Protocolo adaptativo de difusión de mensajes con orden de entrega aproximado y su aplicación en sistemas clave-valor

Trabajo Fin de Máster

Autor:
Iulen Salinas Lizoain

Director:
Aitor González de Mendivil Grau

Master Universitario en Ingeniería Informática

Departamento de Estadística, Informática y Matemáticas
Universidad Pública de Navarra
Pamplona, Navarra, España
9 de junio de 2022

Índice

1. Introducción

El problema de la difusión de mensajes entre un grupo de procesos en entornos tolerantes a fallos es un problema que aparece en un gran número de problemas de coordinación distribuida. La *difusión fiable* de los mensajes (*Reliable Broadcast*) permite que los procesos correctos en el sistema (aquellos que no han fallado en la ejecución) reciban el mismo conjunto de mensajes. Esta propiedad es lo suficientemente débil para que se pueda construir una solución a la difusión fiable en un entorno asíncrono con un número arbitrario de fallos. Aún siendo una propiedad débil tiene utilidad. Recientemente se han desarrollado un gran número de sistemas de almacenamiento distribuido que se construyen en base a la propiedad de *consistencia eventual* [1]. Básicamenente, las operaciones de lectura pueden devoler cualquier valor que se ha escrito en el sistema pero, en ausencia de escrituras, el estado del sistema de almacenamiento de datos converge al mismo estado, y a partir de ese punto, todas las lecturas devuelven el mismo valor con independencia del lugar donde se ejecuten. Este comportamiento se puede implementar con un protocolo de difusión fiable y con un mecanismo de escrituras que asegure que la más *reciente* es la que tiene efecto [2].

El problema de la difusión fiable en entornos con fallos ha sido tratado ampliamente en la literatura de sistemas distribuidos. Los trabajos iniciales se encuentran en la década de los 80, por ejemplo [3][4]. El trabajo de Hadzilacos y Toueg [5] es un excelente resumen de la difusión fiable y problemas relacionados con la difusión. Este trabajo presenta las limitaciones teóricas y extensiones prácticas que podemos encontrar en relación con este tipo de comunicación a grupo. Hadzilacos y Toueg muestran como se implementan protocolos de difusión fiable con las propiedades adicionales de entrega *fifo* y entrega en *orden causal*, utilizando de manera modular un protocolo básico de difusión fiable. De esta manera se amplía la calidad de servicio de la comunicación a grupo en un entorno asíncrono con fallos.

Entre las extensiones más atractivas de aumento de calidad de servicio a la difusión fiable encontramos la posibilidad de que los mensajes se entregen a todos los procesos correctos en el mismo orden (propiedad de *orden total*). Esto es deseable, por ejemplo, cuando interesa obtener un sistema de replicación tolerante a fallos que ofrezca *consistencia atómica*. Con una *difusión fiable en orden total*, es bastante simple implementar el esquema de replicación del estado de la máquina (*state machine replication*). Sin embargo, en un sistema distribuido asíncrono y con fallos de parada, esta propiedad no puede ser implementada. El resultado de *imposibilidad FLP* [6] determina que el problema del *acuerdo distribuido* es imposible implementarlo con un algoritmo determinista en este tipo de sistemas. La razón de la imposibilidad radica en que no es posible diferenciar a un proceso que ha fallado de un proceso correcto que va extremadamente lento en su comportamiento. Se puede demostrar que la difusión fiable en orden total es equivalente a problema del acuerdo distribuido [5] por lo que ambos no son implementables en sistemas asíncronos con fallos de parada. No obstante, la implementación es posible en sistemas que son *parcialmente síncronos*, ver por ejemplo el trabajo seminal de Chandra-Toueg [7]. En un sistema parcialmente sícrono hay un tiempo desconocido a partir del cual el sistema se comporta de forma síncrona: los mensajes que se envían entre los procesos se entregan antes de un cierto intervalo de tiempo fijo (aunque desconocido). Dependiendo del tipo de solución también se require alguna limitación en el número de procesos que pueden fallar en una ejecución. Normalmente, las soluciones al acuerdo distribuido requieren que una mayoría de los procesos sean correctos [7]. El trabajo de Defago [8] presenta un excelente resumen de los mecanismos que se han utilizado a lo largo de los años para obtener protocolos de difusión fiable en orden total.

Hemos indicado anteriormente que los sistemas de replicación que tratan de obtener una con-

sistencia fuerte (atómica) emplean algún tipo de mecanismo de acuerdo distribuido o algoritmo de difusión en orden total. Por ejemplo, Marchetti [9] propone un protocolo de replicación activa utilizando la difusión en orden total. Incluso, en el caso de utilizar replicación con consistencia eventual, para algunas operaciones se necesita que haya un comportamiento atómico (*atomic visivility*). Por ejemplo, en la solución propuesta por Zhao [10] se combina un protocolo de difusión fiable y un protocolo de consenso (RAFT). Este último para obtener el comportamiento atómico de ciertas operaciones. Es un hecho bien conocido que cuando la difusión fiable se implementa en una red de área local (LAN), los mensajes difundidos se entregan casi simultáneamente a todos los receptores, y en consecuencia, la entrega de mensajes a los diferentes receptores en desorden es un evento bastante poco frecuente. Esta idea se aprovecha, por parte de los diseñadores de protocolos de replicación fuerte, de una manera optimista: (i) se entrega el mensaje difundido y la aplicación lo ejecuta; (ii) mediante otro protocolo de orden total se espera la confirmación del orden de los mensajes ya entregados de forma optimista; (iii) aquellos mensajes ya entregados, pero en desorden, obligan a retrotraer la acción ya ejecutada. Esta manera optimista de proceder es eficiente cuando la coincidencia entre el orden optimista y el producido por un protocolo de orden total coinciden la mayor parte del tiempo. F. Pedone utiliza esta técnica optimista en sistemas transaccionales [12] [13].

En los ejemplos de replicación indicados su funcionamiento se basa en un protocolo de acuerdo distribuido o bien, un protocolo de orden total. Los diseñadores deben ser cuidadosos a la hora de elegir la zona del sistema distribuido donde se despliegan los protocolos de orden total (o consenso). Se necesita desde el punto de vista teórico que sea una región con sincronía parcial (para garantizar la entrega de los mensajes), lo que en la práctica se traduce en regiones donde la mayor parte del tiempo su comportamiento es síncrono. Es por tanto razonable, que estos protocolos se desplieguen en un cluster de máquinas sobre LAN en una infraestructura propietaria, o que se elija un centro de datos (DC), en una región geográfica concreta, de los ofrecidos por proveedores de infraestructura como pueden ser Amazon AWS. Mientras que sobre una LAN el orden total espontáneo se produce de forma natural por la propia arquitectura de una red LAN, en un DC podemos encontrar un menor orden espontáneo en la entrega debido a *intercambios* (*swaps*) o pérdidas de paquetes (*packet loss*). Los intercambios suceden debido a disparidades en las distancias entre enviadores y receptores, mientras que las pérdidas de paquetes casi nunca suceden en la red del DC, y si ocurren, es porque el núcleo del sistema operativo de la máquina está sobrecargado cuando la tasa de llegada de paquetes es excesivamente alta.

Este proyecto comparte las ideas de los trabajos relativos a los protocolos de difusión fiable en orden total optimistas sin embargo, en nuestro caso, no planteamos ninguna autoridad[1] que ratifique o invalide el orden de la entrega de los mensajes. Nuestra idea de partida para el diseño es básicamente que los procesos observen localmente si han entregado los mensajes en un cierto orden o no, donde el orden entre los mensajes es establecido mediante un mecanismo simple de marcas de tiempo (*timestamp*). A partir de ese diseño muy simple, estudiamos hasta qué punto retrasar la entrega de los mensajes tiene un efecto positivo en la tasa de mensajes entregados en orden total en todo el sistema, y si esas latencias introducidas afectan significativamente al comportamiento del sistema.

Uno de los objetivos de este proyecto es estudiar el estado del arte de la difusión fiable de mensajes para poder diseñar un algoritmo que garantice un orden de entrega aproximando en un sistema distribuido real. Posteriormente, el algoritmo se desplegará en un Data Center de AWS (*Amazon Web Services*) y se someterá a unas pruebas de carga para comprobar qué precisión

[1]Por autoridad entendemos el protocolo de orden total subyacente que ratifica o no el orden de la entrega ofrecido por un protocolo básico de difusión fiable.

tiene a la hora de ordenar mensajes en un entorno de producción.

Finalmente, se plantea utilizar el algoritmo diseñado y testeado para construir un sistema de almacenamiento distribuido de tipo *clave-valor* con garantías de consistencia eventual en un entorno productivo real.

El resto de este trabajo se organiza de la siguiente forma: en la sección 2 se definen las bases de la difusión fiable de mensajes (*Reliable Broadcast*). En la sección 3 se propone un algoritmo de orden de entrega aproximado. En la sección 4 se diseña e implementa un sistema distribuido de clave-valor utilizando el algoritmo visto en el apartado anterior. Finalmente, en el apartado 5 se presentan las conclusiones obtenidas de la realización de este proyecto.

2. Reliable Broadcast

2.1. Especificación de la difusión fiable

En esta sección presentamos la especificación de la difusión fiable (*reliable broadcast*) y en la siguiente sección mostramos cómo se implementa en un sistema propenso a fallos. La difusión fiable es importante en el diseño de e implementación de sistemas distribuidos tolerantes a fallos donde se necesita una comunicación entre un grupo de procesos con garantías en la entrega de los mensajes difundidos.

Modelo de procesos. El modelo de computación que consideramos está compuesto por un conjunto de procesos secuenciales identificados de manera única. Dicho conjunto lo denotamos por $\Pi = \{p_1, ..., p_N\}$ siendo p_i el identificador del proceso i-ésimo. Donde no haya confusión también lo identificamos por el índice i. El número de procesos N es una constante conocida por los procesos en el sistema. Los procesos son secuenciales y asíncronos. Por secuencial entendemos que el proceso no realiza una nueva operación hasta que completa la anterior; y por asíncrono entedemos que el proceso se ejecuta a su propia velocidad, de manera arbitraria y desconocida para el resto de procesos. Los procesos puede fallar de forma abrupta (*crash failure*). Mientras no fallan realizan su algoritmo de forma correcta y cuando fallan simplemente se paran. Un proceso que no falla durante una ejecucíon es un proceso *correcto*, en otro caso decimos que es *fallido*. El número de fallos posible en cualquier ejecución, f, cumple $1 < f \leq N$. Al menos un proceso en una ejecución es correcto aunque para cada ejecución es desconocida su identidad.

Especificación de la difusión fiable. El algoritmo de difusión fiable ofrece a cada proceso dos primitivas para gestionar el envío y la entrega de mensajes. La operación **broadcast**(m) invocada por un proceso p_i le permite enviar el mensaje m al resto de participantes. El evento **deliver**(m) es disparado por el algoritmo para indicarle al proceso p_i la recepción del mensaje m que debe tratar. **deliver**(m) debe entenderse como una llamada que hace el algoritmo para ejecutar el código del proceso p_i asociado con la entrega de un mensaje. Las propiedades que debe cumplir el algoritmo de la difusión fiable asumen que todos los mensajes difundidos son distintos

B1 *Validez.* Si un proceso entrega un mensaje m, entonces m fue difundido por algún otro proceso.
B2 *Integridad.* Un mensaje se entrega como mucho una vez a cada proceso.
B3 *Terminación 1.* Si un proceso correcto difunde un mensaje m entonces ese mismo proceso entrega el mensaje m.
B4 *Terminación 2.* Si un proceso correcto entrega un mensaje m, entonces todos los procesos correctos entregan el mensaje m.

Como puede observarse, **B1** y **B2** son propiedades de seguridad, mientras que **B3** y **B4** son propiedades de progreso. Debe notarse también que **B3** y **B4** no hacen referencia alguna al tiempo en el que los procesos entregarán los mensajes.

De la propia especificación se deduce la siguiente propiedad. Si en un determinado momento τ, un proceso correcto p_i ha entregado un conjunto de mensajes D^i_τ, entonces hay un momento posterior $\tau' \geq \tau$, tal que otro poceso correcto p_j cumple $D^i_\tau \subseteq D^j_{\tau'}$. Esta propiedad se deduce de la condición **B4**. En otras palabras, los procesos correctos van entregando el mismo conjunto de mensajes a lo largo del tiempo.

La difusión fiable captura una clase de problemas de coordinación distribuida donde la operación broadcast(m) y deliver(m) pueden interpretarse respectivamente como “esto es una orden”

y “ejecuta la orden”, donde la orden viene indicada en el cuerpo del mensaje m. Por la propiedad anterior los procesos correctos ejecutarán las mismas ordenes (en un orden arbitrario), estas órdenes incluyen a las enviados por procesos correctos y algunas enviadas por procesos que han fallado. En la especificación dada no podemos asegurar ninguna condición sobre los mensajes entregados por procesos que fallan en la ejecución. En la subsección 2.3 indicamos otras propiedades que amplían la especificación de la difusión fiable.

2.2. Implementeación de la difusión fiable

Si consideramos un sistema distribuido asíncrono con canales fiables la implementación de la especificación dada es bastante sencilla. En esta sección seguimos la implementación dada en [15].

Subsitema de comunicación. Asumimos que los procesos se pueden comunicar entre sí mediante canales punto a punto asíncronos y *fiables*. La operación **send**(m) **to** p_j se invoca por un proceso p_i para enviar el mensaje m al proceso p_j. El evento **receive**(m) **from** p_i es tratado por el proceso p_j en el momento de la recepción del mensaje m enviado por p_i. Las propiedades que cumplen los canales son:

> **C1** *Validez*. Si un proceso recibe un mensaje m, entonces m ha sido enviado por otro proceso.
> **C2** *Integridad*. Los mensajes son recibidos como mucho una vez.
> **C3** *Terminación*. Si un proceso correcto envía un mensaje m a otro proceso correcto, entonces este último recibirá el mensaje m.

De nuevo se observa que el canal es asícrono en el sentido de que **C3** no acota el tiempo en el que el mesaje será recibido. Una variante de **C3**, un poco más débil, que se suele considerar en la literatura es la siguiente:

> **C4** *Terminación*. Si un proceso (correcto o no) completa la operación de envío de un mensaje m a otro proceso correcto, entonces este último recibirá el mensaje m.

Observamos que **C3** $\Rightarrow$ **C4**. Algunos autores indican que **C1** a **C3** definen canales *fiables*, mientras que **C1** a **C4** constituyen la definición de canales *uniformemente fiables*. Seguiremos esta terminología.

Algoritmo R-Broadcast. La implementación de un algoritmo que satisface las condiciones de la disfusión fiable en un sistema asícrono con fallos de parada y canales fiables se presenta en el Algotimo 1. La variable *Delivered* contiene los mensajes que han sido entregados. La operación Broadcast(m) simplemente envía el mensaje a sí mismo. Cuando un mensaje m es recibido (línea 8), si es la primera vez que se recibe se transmite punto a punto al resto de procesos (líneas 9-10) y finalmente se entrega (líneas 11-12). La operación **send** $\langle \mathtt{m} \rangle$ **to** $\Pi \setminus \{k, i\}$ debe entenderse como una forma resumida de escribir
for each j **in** $\Pi \setminus \{k, i\}$ **do send** $\langle \mathtt{m} \rangle$ **to** j
Observamos que si un proceso falla durante la ejecución de esta sentencia no hay ninguna garantía de recepción del mensaje para los procesos indicados en la misma.

Algorithm 1 Algoritmo R-Broadcast. Código del proceso p_i

```
 1: Variables
 2: Delivered := ∅                                   ▷ set of delivered messages
 3:
 4: operation Broadcast(m)
 5:     send ⟨m⟩ to p_i
 6: end operation
 7:
 8: when ⟨m⟩ is received from p_k
 9:     if m ∉ Delivered then                        ▷ first reception of m
10:         send ⟨m⟩ to Π \ {k, i}
11:         Deliver(m)                               ▷ m is delivered
12:         Delivered := Delivered ∪ {m}
13:     end if
14: end when
```

Prueba de corrección y coste. El Algoritmo 1 cumple la especificación de la difusión fiable. Asumimos que los mensajes difundidos son todos distintos.

- Por el propio código, **B1** se cumple, puesto que sólo se entregan mensajes que han sido difundidos por algún proceso.
- Por otra parte, como los mensajes son distintos, una vez que un mensaje m es entregado, se incluye en *Delivered* (línea 12), $m \in Delivered$, por lo que si el proceso vuelve a ejecutar la línea 9 posteriormente para el mismo mensaje m, esté no será entregado. Los mensajes se entregan como mucho una vez **B2**. Observamos que los mensajes difundidos son distintos pero eso no implica que canales distintos puedan contener el mismo mensaje en diferentes momentos de la ejecución. Esto es así porque cada vez que se vé un mensaje por primera vez se vuelve a retransmitir. La repetición sirve para asegurar la propiedad **B4**.
- Propiedad **B3**. Si un proceso correcto p_i realiza Broadcast(m), se envía el mensaje a sí mismo. Por la propiedad **C3**, recibe el mensaje m y (la primera vez) ejecuta las líneas 8-13, entregando el mensaje.
- Propiedad **B4**. Si un proceso correcto p_i entrega un mensaje m, línea 11, ha ejecutado antes línea 10, y ha enviado el mensaje a todos menos a sí mismo y al enviador del mensaje. Si el enviador del mensaje p_k es correcto habrá entregado (o entregará) el mensaje m (por línea 11). Por efecto, de la línea 10 y el hecho de que los canales cumplen **C3**, todos los procesos correctos en el conjunto $\Pi \setminus \{k, i\}$ recibirán el mensaje m enviado por p_i. Dado que son correctos llegarán a ejecutar Deliver(m). En conclusión, todos los correctos entregarán el mensaje m.

