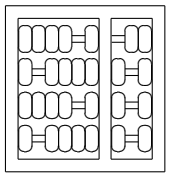

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Time Hybrid Total Order Broadcast: Exploiting the Inherent Synchrony of Asynchronous Broadcast-based Distributed Systems

Daniel Cason *Luiz E. Buzato*

Technical Report - IC-13-26 - Relatório Técnico

September - 2013 - Setembro

Time Hybrid Total Order Broadcast: Exploiting the Inherent Synchrony of Asynchronous Broadcast-based Distributed Systems

Daniel Cason* Luiz Eduardo Buzato*

Abstract

Informally, total order broadcast protocols allow processes to send messages with the guarantee that all processes eventually deliver the messages in the same order. In this paper, we investigate the efficiency and performance of a very simple synchronous total order broadcast protocol, that is built atop a broadcast-based asynchronous distributed system. The performance results allow us to conclude that our Time Hybrid Total Order Broadcast (THyTOB) represents an interesting trade-off between performance and simplicity for total order broadcasts; its simplicity also allows its use as a benchmark for total order broadcast atop Ethernet. Finally, the experimental results that support THyTOB make a strong case for the reconsideration of the common wisdom that total order broadcasts must be always designed for the asynchronous model because it is the best way to guarantee safety and performance.

1 Introduction

Informally, total order broadcast protocols allow processes to send messages with the guarantee that all processes eventually deliver the messages in the same order. The importance these protocols of is attested by their use as key components of several highly available systems [10, 11, 17, 37] and by the intense research associated with them [19]. Yet, despite the steady stream of research, to the best of our knowledge, no one has tried to assess the performance of a simple total order broadcast for broadcast-based networks such as Ethernet or Infiniband, the network technologies that are adopted by the vast majority of current commercial datacenter clusters and high-performance computers [1].

In this paper, we investigate the efficiency and performance of a very simple synchronous total order broadcast protocol, that is built atop a broadcast-based asynchronous distributed system. Thus, to discuss our protocol we have to explain the combined behaviour of two computing models: the timed-asynchronous and the synchronous models. We show that the exploration of the interaction between asynchronicity and synchronicity can lead to a time hybrid total order broadcast protocol presenting both good performance and high reliability. In this context, the main contributions of this work are:

*Institute of Computing, University of Campinas, 13081-970 Campinas, SP, Brasil. Research partially founded by FAPESP under grants 2010/14555-6 and 2011/23705-4, and CNPq grant 473340/2009-7.

- to show experimentally that an asynchronous broadcast-based cluster can indeed be a trustworthy implementation of the synchronous model of computation considered by our total order broadcast. Our experiments allow us to maintain that our assumptions regarding the inherent synchrony of broadcast networks are reasonable for clusters built atop of local area networks.

- to show that our Time Hybrid Total Order Broadcast (THyTOB) is on a par in terms of performance with other total order broadcast protocols, despite its simplicity. This result makes our algorithm a good alternative to more complex protocols for several classes of cluster applications. For example, THyTOB can be used to implement an MPI Broascast (MPI_BCAST) primitive equivalent to the one proposed by [28], but based on a simpler algorithm.

This work explores the fact that the processes and the network of *asynchronous* broadcast-based clusters have an inherent tendency to behave as *synchronous* systems during reasonably long periods of time. This tendency to synchronicity has allowed us design and implement the Time Hybrid Total Order Broadcast protocol (THyTOB) that is discussed in Section 3. Section 4 assesses the performance of THyTOB, and compares it with the performance of other interesting total order broadcasts. From the perspective of the total order broadcast, these two sections can be considered self-contained, in the sense that a good understanding of THyTOB can be obtained without a critical examination of the evidence of inherent synchrony. Despite this fact, an inquisitive reader may readily want to verify that the synchronous system is indeed implementable on top of the asynchronous system, forming a time hybrid system. So, the synchronous and asynchronous models of computation are formally defined in Section 2 and their feasibility is examined experimentally in Section 5. The conclusion is in Section 7 where we summarize our results and discuss briefly why we consider the principles that guide THyTOB's design a promising avenue giving the current trends in the reliability of broadcast networks and commercially available off-the-shelf computers.

2 Models of Computation

In this section we define the asynchronous and synchronous models of computation, including their failure assumptions.

2.1 Asynchronous Model of Computation

An asynchronous distributed system is composed of a fixed set of n processes connected by a broadcast network. Processes are asynchronous, in the sense that there is no bound in the time they take to perform processing steps. Processes can fail by crashing (crash-stop), but they never perform incorrect actions. Communication is accomplished by message passing, with the broadcast network offering a best-effort broadcast primitive. So, communication is one-to-all, asynchronous and unreliable: messages can be lost, duplicated, received out of order, or arbitrarly delayed, but we assume that messages cannot be corrupted.

Processes have access to local clocks, which display monotonically increasing values. The clocks are not synchronized, so the deviations between the processors' clocks can be arbitrarily large, but we assume they to progress most of the time within a narrow envelope of real time. Such clocks are useful to set timeouts, but we are particularly interested in using them to schedule the execution of periodic tasks: given a period Δ, the processes are then expected to be awakened to execute a task approximately every Δ units of real time. As the processes are asynchronous, there is no guarantees that this periodicity is actually met during the whole execution; but when considering sufficiently long periods of execution, the average interval of real time between successive invocations of the scheduled task is expected to be around the period Δ.

In addition to this clock assumption, we aggregate to the model an empirical hypothesis regarding its timing behavior. Given that the load applied to the system is controlled, the time it takes to complete a broadcast of a message with up to S bytes is *likely* to be bound by a constant δ_S. This delay includes the time the process takes to perform a processing phase, which results in the broadcast of a message, and the latency to deliver the message to all correct processes. Observe that this assumption does not impose bounds to the processing delays or network latencies: it only enables us to stipulate maximum delays for the actions to take place in the system, that are respected with high probability by their components.

Whenever the one-way timeout delay δ_S is violated we say that there was a *performance failure*. There is no bounds on the frequency with which performance failures can occur, or on the number of processes or channels that are affected by such failures. However, for the sake of progress, we assume that there is a time beyond which all broadcasts initiated by correct processes are completed within δ_S. What means that after a period of instability there is always a time interval of some given minimum length in which the system behaves *stably*, that is, a period in which the frequency of performance failures is upperbounded.

Thus, we started from an asynchronous crash-stop model with unreliable broadcast channels, to which we aggregated two assumptions based on the timed-asynchronous system model [18]: access to local clocks with bounded drift rates, and probabilistic maximum delays δ_S for the messages with respect to their size. The validity of these additional assumptions is restricted to periods of stability, in which processes and channels exhibit a (predominantly) synchronous behavior. As we consider that this is the behavior of the system *most of the time*, in next section we build, atop the model of computation described in this section, a powerful abstraction from the synchronous model: the abstraction of synchronous rounds [35].

2.2 Synchronous Model of Computation

In the synchronous model the processes have access to what we call a *logical global clock*. It is a clock, in the sense that it periodically ticks at discrete instants of time, and it is global, as it provides a common source of time to the processes. However, it is not a physical or a real-time clock, but a logical mechanism that generates ticks based on the processes' clocks, and on the exchange of synchronization messages.

The ticks of the logical global clock are used by the processes to organize the distributed computation into synchronous rounds. A process is then at round r from the instant it

receives tick r until immediately before the reception of tick $r+1$. The behavior of processes and channels during *failure-free* synchronous rounds is governed by the following rules:

(i) At the beginning of each round every process: processes the messages received in the previous round, broadcasts a message, and then waits to receive some messages broadcast in that round;

(ii) All messages processes broadcast in round r are received by all processes within the same round r. Thus, as in [35], we assume that transfer delays do not exceed the duration of a round, and that the processing delays are insignificant in relation to the communication latencies;

(iii) Every process broadcasts at most one message per round. Thus, when the tick $r+1$ is received, the process knows that all messages broadcast during round r have already been received and processed, and that all processes of the system have now progressed to round $r+1$.

This model is akin to the (partially) synchronous models defined by [22] and [35] but it can only be correctly implemented by a *physically* synchronous system, that enforces real-time guarantees. In this work, one of our goals is to have a *logical* implementation of this model that adheres as much as possible to the specification above, but relying on an asynchronous distributed system—as modeled in previous section. Unfortunately, the total isolation of failures that occurs in asynchronous systems it is not only costly but practically impossible, so that not all failures considered in the asynchronous model could have been masked in the synchronous model, in which we had to introduce the occurrence of performance failures.

Performance failures can affect processes and channels: processes may not participate in some rounds (send omission) and messages may not be delivered to some processes (receive omission). Send omissions result from failures in the logical global clock mechanism, that cause processes to miss some clock ticks. Receive omissions, in turn, reflect the possibility that some of the messages broadcast in a round are not received by their destinations, because they can either be lost or arrive after the end of that round. Note that despite the occurrence of performance failures, *all messages delivered to a process at the end of each round have necessarily been broadcast in that round.* It is based on the content of such "timely"messages, received in the round in which they have been sent, that our synchronous model is implemented.

Now that we have defined the two models of computation that support our time hybrid total order broadcast protocol, we proceed with its description and then with its performance evaluation.

