Universidade Estadual de Campinas
Instituto de Computação

Daniel Cason

The role of synchrony on the performance of Paxos

O papel da sincronia no desempenho de Paxos

CAMPINAS
2017

Daniel Cason

The role of synchrony on the performance of Paxos

O papel da sincronia no desempenho de Paxos

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Luiz Eduardo Buzato

Este exemplar corresponde à versão final da Tese defendida por Daniel Cason e orientada pelo Prof. Dr. Luiz Eduardo Buzato.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2011/23705-4
ORCID: http://orcid.org/0000-0003-1598-106X

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C27r

Cason, Daniel, 1987-
The role of synchrony on the performance of Paxos / Daniel Cason. – Campinas, SP : [s.n.], 2017.

Orientador: Luiz Eduardo Buzato.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas distribuídos. 2. Tolerância a falha (Computação). 3. Consenso distribuído (Computação). 4. Algoritmos de ordenação total. 5. Sincronização. I. Buzato, Luiz Eduardo,1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: O papel da sincronia no desempenho de Paxos
Palavras-chave em inglês:
Distributed systems
Fault-tolerant computing
Distributed consensus (Computer science)
Total order algorithms
Synchronization
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Luiz Eduardo Buzato [Orientador]
Fernando Pedone
Fernando Luís Dotti
Edmundo Roberto Mauro Madeira
Islene Calciolari Garcia
Data de defesa: 31-03-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Daniel Cason

The role of synchrony on the performance of Paxos

O papel da sincronia no desempenho de Paxos

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação / Universidade Estadual de Campinas
- Prof. Dr. Fernando Pedone
  Faculty of Informatics / University of Lugano
- Prof. Dr. Fernando Luís Dotti
  Faculdade de Informática / Pontifícia Universidade Católica do Rio Grande do Sul
- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação / Universidade Estadual de Campinas
- Profa. Dra. Islene Calciolari Garcia
  Instituto de Computação / Universidade Estadual de Campinas

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 31 de março de 2017

Living is a very dangerous business.

The Devil to Pay in the Backlands
João Guimarães Rosa

This was our first, remote contact with the Plan. I could easily be somewhere else now if I hadn't been in Belbo's office that day. I could be—who knows?—selling sesame seeds in Samarkand, or editing a series of books in Braille, or heading the First National Bank of Francesco Giuseppe Land. Counterfactual conditionals are always true, because the premise is false. But I was there that day, so now I am where I am.

Foucault's Pendulum
Umberto Eco

"When *I* use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."

"The question is," said Alice, "whether you *can* make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master—that's all."

Through the Looking Glass
Lewis Carroll

Resumo

Algoritmos de consenso e de difusão totalmente ordenada são centrais para a implementação de aplicações distribuídas tolerantes a falhas. Paxos é um algoritmo assíncrono de consenso comumente empregado para a implementação de difusão totalmente ordenada. De forma breve, Paxos proporciona que mensagens enviadas a um conjunto de processos sejam a eles entregues em uma mesma ordem total. Uma funcionalidade que é simples de se obter sob condições normais de operação de um sistema distribuído, mas que se torna reconhecidamente complicada quando se observa uma mescla de assincronia e falhas. O projeto assíncrono de Paxos visa assegurar um comportamento consistente sob condições particularmente adversas de operação: a robustez é o seu maior atributo. Mas quando se trata da operação regular do sistema, sem falhas e predominantemente síncrona, Paxos peca por não oferecer um desempenho compatível com os recursos disponíveis.

A busca por formas de se melhorar o desempenho de Paxos ganha importância quando o algoritmo passa a compor o núcleo de várias soluções de replicação. De fato, múltiplas variações de Paxos destinadas a obter difusão totalmente ordenada de alto desempenho foram propostas em paralelo ao nosso Doutorado. Elas abordam limitações do algoritmo original e sugerem alterações, que vão desde a reorganização da topologia até otimizações no uso da rede pelos processos. A nossa abordagem difere das existentes, ou paralelas, por envolver a adoção explícita de pressupostos de sincronia como forma de aprimorar o desempenho de Paxos. Trata-se de incorporar a um algoritmo essencialmente assíncrono abordagens empregadas em modelos mais restritivos, como o modelo síncrono.

Esta tese relata os resultados obtidos a partir da abordagem de agregar hipóteses de sincronia a Paxos como forma de aprimorar seu desempenho. Nossas contribuições podem ser sintetizadas da seguinte forma. Primeiramente, mostramos que uma rede local, dado que a ela se apliquem mecanismos de controle de carga, se comporta na maior parte do tempo de forma predominantemente síncrona. Este resultado é atestado por, e propiciou o desenvolvimento de dois algoritmos de difusão totalmente ordenada. O Time Hybrid Total Order Broadcast (THyTOB) representa a possiblidade efetiva de se implementar um algoritmo de difusão totalmente ordenada essencialmente síncrono que opera sobre um sistema tipicamente modelado como assíncrono. THyTOB não apenas apresenta um desempenho, na ausências de falhas, superior a implementações tradicionais de Paxos, como também se destacou por sua reduzida variação de latências: é um algoritmo estável com desempenho bastante previsível. Apresentamos também nosso On-Time Fast Paxos, um algoritmo que usa tempo para gerar uma ordenação total para a as mensagens, que é então ratificada por instâncias de Fast Paxos. On-Time Fast Paxos apresenta altas vazões, da ordem da capacidade da rede, enquanto provê latências baixas e bem condicionadas. Trata-se da comprovação de nossa tese: sincronia melhora o desempenho de Paxos.