Coste computacional. Se puede argumentar que no es necesario enviarse el mensaje a si mismo en la ejecución de Broadcast(m) pero es una modificación menor. El *peor tiempo* entre Brodacast(m) y Deliver(m), medido como la secuencia más larga posible de mensajes entre Broadcast(m) ejecutado por p_i y Deliver(m) ejecutado por otro proceso distinto p_j es líneal $\mathcal{O}(N)$. El peor coste que tiene el algoritmo es que por cada mensaje difundido se generan $\mathcal{O}(N^2)$ mensajes. Este coste es el precio que se paga por tener tolerancia a fallos de parada.

2.3. Variantes de la difusión fiable

Es posible aumentar la garantía de entrega ofrecida por el algoritmo de difusión fiable básico que hemos presentado en el Algoritmo 1. Si se consideran canales de comunicación entre procesos que cumplen la propiedad **C4**, los canales se consideran uniformemente fiables. En ese caso, la misma implementación dada en el Algoritmo 1 cumple la propiedad

B5 *Terminación 2 uniforme.* Si un proceso (correcto o no) entrega un mensaje m, entonces todos los procesos correctos entregan el mensaje m.

En ese caso, los mensajes entegados por procesos que fallan son un subconjunto de los mensajes entregados por los procesos correctos. La estructura del Algoritmo 1 en donde primero se envía a todos y luego se entrega el mensaje es la base para que se cumpla la condición de entrega uniforme dada en **B5**.

En el trabajo [5], se muestra cómo añadir nuevas propiedades de seguridad al Algoritmo 1 utilizando este mismo algoritmo como un módulo sobre el que construir, mediante técnicas bloqueantes, nuevos algoritmos de difusión. Entre las propiedades de seguridad que se pueden añadir con estas técnicas encontramos la entrega *fifo* y la entrega *causal*:

B6 *Orden Fifo.* Para cualquier par de procesos p_i y p_j, si p_i entrega primero un mensaje m y más tarde m', ambos mensajes difundidos por p_j, entonces ningún otro proceso entrega m' sin haber entregado antes m.

B7 *Orden Causal.* Si un mensaje m precede causalmente a m', entonces ningún proceso entrega m' a menos que haya entregado antes m.

Las implementaciones de los algoritmos que satisfacen las propiedades anteriores sólo requieren información local a los procesos para tomar la decisión de entrega de los mensajes. Este no es el caso si deseamos que los mensajes se entreguen en *orden total.* La propiedad de entrega en *orden total* se puede enunciar de forma simple (hay diferentes variantes en la literatura, nosotros consideramos la ofrecida en [5]):

B8 *Orden Total.* Si dos procesos p_i y p_j entregan ambos m y m', entonces p_i entrega m antes que m' si y sólo si p_j entrega m antes que m'.

Obtener esta propiedad requiere que los procesos *decidan el mismo orden de entrega* y esto es posible si, efectivamente, llegan a un acuerdo en el orden de la entrega de los mensajes. Esta propiedad es imposible en un sistema asíncrono con fallos en virtud de la imposibilidad FLP y del hecho de que la difusión con orden total y el problema del acuerdo distribuido son equivalentes. En este trabajo de investigación analizaremos una versión mucho más débil del orden total, aquella en el que los procesos proporcionan una entrega en orden total aproximado.

2.4. Implementeción de la difusión fiable sobre canales FIFO

Una de las cuestiones que se deben resolver en el Algoritmo 1 para llevarlo a la práctica es recolectar los mensajes que se encuentran alamacenados en *Delivered* y que ya no van a ser utilizados. La variable *Delivered* crece sin control y por tanto, se debe acotar su contenido sin variar el correcto comportamiento de la difusión fiable. Cuando se realiza la modificación del Algoritmo 1 para obtener el orden fifo se obtinene una solución donde la comprobación realizada en la línea 9 se construye mediante una tabla donde para cada proceso se almacena el siguiente mensaje que se debe recibir en orden fifo. Esto se implementa de manera sencilla si cada mensaje difundido tiene un identificador de la forma $(origin, sqn)$, donde *origin* es el identificador del proceso que difunde el mensaje y *sqn* es el número de secuencia de dicho mensaje generado por *origin* de forma autoincremental. Por otro lado, si consideramos que los canales punto a punto entre los procesos son fifo,

C5 *fifo.* Si el proceso p_i envía primero m y después m' al proceso p_j, entonces p_j no recibe m' si no ha recibido antes m.

no es difícil comprobar que el Algoritmo 1 implementado sobre canales fiables fifo cumple la propiedad **B6** *Orden Fifo*. En nuestro caso la implementación la realizaremos sobre canales fifo por lo que para recolectar la basura de los mensajes utilizamos prácticamente la misma estrategia que la dada para la implementación de la difusión fiable con orden fifo sobre canales simplemente fiables. El Algoritmo 2 presenta la solución propuesta.

Algorithm 2 Algoritmo Fifo R-Broadcast (canales Fifo). Código del proceso p_i

```
 1: Variables
 2: sqn := 0                                   ▷ sequence number for a protocol message
 3: lastsq[1..N] := (0, .., 0)                 ▷ lastsq[j] the last sqn of a delivered message with origin p_j
 4: id := (i, 0)                               ▷ identity of the protocol message with origin p_i
 5:
 6: operation Broadcast(m)
 7:     sqn := sqn + 1
 8:     id := (i, sqn)                         ▷ identity, a pair (origin, sqn)
 9:     send ⟨m, id⟩ to p_i
10: end operation
11:
12: when ⟨m, mid⟩ is received from p_k
13:     if not isdelivered(mid) then           ▷ first reception of m
14:         send ⟨m, mid⟩ to Π \ {k, i}
15:         Deliver(m)                         ▷ m is delivered
16:         lastsq[mid.origin] := mid.sqn
17:     end if
18: end when
19:
20: function isdelivered(mid): boolean
21:     if lastsq[mid.origin] ≥ mid.sqn then
22:         return(true)
23:     end if
24:     return(false)
25: end function
```

Este nuevo algoritmo implementa 3 nuevas variables:

- *sqn*: número de secuencia actual (cada vez que se envía un mensaje nuevo se incrementa la variable).

- *lastsq*: array que contiene el último número de secuencia recibido por cada proceso (cada posición del array representa un proceso).
- *id*: identificador único de cada mensaje (tupla compuesta por el número de proceso y el número de secuencia).

Observe que este algoritmo dispone de una función adicional llamada *isdelivered()*. Esta función comprueba si un mensaje recibido ha sido previamente entregado o no (para hacer esto comprueba el contenido de la variable *lastsq*).

2.5. Discusión

Hemos indicado en la sección de introducción y en la sección 2.3 que la difusión fiable con orden total es imposible en un sistema asíncrono con fallos. Las soluciones a este problema se pueden implementar en sistemas parcialmente síncronos. Pero para asegurar la terminación de la difusión con la garantía del orden total se suelen emplear entornos de despliegue que tengan un comportamiento síncrono lo más pronto posible. Estos sistemas tienen un buen comportamiento en cuanto a los retrasos de los mensajes y en ellos es bastante plausible que se cumpla la propiedad de orden espontáneo. La cuestión es que los nodos, en la entrega de los mensajes, no tienen *consciencia* de dicho orden y únicamente, si se implementa una difusión con orden total se asegura dicha información. Si abandonamos la propiedad de orden total como algo absoluto que debe cumplir la entrega de mensajes podemos intentar diseñar un algoritmo que intente hacer el mayor esfuerzo posible para generar un orden en la entrega y que este orden sea un *orden aproximado* para todos los nodos. Dado que no hay un acuerdo global en el orden en que deben entregarse los mensajes (impuesto por un protocolo de orden total) debemos confiar en que exista un orden *a priori* que permita a los nodos identificar *un cierto orden*. La forma mas simple para generar un orden entre los mensajes es basasarnos en algún mecanismo de *marcas de tiempo*. Con esa idea, es bastante simple que los nodos puedan decidir si *localmente* un mensaje se entrega en el orden impuesto por las marcas de tiempo. La cuestión es entonces si localmente un nodo puede controlar la entrega de mensajes para producir un mayor orden aproximado en todo el sistema. Es obvio que lo que puede controlar un nodo es la historia de mensajes ya entregados, el retraso en la entrega de un mensaje o el retraso en el envío del mismo. Todos los retrasos que se introduzcan y que produzcan un efecto positivo (o negativo) en el orden de entrega de mensajes representan un tira y afloja contra el comportamiento del entorno. La cuestión es entonces comprobar si el control ejercido por los nodos sobre el orden de entrega tiene un coste que es compensado por su beneficio potencial.

En lo que sigue analizamos primero el mecanismo de marcas de tiempo que vamos a utilizar en nuestro trabajo para posteriormente definir la noción de difusión fiable con orden aproximado. Sobre esta noción es sobre la que diseñaremos un protocolo para comprobar experimentalemente en entornos LAN y en entornos ofrecidos por un DC (centro de datos) la relación entre el orden (total) conseguido por todos los nodos y los costes introducidos para obtenerlo.

2.6. Mecanismo de marcas de tiempo: HLC

En el sistema distribuido que consideramos no hay una noción de *tiempo verdadero* que los procesos puedan utilizar. Podemos referinos al tiempo verdadero sólo desde nuestro punto de vista cuando razonamos sobre el comportamiento del sistema como observadores omniscientes. Cada proceso puede acceder a un reloj en la máquina donde se ejecuta. Desde un punto de vista práctico consideramos que las máquinas se sincronizan empleando el protocolo NTP (*Network Time Protocol*) [17]. Ya que la sincronización perfecta de los relojes no es posible, asumiremos que hay intervalos de incertidumbre asociados con los relojes. En este escenario podemos utilizar el mecanismo de marcas de tiempo propuesto por [16] denominado HLC (*Hybrid Logical Clock*). Las ventajas que presenta este mecanismo en comparación con otros es su buena capacidad de tolerar las divergencias comunes de NTP; además, es auto-estabilizado, y es resistente a posibles corrupciones de las variables del reloj.

Para enteder algunas de las propiedades que ofrece HLC recordamos aquí la noción de *relación causal entre eventos*. Por evento entendemos las acciones internas, el envío o recepción de mensajes que realiza un proceso (secuencial). Un evento e *ocurre antes* que f, denotado e hb f, (i) si e y f son del mismo proceso y e ocurre antes que f; o bien (ii) e es un evento de envío de un mensaje y f es su correspondiente evento de recepción del mensaje. La relación completa hb es el cierre transitivo de lo indicado en (i) y (ii) para todos los eventos. Dos eventos son *concurrentes*, denotado $e||f$, si y sólo si, $\neg(e\, hb\, f) \wedge \neg(e\, hb\, f)$.

En el algoritmo HLC, Algoritmo 3, cada evento tiene asociado una pareja de valores (l, c). El valor l se corresponde con un reloj físico, pt, y el valor de c se corresponde con un entero para capturar la causalidad cuando los valores de l de los eventos son iguales. Los procedimientos indicados en el Algoritmo 3 son fáciles de comprender. Básicamente l se actualiza con el reloj fśico y en caso de empate se incrementa c. En lo que sigue se asume[2]:

AS1 *Suposición de sincronización de relojes físicos*: No existen dos eventos e y f en el sistema distribudo tales que $e\, hb\, f$ y $pt.e > pt.f + \varepsilon$. Donde ε denota la incertidumbre de la sincronización de los relojes (aproximadamente dos veces el valor de compensación (offset) del NTP).

Las propiedades [16] que ofrece este mecanismo en relación a la causalidad son las siguientes:

HLC1 *Causalidad direccional*. Para dos eventos cualesquiera e y f:
$e\, hb\, f \Rightarrow (l.e, c.e) < (l.f, c.f)$.
HLC2 Para cualquier evento f: $l.f \geq pt.f$.
HLC3 *$l.f$ denota el máximo valor de reloj del que f es consciente*. Para cualquier evento f: $l.f > pt.f \Rightarrow (\exists g : g\, hb\, f \wedge pt.g = l.f)$
HLC4 *Acotación*. Para cualquier evento f: $|l.f - pt.f| \leq \varepsilon$
HLC5 Para cualquier evento f:

$$c.f = k \wedge k > 0 \Rightarrow$$
$$\exists g_1, g_2, ..., g_k : ((\forall j : 1 \leq j < k : g_i\, hb\, g_{i+1})$$
$$\wedge (\forall j : 1 \leq j \leq k : l.(g_i) = l.f) \wedge g_k\, hb\, f)$$

HLC6 *Acotación de c*. Para cualquier evento f: $c.f \leq |\{g : g\, hb\, f \wedge l.g = l.f\}|$
HLC7 *Acotación de c*. Para cualquier evento f: $c.f \leq N \times (\varepsilon + 1)$

[2]Mantenemos la notación de Kulkarni [16], para un evento e su pareja asociada es $(l.e, c.e)$. El valor físico del reloj cuando sucedió el evento e se denota, $pt.e$. Por otro lado, la relación $<$ se establece como $(a, b) < (c, d) \Leftrightarrow (a < c) \vee ((a = c) \wedge (b < d))$.

Algorithm 3 Algoritmo HLC. Código del proceso p_i

```
1: Variables
2: pt                                  ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                        ▷ timestmap, a pair (l, c). Updated via HLC procedures
4:
5: procedure update_ts-snd()
6:     Let l' := ts.l                  ▷ the previous logical time
7:     ts.l := max(l', pt)             ▷ the new value of logical time ts.l
8:     if ts.l = l'
9:         ts.c := ts.c + 1
10:    else
11:        ts.c := 0
12:    end if-else
13: end procedure
14:
15: procedure update_ts-rcv((lm, cm))
16:    Let l' := ts.l                  ▷ the previous logical time
17:    ts.l := max(l', lm, pt)         ▷ the new value of logical time ts.l
18:    if ts.l = l' = lm
19:        ts.c := max(ts.c, cm) + 1
20:    else if ts.l = l'
21:        ts.c := ts.c + 1
22:    else if ts.l = lm
23:        ts.c := cm + 1
24:    else
25:        ts.c := 0
26:    end if-else
27: end procedure
```

3. AOR Broadcast

3.1. Especificación AOR

En la difusión fiable con orden de entrega aproximado, cada proceso p_i es consciente de un cierto orden de entrega de mensajes, siendo el orden observado por p_i compatible con el orden observado por otro proceso distinto p_j. La especifiación de la difusión fiable con orden aproximado tiene las mismas propiedades que la difusión fiable y una propiedade débil sobre el orden de la entrega de mensajes.

Definición de AOR-Broadcast. En la especificación de la difusión fiable con orden aproximado consideramos que los procesos disponen de la operación Broadcast(.) para difundir mensajes y dos tipos de eventos de entrega, O-Deliver(.) y U-Deliver(.).

> Cuando se ejecuta O-Deliver(m) o bien U-Deliver(m) por un proceso decimos que el "mensaje m ha sido entregado por el proceso"(como en el caso de la difusión fiable).
>
> O-Deliver(m) ejecutado por el proceso p_i indica que el mensaje m se ha entegado localmente en *orden*. Este orden local de los mensajes visto por el proceso p_i es básicamente construido por la siguiente relación: Decimos que dos mensajes m1 y m2 entregados por el proceso p_i, están en orden local a p_i, denotado $m1 <_O^{p_i} m2$, si y sólo si O-Deliver(m1) y O-Deliver(m2) han sido ejecutados por p_i y O-Deliver(m1) hb O-Deliver(m2). Sea $O_i(\tau)$ el conjunto de mensajes en orden entregados por p_i hasta un cierto momento τ. Obviamente, $O_i(\tau)$ está totalmente ordenado con la relación de entrega ordenada $<_O^{p_i}$.
>
> U-Deliver(m) ejecutado por el proceso p_i en un cierto momento τ indica que el mensaje m se ha entregado localmente en desorden. En otras palabras, $m \notin O_i(\tau)$ por la definición dada anteriormente.

Como en el caso de la difusión fiable asumimos que todos los mensajes difundidos son distintos. Las propiedades que debe cumplir cualquier algoritmo que implemente una difusión fiable con orden aproximado son las mismas propiedades de la difusión fiable dadas en la sección 2.1, a saber, **B1** *Valided*, **B2** *Integridad*, **B3** *Terminación 1*, **B4** *Terminación 2*, y la siguiente propiedad,

> **AOB** *Orden total aproximado.* Si dos procesos p_i y p_j y dos mensajes m y m' cumplen $\{m, m'\} \subseteq O_i(\tau)$ y $\{m, m'\} \subseteq O_j(\tau)$ en un cierto momento τ, entonces $m <_O^{p_i} m' \Leftrightarrow m <_O^{p_j} m'$.