3 Time Hybrid Total Order Broadcast

This section presents THyTOB, an uniform total order broadcast protocol designed for the synchronous model of computation described in Section 2.2. Formally, THyTOB is defined through the two primitives it exports: a *broadcast* primitive, invoked by a process to request

the sending of a message to all processes, and a *deliver* primitive, invoked by the processes to offer messages to the application. Based on these primitives, and considering that messages are uniquely identified, THyTOB ensures the following four properties:

- VALIDITY: If a correct process broadcasts a message m, then the process eventually delivers m.
- UNIFORM AGREEMENT: If any process delivers any message m, then every correct process eventually delivers m.
- UNIFORM INTEGRITY: For any message m, every process delivers m at most once, and only if m was previously broadcast by some process.
- UNIFORM TOTAL ORDER: If some process delivers message m' before message m, then any process delivers m only after it has delivered m'.

THyTOB is designed to be safe under asynchrony and in the presence of partial process failures, but progress is only guaranteed while the distributed system behaves synchronously and there are no process failures. Therefore, it is a protocol intended to be used during periods of "good" behavior of the system, when it is possible to build consistent global views of the computation total order broadcast in a straightforward way. We first describe THyTOB considering a (perfectly) synchronous computation, then we present the procedure to tolerate performance failures, finally in Section 3.4 we describe how the protocol copes with crash failures.

3.1 Basic Protocol

To illustrate how the THyTOB works we first consider rounds in which the system behaves synchronously, i.e. rounds in which performance failures does not occur. At the beginning of a round r, a process p sends a message to all processes, composed by an application message to be total ordered and its sequence number i. We assume that all processes are initially "synchronized" with p, so that they broadcast in round r messages with the same sequence number i used by p. Given that there is no failures, p receives the n messages broadcast in round r, one from each process, and all messages have the same sequence number. Therefore, at the end of round r process p knows all messages with sequence number i that could have been broadcast, since p received messages from all processes and the assignment of sequence numbers for messages is unique. We say then that r was a *successful* round for p, as no failures or desynchronized processes were detected by p in the round, and that p *succeeded* in receiving the messages i in round r.

Having succeeded in receiving the messages i in round r enables p to broadcast, in the round which follows r, another application message with sequence number $i + 1$. This is actually the progress condition of the THyTOB, which associates the broadcast of new messages to the knowledge of the previous ones:

Condition 1. *A process can only broadcast a message with sequencer number $i+1$ if it has succeeded in receiving all messages with sequence number i in a previous round.*

As a consequence of Condition 1, when in a round $r' > r$ process p succeeds in receiving the messages $i + 1$, it in particular learns that the remaining processes also succeeded in receiving the messages i in a previous round. The set of messages p received in round r then became stable, and can be safely delivered to the application at the end of round r'. In the absence of failures $r' = r + 1$, so that in the best case the messages are delivered in two rounds, latency which is minimal for uniform total order broadcast—by reduction to uniform consensus, which require at least two communication steps [30]. The THyTOB then need a second enabling condition, for the sake of *uniform delivery*, which is stated as follows:

Condition 2. *A process only delivers the messages with sequence number i after having been successful in a round in receiving all messages with sequence number $i + 1$.*

In order to turn these two conditions into a total order broadcast we make two further considerations. First: whenever Condition 1 allows a process to use a new sequence number, it broadcasts a new message. Even when the process does not have application messages to broadcast, it assigns a sequence number and broadcast a *null* message, that is not delivered to the application but is required to enable Condition 2. Second: whenever Condition 2 becomes valid for the first time for a given sequence number, the process delivers the set of messages that became stable. In order to ensure total order delivery, the process applies a predefined deterministic ordering function to the messages—which considers for example the ids of their senders and their sequence number—before delivering them to the application. As a result, the first time that a process succeeds in receiving the messages i, it delivers all messages $i - 1$ using a predefined order, and it broadcasts a new application message (or a *null* message) with sequence number $i + 1$.

The correctness of this algorithm in the absence of failures is straightforward, and essentially relies on the properties of the rounds. The rounds are communication-closed, so that all processes that succeed in a round receive the same set of messages, composed by all messages that could have been broadcast in that round. The processes apply the same ordering function to the messages received in each round, therefore they build the same sequences $i \in \{1, 2, 3, \ldots\}$ of messages at the end of every successful round. The trivial total order achieved through the concatenation of such sequences of messages is the foundation of the THyTOB, which postpones in a round the deliver of (already ordered) messages just to ensure uniformity.

3.2 Tolerating Performance Failures

Given the ordering mechanism and the behavior of THyTOB in the absence of failures, in this section we discuss how the protocol handles rounds in which the occurrence of performance failures prevent some or all processes to succeed. Remember that a process succeed in a round when: (i) it receives n messages, one from each process; and (ii) all messages received have the same sequence number. We discuss the violation of predicates (i) and (ii) separately, illustrating two scenarios when they can occur.

The first scenario consists in a round r in which no process succeed because no process received the n messages expected for the round. This scenario may occur, for example,

when a process p miss round r and then does not broadcast or receive any message. In this case, p knows that no process could succeed in round r, thus p broadcast in the next round the same message i it should have broadcast in the round it missed. The remaining processes, although unaware of what really happened, will assume that no process succeeded in round r, so that they will broadcast their messages i again in the following round. As a result, when in a round $r' > r$ the system behaves synchronously again the processes might succeed in receiving the messages i, since they will continue to broadcast their messages i until that happens.

The second scenario starts from a round r in which some processes succeed in receiving the messages i, while others do not. It is a round in which all processes participate and broadcast their messages i, but for several reasons—synchronization failures, instability in latencies, message loss, etc.—only some of them receive the n messages broadcast. As a consequence, in the round which follows r some processes will broadcast messages $i + 1$, while others will broadcast their messages i again—following the procedure described for the first scenario. This scenario then leads to an impasse: rounds in which, even in the absence of performance failures, the processes can not succeed because they are desynchronized.

The procedure to circumvent this impasse is to make the processes that are "ahead"—the ones that succeeded in the round r illustrated above—to *retrograde* to lower sequence numbers, in order to enable the processes "behind" to resynchronize with them. More specifically, a process that broadcasts a message i in a round but receives a message $j < i$ from some process, have to retrograde to the sequence number j. Processes that retrograde to j behave like processes that never succeeded in receiving the messages j: they continue to broadcast their messages j until they eventually succeeds in receiving the messages j. Once it happens, the processes that retrograded will broadcast their messages $j+1$, but they will not deliver the messages $j - 1$ again. Note that for this procedure to succeed, all processes that were "ahead" have to retrograde to j, and all processes have to succeed in receiving the messages j in the same round—otherwise they fall again in the second scenario. Such conditions are actually sufficient to circumvent the impasse, since the enabling conditions of THyTOB ensure that the processes that retrograde, can only retrograde to the sequence number immediately before the highest sequence number used by any process.

Lemma 1. *Given that some process has broadcast a message i in a round r, if any process broadcast a message $j < i$ in any round $r' \geq r$, then necessarily $j = i - 1$.*

Proof. Consider r^* as the first round in which a process p succeeded in receiving the messages $i - 1$. Since p succeeded in round r^*, all processes broadcast messages $i - 1$ in round r^*. Thus in the round which follows r^* the processes could only broadcast: (i) a message i, if the process succeeded in round r^*; or (ii) a message $i - 1$, if the process did not succeed in round r^*. Note that from round r^* there is no possibility of a process to broadcast messages $j < i - 1$. From Condition 1, if p has broadcast a message i in round r it already succeeded in receiving the messages $i - 1$ in a round previous to r. Therefore, as $r > r^*$, in particular in any round $r' > r > r^*$ no process can broadcast messages with sequence numbers $j < i - 1$. □

As an immediate corollary of Lemma 1, we have an important property of the protocol:

despite the occurrence of failures in any round the processes can, either broadcast messages with the same sequence number i, or broadcast messages with consecutive sequence numbers i and $i + 1$. The protocol can only achieve progress in the first case, and given that all processes succeed in the round. Otherwise, if only part of the processes succeed in the round, we have the second scenario, which can be only reverted when the processes "ahead" retrograde to the sequence number i—i.e. when the system returns to the first case. Consequently, the progress of THyTOB depends on rounds in which there is no failures, so that all processes succeed in receiving the messages i. Some of them could have retrograded to the sequence number i, but necessarily some process succeeds in receiving the messages i for the first time in that round, therefore it will deliver the sequence $i - 1$ of messages and broadcast its message $i + 1$ for the first time.

3.3 Failure-free Behavior

The pseudo-code of the THyTOB protocol in the absence of process failures is depicted in Algorithm 1. The main procedure of the protocol, executed by every process which participates in each round r, consists in processing the set M of messages received by the process during the round. The conditions for a process to succeed in the round are checked in Line 7: it received messages from all processes and all messages have the same sequence number, which is the currently in use by the process (stored in the variable `current`). The first time a process succeeds in receiving the messages i, it performs the following actions:

(i) it applies the predefined ordering function to the set M of messages received in the round in order to build the ith sequence of messages (Line 10);

(ii) it delivers the sequence $i - 1$ of messages, the last it built, which became stable (Line 9);

(iii) it broadcasts the next application message available or a *null* message (if there is no new messages to broadcast) with sequence number $i + 1$ (Line 12).

