su extensa documentación y a su fácil instalación, además de que ya las había utilizado anteriormente durante las becas de colaboración a las que pertenecí durante mis dos últimos años en la universidad.

Un entorno realmente nuevo para mí fue la creación, uso, y manejo de un clúster de máquinas sobre un sistema Proxmox y contenedores Docker. Al principio resultó algo complejo pero con el tiempo pude ver el increíble potencial que este tipo de tecnologías proporcionan. Rápidamente y sin darnos cuenta ya teníamos un clúster de máquinas y cantidad de contenedores perfectamente configurados para realizar nuestras pruebas.

En cuanto al desarrollo de la implementación hemos de decir que estamos muy contentos con cómo ha terminado el proyecto y con los resultados que hemos obtenido. Creemos que el algoritmo puede ser realmente útil para muchas aplicaciones distribuidas que se basen en la comunicación de nodos, y esperamos que el trabajo realizado sirva incluso de objeto de estudio para futuros estudiantes de esta universidad y de otras.

Referencias

[1] F. Schneider. Implementing fault tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4, 299—319, 1990.

[2] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12, 3, 463–492, 1990.

[3] K. P. Birman et al. Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. 9, 3, 272—314, 1991.

[4] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY., 1991.

[5] C. Marchetti et al. Fully Distributed Three-Tier Active Software Replication. IEEE Trans. on Parallel and Distributed Systems, 17, 7, 633-645, 2006.

[6] T. D. Chandra y S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43, 2, 225–267, 1996.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. Journal of the ACM 43, 4, 685—722, 1996.

[8] M. J. Fischer, N. A. Lynch, and S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2, 374—382, 1985.

[9] L. Lamport. Fast Paxos. Distributed Computing, 19, 6, 79–103, 2006.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 7, 558—565, 1978.

[11] Diego Ongaro. Consensus: Bridging Theory and Practice. Doctoral Thesis. Stanford University, 2014.

[12] Miguel Matos, Hugues Mercier, Pascal Felber, Rui Oliveira, José Pereira. EpTO: An Epidemic Total Order Algorithm for Large-Scale Distributed Systems. Middlewar 2015, Pags 100-111.

[13] 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*.

[14] Jim. R Wilson. Node.js the Right Way. Practical, Server-Side Javascript That Scales. The Pragmatic Programmers, edited by Jacquelyn Carter.

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

[16] Michel Raynal. Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. Morgan&Claypool Publishers, 2010.

[17] X. Defago et al. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. ACM Computing Surveys, 36, 4, 372—421, 2004.