Debe observarse que **AOB** es la misma propiedad **B8** *Orden Total* aplicada únicamente a los mensajes entregados en orden, en otras palabras, **AOB** indica que si dos procesos p_i y p_j ambos entregan en orden m y m', entonces p_i entrega en orden m antes que m' si y sólo si p_j entrega en orden m antes que m'. Es bastante obvio que en una ejecución muy favorable de la difusión fiable con orden aproximado en la que todos los mensajes se entreguen en orden el resultado es indistinguible de una ejecución producida por una difusión fiable con orden total.

3.2. Implementación AOR

El protocolo de difusión fiable presentado en el Algoritmo 1 se puede modificar de manera muy simple para obtener un algoritmo de difusión fiable con orden aproximado. El Algoritmo 4 asume que el mecanismo de marcas de tiempo empleado se basa en HLC (ver sección 2.6). El

Algoritmo 4 utiliza la variable $lastTS$ para almacenar la marca de tiempo del último mensaje entregado en orden. En base al valor de esta variable el siguiente mensaje a ser entregado se clasifica como ordenado o desordenado. Como es posible que pueda haber dos marcas de tiempo iguales, la marca de tiempo HLC, (l, c), se extiende con el identificador del proceso origen que genera dicha marca. De esta manera todos los mensajes difundidos tienen un identificador único cuya estructura es $(origin, (l, c))$ (origen y marca de tiempo). La comparación entre las marcas de tiempo extendidas (identificadores de los mensajes) sigue la regla siguiente:
$(i, (l, c)) < (j, (l', c') \Leftrightarrow (l < l') \vee (l = l' \wedge c < c') \vee ((l, c) = (l', c') \wedge i < j))$.

Algorithm 4 Algoritmo AOR-Broadcast (trivial). Código del proceso p_i

```
 1: Variables
 2: pt                                  ▷ a read-only variable. It contains physical time via NTP protocol
 3: ts := (0, 0)                        ▷ timestmap, a pair (l, c). Updated via HLC procedures Alg. 3
 4: Delivered := ∅                      ▷ set of delivered messages
 5: TS := (i, ts)                       ▷ extended timestamp of node p_i, it is a pair (origin, timestamp)
 6: lastTS := (0, (0, 0))               ▷ extendend timestamp of the last delivered message in order
 7:
 8: operation Broadcast(m)
 9:     update_ts-snd()                 ▷ update timestamp on sending
10:     TS := (i, ts)
11:     send ⟨m, TS⟩ to p_i
12: end operation
13:
14: when ⟨m, mTS⟩ is received from p_k
15:     update_ts-rcv(mTS.ts)           ▷ update timestamp of p_i on receiving
16:     if m ∉ Delivered then
17:         send ⟨m, mTS⟩ to Π \ {k, i}
18:         if mTS < lastTS then U-Deliver(m)      ▷ m is delivered unordered
19:         else                                   ▷ strict total order lastTS < mTS
20:             O-Deliver(m)                       ▷ m is delivered in order
21:             lastTS := mTS
22:         end if-else
23:         Delivered := Delivered ∪ {m}
24:     end if
25: end when
```

Observe que las líneas 9-10 del Algoritmo 4 aseguran que todos los mensajes difudidos por el algoritmo son distintos. Esto es así por la propiedad **HLC1** y el hecho de que los identificadores de los procesos son distintos. Dado que las líneas 18-22 determinan la entrega en orden o en desorden de manera exclusiva, con independencia del valor de $lastTS$, es equivalente su comportamiento a la línea 11 del Algoritmo 1. Como los mensajes una vez entregados se almacenan en *Delivered* concluimos que el Algoritmo 4 satisface las mismas propiedades que la difusión fiable. La línea 15 asegura la causalidad direccional **HLC1** entre los diferentes eventos de los procesos y hace un mayor esfuerzo para que los siguientes mensajes difundidos por el proceso se entreguen en orden. Esto es así puesto que se puede demostrar fácilmente que en cualquier estado de la ejecución del algoritmo para el proceso p_i se cumple que $lastTS < (i, ts)$.

La propiedad **AOB** se cumple trivialmente entre dos procesos que entregan en orden los mismos mensajes puesto que el orden viene impuesto por sus marcas de tiempo extendidas. Concluimos que el Algoritmo 4 satisface la especificación de la difusión fiable con orden aproximado.

La versión del Algoritmo 4 para canales fifo es el Algoritmo 5.

Algorithm 5 Algoritmo AOR-Broadcast (trivial) sobre canales fifo. Código del proceso p_i

```
1: Variables
2: pt                                   ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                         ▷ timestmap, a pair (l, c). Updated via HLC procedures Alg. 3
4: lastts[1..N] := ((0,0), .., (0,0))   ▷ lastts[j] the last ts of a delivered message with origin p_j
5: TS := (i, ts)                        ▷ extended timestamp of node p_i, it is a pair (origin, timestamp)
6: lastTS := (0, (0,0))                 ▷ extendend timestamp of the last delivered message in order
7:
8: operation Broadcast(m)
9:     update_ts-snd()                  ▷ update timestamp on sending
10:    TS := (i, ts)
11:    send ⟨m, TS⟩ to p_i
12: end operation
13:
14: when ⟨m, mTS⟩ is received from p_k
15:    update_ts-rcv(mTS.ts)            ▷ update timestamp of p_i on receiving
16:    if not isdelivered(mTS) then     ▷ first reception of m
17:        send ⟨m, mTS⟩ to Π \ {k, i}
18:        if mTS < lastTS then U-Deliver(m)   ▷ m is delivered unordered
19:        else                         ▷ strict total order lastTS < mTS
20:            O-Deliver(m)             ▷ m is delivered in order
21:            lastTS := mTS
22:        end if-else
23:        lastts[mTS.origin] := mTS.ts
24:    end if
25: end when
26:
27: function isdelivered(mTS): boolean
28:    if lastts[mTS.origin] ≥ mTS.ts then
29:        return(true)
30:    end if
31:    return(false)
32: end function
```

3.3. Retrasar la entrega de mensajes

Uno de los principales inconvenientes de trabajar con sistemas distribuidos asíncronos es que no se puede garantizar que no ocurran casos como el que observamos en la figura 1. En este caso aislado vemos que si tenemos 3 procesos $\{p_1, p_2, p_3\}$ y p_1 envía el mensaje m_{11} y seguidamente p_2 envía el mensaje m_{21}, al ser un sistema asíncrono, se puede dar el caso en el que el proceso p_3 reciba primero el mensaje m_{21} y después reciba el mensaje m_{11}. El orden de recepción de los mensajes sería "incorrecto", ya que podemos ver que claramente m_{11} se emitió antes que m_{21}.

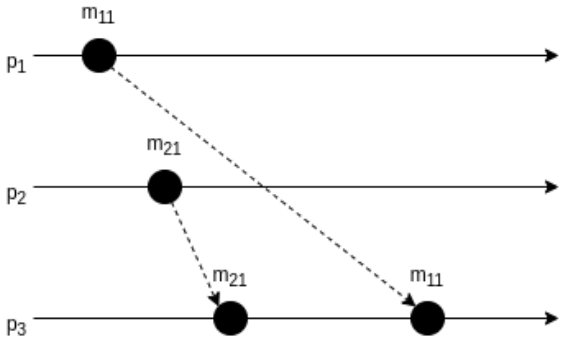

Figura 1: Entrega Desordenada

Para solventar este problema, se propone implementar una medida que mediante ventanas de tiempo Δ retrase la entrega de mensajes. Es decir, cada vez que se reciba un mensaje, se almacena y ordena utilizando la misma medida de marcas de tiempo que describimos en el apartado 2.6. Mientras tanto, un hilo que trabaja en segundo plano envía los mensajes de forma ordenada cada Δ unidades de tiempo. En la figura 2 podemos visualizar cómo funcionaría esta medida.

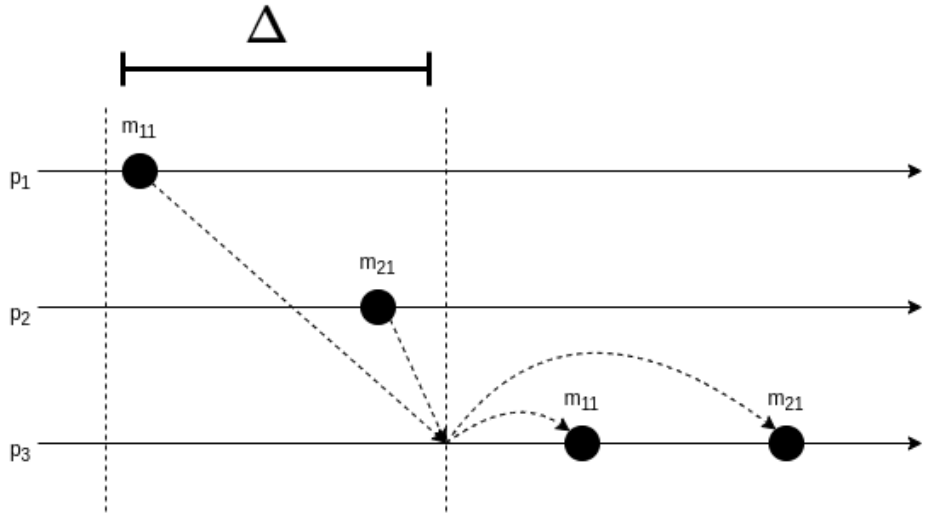

Figura 2: Ventana de tiempo

Fijando la entrega de mensajes por intervalos de tiempo Δ reducimos la posibilidad de que se entreguen mensajes en desorden cuando se dan casos como en el de la figura 1. De este modo

mantenemos la entrega de mensajes "bajo control" al ser capaces de ordenar los mensajes utilizando la misma medida de comparación de marcas de tiempo que en 3.2.

3.3.1. Implementación

Utilizando la medida descrita anteriormente y con el objetivo de incrementar el porcentaje de mensajes entregados en orden, se propone el Algoritmo 6, cuya implementación parte de la versión 5 y añade nueva lógica.

En este nuevo algoritmo la recepción de mensajes se trata de la misma forma que en el algoritmo 8, con la excepción de que cada vez que llega un mensaje "ordenado", en lugar de entregarlo directamente con un evento de tipo *O-Deliver*, se le añade una marca de tiempo que representa el tiempo de recepción del mensaje y se introduce en *Rec*, un array que contiene mensajes "ordenados" pendientes de entregar y que, además, está ordenado de menor a mayor.

Adicionalmente, esta versión dispone de una nueva función (línea 31) que se ejecuta cada Δ milisegundos y que recorre el array *Rec* (de forma ordenada) entregando aquellos mensajes que cumplan la condición de que la diferencia entre el tiempo actual y el tiempo en el que se recibió el mensaje que se está tratando sea mayor o igual que Δ. Si se cumple la condición, se entrega el mensaje en cuestión, se actualiza el $lastTS$ y se elimina el mensaje del array *Rec*.

Algorithm 6 ATO-URB Algorithm v6 (FIFO channels). Code for process p_i

```
1: Variables
2: pt                                   ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                         ▷ timestmap, a pair (l, c). Updated via HLC procedures Alg. 3
4: delivered[1..N] := ((0,0),..,(0,0))  ▷ delivered[j] the last ts of a delivered message with origin p_j
5: TS := (i, ts)                        ▷ extended timestamp of node p_i, it is a pair (origin, timestamp)
6: lastTS := (0, (0, 0))                ▷ extendend timestamp of the last delivered message in order
7: Rec := ∅   ▷ a map of tuples (msg, msgTS, mpt) where msg is a message, msgTS is its extended
   timestamp (key), and mpt is the physical time when msg was received by the first time
8:
9: operation Broadcast(m)
10:     update_ts-snd()
11:     TS := (i, ts)
12:     send ⟨m, TS⟩ to p_i
13: end operation
14:
15: when ⟨m, mTS⟩ is received from p_k
16:     update_ts-rcv(mTS.ts)                ▷ update timestamp of p_i on receiving
17:     if not isdelivered(mTS) then
18:         if mTS ∉ Rec then                ▷ first reception of m
19:             send ⟨m, mTS⟩ to Π \ {k, i}  ▷ message m is retransmitted
20:             if mTS ≤ lastTS
21:                 U-Deliver(m)
22:                 delivered[mTS.origin] := mTS.ts
23:             else                         ▷ mTS > lastTS
24:                 Rec := Rec ∪ {(m, mTS, pt)}
25:             end if-else
26:         end if
27:     end if
28: end when
29:
30: Task::
31: repeat  each Δ time
32:     Let todeliver[ ] := sortbyTS(Rec)    ▷ Array from Rec sorted by the key
33:     Let L := todeliver.length            ▷ Length of previous array [1..L]
34:     Let p := 1
35:     while p ≤ L and pt − todeliver[p].mpt ≥ Δ do
36:         Let (m, mTS, mpt) := todeliver[p]
37:         O-Deliver(m)
38:         lastTS := mTS
39:         Rec := Rec \ {(m, mTS, mpt)}
40:         delivered[mTS.origin] := mTS.ts
41:     end while
42: end repeat
43:
44: function isdelivered(mTS): boolean
45:     if delivered[mTS.origin] ≥ mTS.ts then
46:         return(true)
47:     end if
48:     return(false)
49: end function
```

3.4. Algoritmo adaptativo

El algoritmo propuesto en el apartado anterior (6) presenta el problema de que el parámetro Δ tiene un valor fijado y no puede adaptarse según la carga de mensajes que esté sufriendo el protocolo. Si la carga de mensajes aumenta, el valor de Δ será demasiado pequeño como para satisfacer la cantidad de mensajes que se reciben y la ventana de tiempo utilizada dejará de ser efectiva, produciendo así un aumento en la entrega de mensajes en desorden. Por otro lado, si disminuye la carga de mensajes, el valor de Δ será demasiado grande y a pesar de que la ventana de tiempo siga siendo efectiva, se estará retrasando la entrega de mensajes más tiempo del realmente necesario.

Para solucionar esta deficiencia se propone el algoritmo 7 que ofrece una medida que es capaz de adaptar el valor del parámetro Δ para que aumente o disminuya según la carga de mensajes recibida.

Algorithm 7 Algoritmo Adaptativo. Código del proceso p_i

```
 1: Variables
 2: pt                          ▷ a read-only variable. It contains physical time via NTP protocol
 3: delta := 0                                                          ▷ value of the parameter Δ
 4: delta_max := 0                          ▷ maximum physical time read from received messages
 5: delta_min := 0                          ▷ minimum physical time read from received messages
 6:
 7: procedure update_bounds(l)
 8:     if delta_max = 0 ∧ delta_min = 0 ∧ (pt − l) > 0
 9:         delta_max := pt − l
10:         delta_min := delta_max
11:     end if
12:     if (pt − l) > delta_max ∧ (pt − l) > 0
13:         delta_max := pt − l
14:     end if
15:     if (pt − l) < delta_min ∧ (pt − l) > 0
16:         delta_min := pt − l
17:     end if
18: end procedure
19:
20: procedure update_delta()
21:     Let diff := delta_max − delta_min      ▷ the difference between delta_max and delta_min
22:     if diff ≤ 1
23:         diff := 1
24:     end if
25:     if diff > 5,0
26:         diff := 3
27:     end if
28:     delta := 0,7 ∗ diff + 0,3 ∗ delta
29:     delta_max := 0
30:     delta_min := 0
31: end procedure
```

La función ***update_bounds(l)*** recibe como argumento la variable l que representa una marca de tiempo físico (*timestamp*) y la utiliza para actualizar las variables globales *delta_max* y *delta_min*, que representan (respectivamente) la máxima y mínima diferencia de tiempo registradas.

La función ***update_delta()*** actualiza la variable global *delta* utilizando las variables globales *delta_max* y *delta_min*. En la línea 28 podemos ver que el cálculo del nuevo delta se realiza aplicando la siguiente fórmula: $\Delta = 0{,}7 \times \mathit{diff} + 0{,}3 \times \mathit{delta}$. Dónde *diff* representa la diferencia entre *delta_max* y *delta_min*. El motivo por el que hemos seleccionado esos valores $\{0{,}7; 0{,}3\}$ es para que en caso de que la diferencia de tiempo entre *delta_max* y *delta_min* sea muy alta, no afecte en exceso al nuevo valor de la variable *delta*.

3.4.1. Implementación

Si integramos las funciones que ofrece el algoritmo 7 en el código del algoritmo 6 obtenemos una solución que se adapta automáticamente a la carga de mensajes que esté recibiendo el protocolo en cualquier momento. Así pues, se propone el algoritmo 8 que incluye las funciones mencionadas.

La función ***update_bounds(l)*** es llamada cada vez que se recibe un mensaje nuevo (línea 19) y recibe como argumento la marca de tiempo física del mensaje recibido, para que actualice los valores de *delta_max* y *delta_min* en base a su valor.

La función ***update_delta()*** es llamada justo antes de finalizar la ejecución del hilo que se encarga de realizar la entrega de los mensajes de tipo *O-Deliver* (línea 17). De esta forma la siguiente vez que se ejecute el hilo, lo hará tras haber esperado lo equivalente al nuevo valor de la variable *delta*.