Otherwise, when the process succeeds in receiving the messages i in a round but it already built the ith sequence of messages in a previous round, it does not build or deliver any sequence. The process, which necessarily retrograded to the sequence number i in some previous round, assumes that all processes also succeeded in receiving the messages i in the round, and broadcasts its message $i + 1$ gain.

When the round is not successful for the process, it does not perform any of the actions listed above, and its default behavior is to broadcast again the message it had broadcast on that round (stored in the variable `current`). However, if some message received during the round had a sequencer number j lower than the currently in use by the process (Line 14), the process have to retrograde and to broadcast again its message j. As stated by Lemma 1, the process can only retrograde to the sequence number immediately before the highest sequence number it used, which is stored in the variable `last`—updated only when a new message is broadcast (Line 11). Thus, when the process retrogrades, the value of the variable `current` become equal to `last - 1`, what in particular prevents the process to deliver the messages $j - 1$ again (Line 8).

```
1: upon Init do
2:     sequence[0] ← ∅
3:     current ← last ← 1
4:     messages[last] ← loadNextMessage()
5:     broadcast(last, messages[last])

6: upon End of round r with message set M do
7:     if |M_r| = n and ∀m ∈ M : m.seq = current then
8:         if current = last then
9:             last ← last + 1
10:            deliver(sequence[current − 1])
11:            sequence[current] ← sortMessages(M)
12:            messages[last] ← loadNextMessage()
13:        current ← current + 1
14:    else if ∃m ∈ M : m.seq < current then
15:        current ← min{m.seq : m ∈ M}
16:    broadcast(current, messages[current])
```

Algorithm 1: Pseudo-code of the THyTOB protocol.

3.4 Tolerating Crash Failures

Once its progress depends on the ability to build global views of the system, THyTOB does not natively tolerate the occurrence of process failures. The same applies when some processes, or the system as a whole, behave asynchronously over a long period of time: once the processes do not succeed in the rounds, THyTOB becomes unable to achieve progress. In this section we present the procedure THyTOB uses to handle the presence of crashed processes or long periods of asynchrony. This recovery procedure relies on an underline failure detection mechanism and on any asynchronous implementation of Uniform Consensus.

The procedure consists in a task that runs parallel to the protocol main task (depicted in Algorithm 1). This task is responsible for detecting when the protocol becomes unable to achieve progress, due to both asynchrony or process failures, for a given period of time or for a given maximum number of rounds. Note that as the underline model of computation is asynchronous, crashed processes can not be deterministically distinguished from slow processes [24], thus this performance-based failure detection mechanism will always be unreliable. That is why the recovery procedure relies on an asynchronous consensus algorithm, which is safe under asynchrony and indulgent to the failure detection mechanism mistakes [26], but ensures liveness under minimal conditions of synchrony provided that up to $f < n/2$ processes fail by crashing [12, 21].

Just as total order broadcast, consensus is an agreement problem that is central to the implementation of fault-tolerant distributed systems. The consensus problem is stated in

terms of processes that *propose* values and eventually *decide* the same value, and an uniform consensus algorithm must ensure the following properties [12, 26]: (i) if a process decides on some value, then this value was proposed by some process; (ii) two processes do not decide different values; and (iii) every correct process eventually decides on some value. Total order broadcast and consensus are equivalent problems [12], and the former can be reduced to the resolution of several instances of consensus in which the input values are sequences of messages, so that the value decided in the ith instance of consensus is the ith sequence of messages the processes will deliver. This reduction, that is trivial in THyTOB, is the basis for the recovery procedure detailed below.

When a process detects that the protocol is not live, it abandons the THyTOB main task and initiates some consensus *instances* in order to decide whether the pending messages can be safely delivered to the application. Considering that the highest sequence number used by the process—the value of variable `last` in Algorithm 1—when it abandoned the THyTOB main task was i, the scenario is as follows:

(i) From Condition 1, the process knows all sequences of messages up to the sequence $i-1$;

(ii) From Condition 2, the process has already delivered all sequences of messages up to $i-2$;

(iii) From (i) and Conditions 1 and 2, all processes have delivered all sequences of messages up to $i-3$;

(iv) As the process never broadcast its message $i+1$, no process can have delivered sequence of messages i.

Mapping to consensus instances, the value decided in every instance $j \leq i-2$ is the sequence j built by the process during the ordinary execution of the protocol. The only pending sequence is the sequence $i-1$, that some processes may have delivered in THyTOB. It is also the next sequence of messages the process should deliver, so that it proposes the sequence $i-1$ (it has already built but is not yet stable) as value for consensus instance $i-1$ The value decided in this instance is the next sequence of messages the process delivers. Note that processes which did not delivered the sequence $i-2$ start the procedure with `last` $= i-1$, thus they will propose in consensus instance $i-2$ the same sequence the process decided in the instance. This consistency between sequences of messages delivered by THyTOB and values decided in the corresponding consensus instances supports the correctness of the recovery procedure.

The algorithm executed by a process when it suspects that THyTOB may not achieve progress anymore, or it detects that other process started some recovery consensus instance is the following:

1. The process interrupts the THyTOB main procedure (Algorithm 1), and joins any consensus instance that is initiated by other processes;

2. In consensus instance `last - 2` the process *decide* the value `sequence[last - 2]`;

3. In consensus instance `last - 1` the process *propose* the value `sequence[last - 1]`;
4. In consensus instances from `last` the process can *propose* any value received from other process, or the special value *stop*;
5. If in any consensus instance the process *decides* the special value *stop*, the procedure is completed.

Step by step, the recovery procedure works as follows. In Step 1 the process abandons the THyTOB protocol, forcing the remaining processes—which will not succeed anymore in the rounds—to eventually start this procedure. Step 2 only enables processes that have not yet delivered the sequence of messages $i-2$ (i is the value of variable `last`) to deliver this sequence without conducting a complete consensus instance. Step 3 is required to ensure Uniform Agreement: if any process has delivered the sequence $i-1$ then all processes that propose values for consensus instance $i-1$ are forced to propose the same sequence of messages it has delivered. Since only values proposed can be decided, if a process has delivered the sequence $i-1$ then all correct processes eventually deliver the same sequence of messages $i-1$. Finally, in the last Steps the process completes the procedure by proposing the especial value *stop* in instances $j \geq i$ for which it do not have sequences j to propose. Note that such sequences were not delivered by any process in THyTOB, so that the process can accept any value proposed for these instances. When the first *stop* is "delivered" the process learns that there are no more pending messages, and the procedure is concluded.

From this point, there are two alternatives for the total order broadcast solution to proceed. The first alternative is to get the processes, once they deliver the *stop* command, to start the execution of any other asynchronous and fault-tolerant order broadcast protocol. When the system eventually restores its original synchronism and all (possibly crashed) processes recover, a new execution of THyTOB can be started. The second alternative consists in selecting, once recovery procedure is concluded, a new set of correct processes to start a new execution of THyTOB. What can be achieved by relying in a group membership service [8], or using special consensus instances in order to reconfigure the system [34]. Note that to ensure safety, the size n' of the new set of processes must be chosen so that $n' > 2f$, therefore the system reconfiguration is actually intended to replace crashed or misbehaving processes that prevent THyTOB to progress.

4 Performance Evaluation

This section presents the experiments realized to assess the performance of the THyTOB protocol. The experiments evaluate scenarios in which there is no process failures, that are expected to be rare events in our target system, but performance failures occur relatively often, because of the system asynchrony which becomes particularly evident under high network loads. In particular, the results show that the performance of THyTOB is quite *predictable*, what reinforces our assumption that an asynchronous system, given that the load applied by the protocol is controlled, present a synchronous behavior most of the time.

We first present the experimental settings we used in the experiments and next study two performance metrics: throughput, which measures the number of broadcasts that the

processes can complete per time unit, and latency, which measures the time required to complete a single broadcast without contention. In Section 4.6 we then compare the performance of THyTOB with the performance of other interesting total order broadcasts, which were designed for the same computing environment considered by THyTOB.

4.1 Experimental Setup

All experiments reported in this work were carried out on a cluster of machines equipped with two quad core 2.40 GHz Intel-Xeon and 12 GB of main memory. Machines ran Gnu/Linux Debian 6.0 with Linux 2.6.32 SMP 64-bits kernel, and experiments were carried out using the Sun Java SE Runtime Environment 1.6. Machines were interconnected using a 3Com 4200G Gigabit Ethernet switch with 24 ports and 0.2ms of round-trip time. The socket's receive buffers had 128 KB, the default size for the operating system.

THyTOB was implemented in Java, in about 1000 lines of code. Processes communicate through UDP (for unreliable datagram transport) and the network-level multicast primitive provided by the IP protocol over Ethernet. The THyTOB implementation includes a protocol that implements the round-based model of computation described in Section 2.2. The description of this protocol, including its implementation and the main characteristics of the synchronous rounds it generates, has been left to Section 5.

The load for the experiments was generated by a fake application that requests, whenever a sequence of messages is delivered, the broadcast of a random message composed by S bytes, where S is a parameter for the experiment. Are also parameters for the experiments the reference duration Δ for the synchronous rounds, given in microseconds ($1\mu s = 10^{-6}s$), and the number of processes participating in the protocol. Experiments lasted about 10 minutes (300 thousand rounds), and were repeated at least five times.

