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.

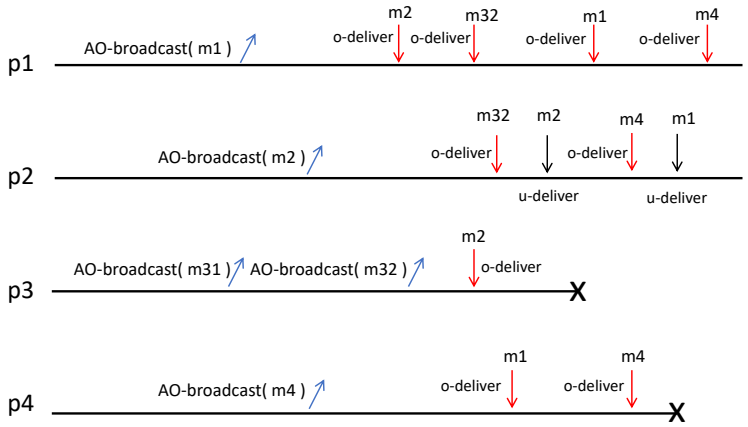

Figure 1 An example of the AOR-Broadcast delivery guarantees

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,

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

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: $\Delta_{max} := 0$
8: $\Delta_{min} := 0$ ▷ variables for approximating Δ-Timeliness property
9: $\delta := 1$ ▷ 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**($\langle$m, $ets\rangle$)
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: $(\Delta_{max}, \Delta_{min}) :=$ **g1**($now -$ mets.$ts.l$, $(\Delta_{max}, \Delta_{min})$)
20: **if** mets $\leq last\_ets$
21: **u-deliver**(m)
22: **else** ▷ mets $> last\_ets$
23: $Rec :=$ **insert**(Rec, ⟨(m, mets), $now\rangle$) ▷ Rec is ordered by extended timestamps
24: **end if-else**
25: **end when**
26:
27: Task::
28: **repeat** each max(1, $\delta/2$) time
29: **update_ts**()
30: $\delta :=$ **g2**($\Delta_{max} - \Delta_{min}$, δ)
31: Let $L := Rec.length$ ▷ Length of Rec array, indexed by $1..L$
32:
33: Let $p_{max} := \max\{p : 1 \leq p \leq L \wedge (\forall j : 1 \leq j \leq p : (now - Rec[p].pt) \geq \delta)\}$
34: ▷ particular case: $p_{max} = 0$ when $\max(\emptyset)$
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.

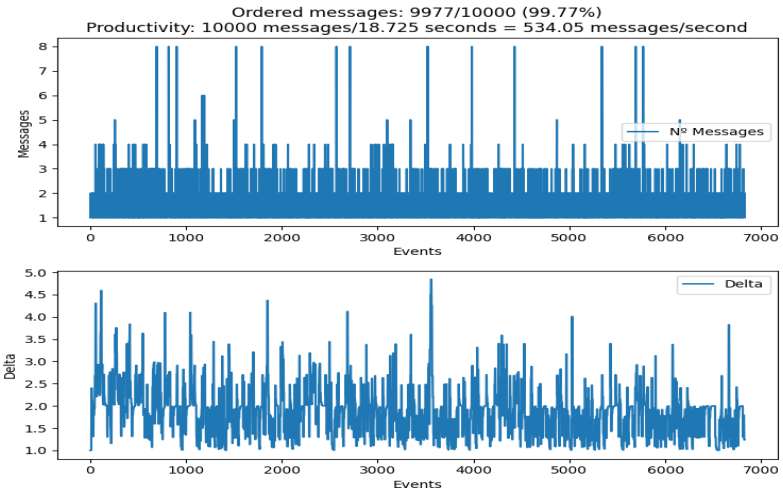

Figure 2 An example of how delta evolves using the Adaptive AOR-Broadcast Algorithm.

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
```