Algorithm 8 ATO-URB Algorithm v7 (FIFO channels). Code for process p_i

```
1: Variables
2: pt                                   ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                         ▷ timestmap, a pair (l, c). Updated via HLC procedures Alg. 3
4: delivered[1..N] := ((0,0),..,(0,0))  ▷ delivered[j] the last ts of a delivered message with origin p_j
5: TS := (i, ts)                        ▷ extended timestamp of node p_i, it is a pair (origin, timestamp)
6: lastTS := (0, (0, 0))                ▷ extendend timestamp of the last delivered message in order
7: Rec := ∅   ▷ a map of tuples (msg, msgTS, mpt) where msg is a message, msgTS is its extended
   timestamp (key), and mpt is the physical time when msg was received by the first time
8: delta := 0                                                   ▷ value of the parameter Δ
9: delta_max := 0                                ▷ maximum physical time read from received messages
10: delta_min := 0                               ▷ minimum physical time read from received messages
11:
12: operation Broadcast(m)
13:     update_ts-snd()
14:     TS := (i, ts)
15:     send ⟨m, TS⟩ to p_i
16: end operation
17:
18: when ⟨m, mTS⟩ is received from p_k
19:     update_bounds(mTS.ts.l)                  ▷ update delta_max and delta_min
20:     update_ts-rcv(mTS.ts)                    ▷ update timestamp of p_i on receiving
21:     if not isdelivered(mTS) then
22:         if mTS ∉ Rec then                                   ▷ first reception of m
23:             send ⟨m, mTS⟩ to Π \ {k, i}                     ▷ message m is retransmitted
24:             if mTS ≤ lastTS
25:                 U-Deliver(m)
26:                 delivered[mTS.origin] := mTS.ts
27:             else                                            ▷ mTS > lastTS
28:                 Rec := Rec ∪ {(m, mTS, pt)}
29:             end if-else
30:         end if
31:     end if
32: end when
33:
34: Task::
35: repeat  each delta time
36:     Let todeliver[ ] := sortbyTS(Rec)            ▷ Array from Rec sorted by the key
37:     Let L := todeliver.length                    ▷ Length of previous array [1..L]
38:     Let p := 1
39:     while p ≤ L and pt − todeliver[p].mpt ≥ delta do
40:         Let (m, mTS, mpt) := todeliver[p]
41:         O-Deliver(m)
42:         lastTS := mTS
43:         Rec := Rec \ {(m, mTS, mpt)}
44:         delivered[mTS.origin] := mTS.ts
45:     end while
46:     update_delta()                               ▷ update delta
47: end repeat
48:
49: function isdelivered(mTS): boolean
50:     if delivered[mTS.origin] ≥ mTS.ts then
51:         return(true)
52:     end if
53:     return(false)
54: end function
```

4. Sistema distribuido de clave valor

Como explicamos al principio de este trabajo, el objetivo de este proyecto es construir un sistema distribuido con una base de datos de tipo *clave-valor*. Una vez diseñado un algoritmo de difusión de mensajes con orden de entrega aproximado (8), podemos diseñar y posteriormente implementar el sistema.

4.1. Diseño del sistema

El algoritmo de orden aproximado debe disponer de un intermediario (llamémoslo *Server*) para comunicarse con los clientes y para realizar las operaciones contra la base de datos. La decisión de añadir este programa es para evitar cargar al protocolo de orden aproximando (llamémoslo *AOR*) con lógica innecesariamente adicional. Al estar trabajando en sistemas distribuidos, se prima evitar construir sistemas monolíticos y orientarlo más a microservicios.

Cada instancia de tipo *Server* se comunicará con una instancia del protocolo *AOR* y dispondrá de una BBDD propia. De este forma, cada servidor actuará como una réplica y, tal y como está diseñado el protocolo de orden aproximado, eventualmente todas las réplicas convergerán al mismo estado; es decir, serán réplicas idénticas.

El sistema a construir debe disponer de un balanceador de carga a nivel de red para que los clientes dispongan de servicio en cualquier momento independientemente de la carga que esté teniendo el sistema. Un balanceador de carga actúa como único punto de contacto para los clientes y distribuye el tráfico entrante entre varios destinos, aumentando así la disponibilidad de la aplicación. En caso de que uno de los *Servers* falle, el balanceador de carga redirigirá el tráfico de los clientes al resto de *Servers* que estén operativos, haciendo que el sistema sea accesible en cualquier momento.

En la figura 3 podemos visualizar el diseño y la arquitectura final del sistema.

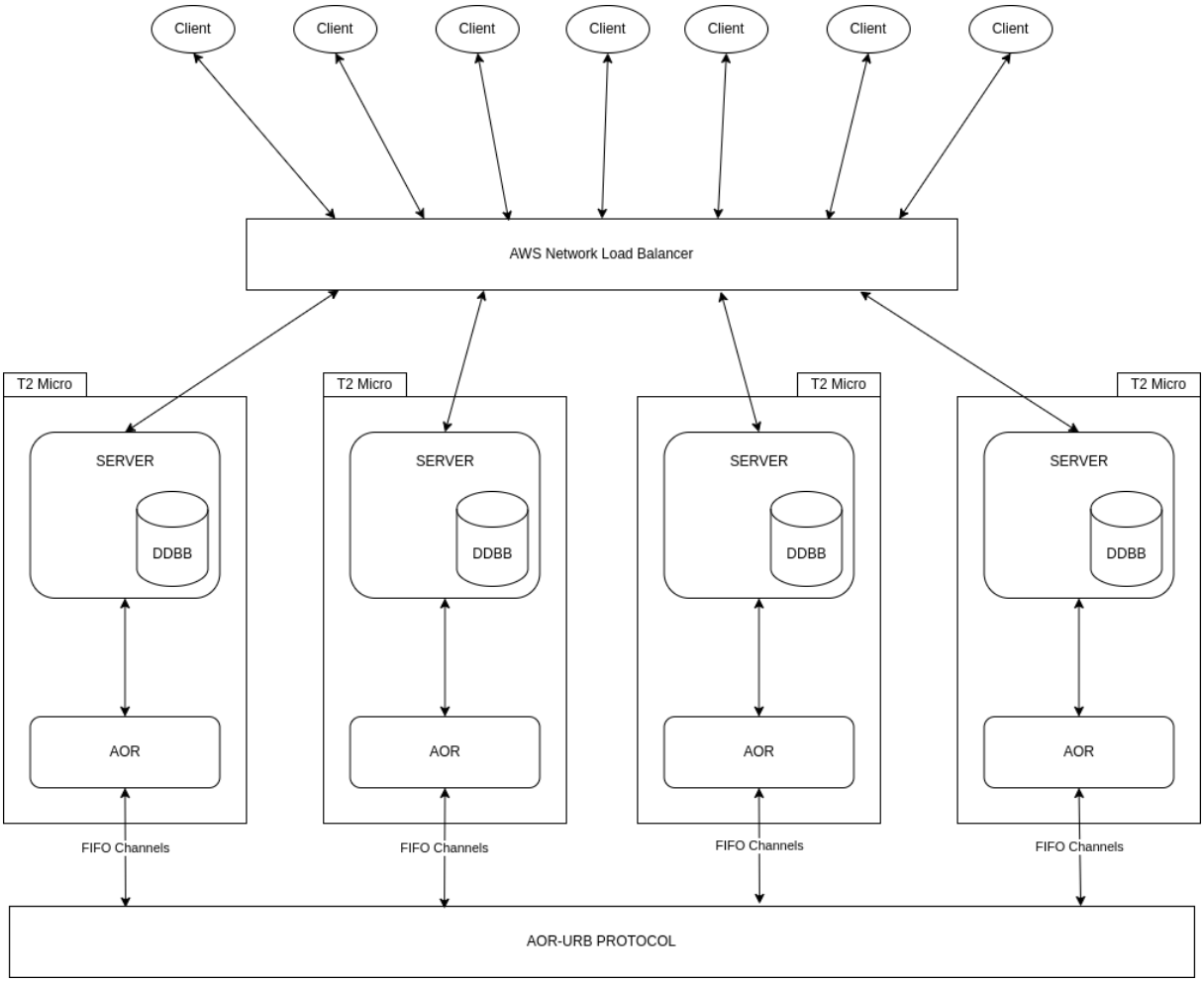

Figura 3: Arquitectura del Sistema

4.2. Implementación del sistema

Todos los componenetes del sistema se han implementado en el lenguaje de programación *NodeJS* (versión v10.19.0). Para la comunicación entre procesos hemos utilizado la librería *ZeroMQ* (versión v2.5.8), la cual utiliza sockets *TCP/IP*.

Los clientes se conectan a un *Server* y envían un mensaje que contiene una consulta a la base de datos, ya sea de escritura o de lectura. Las consultas de lectura tiene el formato {`get <key>`} y las de de escritura {`put <key, value>`}. Una vez recibida la respuesta, los clientes esperan un intervalo de tiempo, al cuál denominaremos *Thinking Time* (en adelante TT), y después envían la siguiente consulta.

El componente *Server* recibe peticiones de los clientes. Si la petición contiene una consulta de tipo {`get <key>`}, el servidor devuelve el valor de esa clave directamente de la base de datos local de la que dispone. Si la consulta es de tipo {`put <key, value>`}, el servidor encapsula esa consulta en un mensaje que respete el formato del protocolo y se lo envía al componente *AOR*. Aquí es donde la consistencia eventual juega su papel, ya que si la consulta es de lectura el servidor devuelve el valor de la lectura directamente de su base de datos local, pero si es de escritura, envía el mensaje a través del protocolo para que todos los servidores reciban esa orden de escritura y actualicen sus bases de datos locales. Como se ha explicado anteriormente, cuando las escrituras cesen, eventualmente todas las bases de datos convergerán al mismo estado.

El componente *AOR* se conecta con un único *Server* y con todos los demás componentes *AOR*. La lógica de este componente es la mimsa que al del algoritmo 8, pero adaptada al lenguaje *NodeJS*. Cuando este componente recibe un mensaje del servidor, lo envía a través del protocolo para que todos los servidores actualicen el valor de la clave contenida en la consulta en sus bases de datos locales.

4.3. Entorno de ejecución

El sistema se ha desplegado en un cluster de máquinas virtuales en *Amazon Web Services* (en adelante *AWS*). El entorno está compuesto por 9 máquinas virtuales, cada una con $1GB$ de memoria RAM y un procesador *Intel Xeon* mononúcleo de $2{,}40GHz$. Este tipo de máquina es la denominada "t2 micro" (es una instancia de mínimas prestaciones que ofrece AWS).

Las 9 máquinas están situadas en 3 zonas de disponibilidad distintas (3 en cada zona). En las figuras 4 y 5 se pueden observar las diferentes latencias entre máquinas que están en la misma zona ($\simeq$ 0,5 milisegundos) frente a máquinas que están en distintas zonas ($\simeq$ 0,8 milisegundos).

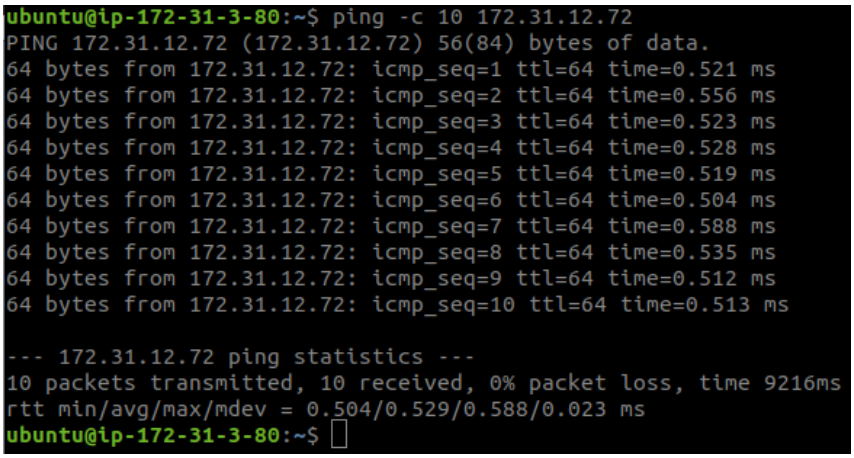

Figura 4: Latencia entre nodos de la misma zona

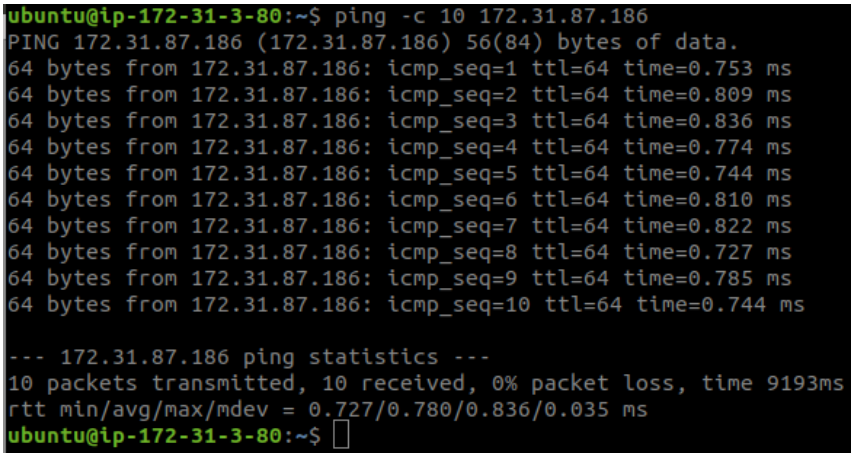

Figura 5: Latencia entre nodos de distinta zona

Las máquinas de AWS están conectadas entre sí mediante el protocolo NTP, donde una de las máquinas hace de servidor de tiempo y el resto lo consume. Si pedimos al demonio de NTP de uno de los consumidores que nos muestre información acerca de la sincronización del tiempo obtenemos lo que podemos observar en la figura 6. La sincronización tiene muy poca latencia, ya que el *delay* [3] es menor de 1 milisegundo y el *offset* [4] es muy bajo.

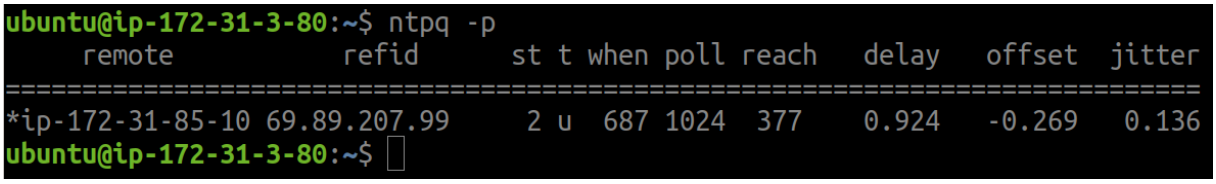

Figura 6: Información que ofrece NTP

[3] El delay es el tiempo de ida y vuelta (RTT) de un paquete. Se representa en milisegundos.

[4] El offset generalmente se refiere a la diferencia de tiempo entre una referencia de tiempo externa y el tiempo en una máquina local. Cuanto mayor sea el offset, más imprecisa será la fuente de temporización. Generalmente se mide en milisegundos.

4.4. Pruebas y resultados

Para probar el sistema, se utilizaron las 9 instancias del cluster. En cada máquina se ejecutó un proceso de tipo *Server* y un proceso de tipo *AOR*. Se realizaron 2 tipos de pruebas distintas:

4.4.1. Pruebas sin el balanceador de carga

Estas pruebas tienen como objetivo llevar el sistema a la máxima capacidad posible para estudiar sus limitaciones y ver cómo se adapta el protocolo AOR cuando la carga es demasiado alta. Al estar todos los componentes del sistema en el mismo *Data Center*, la latencia va a ser muy baja.

Se ejecutaron 9 clientes desde la misma instancia de AWS en la que se encuentran el *Server* y el *AOR*. Es decir, cada máquina ejecuta un cliente, un servidor y un AOR. Como se ha explicado antes, cada cliente envía una petición, espera hasta obtener respuesta y después espera un determinado tiempo (TT) hasta enviar la siguiente petición. En total se han realizado 6 ejecuciones con un TT de $\{0, 1, 2, 5, 10, 15\}$.

En las figuras 7-12 podemos ver los resultados de las ejecuciones. Cada figura representa los resultados de uno de los 9 nodos *Server* durante la ejecución del test. Existen 9 figuras por cada ejecución de test, pero se ha seleccionado el resultado de uno de los nodos al azar como ejemplo.

En cada figura podemos leer el número de mensajes en orden y la productividad junto con 2 gráficas cuyo significado explicaremos más adelante.

El número de mensajes en orden es un simple cálculo que se realiza para saber qué porcentaje de mensajes en orden está entregando el nodo:

$$\frac{número\_de\_mensajes\_entregados\_en\_orden}{número\_total\_de\_mensajes\_entregados} \times 100 \tag{1}$$

La productividad consiste en contar el número total de mensajes que se han recibido y dividirlo por el tiempo que ha durado la ejecución del test. Con esto obtenemos la productividad del nodo (en $mensajes/segundo$):

$$\frac{número\_total\_de\_mensajes\_entregados}{tiempo\_total\_de\_ejecución} \tag{2}$$

Para entender qué representan las 2 gráficas que aparecen por cada figura es necesario especificar algunos detalles sobre la implementación. En el algoritmo adaptativo 8 que hemos implementado en *NodeJS*, la línea 41 daba problemas cuando el número de mensajes en orden a entregar era demasiado grande, ya que se genera un evento de tipo *O-Deliver()* por cada mensaje y eso hace que la cola de eventos crezca de manera desproporcionada, haciendo que el proceso se ralentice y se quede "atascado" recorriendo el array mientras el hilo principal sigue introduciendo mensajes en *Rec*.