4.2 Throughput

Figure 1 presents the throughput achieved by THyTOB in experiments with five processes and messages of 10 KB, with respect to the reference duration for the rounds. Each experiment was repeated five times, and the points and bars represent, respectively, the averages and standard deviations for the mean throughput measured in each execution. The dashed curve represents the optimal throughput for each experiment, i.e. the throughput that should be achieved in the absence of performance failures, when all processes succeed in every round. Note that the higher the reference duration Δ for the rounds, the closer to the optimal is the throughput achieved by the protocol, what means that the bigger was the proportion of rounds executed in which the protocol completed broadcasts. It is an expected behavior, since with increased duration for the rounds, small fluctuations in the processing delays or in the latencies for the messages are less likely to result in performance failures, and therefore to prevent the rounds to be successful.

In the rightmost point in Figure 1, with rounds of $1750\mu s$, THyTOB achieves 28.01 ± 0.02 MB/s of throughput, what corresponds to 98% of the optimal throughput for the experiment, which is 28.57 MB/s. We say that THyTOB had an efficiency of 98% in that experiment, what means that the protocol does not complete broadcasts in only 2% of the

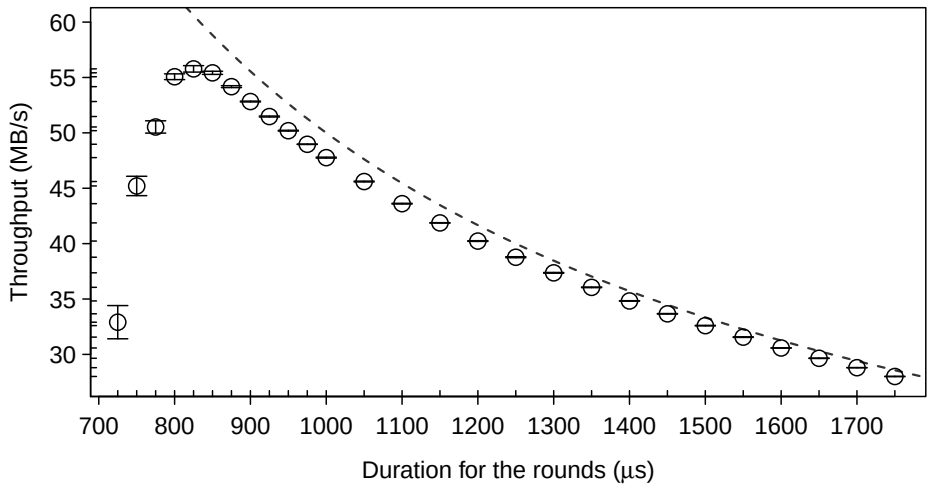

Figure 1: Throughput with respect to the reference duration Δ for the rounds in experiments with 5 processes and messages of 10 KB. The dashed curve is the optimal throughput for the experiments in function of Δ, given by $(5 * 10000/\Delta)$ MB/s.

rounds executed. When we progressively reduce the duration for the rounds there is an slight increase on the frequency of performance failures, so that the efficiency decreases slowly, reaching about 95% with $\Delta = 900\mu s$, and 92% with $\Delta = 825\mu s$. But as the optimal throughput is inversely proportional to the duration for the rounds, the throughputs increase with the reduction of Δ, reaching its maximum value 55.78 ± 0.29 MB/s at $\Delta = 825\mu s$. From this point, small reductions of Δ cause large drops in efficiency (88% with $\Delta = 800\mu s$, 78% with $\Delta = 775\mu s$), which are no longer offset by the increasing of the optimal throughput. As a result, in the leftmost point in Figure 1, with $\Delta = 750\mu s$, the measured throughput was 33.20 ± 1.42 MB/s, which corresponds to only 48% of efficiency. Observe that not only the average throughputs decreased but also the standard deviations increased, what reflects the instability of the system when the rounds become too short to support the load applied by the protocol.

4.3 Latency

Figure 2 presents the average latencies for delivering the messages in the same experiments described in previous section, with five processes and messages of 10 KB. The points in black and the bars represent the average and standard deviations for the mean latencies measured in five executions of the protocol. The dashed curve is the optimal latency for the experiments, which depends only on the reference duration Δ for the rounds. Similarly to the throughput, the measured latencies are closer to the optimal with the increasing of the duration for the rounds, but they degrade rapidly when the rounds become too short. In fact, the operation of the protocol causes throughput and latency to be closely related: as the processes can only broadcast new messages when they complete broadcasts, the more frequently they deliver messages, the lower is the latency achieved. Thus, as would be

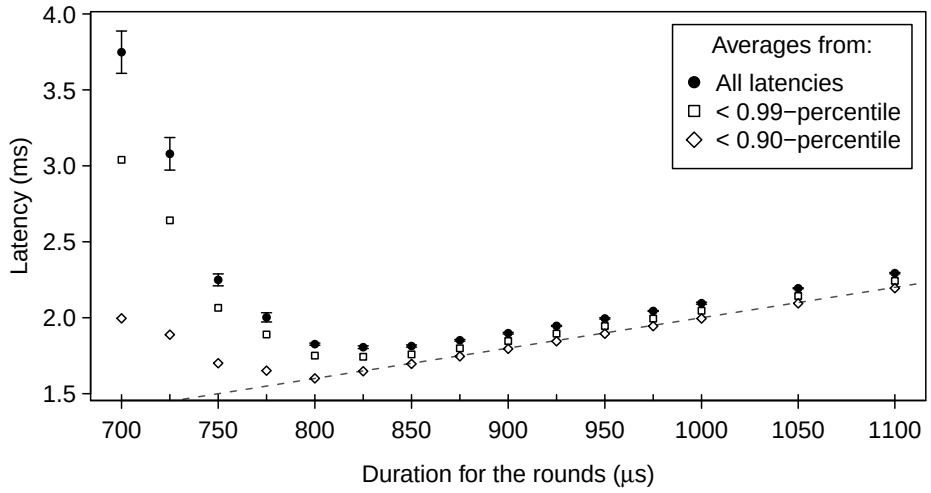

Figure 2: Latency with respect to the reference duration Δ for the rounds in experiments with 5 processes and messages of 10 KB. The dashed curve is the optimal latency for the experiments in function of Δ, given by $(2 * \Delta/1000)$ ms.

expected, the best latency for this configuration was achieved at $\Delta = 825\mu s$, which is the inflection point for both curves of latency and throughput. The latency for this point was 1.80 ± 0.01 ms, which is only 9.4% above the optimal value $2\Delta = 1.65$ ms.

The white points in Figure 2, in turn, represent the average latencies computed when the higher 1% (squares) and 10% (losangles) measured latencies are disregarded. Note that from $\Delta = 800\mu s$ the latter are almost equal—and the former are getting closer and closer—to the optimal latencies for the experiments. For instance, with $\Delta = 825\mu s$ the 90-percentile latency was 2.08Δ and the 99-percentile was 5.89Δ, what means that (at least) 90% of messages had an optimal latency of two communication rounds, and 90% of the remaining had latencies of at most 6 rounds. These results, in conjunction with the previous that messages are delivered in 92% of the rounds, allows to conclude that the executions of the protocol are mainly composed by long sequences of successful rounds, in which the processes deliver n messages per round with latency of two rounds. Such sequences are typically interrupted by few unsuccessful rounds, resulting in latencies of 3-5 rounds, and only occasionally by longer periods of instability, which are also controlled: from $\Delta = 800\mu s$ to $1750\mu s$ the 99.9-percentile latencies are on the range 12-14 ms.

4.4 Impact of Message Size

The same behavior of the protocol in the configuration presented in the last two sections, with five processes and messages of 10 KB, was observed in several configurations of execution. Specifically, for every size for the messages broadcast by the processes, there is an optimal reference duration Δ_{opt} for the rounds in which both the throughput achieved by the protocol is maximal and the average latency for the messages is minimal. The performance of the protocol in this sort of saturation point of the system in configurations

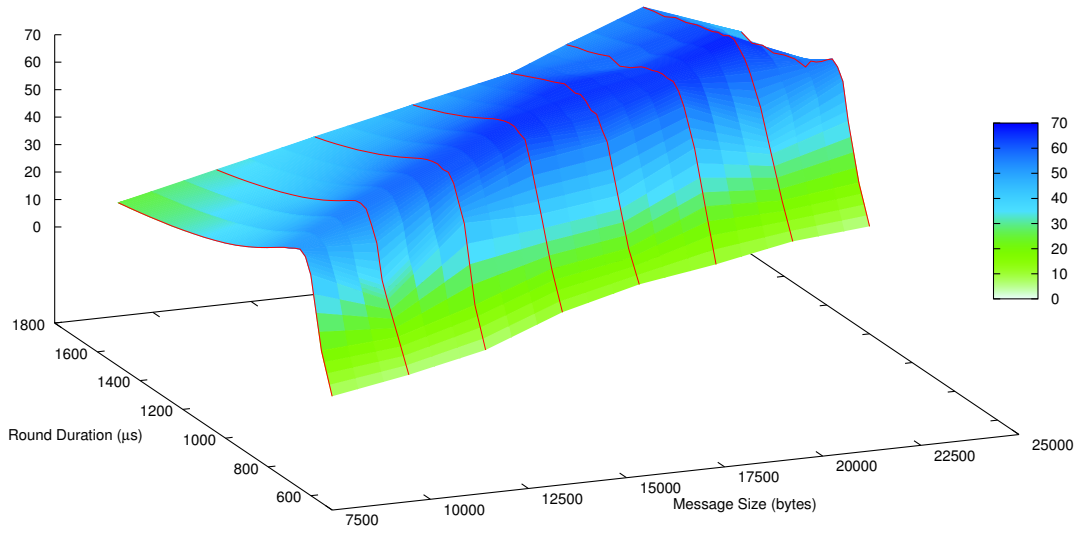

Figure 3: Throughput with respect to the message size and the duration for the rounds in experiments with 5 processes.

Message Size	7.5 KB	10 KB	12.5 KB	15 KB	17.5 KB	20 KB	22.5 KB	25 KB
Round Duration	$650\mu s$	$825\mu s$	$925\mu s$	$1025\mu s$	$1175\mu s$	$1250\mu s$	$1375\mu s$	$1375\mu s$
Protocol Efficiency	90.4%	92.0%	90.2%	89.4%	87.6%	81.0%	81.8%	61.5%
Mean Latency (ms)	1.45	1.80	2.06	2.30	2.69	3.10	3.39	4.53
Throughput (MB/s)	52.2	55.8	60.9	65.4	65.2	64.8	67.0	55.9

Table 1: Best performance of THyTOB with 5 processes and several message sizes.

with five processes and several message sizes is detailed in Table 1, and an overview of the throughput achieved in such configurations with respect to the duration of the rounds is presented in Figure 3.

The first element observed in Figure 3 is that the shape of the curves for each configuration (plotted in red) are similar to the shape of Figure 1. The throughput increases with the reduction of Δ, accompanied by a slight reduction of efficiency, until Δ_{opt} is reached; around Δ_{opt} the system is saturated and the protocol achieves their best throughputs (regions in darker blue); then small reductions of Δ cause large drops in the efficiency, and the throughput decreases to almost zero (green regions). What changes with the increase of the message size—besides the predictable increase of Δ_{opt}, and the consequent increase in the mean latencies—is that the peak of maximum throughput is less prominent in the curves, and there is a higher range of Δ in which the throughputs are very close to the maximum. This behavior is explained by the increasing load applied to the system when the messages are larger, which causes a decrease in the efficiency, and therefore limits the throughput achieved by the protocol, as can be seen in Table 1.

A second element observed in Figure 3 is the curve of throughputs with respect to the message size, which has an (roughly) increasing pattern until it reaches 67.0 ± 0.7 MB/s,

Number of Processes	$n = 5$	$n = 10$	$n = 15$
Message Size	15 KB	7.5 KB	5.0 KB
Round Duration	$1025\mu s$	$1150\mu s$	$1300\mu s$
Protocol Efficiency	89.4%	86.7%	80.5%
Mean Latency (ms)	2.30	2.65	3.23
Throughput (MB/s)	65.4	56.6	46.2

Table 2: Performance of THyTOB with respect to the number of processes, when 75 KB of messages were broadcast per round.

with messages of 22.5 KB. From this point efficiency drops rapidly with the increasing of the message size, indicating a saturation of the network: from 81.8% with 22.5 KB to 61.5% with 25 KB, and to less than 20% with 27.5 KB. With smaller messages, in turn, the reduction of the load reflects in the increase of efficiency, that reaches 92% with messages of 10 KB, which is not offset by the decreasing of the optimal throughput. This limitation is more evident with messages of 7.5 KB, in which the optimal reference duration for the rounds was only $650\mu s$ and the throughput was 52.2 ± 0.3 MB/s. For best throughput would be necessary to further reduce the duration for the rounds, but for shorter rounds the system can hardly comply the deadlines imposed, so that the performance in this configuration is clearly limited by its reduced optimal throughput.

4.5 Impact of the Number of Processes

In all experiments presented so far the system was composed by five processes, what means that the protocol tolerated the crashing of two processes. In order to assess the scalability of THyTOB we selected the configuration with 5 processes and messages of 15 KB, which has a good trade-off between throughput and latency, and distributed the load of 75 KB of messages that are broadcast per round among a larger number of processes. The performance with 5, 10 and 15 processes is summarized in Table 2.

With the increasing in the number of processes which participates in the protocol there is a smooth decreasing in the throughput achieved, and a proportional increasing in the mean latencies: of about 13% when n is doubled and 30% when n is tripled. This loss of performance results from the combination of two factors, which were also observed with the increasing of the message size: the increase of Δ_{opt} for the configuration and the reduction of the efficiency achieved by the protocol. The former can be justified by the higher network latencies and processing delays required to receive and to process more messages in every round. The latter can be seen as a consequence of the progress condition of the protocol: with the increasing number of processes the probability of a performance failure to affect any process or message during a round tends to increase, so that the slight loss of efficiency observed was somehow expected.

Protocol	*Throughput*	*Latency*	*Message Size*	*Synchrony*	*Topology*	*Channels*	*Failures*
LCR	950 Mb/s	4.6 ms	32 KB	Strong	Ring	TCP	Membership
Ring-Paxos	900 Mb/s	4.2 ms	8 KB	Weak	Wheel	UDP	Consensus
THyTOB	**540 Mb/s**	**3.4 ms**	**22.5 KB**	**Weak**	**Star**	**UDP**	**Consensus**
THyTOB	**525 Mb/s**	**2.3 ms**	**15 KB**	**Weak**	**Star**	**UDP**	**Consensus**
THyTOB	**450 Mb/s**	**1.8 ms**	**10 KB**	**Weak**	**Star**	**UDP**	**Consensus**
Spread	180 Mb/s	5.7ms	16 KB	Weak	Wheel	UDP	Membership
Treplica	105 Mb/s	4.3ms	10 KB	Weak	Star	UDP	Consensus
Paxos4SB	65 Mb/s	4.6ms	4 KB	Weak	Star	UDP	Consensus

Table 3: Comparison of uniform total order broadcast protocols.

4.6 THyTOB versus other protocols

Table 3 compares the performance of THyTOB to that of other five total order broadcast protocols: LCR [27], Ring-Paxos [36], Spread [3], Treplica [38], and the protocol presented in [4] that hereafter we refer as Paxos4SB. The results were obtained in experimental settings very similar to the one used in the experiments with THyTOB. The performance data for Ring-Paxos, LCR and Spread were taken from the paper that describes the former [36], and for Paxos4B we considered the experiment with best throughput from the presented in [4]. All results in Table 3 refers to experiments with five processes, excepting Spread and Paxos4SB, from which the best results available were with three and twelve processes.

Interestingly, the protocols with best throughputs arrange the processes in a logical ring. In LCR every process maintains a TCP connection with its successor in the ring, and the messages that circulate this ring are totally ordered using logical clocks [32]. LCR achieves an almost-optimal throughput, corresponding to 95% of the network capacity, at cost of latencies that increases linearly with the number n of processes in the ring. To tolerate process failures LCR relies in a group membership service, and requires perfect failure detection, an abstraction which implies strong synchrony assumptions. Ring-Paxos, in turn, disposes $f + 1$ processes in a logical directed ring, where $f < n/2$ is the number os process failures tolerated. The first process of the ring is the only that broadcast messages (using IP-multicast), and the acknowledgments circulate through the ring via unicast. As a result of this strategy, the throughput of Ring-Paxos is slightly lower than LCR, but its latency is also smaller (better), and almost constant with the number of processes. Ring-Paxos tolerates message loss and process failures using instances of Paxos [33]—the consensus protocol of which Ring-Paxos is an optimization—, that are also responsible for reforming the ring.

In the lower portion of Table 3 we have Spread, which is one of the most-used group communication systems, and two consensus-based total order broadcasts implementing Paxos: Paxos4SB and Treplica—the latter developed by our research group. Spread implementation's of total order broadcast is based on Totem [5], a ring-based protocol that allows a single process to broadcast messages at time. This privilege circulates the ring in the form of a token, and the process holding the token broadcasts its messages, that are only

delivered after two complete revolutions of the token. To tolerate process and network failures, Spread implements a group membership service, responsible for reconstructing the ring and regenerating the token. Thus, Spread adopts the same logical topology of Ring-Paxos, in which a single process can IP-multicast messages at time, but suffers from the same $2n$ latency factor of LCR. The consensus-based protocols, in turn, adopt the same logical topology of THyTOB: n processes connected to a central node, which is the Ethernet switch, that communicate through IP-multicast. The relative poor performance achieved by these protocols can be explained by the inherent complexity of consensus [30], and by the network instability observed when several processes broadcast messages concurrently and uncoordinately [36].

Although presenting throughputs inferior to the observed for LCR and Ring-Paxos, on the order of 45% to 55% of the network capacity, THyTOB present latencies significantly lower than both protocols. What can be explained when we compare the number of communication steps required to complete a broadcast in each protocol: THyTOB is optimal, requiring only two communication steps in best case, while Ring-Paxos requires at least $f+3$ steps, and LCR requires two complete revolutions in the ring ($2n$ steps). Moreover, despite the adoption of a star topology, the way in which THyTOB *coordinates* the processes and *conditions* the load applied to the network, prevents the considerable loss of performance observed in asynchronous total order broadcasts in which processes communicate through the IP-multicast primitive.