Para solucionar este problema, se propone la siguiente modificación (9) que almacena los mensajes a enviar en un array (máximo 8) y, tras salir del bucle, envía todos los mensajes en

forma de array. Nótese que también se envía el el valor de *delta* en ese momento.

Algorithm 9 Algoritmo Mejorado. Code for process p_i

```
1:  Task::
2:  repeat  each delta time
3:      Let todeliver[] := sortbyTS(Rec)          ▷ Array from Rec sorted by the key
4:      Let L := todeliver.length                 ▷ Length of previous array [1..L]
5:      Let p := 1
6:      Let messages_to_deliver := ∅
7:      while p ≤ L and p ≤ 8 and pt − todeliver[p].mpt ≥ delta do
8:          Let (m, mTS, mpt) := todeliver[p]
9:          lastTS := mTS
10:         Rec := Rec \ {(m, mTS, mpt)}
11:         delivered[mTS.origin] := mTS.ts
12:         messages_to_deliver := messages_to_deliver ∪ {m}
13:     end while
14:     if messages_to_deliver != ∅ then
15:         O-Deliver(messages_to_deliver, delta)
16:     end if
17:     update_delta()                            ▷ update delta
18: end repeat
```

Las gráficas muestran el número de mensajes que se han enviado y el valor de Δ en cada evento. Estos valores nos ayudan a determinar cuánto se ha saturado cada nodo durante la ejecución del test. Podemos observar que cuanto más bajo es el TT más alta es la productividad (o *throughput*) y más mensajes se envían por cada evento. En consecuencia, la variable *delta* se tiene que adaptar a la alta demanda y eso produce que tenga picos en los que llega a alcanzar el máximo valor permitido por el algoritmo 7. Es decir, cuanto menos esperan los clientes para enviar el siguiente mensaje, más se satura la red, pero cuando empiezan a esperar más tiempo entre mensaje y mensaje, la carga va disminuyendo y el Δ y el número de mensajes que se envían por cada evento comienzan a bajar.

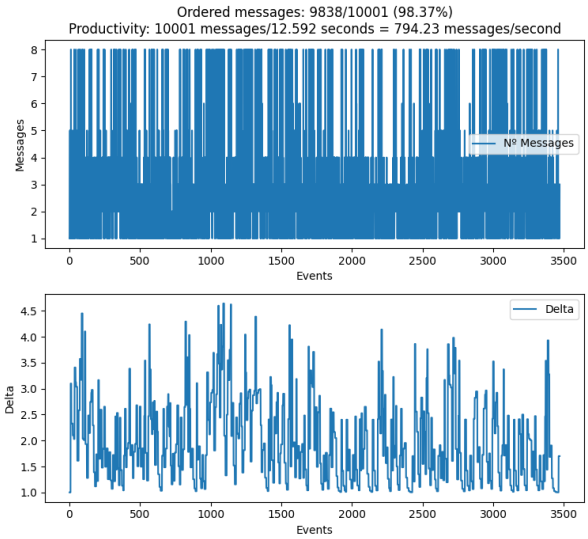

Figura 7: *Thinking Time* 0 milisegundos

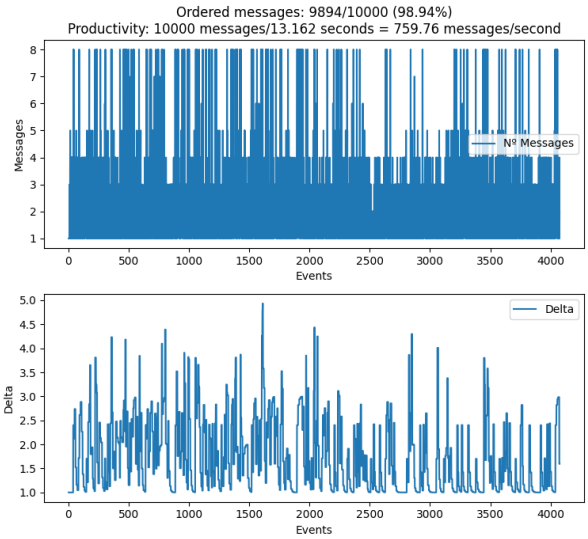

Figura 8: *Thinking Time* 1 milisegundos

Es de destacar que a medida que aumenta el *Thinking Time* también lo hace el número de mensajes entregados en orden. Esto sucede porque al disminuir la carga del sistema, el algoritmo se satura menos y es capaz de entregar más mensajes en orden.

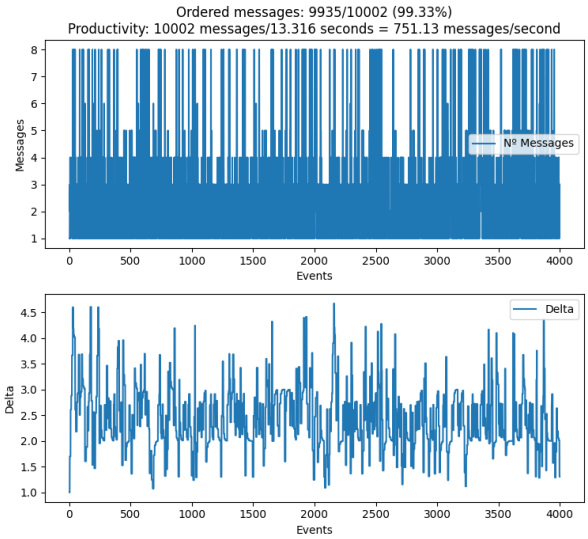

Figura 9: *Thinking Time* 2 milisegundos

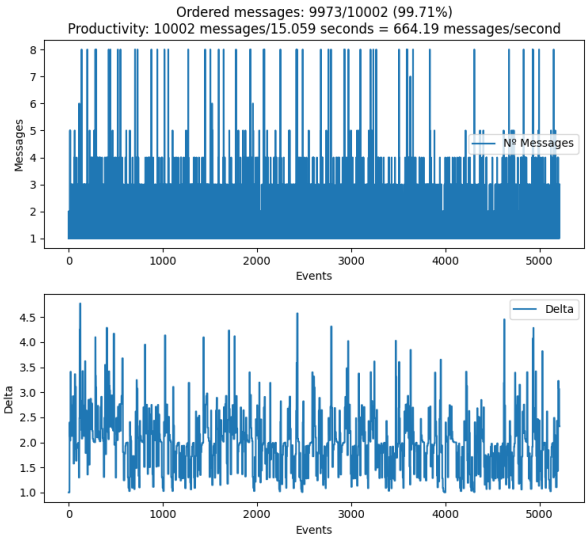

Figura 10: *Thinking Time* 5 milisegundos

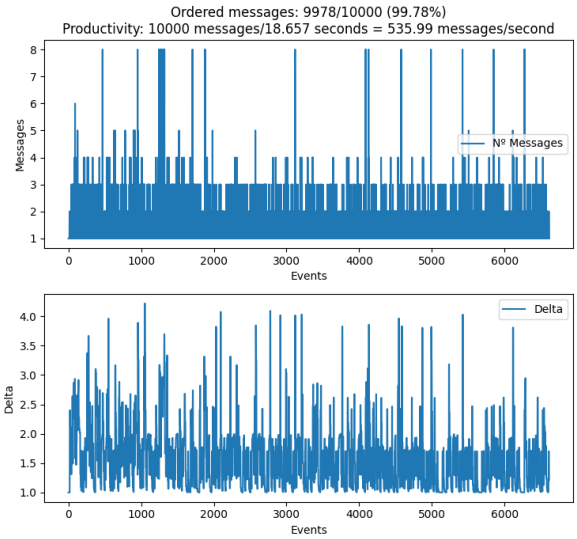

Figura 11: *Thinking Time* 10 milisegundos

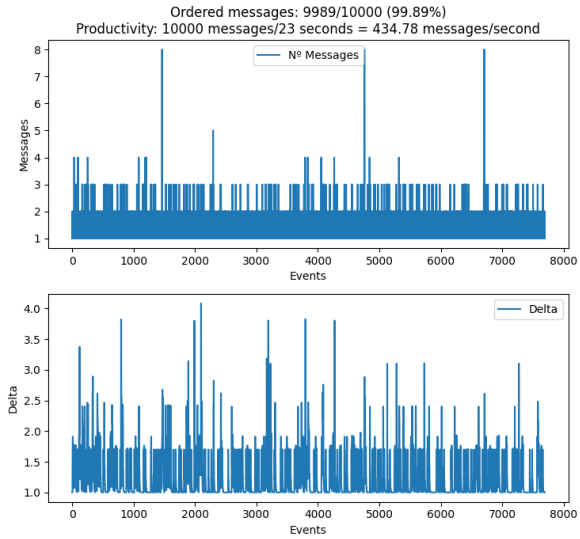

Figura 12: *Thinking Time* 15 milisegundos

4.4.2. Pruebas con el balanceador de carga

En estas pruebas se ejecutaron 18 clientes desde un ordenador situado en el *laboratorio de Lenguajes y Sistemas Informáticos*. El objetivo de estas pruebas es simular cómo se comportaría el sistema diseñado en un entorno productivo real en el que los clientes pueden hacer peticiones desde cualquier parte del mundo y donde la latencia es más alta ($\simeq$ 100 milisegundos). En total se han realizado 9 ejecuciones con un TT de $\{0, 1, 2, 5, 10, 25, 50, 100, 1000\}$.