5 Time Hybrid System: Experimental Validation

The previous sections have shown that THyTOB is both simple and produces competitive throughput while providing low latency. What remains to be done is to verify experimentally that: (i) a broadcast-based cluster is a trustworthy implementation of the asynchronous model, and (ii) the synchronous model can be implemented upon the asynchronous model. Instead of designing separate sets of experiments to test each of the premisses specifically, we have designed a single set of experiments that when viewed as whole provide enough evidence that both premises are valid for an Ethernet-based commodity cluster.

The models of computation referred above were defined in Section 2. The synchronous model organizes the distributed computation in rounds using a logical global clock (Section 2.2). In next section we describe an implementation for this abstraction through a protocol that periodically generates and diffuses the *ticks* that determine the succession of rounds: the rounds protocol. Next, we describe a set of experiments that both assess the rounds protocol, and verifies the assumptions of the asynchronous model (Section 2.1).

5.1 The Rounds Protocol

We consider that a process is elected to execute the rounds protocol: the *synchronizer*. The synchronizer implements a timer that triggers a tick generation event every Δ time units. The period Δ, the reference duration for the rounds, is a function of the number of processes and the maximum size of the messages exchanged by the processes, excluding messages generated by the rounds protocol. A round identifier, made up by an epoch

number juxtaposed with a sequencer number, that sequence number, is used to uniquely identify the synchronizer and the ticks it broadcasts. Using the round identifier processes are able to detect missed, duplicated or out of order clock ticks that are disregarded. The processes only accept as valid the round identifiers generated by the process they consider the correct synchronizer, and a process acting as a synchronizer is demoted from its role as soon as it receives a tick with an epoch number higher than its own epoch number. Thus, the round identifier is also used to implement a mechanism that guarantees that the system can eventually converge to a state where a single process acts as the synchronizer.

Whenever the tick generation event is triggered, the synchronizer broadcasts a new tick message using the IP-multicast primitive. Tick messages are composed by the epoch identifier e, a sequence number i, the period Δ, and some debugging information used to build the distributions presented in the next sections, totaling about 100 bytes. Processes listen to the address to which the ticks are broadcast, and whenever a tick is received they check whether it can be accepted. A process that executes round i of epoch e accepts a tick (e', j) only if $e' \geq e$ and $j > i$: if $e' = e$ the process starts round j; otherwise the process joins epoch e' from round j. When round j is started the messages received during the last round i are passed to the client protocol (e.g. to THyTOB) to processing, and the client returns the next message to be broadcast. This message is then labeled with the round identifier (e, j) and the process id—the headers added to such ordinary messages had about 240 bytes– and is broadcast through the IP-multicast primitive.

Similarly to the ticks, whenever a message is received the processes check whether it can be accepted. Given a message labeled with round (e', j) received by a process during round (e, i) we have three scenarios. If $e' = e$ and $j = i$, and if no message from the same process was received during the round, the message is *timely*, and it will be delivered to the protocol at the end of the round. But if $e' \neq e$ or $j < i$ this message is either from another epoch or from some previous round, then it is just dropped. This is actually a major characteristic of round-based computations: *late* messages are always dropped, so that they are equivalent to lost messages [14, 22]. Finally, if $e' = e$ and $j > i$ the process received an *early* message, that is buffered to be checked again in future. When round j eventually starts the process checks the buffer for messages j, but if round j is missed—because tick j was not received—all messages j in the buffer are dropped.

Given the description of the rounds protocol, in the following sections we assess its efficiency, starting from the accuracy with which the synchronizer periodically generates and broadcasts the ticks.

5.2 Accuracy of the Synchronizer

In the first set of experiments the system was composed of five processes and the reference duration for the rounds was $\Delta = 500\mu s$. This is a very tight duration for the rounds—as shown below, the latencies for delivering messages with several sizes are often larger than $500\mu s$— but with this value of Δ we might explore the behavior of the synchronization protocol under several scenarios of network load. In the first scenario the processes broadcast messages with empty payloads, resulting in a low load, of about 20 Mb/s. In the second scenario the message size was computed in order to induce a load of 500 Mb/s, corresponding

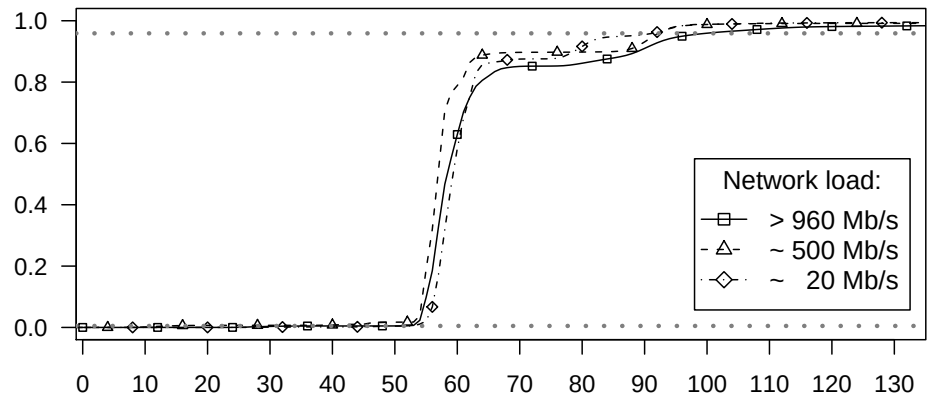

Figure 4: CDF for the processing delays, measured in μs, in the broadcast of 1.5 million ticks in three scenarios of load.

to half of the network capacity. And in the third scenario the message size was computed in order to induce 1 Gb/s of load with the messages broadcast at the beginning of the rounds, what means a completely congested network. As a result, the network loads measured on the last scenario fluctuated slightly during the experiments, but they remained above 960 Mb/s, which is used as reference value.

The first aspect assessed in the experiments was the accuracy of the synchronizer in scheduling the broadcast of ticks. The synchronizer schedules the generation of an infinite sequence of ticks to instants $t_1, t_2, t_3, \ldots$ of its local clock—which is the processor's clock, kept by the operating system. Thus given that a process becomes the synchronizer at the instant t_0 according its clock, the generation of the ith tick is scheduled to the instant $t_i = t_0 + i\Delta$, but the tick will actually be broadcast at some instant $ts_i \geq t_i$, also according the synchronizer's clock. We call the difference difference $ts_i - t_i$ the *processing delay* for tick i, and it reflects the accuracy with which the system (namely, the processor, the operating system and the Java virtual machine) wakes up the synchronizer for generating ticks. The less stable the processing delays, less close to Δ the intervals between the broadcast of ticks, thus less accurate the rounds protocol.

Figure 4 presents the Cumulative Distribution Function (CDF) for the processing delays measured in five executions with five different synchronizers, which broadcast 300 thousands ticks in three scenarios of network load. For expected loads of 20, 500 and 1000 Mb/s, respectively, 98.4%, 97.1% and 95.5% of the processing delays were between 50 and 100μs, and more than 80% of them were concentrated in the range 50 to 65μs. What means that in most cases the interval between the broadcast of ticks was very close to the expected—with the higher load 95.6% of the intervals were in $\Delta \pm 50\mu s$— and the synchronizer proved able to meet most schedules with high accuracy, even with the tight period of 500μs and rate of 2000 ticks per second. On the other hand, respectively, 0.47%, 0.70% and 1.34% of the processing delays measured were above 500μs, what means that these ticks were broadcast after the time scheduled to the generation of the next tick, resulting in very short intervals between ticks, lower than 100μs. This undesirable behavior was observed, albeit at lower

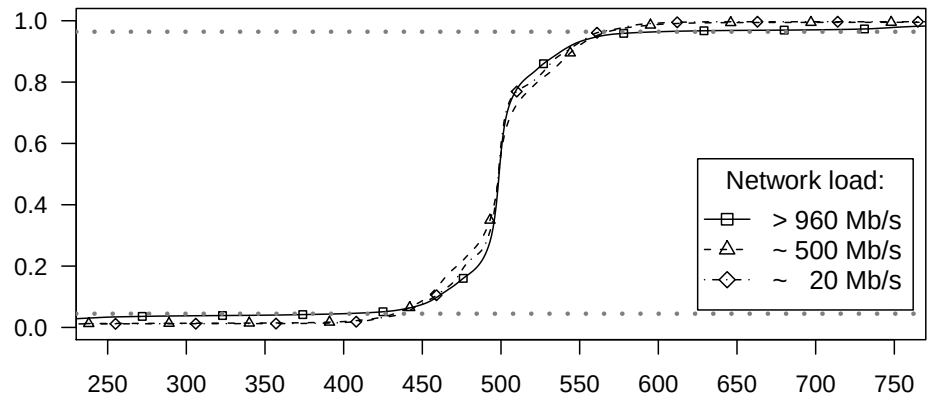

Figure 5: CDF for the duration of 1.5 million rounds, measured in μs by five processes, with reference value $\Delta = 500\mu s$.

percentages, for longer periods Δ and in different scenarios of load, what indicates that the processing delays, although reduced and stable most of the time, are in fact unbounded.

5.3 Duration of the Rounds

The second aspect assessed in the experiments described in last section was the duration of the rounds induced by the synchronizer. The duration of a round i according some process p is given by the difference on the time at which the two ticks which delimit the round—the tick i which starts it, and the tick $j > i$ which ends it—are delivered to p. Ideally, the durations of the rounds should be around the parameter Δ of the experiment, but in practice they are determined by the following factors: (i) the intervals between the broadcast of successive ticks; (ii) the fluctuations on the latencies of the tick messages; and (iii) ticks missed or received out of order—which cause rounds i to be terminated by ticks $j > i + 1$. The first factor was verified in the last section, when we found that at least 98% of the ticks were broadcast with intervals in the range $\Delta \pm 100\mu s$, then in this section we assess the second and third factors.

Figure 5 presents the CDF for the duration of the 1.5 million of rounds induced in the experiments, measured by the five process which participated of them. The two dotted lines delimit the portion of the rounds with durations in the range 400 to $600\mu s$ (i.e. $\Delta \pm 100\mu s$), that were 91.9% in the higher, 96.9% in the intermediate, and 97.6% in the lower loads. These percentages are 6.7%, 2.1% and 1.6% below the portions of ticks which were broadcast with intervals in $\Delta \pm 100\mu s$, what allows to quantify the impact of fluctuations on the latencies of the ticks in the duration of the rounds, with respect to the load applied to the system. Ticks lost or received out of order, in turn, were not observed in the experiments with 20 Mb/s of load, and for reference loads of 500 Mb/s and 1 Gb/s they were 0.17% and 0.82%. The rounds induced by these ticks were not executed by the processes, and for building the distribution their durations were computed as infinite. At the other extreme, the rounds with durations up to $100\mu s$—very short and probably useless for the processes—

were respectively 1.03%, 1.08% and 1.63%, percentages that are similar to the portions of ticks broadcast by the synchronizer with intervals up to $100\mu s$ between them.

The results indicate a high precision in the synchronization achieved by this simple pulsing mechanism with five processes, especially for the intermediate load, in which about 97% of the rounds had durations around $\Delta = 500\mu s$. We then repeated the experiment with reference load of 500 Mb/s doubling and tripling the number of processes, and the results were similar: 96.1% of durations in $\Delta \pm 100\mu s$ for $n = 10$, and 93.8% for $n = 15$; missed rounds were 1.60% and 3.30%; and very short rounds were 1.52% and 2.05%. Observe that, similarly to the load applied to the network, the number of processes impacts the synchronization, resulting in an increase of tick loss, and in larger fluctuations in the latencies of the ticks. The increase of message loss when more processes broadcast messages simultaneously, especially under higher loads, was observed in several scenarios of execution, and was also reported by other authors [36]. But for intermediate loads, the impact on precision when we tripled the number of processes was only 3%.

The experiments presented so far allowed us to validate the hypothesis that is possible to synchronize the processes through the periodic broadcast of ticks, and thus to organize computations as a sequence of synchronous rounds with a predefined duration Δ. In next section we study the behavior of the messages broadcast by the processes at the beginning of the rounds, regarding the distribution of their latencies. We expect to observe the behavior modeled in Section 2.1, that is, to find probabilistic upper bounds δ_S for the latencies, with respect to the message size S, that are respected by the system most of the time.

5.4 Latency Bound

The experiments conducted in this section consisted of five executions in which different syncronizers generated distributed computations with a duration of 300.000 rounds each. In every round the processes broadcast a message carrying a random payload with size S, that is a parameter of the experiment. The synchronizer then computes the Round Trip Time (RTT) of the messages, consisting on the difference between the time when the messages were delivered and the timestamp of the ticks which triggered their broadcast. The reference duration for the rounds $\Delta = 1000\mu s$ was chosen in order to minimize the tick loss, that was lower than 0.15%, 0.25% and 0.45% in executions with five, ten and fifteen processes.

Figure 6 presents the CDFs for the RTT of messages broadcast by five processes with respect to the size S of their payloads. In the first scenario, the payloads were empty, so the messages had essentially the same size of the tick messages, while in the remaining scenarios the payloads had 5, 10 and 15 KB, resulting in loads of about 200, 400 and 600 Mb/s. In the left-most portion of the graphs, representing about 20% of the measurements, are observed the RTTs for messages broadcast by the process which hosts the synchronizer. These messages are delivered locally, through the *loopback* interface, so that their RTTs did not include network latencies. Next, there is a "silence" interval, in which almost no message is delivered, followed by the interval of RTTs in which most messages were delivered. For payloads of 5, 10 and 15 KB the RTTs were mainly concentrated in the intervals 290 to $689\mu s$, 395 to $950\mu s$, and 457 to $1163\mu s$, and the upper bounds of such intervals corresponds

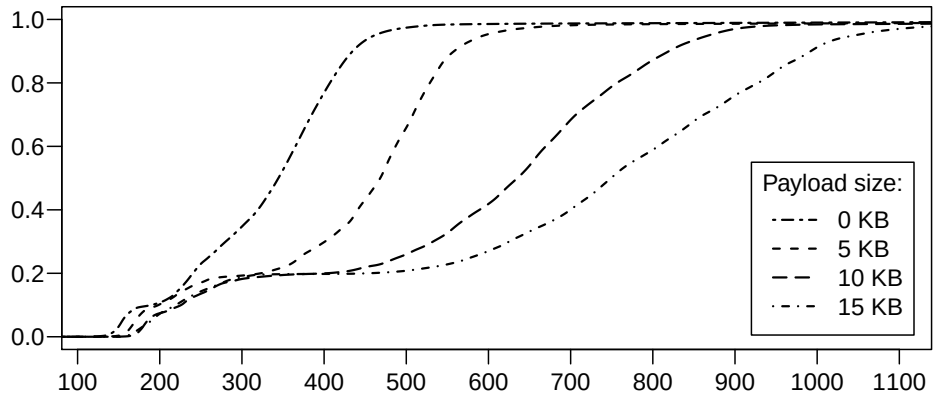

Figure 6: CDF for the RTT, given in μs, of messages broadcast by five processes, in rounds with reference duration $\Delta = 1000\mu s$.

approximately to the 98%-percentile of the distributions. Finally, in the right-most portion of the graphs are observed the 2% higher RTTs, sparsely distributed up to $4000\mu s$ (densities of less than 0.01%), which include the missed messages, with RTTs computed as infinite.

The results of the experiments are summarized in Table 4 which presents the portion of messages that have been classified by the synchronizer as timed, late and lost. Timed messages are the ones which were received before the end of the round in which they have been broadcast, therefore the only messages that were delivered to the processes. Late messages, in turn, were received but discarded by the processes, and the lost messages were not even received. The portions of timed and late messages are closed related to the percentages of messages with RTT up to the reference duration for the rounds, with some fluctuation due to inaccuracies in the synchronization. For instance, for payloads of 15 KB about 5.8% of the RTTs were in the range 1000 to $1100\mu s$, and most of such messages were classified as timed. The message loss, in turn, is more related to the load applied to the network than to the size of the messages broadcast.

Payload size	0 KB	5 KB	10 KB	15 KB
RTT $\leq 1000\mu s$	99.00%	98.74%	98.42%	91.24%
Timed messages	98.95%	98.73%	98.52%	96.53%
Late messages	0.97%	1.04%	1.18%	2.99%
Lost messages	0.08%	0.23%	0.30%	0.47%

Table 4: Efficiency of the rounds with respect to the payload's size, with 5 processes and duration of the rounds $\Delta = 1000\mu s$.

In a second set of experiments we analyze the impact of the number of processes in the latencies of the messages. We selected the experiment with five processes and 10 KB of payload, and distributed the load of 410 Mb/s applied in that scenario among ten and fifteen processes. While with five processes 98.1% of the latencies were mainly concentrated

up to 950μs (densities above 0.01%), with ten processes 97.6% of them were up to 982μs, and with fifteen processes 97.1% of them were up to 1084μs. Thus, there is an increase in the latencies when more processes broadcast messages in the rounds, even if the number of bytes sent is the same. In addition, there is an increase in the rate of missed rounds and of missed messages, despite the same load applied to the network. As a result, when we doubled and tripled the number of processes the efficiency of the rounds dropped slightly (by about 0.5% and 1.5%), as summarized in Table 5.

Number of Processes	$n = 5$	$n = 10$	$n = 15$
RTT $\leq 1000\mu s$	98.42%	97.78%	95.23%
Timed messages	98.52%	97.98%	97.07%
Late messages	1.18%	1.39%	1.91%
Lost messages	0.30%	0.63%	1.02%

Table 5: Efficiency of the rounds with respect to the number of processes, for rounds with $\Delta = 1000\mu s$ and 410 Mb/s of load.

This section concludes the experimental evaluation of our time hybrid system. As previously observed in Section 4, for each configuration of execution—number of processes and size of the messages—is possible to determine a suitable reference duration for the rounds, so that most of the messages broadcast in a round are received by all correct processes before the end of that round, enabling THyTOB to progress.

6 Related Work

There exists a considerable amount of literature on total order broadcast, and many algorithms have been proposed to solve this problem. In the following sections we briefly survey the five classes of total order broadcast algorithms that have been distinguished in the literature [19]: fixed sequencer, moving sequencer, privilege-based, communication history and destination agreement. We focus on the performance expected for algorithms falling into each class in broadcast networks (complementing some observations made in Section 4.6), but we also describe total order broadcasts that share structural aspects with THyTOB.