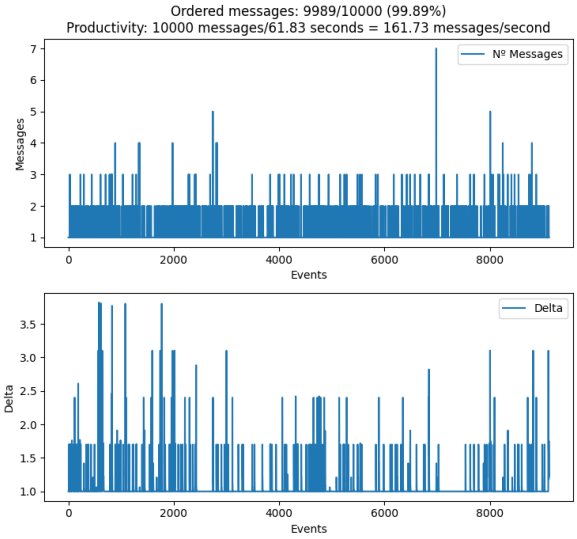

Figura 13: *Thinking Time* 0 milisegundos

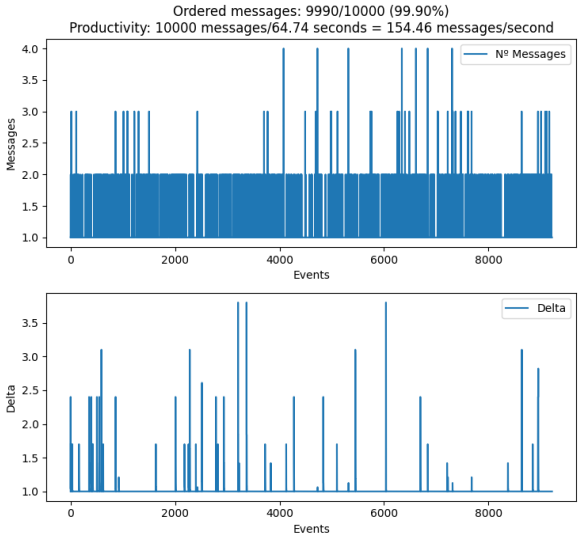

Figura 14: *Thinking Time* 1 milisegundos

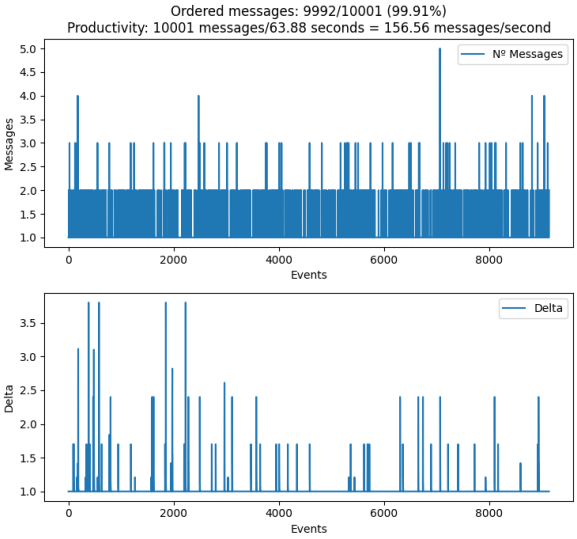

Figura 15: *Thinking Time* 2 milisegundos

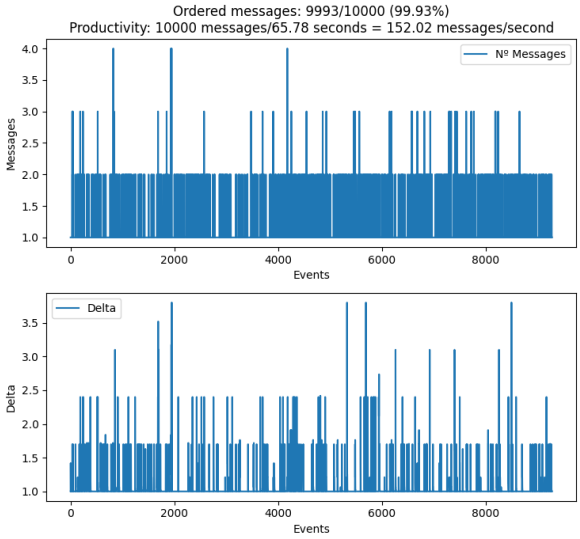

Figura 16: *Thinking Time* 5 milisegundos

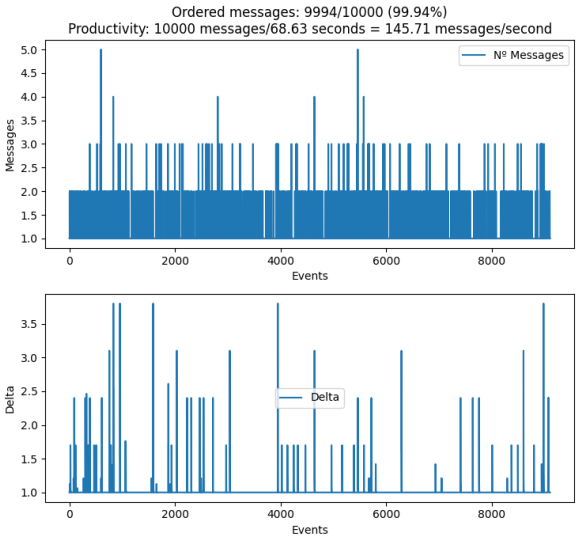

Figura 17: *Thinking Time* 10 milisegundos

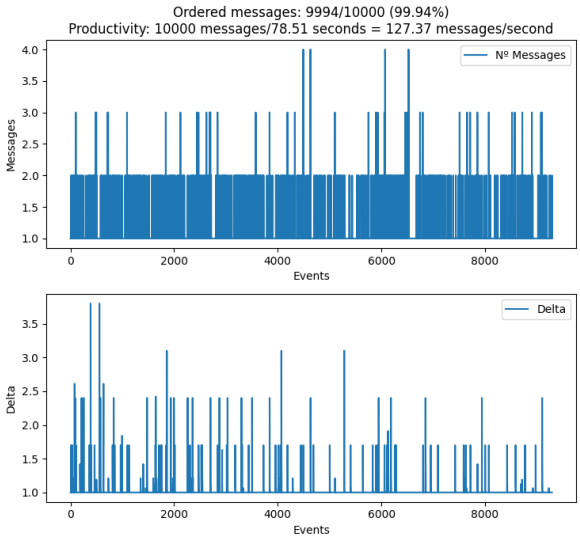

Figura 18: *Thinking Time* 25 milisegundos

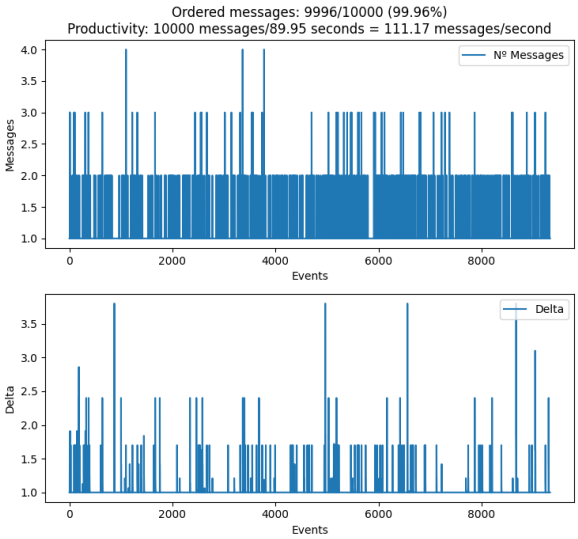

Figura 19: *Thinking Time* 50 milisegundos

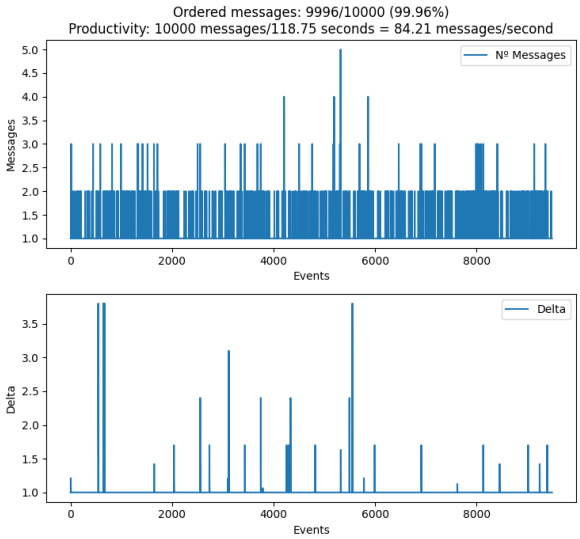

Figura 20: *Thinking Time* 100 milisegundos

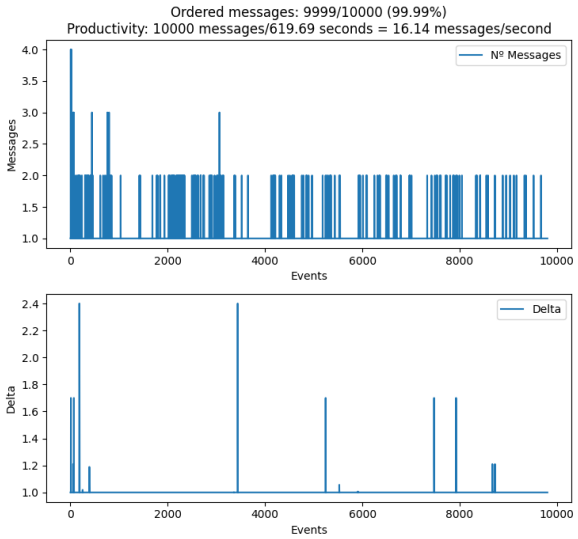

Figura 21: *Thinking Time* 1000 milisegundos

En las figuras 13-21 podemos ver los resultados de las ejecuciones. Observamos que en estas pruebas la productividad es muy baja en comparación con las pruebas realizadas en el apartado anterior. En estas pruebas, cuando el *Thinking Time* es de 0 milisegundos, hemos obtenido una productividad de 161,73 $mensajes/segundo$, mientras que en el apartado anterior hemos obtenido una productividad de 794,23 $mensajes/segundo$. Esto se debe a que la alta latencia provoca que los clientes tarden más tiempo en enviar y recibir los mensajes y, aunque haya 18 clientes (el doble que en las pruebas del apartado anterior), no consigen saturar la red del protocolo.

Como consecuencia, el número de mensajes que se envían cada evento rara vez es mayor de 4 y el delta adopta valores más bajos y más estables. También destaca que en todas las ejecuciones realizadas, el porcentaje de mensajes entregados en orden es superior al 99,9 %.

5. Conclusiones

Como se ha podido observar en los resultados obtenidos, el sistema es completamente operativo y puede garantizar una consistencia eventual sólida, ya que el protocolo AOR es capaz de entregar un $99,9\,\%$ de mensajes en orden incluso cuando la carga es muy alta. Además, es tolerante a fallos; es decir, está preparado para adaptarse en caso de que algunos nodos fallen para seguir funcionando con normalidad y que el servicio esté siempre disponible.

Este sistema de almacenamiento distribuido de *clave-valor* es en esencia una base de datos distribuida. Con unas pocas modificaciones se podría productivizar y ofrecerse como un servicio de almacenamiento. Las aplicaciones de una base de datos distribuida son muy variadas. Puede emplearse en entornos de fabricación, especialmente cuando la producción se lleva a cabo en múltiples plantas, también es la opción escogida por las aerolíneas y cadenas de hoteles, grupos multinacionales y, en general, la alternativa que se plantea cualquier organización con una estructura organizativa descentralizada.

Este proyecto ha resultado muy satisfactorio en lo que a realización presonal se refiere, pero no refleja todo el trabajo que ha conllevado para llegar a unos resultados tan prometedores. Se ha realizado una investigación previa muy extensa sobre el protocolo *URB* y los algoritmos distribuidos, la cual se ha publicado como una memoria de investigación. Adicionalmente, esta investigación y parte de los resultados han sido de ayuda para publicar un artículo que se presentó en las Jornadas de Concurrencia y Sistemas Distribuidos 2022. Este documento se encuentra anexo al final de esta memoria.

Referencias

[1] E. Brewer. CAP twelve years later: how the rules have changed. IEEE Computer, February, 2012.

[2] P. Bailis and A. Ghodsi. Eventual Consistency Today: Limitatios, extensions, and beyond. ACMQUEUE, vol 11 (3), 2013.

[3] K. Birman et. al. Reliable Communication in Presence of Failures. ACM Transactions on Computer Systems, vol 5(1), 1987.

[4] J. Chang and N. Maxemchuck. Reliable Broadcast Protocols. ACM Transactions on Computer Systems, vol 2 (3), 1984.

[5] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcast and Related Problems. Technical Report, Cornell University, 1994.

[6] M. Fischer, N. Lynch and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, vol 32 (2), 1985.

[7] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, vol 43 (2), 1996.

[8] X. Defago. Total Order Broadcast and Multicast Algorthms: Taxonomy and Survey. ACM Computing Surveys, vol 36 (4), 2004.

[9] C. Marchetti et. al. Fully Distributed Three-Tier Active Software Replication. IEEE Trans. on Parallel and Distributed Systems, vol 17 (7), 2006.

[10] X. Zhao, P. Haller. Replicated data types that unify eventual consistency and observable atomic consistency. Journal of Logical and Algebraic Methods in Programming, vol 114, 2020.

[11]

[12] F. Pedone, A. Schiper. Optimistic Atomic Broadcast. in DISC, 1998.

[13] B. Kemme, et. al. Processing transactions over optimistic atomic broadcast protocols. In 19th International Conference on Distributed Computing Systems, 1999.

[14] A. Sousa et. al. Optimistic Total Order in Wide Area Networks. in SRDS, 2002.

[15] M. Raynal. Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. Synthesis lectures on Distributed Computing Theory, Morgan&Claypool Publishers, 2010.

[16] S. Kulkarni, et. al. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases. Technical report cse.bufalo.edu, 2014.

[17] D. Mills. A brief history of NTP time: Memoirs of an internet timekeeper. ACM SIGCOMM Computer Communication Review, vol 33 (2), 2003.

Approximate Order Reliable Broadcast

Aitor Mendívil-Grau
Dpto. Estadística, Informática y Matemáticas. Universidad Pública de Navarra, Campus Arrosadía s/n, Pamplona, Spain

Iulen Salinas
Dpto. Estadística, Informática y Matemáticas. Universidad Pública de Navarra, Campus Arrosadía s/n, Pamplona, Spain

J.R. González de Mendívil
Dpto. Estadística, Informática y Matemáticas. Universidad Pública de Navarra, Campus Arrosadía s/n, Pamplona, Spain

Abstract

Total order broadcast demands consensus in order to ensure that all the participating processes deliver the broadcast messages in the same order. Consensus may not be achieved in asynchronous systems where processes may fail. However, the spontaneous order in reliable broadcast protocols may be close to total order when those protocols are deployed on local area networks. Additionally, if messages are tagged in their broadcast action using globally ordered identifiers or (logical) clock values and shortly buffered at reception time, then the resulting approximate total order will only deliver very few unordered messages. We propose an adaptive approximate (total) order algorithm based on those mechanisms. The results of an experimental evaluation confirm that it provides a good throughput and its number of messages delivered in an unordered way may be easily kept below 1%, even when processes are deployed in different zones of a data center.

2012 ACM Subject Classification Dependable and fault-tolerant systems and networks → Availability; Dependable and fault-tolerant systems and networks → Redundancy; Network protocols → Application layer protocols; Distributed computing methodologies → Distributed algorithms; Middleware for databases → Data replication tools

Keywords and phrases Reliable broadcast, Total-order broadcast, Approximate-order broadcast

Acknowledgements We want to thank the company Veridas (veridas.com) for providing the infraestructure to carry out the experimental tests of this paper. We also want to thank Prof. F. Muñoz (Universidad Politécnica de Valencia) and Prof. F. Fariña (Universidad Pública de Navarra) for their comments that have helped us improve the content of this paper.

1 Introduction

The problem of message broadcasting among a group of processes in fault-tolerant environments is a problem that appears in many distributed coordination problems. Reliable Broadcast (R-Broadcast) allows the set of correct processes in the system (those that have not failed in the execution) to receive the same set of messages. This property is sufficiently weak to allow us to design a R-Broadcast protocol in an asynchronous distributed system in where an arbitrary number of processes may crash (*crash failures*). From a practical perspective, R-Broadcast is useful, e.g., data replication distributed systems with *eventual consistency* [4] can be implemented by using an R-Broadcast protocol and Last Write Wins rule [2]. Initial works on R-Broadcast are found in the 1980s [3, 6]. Hadzilacos and Toueg [10] present an excellent summary of R-Broadcast and related problems. They show how to implement R-Broadcast protocols with additional delivery guarantees in a modular way, e.g., FIFO and Causal Order (CO) delivery guarantees can be built on top of a simple R-Broadcast. Extending the quality of service of R-Broadcast has practical implications. For instance, total order (TO) message delivery is adequate for simplifying the design of data

replication systems with *atomic consistency* [15] or with eventual consistency and *atomic visibility* for some type of operations [20].

Unfortunately, TO-R-Broadcast cannot be implemented in an asynchronous distributed system with crash failures. This is due to (i) the equivalence of TO-R-Broadcast problem and *distributed agreement problem* [10]; and (ii) the *FLP impossibility* result [9]. FLP impossibility determines that the distributed agreement problem cannot be solved with a deterministic algorithm in this kind of systems. However, its implementation is possible in the so called *partially synchronous* systems [5]. Typically, solutions for distributed agreement also require a majority of correct processes. Défago [8] surveys the mechanisms that have been used over the years to obtain TO-R-Broadcast protocols.

It is a well-known fact that when R-Broadcast is implemented in a cluster of nodes over a Local Area Network (LAN), delivered messages at different processes appear in same order most of the time. This idea is exploited by the designers of data replication systems in an optimistic way: (i) the broadcast message is optimistically delivered and the application executes its operation; (ii) by means of another TO-R-Broadcast protocol, the confirmation of the order of the messages already delivered in an optimistic way is expected; (iii) those messages already delivered, but in disorder, force the action already executed to be rolled back. This optimistic way of proceeding is efficient when the coincidence between the optimistic order and that produced by the TO-R-Broadcast protocol coincide most of the time. Pedone uses this optimistic technique in transactional systems [18, 12], and Sousa provides a solution in Wide Area Networks [7].

In general, designers of data replication systems must be careful in choosing the communication environment and geographic area where TO-R-Broadcast protocols have to be deployed. From a theoretical point of view, this area has to be a region with partial synchrony (to guarantee message delivery), which in practice translates into regions where most of the time their behavior is synchronous. Thus, it is reasonable to deploy these protocols on a LAN or a Data Center (DC) over a specific geographic region (from those data centers offered by infrastructure providers).

In this paper, we address the design of an R-Broadcast protocol in which processes are aware that some messages are delivered in order and that the local (total) orders, that are established by these ordered deliveries, are compatible among different processes. We call this protocol Approximate Order Reliable Broadcast (AOR-Broadcast). Since not all messages are delivered in the same (total) order in all the processes, it is possible its implementation in an asynchronous distributed system with arbitrary number of crash failures as opposed to what happens with TO-R-Broadcast or Optimistic TO-R-Broadcast protocols. On the other hand, in the same practical distributed environments where it is possible to implement TO-R-Broadcast, we analyze the amount of messages delivered in the same order among processes. This measure establishes the degree of approximation of the AOR-Broadcast with respect to an ideal TO-R-Broadcast. In addition, we propose adaptation mechanisms so that the amount of ordered messages delivered at each process may be increased in these practical environments.

The rest of the paper is organized as follows. Section 2 defines and specifies AOR-Broadcast. Section 3 provides a basic algorithm that complies with that specification and experimentally evaluates its throughput and approximate order measure. Section 4 proposes a second algorithm that dynamically adapts message management in order to increase totally ordered delivery and compares its performance with that of the basic algorithm. In Section 5, we outline a possible impossibility result related with the specification of AOR-Broadcast. Finally, Section 6 concludes the paper.

2 Specification of AOR-Broadcast

The *Approximate Order Reliable Broadcast* protocol (AOR-Broadcast) proposed in this paper allows a process to broadcast messages, and to deliver messages in two different ways: u-deliver(.) and o-deliver(.). Both types, u-deliver(.) and o-deliver(.), are delivery events that satisfy the common properties of a simple Reliable Broadcast [10, 19]. However, o-deliver(.) events are intended to indicate the process that the set of messages delivered in this way satisfy a simple local (total) order relation[1]. Local order relations of o-deliver events in different processes are compatible. Basically, if two different processes p_i and p_j o-deliver the same pair of messages m and m', then p_i o-delivers m before m' if and only if p_j o-delivers m before m'. Let us observe that this property is the common safety property used in the TO-R-Broadcast [10] specification. In our case, this property is applied only to o-deliver messages. In the following, we provide a more formal specification of AOR-Broadcast.

2.1 Distributed system model

Let us consider a distributed system that is composed of a set of $N \geq 2$ asynchronous sequential processes. Processes are identified by p_i (or simply by the number i) with $i \in P$, where $P = \{1, 2..., N\}$. Processes are asynchronous and sequential, i.e., each process proceeds at its own speed. Each process behaves correctly according to its local algorithm until it possibly *crashes*. After a crash event, a process stops its execution. A process is *correct* if it never crashes, otherwise it is *faulty*. We assume that there is at least one correct process in every execution of the system, i.e., the possible number of processes that may fail in an execution is at most $N - 1$.

Processes may communicate among them by sending and receiving messages over asynchronous reliable channels. However, in order to simplify the presentation, we assume that the distributed system is equipped with a R-Broadcast protocol. In fact, R-Broadcast can be implemented in the considered distributed system (see some implementations in [10, 19]). The basic R-Broadcast protocol provides the operation broadcast(m). This operation is executed by a process to send m to all participant processes. When the event deliver(m) is triggered at a process, it delivers m to that process. The R-Broadcast is characterized by the following three properties [19], where it is assumed that all broadcast messages are different[2]:

(**B1**) *Validity.* If a process delivers a message m, then m was broadcast by a process.
(**B2**) *Integrity.* A message is delivered at most once by each process.
(**B3**) *Termination.* If a *correct* process (a) broadcasts a message m or (b) delivers a message m, then each *correct* process eventually delivers m.

In the previous specification, we have followed the common terminology: (i) when a process invokes the operation broadcast(m), we say that it 'broadcasts a message m'; and (ii) when deliver(m) is triggered at a process, we say that it 'delivers m'.

In the proposed distributed system, there is not a notion of global true-time that processes are able to utilize in their algorithms: each process may only use their local clock and different local clocks are not synchronized. However, we assume that there exists an omniscient observer that can specify or derive some properties of the system by using a real time notion. For

[1] This notion is more simple than the *local order* property used to show that *fifo+local order* is equivalent to *causal order* [10].
[2] Messages are considered unique without any other assumption, i.e., for two different broadcast operations sending m and m', we only know that $m \neq m'$.

example, for each process p_i, let us define $D_i(\tau)$ as the set of messages delivered by p_i up to real time τ via the R-Broadcast protocol. By (**B3**), if a correct process p_i has delivered $D_i(\tau)$ up to time τ, then there is a time $\tau' \geq \tau$ such that for any other correct process p_j, $D_j(\tau') \supseteq D_i(\tau)$ holds. This is the reason to say informally that correct processes deliver the same set of messages using a R-Broadcast protocol.

2.2 Definition of AOR-Broadcast

AOR-Broadcast provides processes with the operation AO-broadcast(m) to broadcast a message m, and two types of delivery events, u-deliver(.) and o-deliver(.). Again, all messages that are AO-broadcast are different. When u-deliver(m) or o-deliver(m) have been triggered at process p_i, in both cases, we say that p_i 'delivers m', and we use the binary classification given by the letters 'u' or 'o' to specify the particular type of event. With this simple description, the AOR-Broadcast must satisfy the properties of *validity*, *integrity* and *termination* for any deliver event (see properties (**B1**) to (**B3**) above) of a R-Broadcast.