6.1 Fixed Sequencer

In fixed sequencer algorithms, a distinguished process is elected as sequencer and is responsible for ordering the messages. The role of sequencer is unique, and the sequencer is only replaced in case of failure. The sequencer may become a bottleneck because it have to receive all messages to be broadcast, and also the acknowledgments (acks) from all processes. For this reason, most fixed sequencer protocols [7, 9, 29] do not require all processes to send acks back to the sequencer, or allow processes to delivery messages before receiving acks from other processes. As a result, algorithms that fall into this class are rarely uniform [23], what means that processes suspected (even incorrectly) of being faulty may violate the total order.

However, this strategy of restricting the broadcast of messages to a distinguished process can result in high throughputs, especially in broadcast-based networks, as with Ring-Paxos [36] (see Section 4.6), which uses a ring-overlay topology in order to prevent the sequencer to handle all acknowledgments.

6.2 Moving Sequencer

In moving sequencer algorithms, unlike fixed sequencer ones, the role of sequencer is constantly passed from one process to another, even if there is no failures. The motivation is to distribute the load of ordering and broadcast messages among several sequencers, thus avoiding the bottleneck caused by having a single sequencer. The role of sequencer is represented by a token, that circulates among the processes, and that also carries acknowledgments, what simplifies the detection of message stability. All moving sequencer protocols we are aware of [15, 31] are optimizations of the by Chang and Maxemchuk's protocol [13]. Unfortunately, we are unaware of implementations of these protocols, but as they allow processes to broadcast messages concurrently and uncoordinately, they probably achieve poor performance in broadcast networks [36].

6.3 Privilege-based

Privilege-based algorithms rely on the idea that processes can broadcast messages only when they are granted the privilege to do so. In asynchronous privilege-based algorithms, like Totem [5] (presented in Section 4.6) and On-Demand [16], this privilege circulates from process to process in the form of a token. When a process receives the token, it broadcasts some messages along with the token, which also gathers acknowledgments for all messages previously broadcast. As processes have to wait for the token to broadcast messages, and messages are only delivered after a full round-trip of the token, privilege-based algorithms present the worst latency from all classes [23]. However, as broadcasts are coordinated, they theoretically should achieve the best throughputs, especially when considering broadcast networks [23, 36].

In synchronous privilege-based algorithms the privilege of broadcast message is determined by predefined time slots, using the technique also known as timed division multiple access (TDMA). The only total order broadcast that proceeds in synchronous rounds we are aware of was proposed by Gopal and Tueg [25]; it is a privilege-based algorithm, based on the TDMA technique, and works as follows. For each round r a process is designated the *transmitter*. The transmitter of a round is the only process that broadcast applications messages in that round, while the remaining processes broadcast acknowledgments for previous messages. Messages are delivered once they are acknowledged, three rounds after the initial broadcast, and at most a message is delivered per round—while THyTOB delivers one message from each process per round with latency of two rounds. Unfortunately, we are unaware of implementations or follow-ups of this protocol.

6.4 Communication History

In communication history algorithms the message ordering is also defined by the senders, but unlike in privilege-based algorithms, processes can broadcast messages at any time. Messages carry physical or logical clocks that allow processes to observe the messages other processes have broadcast and received, that is, to build the communication history in the system. Based on the communication history and a predefined ordering strategy, processes learn when it is safe to deliver messages without violating the total order.

THyTOB is a communication history algorithm, and is also LCR [27] (presented in Section 4.6). Both depend on processes to maintain global views of the system state, and require all processes to periodically report their states, even when they have no messages to broadcast. To tolerate process failures LCR relies in a group membership service and requires perfect failure detection, abstraction that can only be implemented in systems with strong synchrony guarantees. Similarly to some communication history algorithms [2, 32], THyTOB does not natively tolerate process failures, but it relies in a simple recovery procedure which is live under minimal conditions of synchrony and optimal regarding the number of failures tolerated.

Regarding performance, communication history algorithms theoretically have the best throughput and latency of all classes when considering small systems with high load of broadcasts, evenly distributed among the processes [23]. In pratice, the performance of such algorithms is limited by the network contention, and by the instability of the system in presence of concurrent broadcasts. LCR circumvents these limitations by arranging the process in a ring, thus preventing messages from different processes to collide. THyTOB, in turn, coordinates the processes and conditions the load in order to achieve high performance.

6.5 Destination Agreement

In the last class of algorithms, the delivery order is obtained in a distributed fashion, and results from an agreement between the destination processes. Thus, destination agreement algorithms achieve total order broadcast through the resolution of a sequence of instances of consensus (as described in Section 3.4). In some algorithms the subject of the agreement is the sequence number for a message (e.g., [8]); in others is the acceptance of a message order proposed by some process (e.g., [6]); but in most algorithms the ith instance of consensus determines the ith sequence of messages to be delivered [12, 20, 33]. Protocols that fall into this class are known for their relative poor performance on the absence of failures [23], which was observed in Section 4.6 considering two implementations of Paxos [33]. The main advantage of destination agreement algorithms is to inherit properties from the consensus algorithm they rely on, thus they ensure progress under minimal conditions of synchrony, and natively tolerate an optimal number of failures [12].

An interesting destination agreement algorithm, based on the concept of communication-closed rounds, is ATR [20]. It is based in a distributed execution model called Synchronized Phase System (SPS), in which, similarly to our time hybrid system, processes try to proceed in rounds like in synchronous systems. But, unlike our system model, rounds have asynchronous semantics: a process only proceeds to round $r + 1$ when it receives all messages

sent in round r by processes it trusts; otherwise, if the process suspects that some process failed, it starts a new phase from round zero. A phase only succeeds when all processes trust in the same subset of processes, so that all processes in the phase reach round one. In successful phases processes exchange set of messages, that are delivered in a consensual total order. ATR assumes asynchronous reliable channels and eventually perfect failures detectors [12]—assumptions that are stricter than those considered by THyTOB. Unfortunately, we are also unaware of implementations or follow-ups of this protocol.

7 Conclusion

The performance results allow us to conclude that our time hybrid solution represents an interesting trade-off between performance and simplicity for total order broadcasts. The simplicity of the algorithm also allows its use as a benchmark for total order broadcast atop Ethernet. All in all, THyTOB makes a strong case for the reconsideration of the common wisdom regarding the design and implementation of total order protocols that is: total order protocols must be always designed for the asynchronous model because it is the best way to guarantee safety and liveness (performance).

The experiments show that indeed clusters can behave synchronously and without failures for periods long enough to warrant THyTOB a potential throughput advantage over more complex total order broadcast protocols originally devised for the crash-recover asynchronous model. This observation allows us to consider that a reconfigurable combination of synchronous and asynchronous total order protocols, for example THyTOB and Paxos [33], can probably provide the best overall performance for replicated applications. During the long synchronous failure-free periods THyTOB is used and during the asynchronous failure-prone periods Paxos is used. So, it seems that there is still some promising design choices to explore in the creation of total order broadcast protocols. Future work is going to further address the ways synchronicity and asynchronicity, different failure assumptions, and reconfiguration can be used to improve the overall performance of total order broadcasts.

References

[1] Dennis Abts and Bob Felderman. A guided tour of data-center networking. *Commun. ACM*, 55(6):44–51, June 2012.

[2] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, D(91128):209–218, 2000.

[3] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University, 2004.

[4] Yair Amir and Jonathan Kirsch. Paxos for system builders. In *LADIS '08: Proceedings of Large-Scale Distributed Systems and Middleware*, New York, September 2008.

[5] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, and Paul Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, November 1995.

[6] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 292–301, 1997.

[7] Bela Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, Cornell University, 1998.

[8] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[9] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions Computer Systems*, 9(3):272–314, August 1991.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[11] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[12] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[13] Jo-Mei Chang and Nicholas F. Maxemchuk. Reliable broadcast protocols. *ACM Transactional on Computer Systems (TOCS)*, 2(3):251–273, August 1984.

[14] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71, 2009. 10.1007/s00446-009-0084-6.

[15] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Autonomous Decentralized Systems, 1995. Proceedings. ISADS 95., Second International Symposium on*, pages 215–221, 1995.

[16] Flaviu Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[17] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. In *Proceedings of the 4th workshop on ACM SIGOPS European workshop*, EW 4, pages 6–17, New York, NY, USA, 1990. ACM.

[18] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, 1999.

[19] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[20] Carole Delporte-Gallet and Hugues Fauconnier. Real-time fault-tolerant atomic broadcast. In *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pages 48–55, 1999.

[21] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[22] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[23] Xavier Défago, André Schiper, and Péter Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[24] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[25] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 110–123. Springer Berlin / Heidelberg, 1989. 10.1007/3-540-51687-5_36.

[26] R. Guerraoui and M. Raynal. The alpha of indulgent consensus. *The Computer Journal*, 50(1):53, 2007.

[27] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[28] T. Hoefler, C. Siebert, and Wolfgang Rehm. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[29] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the Amoeba group communication system. *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 0:436, 1996.

[30] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *SIGACT News*, 32(2):45–63, 2001.

[31] Jongsung Kim and Cheeha Kim. A total ordering protocol using a dynamic token-passing scheme. *Distributed Systems Engineering*, 4(2):87, 1997.

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[33] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[34] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41:63–73, March 2010.

[35] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[36] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN 2010: 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 527–536, Chicago, USA, June 2010.

[37] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 224–237, New York, NY, USA, 1997. ACM.

[38] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, May 2008.