Let us denote by $O_i(\tau)$ ($U_i(\tau)$) the set of o-delivered (u-delivered resp.) messages by a process p_i up to time τ. The set of all delivered messages at p_i up to τ is $D_i(\tau) = O_i(\tau) \cup U_i(\tau)$. Obviously, by (**B2**), at any τ and τ' times, $O_i(\tau) \cap U_i(\tau') = \emptyset$, i.e., these sets are disjoint at any time. Here, we establish the main difference between these sets at process p_i. For any time $\tau \geq 0$, each set $O_i(\tau)$ has associated a total order relation, denoted $<_{O_i}$, that is defined as follows: for any m, $m' \in O_i(\tau)$, $m <_{O_i} m'$ if and only if o-deliver(m) happens before o-deliver(m') at process p_i. Each relation $<_{O_i}$ is a *local relation* for each process p_i. However, we establish that these local relations are compatible. Thus, AOR-Broadcast must satisfy the property:

(**AOB4**) *Approximate Order*. If two processes (correct or faulty) p_i and p_j satisfy $\{m, m'\} \subseteq O_i(\tau) \cap O_j(\tau)$ at time τ then $m <_{O_i} m' \Leftrightarrow m <_{O_j} m'$

A simple example of the AOR-Broadcast delivery guarantees appears in Figure 1.

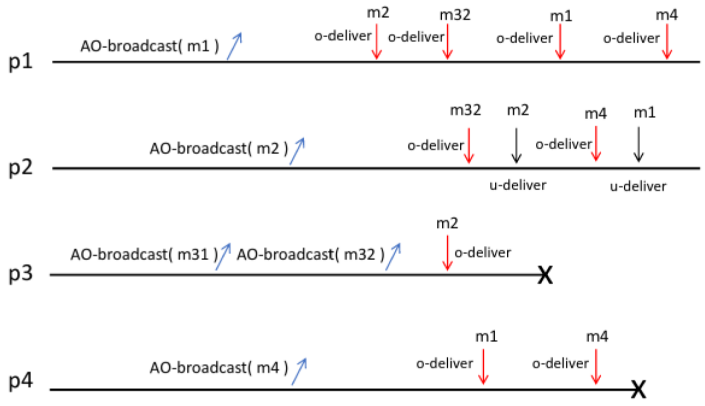

Figure 1 An example of the AOR-Broadcast delivery guarantees

In this example, processes p_1 and p_2 are correct while p_3 and $p4$ are faulty. From the perspective of the properties (**B1**)-(**B3**), the correct processes deliver the same set of messages D while each faulty process deliver a subset of D. Let us observe that $p3$ delivers

none of the messages $m31$ and $m32$ that it has broadcast but $m32$ that has been sent after $m31$ is delivered by the correct processes while $m31$ is not. With respect to the property (**AOB4**): $O_1 \cap O_2 = \{m32, m4\}$. These o-delivered messages satisfy $m32 <_{O_1} m4$ at p_1 and $m32 <_{O_2} m4$ at p_2. In the faulty process p_4, $O_1 \cap O_4 = \{m1, m4\}$. Again, $m1 <_{O_1} m4$ at p_1 and $m1 <_{O_4} m4$ at p_4. The execution satisfies the *approximate order* property that applies to o-delivered messages only.

Let us observe that the previous guarantees for an AOR-Broadcast protocol do not exclude the possibility of constructing trivial algorithms satisfying them; e.g., a reliable broadcast protocol that only produces u-deliver(.) events. Given an execution α that satisfies the previous guarantees, we can define the correct process in α by the set of non-faulty processes in α denoted by $Correct(\alpha)$. In order to exclude trivial solutions, AOR-Broadcast must guarantee the following property:

> (**AOB5**) *Non triviality.* For any $k \geq 0$ and any subset $S \subseteq 2^P$ with $S \neq \emptyset$, there exists an execution α and a time τ such that $S = Correct(\alpha)$ and $k = \| \bigcap_{i \in S} O_i(\tau) \|$

This guarantee establishes that there are executions that provide some amount of ordered messages via o-deliver(.) events and in an extreme case, all messages may be ordered for all the correct processes by the *Approximate Order* property. In addition, we can measure the amount of ordered messages in a given execution up to time τ by the equation,

$$\textit{Approximate Order Measure: } AO(\tau) = \frac{\| \bigcap_{i \in P} O_i(\tau) \|}{\| \bigcup_{i \in P} D_i(\tau) \|} \tag{1}$$

3 An implementation of AOR-Broadcast

Let us provide an algorithm to solve the AOR-Broadcast on top of a R-Broadcast protocol. In the previous specification (section 2), property (**AOB4**) establishes that for two different processes i and j the local (total) orders $<_{O_i}$ and $<_{O_j}$ have to be compatible between them when cardinal $\|O_i \cap O_j\|$ is greater than or equal to 2 in an execution. The simplest way to obtain this compatibility is that all delivered messages be *comparable* using a totally ordered information associated to the messages. As any timestamp mechanism (e.g., Lamport's logical clocks [14]) provides a total order in the set of the events of an execution, the proposed solution is based on a timestamp mechanism. In addition, the obtained algorithm is non-blocking. Algorithm 1 uses a variable *last_ets* to store the timestamp of the latest o-delivered message. In that case, the code (in line 14) compares the timestamp of next delivered message by the underlaying R-Broadcast in order to decide if it is greater than the *last_ets*. If this happens the message is o-delivered and the *last_ets* variable is updated (lines 17-18). Otherwise the message is u-delivered (line 15). This simple idea assures that for each process i, $<_{O_i}$ is a total order and as all messages are *totally ordered* by the timestamps, these local orders are compatible between them.

Notes about correctness of Algorithm 1. Let us observe that the associated operations for timestamps (update_ts(.) and update_ts-rcv(.)) assure the common property: $e \, hb \, f \Rightarrow e.ts < f.ts$ for two events e and f where hb denotes the happen before relation. The correctness of Algorithm 1, properties (**B1**) to (**B3**) of AOR-Broadcast, relies on the properties (**B1**) to (**B3**) of the underlaying R-Broadcast protocol. Since every AO-broadcast message is different then AO-broadcast$_i(m)$ happens only once in an execution for each message m. Thus broadcast$_i(\langle m, ets \rangle)$ happens only once in an execution for each message m. Then, there are not two different occurrences of deliver$_j(\langle m, mets \rangle)$ and deliver$_k(\langle m, mets' \rangle)$ with $mets \neq mets'$ for the same message m in an execution. In addition, lines 14-19 assure that

Algorithm 1 AOR-Broadcast Algorithm. Code for process p_i

```
1: Variables
2: ts := ⊥                                   ▷ timestmap. ⊥ denotes its minimum value.
3: ets := (ts, i)                            ▷ extended timestamp: a pair (timestamp, origin)
4: last_ets := (ts, 0)                       ▷ extendend timestamp of the last o-delivered message
5:
6: operation AO-broadcast(m)
7:     update_ts()
8:     ets := (ts, i)
9:     broadcast(⟨m, ets⟩)
10: end operation
11:
12: when ⟨m, mets⟩ is delivered              ▷ delivered by the underlaying reliable broadcast
13:     update_ts-rcv(mets.ts)               ▷ update timestamp of p_i
14:     if mets ≤ last_ets
15:         u-deliver(m)
16:     else                                 ▷ mets > last_ets
17:         last_ets := mets
18:         o-deliver(m)
19:     end if-else
20: end when
```

o-deliver$_i$(m) is exclusive with respect to u-deliver$_i$(m). Thus, $O_i(\tau) \cap U_i(\tau) = \emptyset$ for any execution and time τ. These previous observations allow us to conclude that properties (**B1**) to (**B3**) of AOR-Broadcast are fulfilled by Algorithm 1 because (**B1**) to (**B3**) are satisfied by the underlaying R-Broadcast.

Property (**AOB4**) may be proved by contradiction. To this end, let us note that in each process i, (a) $last\_ets_i$ satisfies that is greater or equal to the value $mets$ of the last o-delivered message $\langle m, mets\rangle$; and (b) the set of extended timestamps (see ets in line 3) is a totally ordered set. Then, let us consider as the base hypothesis that $\{m, m'\} \subseteq O_i(\tau) \cap O_j(\tau)$ at time τ, and w.l.o.g. $m <_{O_i} m'$ in p_i, but in p_j: $m' <_{O_j} m$. Then, $m <_{O_i} m'$ implies that $mets_m < mets_{m'}$ at p_i. Because of (b), $mets_m < mets_{m'}$ will also hold at p_j. So, if m was delivered before m' at p_j, then $m <_{O_j} m'$ and this contradicts the base hypothesis. However, it may also happen that m' be delivered before m at p_j. If that arises, at m delivery to p_j, its $mets_m$ will be lower than $last\_ets$ (note that the latter is greater or equal to $mets_{m'}$, since p_j has already delivered m'). This makes true the condition assessed at line 14 and compels to u-deliver m. Again, this contradicts the base hypothesis, since m was not o-delivered in this second case. Therefore, in both cases, $m' <_{O_j} m$ was false, and a contradiction has been reached. Thus, $m <_{O_i} m' \wedge m <_{O_j} m'$ holds and (**AOB4**) is proven.

The Non Triviality Property (**AOB5**) is proven by induction on $k \geq 0$. Let us assume an execution $\alpha = \Phi_0\pi_1\Phi_1\pi_2...\Phi_{k-1}\pi_k\Phi_k...$ [3]. Let S be the nonempty subset of correct processes in α. Let us assume that for some index τ, $k = \|\bigcap_{i\in S} O_i(\tau)\|$ holds. Then, we delete from α every action $\pi_{\tau'}$ with $\tau' > \tau$. The new execution β is a prefix of α and, thus, it is also an execution. Then, we build $\beta' = \beta$ AO-broadcast$_p$(m) $\Phi_{\tau+1}$ where process p takes the maximum value of $last\_ets_p$ among the processes in S. As AO-broadcast$_p$(m) is an input operation, β' is a valid execution. Since p is correct and $ets_p \geq last\_ets_p$, $\langle m, ets_p\rangle$ may be o-delivered by all the correct processes without violating that the system is asynchronous.

[3] We use the distributed model proposed in Chp. 7 of the book of Attiya and Welch [1].

Therefore, there exists an execution such that $k+1 = \|\bigcap_{i \in S} O_i(\tau')\|$ for some τ'.

3.1 Experimental evaluation

As we have indicated in the introduction, we are going to test the AOR-Broadcast protocol (Algorithm 1) in a practical distributed environment where we could implement a TO-R-Broadcast. Algorithm 1 has been implemented in NodeJS (version v10.19.0). For the communication among processes we have used the ZeroMQ [11] library (version 5.2.8). The protocol has been deployed in an Amazon data center. That deployment uses nine T2micro VMs, in three availability zones, with three VMs per zone. The latency between machines in the same zone is approximately 0.51-0.53 ms and between different zones is 0.74-0.81 ms. On the other hand, the timestamp mechanism used in the implementation of the algorithm is HLC [13] (see Appendix A for further details). Machines are synchronized via NTP. Each client broadcasts messages using the protocol.

In the tests, when the client delivers its last broadcast message, it awaits a thinking time and then broadcasts a new message again. About 10000 messages have been broadcast in the tests and all processes are correct. In this way, we measure the capacity of the protocol, i.e., its throughput: the number of delivered messages per second. The rest of measures are shown in Table 1 and Table 2.

Thinking Time 0 ms		
Process	Ordered O_i	$Throughput_i$
p1	$\frac{6872}{10000}$ (68.72%)	$\frac{10000}{14{,}692s}$ (680.64 mgs/s)
p2	$\frac{6641}{10000}$ (66.41%)	$\frac{10000}{14{,}628s}$ (683.62 mgs/s)
p3	$\frac{6270}{10000}$ (62.70%)	$\frac{10000}{14{,}640s}$ (683.06 mgs/s)
p4	$\frac{7241}{10000}$ (72.41%)	$\frac{10000}{14{,}638s}$ (683.15 mgs/s)
p5	$\frac{7776}{10000}$ (77.76%)	$\frac{10000}{14{,}655s}$ (682.36 mgs/s)
p6	$\frac{8227}{10000}$ (82.27%)	$\frac{10000}{14{,}663s}$ (681.99 mgs/s)
p7	$\frac{6738}{10000}$ (67.38%)	$\frac{10000}{14{,}599s}$ (684.98 mgs/s)
p8	$\frac{7809}{10000}$ (78.09%)	$\frac{10000}{14{,}671s}$ (681,62 mgs/s)
p9	$\frac{6721}{10000}$ (67.21%)	$\frac{10000}{14{,}675s}$ (681,43 mgs/s)

Table 1 Percentage of o-delivered messages at each process using Algorithm 1. Process throughput, when Thinking Time is 0ms, is at the third column. The Approximate Order measure, AO (eq. 1), is 59.87%. Average throughput is 682.43 mgs/s with 1.23 of variance.

Table 1 shows that when the tests introduce no thinking time, then the maximum achievable throughput is slightly greater than 680 msg/s. In that case, with no sending pause, the approximate order measure does not exceed 60%. That means that only fewer than 60% of the broadcast messages have been delivered in total order in all participating processes.

If longer pauses are introduced between successive broadcast actions at each sender (i.e. longer thinking times) then the probability of unordered delivery decreases, as illustrated in Table 2, alghough the obtained throughput also lessens. This suggests that some inter-message pause is needed in order to increase the approximate order measure. However, that pause may be also introduced at reception time (instead of at sending time), before messages are delivered. At reception time, the algorithm may also sort those received buffered messages according to their intended order, as proposed in other previous works [17, 16]. In our case, that order is set by the timestamp included in every message. Let us consider that approach in an adaptive revision of Algorithm 1.

Thinking Time	$[min_i\{\|O_i\|\}, max_i\{\|O_i\|\}]$	AO (eq. 1)	$(\overline{Throughput_i}; \sigma)$
0 ms	$[62,70\%, 82,27\%]$	59,87%	(682,43 mgs/s; 1,23)
1 ms	$[63,49\%, 82,64\%]$	62,60%	(680,91 mgs/s; 1,54)
2 ms	$[64,67\%, 83,10\%]$	62,66%	(678,41 mgs/s; 1,71)
5 ms	$[68,12\%, 84,91\%]$	66,07%	(674,41 mgs/s; 1,41)
10 ms	$[77,79\%, 91,48\%]$	75,68%	(594,96 mgs/s; 0,83)
15 ms	$[89,74\%, 96,01\%]$	87,94%	(481,09 mgs/s; 0,90)

Table 2 Minimum and maximum percentage of o-delivered messages using Algorithm 1. The Approximate Order measure is in the third column. Process thinking time varies from 0ms to 15ms. Average throughput and its variance are in the fourth column.

4 An adaptive implementation of AOR-Broadcast

In the practical environment considered in subsection 3.1, process clocks are synchronized by the NTP protocol. We have also used the HLC timestamp mechanism [13] in lieu of physical/NTP clocks. The practical usage of HLC requires a rough synchronization among clocks. Thus, we assume that there are not two events e and f in the system such that $e\, hb\, f$ and $e.pt > f.pt + \varepsilon$ where pt is the physical clock of each event at its node, and ε denotes the clock uncertainty. This uncertainty quantity is approximately twice the offset of NTP. The idea to improve the Approximate Order measure (eq. (1)) is to improve $\|O_i\|$ for each process. The main idea comes from the *(Uniform) Local-Time Δ-Timeliness* property indicated in [10]: (**LT**) *There is a known constant Δ such that no process p_i delivers a message m after local time $ts(m) + \Delta$ on p_i's clock (where $ts(m)$ denotes the local time at which m was broadcast according to the sender's local clock).* Hadzilacos and Toueg show that if this property holds in the system then it is possible to obtain the Total Order property for a R-Broadcast. Since this is not possible in our system, the simplest way to improve the order is to estimate Δ and to delay the o-deliver() event of ordered messages some δ time for augmenting the possibility that future messages will be o-delivered.

The code of Algorithm 2 is straightforward. Function **g1**() in line 19 updates Δ_{max} or Δ_{min} in comparison with the new Δ estimation $now - \texttt{mets}.ts.l$ where now is the time where $\langle \texttt{m}, \texttt{mets} \rangle$ is received and $\texttt{mets}.ts.l$ is the clock's value of the timestamp when `m` was broadcast. Messages to be o-delivered are stored in timestamp order in the $Rec[]$ array variable (line 23).

Message delivery via the o-deliver() event is performed in the Task indicated on lines 28-40 of the Algorithm 2. This Task is repeated periodically from the delay δ calculated by the function **g2**() in line 30. The Task calculates from the messages in $Rec[]$, which are previously ordered by its timestamp on line 23, those that can be delivered when Task is executed. In particular, a message in $Rec[]$ can be o-delivered when $(now - Rec[p].pt) \geq \delta)$ (line 33), i.e., it has rested more than δ ms in $Rec[]$ and all previous messages in $Rec[]$ have been o-delivered. As we can see in line 36, all messages that can be o-delivered are delivered together in a single o-deliver() event. Once delivered, variables $last\_ets$ and $Rec[]$ are updated respectively (lines 37-38).

The rule for calculating the delay imposed on the messages in $Rec[]$ is the function **g2**() in line 30 of Algorithm 2. This is a simple *Autoregressive Moving Average* that is calculated by the expression,

$$\delta(\tau + 1) := \theta \cdot (\Delta_{max} - \Delta_{min}) + (1 - \theta) \cdot \delta(\tau) \tag{2}$$

Algorithm 2 Adaptive AOR-Broadcast Algorithm. Code for process p_i

```
1: Variables
2: now                          ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                 ▷ timestmap, a pair (l, c). Updated via HLC procedures Alg. 3
4: ets := (ts, i)               ▷ extended timestamp: a pair (timestamp, origin)
5: last_ets := ((0, 0), 0)      ▷ extendend timestamp of the last o-delivered message
6: Rec := [ ]   ▷ an array of tuples ⟨(m, mets), pt⟩ where m is a message, mets is its extended
   timestamp and pt is the time when m was received
7: Δ_max := 0
8: Δ_min := 0                   ▷ variables for approximating Δ-Timeliness property
9: δ := 1       ▷ adaptive delay, initially 1 ms. Maximum time residence of a message in Rec
10:
11: operation AO-broadcast(m)
12:     update_ts()
13:     ets := (ts, i)
14:     broadcast(⟨m, ets⟩)
15: end operation
16:
17: when ⟨m, mets⟩ is delivered          ▷ delivered by the underlaying reliable broadcast
18:     update_ts-rcv(mets.ts)                                  ▷ update timestamp of p_i
19:     (Δ_max, Δ_min) := g1(now − mets.ts.l, (Δ_max, Δ_min))
20:     if mets ≤ last_ets
21:         u-deliver(m)
22:     else                                                    ▷ mets > last_ets
23:         Rec := insert(Rec, ⟨(m, mets), now⟩)        ▷ Rec is ordered by extended timestamps
24:     end if-else
25: end when
26:
27: Task::
28: repeat  each max(1, δ/2) time
29:     update_ts()
30:     δ := g2(Δ_max − Δ_min, δ)
31:     Let L := Rec.length                          ▷ Length of Rec array, indexed by 1..L
32:
33:     Let p_max := max{p : 1 ≤ p ≤ L ∧ (∀j : 1 ≤ j ≤ p : (now − Rec[p].pt) ≥ δ)}
34:                                          ▷ particular case: p_max = 0 when max(∅)
35:     if p_max > 0 then
36:         o-deliver(Rec[1..p_max])
37:         last_ets := Rec[p_max].mets
38:         Rec := delete(Rec, Rec[1..p_max])
39:     end if
40: end repeat
```

where θ is a parameter in the range $(0.0, 1.0)$.

In the next section, we present the experimental results of this adaptive algorithm.

4.1 Experimental evaluation

The experimental study of the Adaptive AOR-Broadcast Algorithm 2 is carried out under the same conditions that have been established in the subsection 3.1 for the AOR-Broadcast Algorithm 1. As we can see in Table 3 and Table 4, the improvement in the delivery of messages by o-deliver() events is very significant. In the extreme case, when thinking time of processes is 0 ms, the Approximate Order measurement goes from 59.87% to 91.62%. In the test carried out, of 10002 messages delivered, 9164 messages are delivered in the same order in all processes and 838 messages are delivered in different order. In the case of 5 ms (Table 4, we go from obtaining 66.07% to 99.01% of messages delivered in order with the adaptive algorithm. An improvement of 33.23% is obtained. For thinking times greater than 5 ms, the measurement of the Approximate Order exceeds 99%.Table 4 shows that throughput of each test is very similar using both algorithms.

Thinking Time 0 ms		
Process	Ordered O_i	$Throughput_i$
p1	$\frac{9726}{10002}$ (97.24%)	$\frac{10002}{12,596s}$ (794.06 mgs/s)
p2	$\frac{9758}{10002}$ (97.56%)	$\frac{10002}{12,583s}$ (794.88 mgs/s)
p3	$\frac{9803}{10001}$ (98.02%)	$\frac{10001}{12,535s}$ (797.85 mgs/s)
p4	$\frac{9745}{10002}$ (97.43%)	$\frac{10002}{12,582s}$ (794.95 mgs/s)
p5	$\frac{9838}{10001}$ (98.37%)	$\frac{10001}{12,592s}$ (794.23 mgs/s)
p6	$\frac{9691}{10005}$ (96.86%)	$\frac{10005}{12,625s}$ (792.48 mgs/s)
p7	$\frac{9534}{10004}$ (95.30%)	$\frac{10004}{12,572s}$ (795.74 mgs/s)
p8	$\frac{9702}{10004}$ (96.98%)	$\frac{10004}{12,650s}$ (790,83 mgs/s)
p9	$\frac{9347}{10004}$ (93.43%)	$\frac{10004}{12,518s}$ (799,17 mgs/s)

Table 3 Percentage of o-delivered messages at each process using the Adaptive AOR-Broadcast Algorithm 2.

Thinking Time	$[min_i\{\|O_i\|\}, max_i\{\|O_i\|\}]$	AO (eq. 1)	$(\overline{Throughput_i}; \sigma)$
0 ms	$[93,43\%, 98,37\%]$	91,62%	(794,91 mgs/s; 2,38)
1 ms	$[96,19\%, 98,80\%]$	94,26%	(758,60 mgs/s; 3,30)
2 ms	$[97,35\%, 99,33\%]$	95,69%	(751,07 mgs/s; 3,03)
5 ms	$[99,30\%, 99,71\%]$	99,01%	(665,35 mgs/s; 1,92)
10 ms	$[99,59\%, 99,78\%]$	99,30%	(535,00 mgs/s; 1,65)
15 ms	$[99,72\%, 99,89\%]$	99,64%	(435,03 mgs/s; 0,79)

Table 4 Results obtained using the Adaptive AOR-Broadcast Algorithm 2.

In figure 2, we can see how the delay (δ) in the delivery of messages is adapted in the test. In this test, the thinking time is 10 ms. Figure 2 shows the behavior of the process p3. The delay is bounded between 1 ms and 4.7 ms. This figure also shows that the number of ordered messages, which are delivered in each o-deliver() event, is bounded between 1 message and 6 messages.

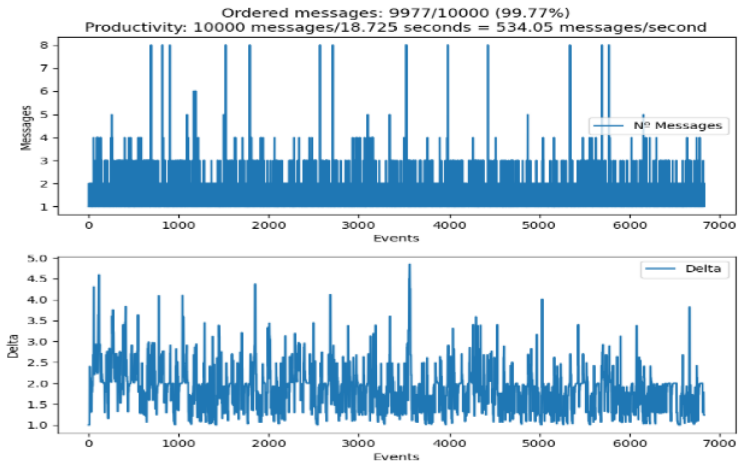

Figure 2 An example of how delta evolves using the Adaptive AOR-Broadcast Algorithm.

In this initial experimentation, obtained results confirm that the adaptive algorithm offers good productivity and that the number of messages delivered in an unordered way can be kept below 1% even when the processes are deployed in different zones (buildings) of a data center.

5 A naive impossibility result

In Theory of Distributed Algorithms, there are some simple impossibility results, e.g., *it is impossible to solve the leader election problem in a ring if processes are anonymous* [1]. Other impossibility results are related to the limits on the number of messages to solve a problem, e.g., *it is impossible to solve the leader election problem in a ring with n processes using a distributed algorithm with message complexity less than* $\Omega(nLog(n))$ [1]. In the considered distributed system model (subsection 2.1), it is impossible to solve the *consensus problem* by the well-known *FLP* impossibility result [9]. The equivalence of the Total Order Reliable Broadcast and consensus [10] makes the former problem also impossible. The specification of AOR-Broadcast is very weak in comparison with TO-R-Broadcast. In addition, we know that other reliable broadcast related problems, e.g., FIFO-R-Broadcast and CO-R-Broadcast, have algorithmic solutions that are built on top of a simple R-Broadcast protocol. In particular, a non-blocking solution exists for CO-R-Broadcast on top of FIFO-R-Broadcast [10][19]. Our question is: *Is it possible to build an non-blocking AOR-Broadcast algorithm on top of a R-Broadcast algorithm without the existence of a total order relation on the set of delivered messages?*. What the previous question indicates is the need for the broadcast messages (and the possible additional information attached to them) to be completely ordered by a pre-established total order. We attempt to explain this informal question in a more formal way.

Let us assume that there is a non-blocking algorithm that implements AOR-Broadcast on top of a R-Broadcast protocol. When a message m is AO-broadcast by a process i, at some point on the code of this operation, it has to execute broadcast$_i(\langle m, s_{bm}[i]\rangle)$, where $s_{bm}[i]$ is the state of process i at this point of the execution. When a process j o-delivers (or u-delivers) m is because deliver$_j(\langle m, s_{bm}[i]\rangle)$ has been received via the underlaying R-Broadcast. As the protocol is non-blocking, decision about this message $\langle m, s_{bm}[i]\rangle$ is taken at the state $s_{dm}[j]$

when deliver$_j(\langle m, s_{bm}[i]\rangle)$ happens. Let us observe that $s_{bm}[i]$ is the maximum information that can be attached to a message. Thus, a non-blocking decision is a deterministic function $dec(s_{dm}[j], \langle m, s_{bm}[i]\rangle) \in \{o, u\}$ (indexes dm and bm satisfy $bm < dm$). What we claim is that *for the possible set* $\{\langle m, s_{bm}[i]\rangle\}$ *of (informed) messages in every execution, there exists a total order relation* $\prec$ *such that, for each process* p, $<_{O_p} \subseteq \prec$ *holds.* We are currently working on offering a convincing proof of this claim.

6 Conclusions

Algorithms that provide a reliable broadcast service may be easily extended with some kind of (logical) timestamp in order to check whether their participating nodes deliver the intended messages in total order or not. Such totally ordered delivery is easily enhanced when senders introduce short pauses between consecutive broadcast actions.

The proposed Approximate (total) Order Reliable Broadcast (AOR-Broadcast), in its adaptive variant, combines those mechanisms with an adaptable short pause between the receiving and the delivery stages, able to highly increase the percentage of messages delivered in total order among all processes. This adaptive pause at the receiving side introduces two benefits. Firstly, when both pauses are short (i.e. shorter than 3 ms), their combined effect on throughput is positive. Secondly, the length of the receiving pause may be substracted from the basic sending pause with better effect on the percentage of messages globally delivered in total order. Thus, AOR-Broadcast has been able to guarantee total-order delivery for more than 99% of the intended messages, while a basic reliable broadcast algorithm in those same scenarios did not exceed 60% of total-order delivery.

References

1 Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998. ISBN 978-0077093525.

2 Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013. doi:10.1145/2447976.2447992.

3 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987. doi:10.1145/7351.7478.

4 Eric A. Brewer. Pushing the CAP: Strategies for consistency and availability. *Computer*, 45(2):23–29, 2012. doi:10.1109/MC.2012.37.

5 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

6 Jo-Mei Chang and Nicholas F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984. doi:10.1145/989.357400.

7 António Luís Pinto Ferreira de Sousa, José Pereira, Francisco Moura, and Rui Carlos Oliveira. Optimistic total order in wide area networks. In *21st Symposium on Reliable Distributed Systems (SRDS)*, pages 190–199, Osaka, Japan, 2002. IEEE Computer Society. doi:10.1109/RELDIS.2002.1180188.

8 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004. doi:10.1145/1041680.1041682.

9 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

10 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, New York, USA, 1994.

11 Pieter Hintjens. *ZeroMQ: Messaging for Many Applications.* O'Reilly Media, Inc., 2013.

12 Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing transactions over optimistic atomic broadcast protocols. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 424–431, Austin, TX, USA, 1999. IEEE Computer Society. doi:10.1109/ICDCS.1999.776544.

13 Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *18th International Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32, Cortina d'Ampezzo, Italy, 2014. Springer. doi:10.1007/978-3-319-14472-6_2.

14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

15 Carlo Marchetti, Roberto Baldoni, Sara Tucci Piergiovanni, and Antonino Virgillito. Fully distributed three-tier active software replication. *IEEE Trans. Parallel Distributed Syst.*, 17(7):633–645, 2006. doi:10.1109/TPDS.2006.89.

16 Emili Miedes and Francesc D. Muñoz-Escoí. Improving the benefits of multicast prioritization algorithms. *J. Supercomput.*, 68(3):1280–1301, 2014. doi:10.1007/s11227-014-1087-z.

17 Akihito Nakamura and Makoto Takizawa. Priority-based total and semi-total ordering broadcast protocols. In *12th International Conference on Distributed Computing Systems (ICDCS)*, pages 178–185, Yokohama, Japan, June 1992. IEEE Computer Society. doi:10.1109/ICDCS.1992.235041.

18 Fernando Pedone and André Schiper. Optimistic atomic broadcast. In Shay Kutten, editor, *12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science*, pages 318–332, Andros, Greece, 1998. Springer. doi:10.1007/BFb0056492.

19 Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. doi:10.2200/S00236ED1V01Y201004DCT002.

20 Xin Zhao and Philipp Haller. Replicated data types that unify eventual consistency and observable atomic consistency. *J. Log. Algebraic Methods Program.*, 114:100561, 2020. doi:10.1016/j.jlamp.2020.100561.

A Hybrid Logical Clocks

In the proposed distributed system model, there is no true time clock that processes can use. Each process can access a physical clock on the machine where it runs. From a practical point of view we consider that the machines are synchronized using the NTP (Network Time Protocol) protocol. Since perfect synchronization of clocks is not possible, we will assume that there are uncertainty intervals associated with the clocks. In this practical scenario, we can use the timestamp mechanism proposed by [13] called HLC (Hybrid Logical Clock). The advantages of this mechanism compared to others are (a) its good ability to tolerate common NTP divergences; (b) it is self-stabilized; and (c) it is resistant to possible corruptions of clock variables. To understand some of the properties offered by HLC we recall here the notion of *causal relationship between events*. An event e occurs before f, denoted $e\ hb\ f$, (i) if e and f are of the same process and e occurs before f; or (ii) e is a message sending event and f is its corresponding message receiving event. The complete relation hb is the transitive closure of (i) and (ii) for all events. Two events are concurrent, denoted $e||f$, if and only if, $\neg(e\,hb\,f) \wedge \neg(e\,hb\,f)$.

In the HLC algorithm, Algorithm 3, each event has a pair of (l, c) values associated with it. The value l corresponds to a physical clock, pt, and the value of c corresponds to an integer to capture causality when the l values of events are equal. Basically l is updated with the physical clock, and in case of a tie c is incremented. In what follows, we assume that for an event e its associated pair is $(l.e, c.e)$. The physical value of the

clock when event e happens is denoted, $pt.e$. On the other hand, the relation $<$ is stated as $(a, b) < (c, d) \Leftrightarrow (a < c) \vee ((a = c) \wedge (b < d))$.

AS1. *Synchronization Assumption of Physical Clocks*: There are no two events e and f in the distributed system such that $e\, hb\, f$ and $pt.e > pt.f + \varepsilon$. Where ε denotes the uncertainty of clock synchronization (approximately twice the offset value of the NTP).

Under the previous assumption, HLC timestamp mechanism provides the following properties [13]:

HLC1 *Directional causality.* For two events e and f:
$e\, hb\, f \Rightarrow (l.e, c.e) < (l.f, c.f)$.
HLC2 For any event f: $l.f \geq pt.f$.
HLC3 *$l.f$ denotes the maximum clock value of which the event f is aware.* For any event f: $l.f > pt.f \Rightarrow (\exists g : g\, hb\, f \wedge pt.g = l.f)$
HLC4 *l is bounded.* For any event f: $|l.f - pt.f| \leq \varepsilon$
HLC5 For any event f:

$$c.f = k \wedge k > 0 \Rightarrow$$
$$\exists g_1, g_2, ..., g_k : ((\forall j : 1 \leq j < k : g_i\ hb\ g_{i+1})$$
$$\wedge (\forall j : 1 \leq j \leq k : l.g_i = l.f) \wedge g_k\, hb\, f)$$

HLC6 *c is bounded.* For any event f: $c.f \leq \|\{g : g\, hb\, f \wedge l.g = l.f\}\|$
HLC7 *c is bounded.* For any event f: $c.f \leq N \times (\varepsilon + 1)$

Algorithm 3 HLC algorithm. Code for process p_i

```
1: Variables
2: pt                                  ▷ a read-only variable. It contains physical time via NTP protocol
3: ts := (0, 0)                        ▷ timestmap, a pair (l, c). Updated via HLC procedures
4:
5: procedure update_ts()                                                    ▷ send or local event
6:     Let l' := ts.l                                                      ▷ the previous logical time
7:     ts.l := max(l', pt)                                                 ▷ the new value of logical time ts.l
8:     if ts.l = l'
9:         ts.c := ts.c + 1
10:    else
11:        ts.c := 0
12:    end if-else
13: end procedure
14:
15: procedure update_ts-rcv((lm, cm))   ▷ receive event of message m and timestamp (lm, cm)
16:    Let l' := ts.l                                                      ▷ the previous logical time
17:    ts.l := max(l', lm, pt)                                             ▷ the new value of logical time ts.l
18:    if ts.l = l' = lm
19:        ts.c := max(ts.c, cm) + 1
20:    else if ts.l = l'
21:        ts.c := ts.c + 1
22:    else if ts.l = lm
23:        ts.c := cm + 1
24:    else
25:        ts.c := 0
26:    end if-else
27: end procedure
```