Abstract

Algorithms of consensus and of total order broadcast are central for the implementation of fault-tolerant distributed applications. Paxos is an asynchronous consensus algorithm commonly used to implement total order broadcast. Briefly, Paxos provides that messages sent to a set of processes are delivered by them in the same total order. A functionality that is simple to implement under the normal operation of a distributed system, but which becomes admittedly complicated when a mix of asynchrony and failures is observed. The asynchronous design of Paxos aims to secure a consistent behavior under a particularly adverse ensemble of operating conditions: robustness is its major attribute. But when it comes to the regular, fail-free and predominantly synchronous, operation of the system, Paxos fails to deliver a performance compatible with the available resources.

The quest for ways to improve the performance of Paxos gains importance when the algorithm progressively becomes part of the core of multiple replication solutions. In fact, several variations of Paxos designed for achieving high-performance total order broadcast were proposed in parallel with our PhD research. They address limitations of the original algorithm and suggest changes, from the reorganization of the topology to optimizations in the use of the network by the processes. Our approach differs from existing, or parallel approaches because it involves the explicit adoption of synchrony assumptions as a way to improve performance of Paxos. This is about incorporating to an essentially asynchronous algorithm approaches adopted by stricter models, such as the synchronous model.

This thesis reports the results achieved from the approach of augmenting Paxos with synchronous assumptions as a way to improve its performance. Our contributions can be summarized as follows. First, we demonstrate that a local area network, given that load control mechanisms are adopted, behaves in a predominantly synchronously manner most of the time. This result is attested by, and lead to the development of two total order broadcast algorithms. Our Time Hybrid Total Order Broadcast (THyTOB) represents the ability of implementing an essentially synchronous total order broadcast algorithm atop of a system typically modelled as asynchronous. THyTOB not only performs, in the absence of failures, better than traditional implementations of Paxos, but also has a very low latency variation: it is a stable and very predictable algorithm. Then we have On-Time Fast Paxos, an algorithm that uses time to generate a total order for the broadcast messages, which is ratified by instances of Fast Paxos. On-Time Fast Paxos sports high throughput, close the network capacity, while providing low and conditioned latencies. This proves our thesis: synchrony actually improves the performance of Paxos.

Contents

Chapter 1

Introduction

Consensus and total order broadcast are fundamental building blocks for the development of fault-tolerant distributed applications. The solution of the consensus problem demands from a set of processes the selection of a single value out of a set of proposed values. Total order broadcast, in turn, allows messages to be sent to a set of processes, with the guarantee that they will be delivered to all processes in the same total order. They are both agreement problems—processes agree on a single value, or on the delivery order for messages—whose solutions become particularly challenging in the presence of failures that affect part of the components hosting the distributed system execution. Total order broadcast and consensus are reducible to each other, so the interaction between these two problems has long been of practical and theoretical interest.

Consensus is probably *the* central problem of fault-tolerant distributed computing. Formalized in the early 1980s [45, 70, 78], consensus gains notoriety on an account of a result published in 1985, commonly referred to by the initials of their authors: Fischer, Lynch, and Paterson (FLP) [46]. The FLP result states that consensus can not be deterministically solved in an asynchronous distributed system even if only one of its processes may fail by crashing. By modeling a distributed system as asynchronous, the authors make no assumptions about the relative speeds of processes or about the latencies for delivering messages. This is equivalent to assume that there are no bounds on processing and on communication delays, which therefore can be completely arbitrary. Under such conditions, in particular, a crashed process, which abandoned the computation unannounced, becomes indistinguishable from a correct but relatively slow process, that takes too long to respond. This kind of uncertainty, brought by the combination of asynchrony and partial failures, is at the core of the FLP result and makes some problems of distributed computation particularly difficult in the asynchronous model.

If solving consensus in the asynchronous model is challenging, at the opposite end of the temporal spectrum, in the synchronous model, its resolution becomes fairly trivial. A synchronous system enforces deadlines for the processes to complete processing steps and for communication channels to deliver messages to the destinations. There are thus bounds on the processing delays and communication latencies, which are known by the processes and used by them to coordinate actions and detect failures. Solving consensus under such conditions, in practice, comes down to ensuring that all processes are aware of the same set of proposed values, among which they deterministically pick one as their

decision value. A strategy that is effective in the synchronous model because processes know how much time they have to wait in order to ensure that all proposals issued by non-crashed processes are received [83]. The same does not apply to the asynchronous model, where a process has at every step to choose between: (i) wait for additional information about the proposed values, with the risk of waiting forever for reports coming from a crashed process; or (ii) decide based on the information already obtained, with the risk of processes deciding differently when considering only partial information.

Unfortunately, unless specialized hardware and software components with real-time guarantees are employed, distributed systems in general are not strictly synchronous. In fact, even under "good" conditions of execution for dedicated systems operating under controlled load, we can observe some operations that take a long time to complete, when compared to median delays. Due to such outliers, deriving safe and practical bounds on the delays experienced by a distributed system is a challenging task, if at all possible. So, the difficulty with synchronicity encourages the adoption of the asynchronous model, that is based on a less restrictive set of assumptions and thus applies to a wider range of systems, regardless of the synchronism they actually present. This is also a choice for preserving the *safety* of distributed algorithms, at the expense of possibly hindering their *liveness* in the presence of failures or unstable system behavior. The safety of an algorithm is defined by invariants, that usually describe situations that can not occur at any point of the execution; for example, in consensus no two processes decide differently. Liveness properties, in turn, describe a state that must be reached, or something that must happen at some point of the execution; in consensus, for example, every correct process eventually decides. An alternative interpretation of the FLP result states that no consensus algorithm can be both safe and live in the asynchronous model: as the usual choice is to preserve safety, additional assumptions of asynchrony are needed to ensure liveness, in particular in the presence of failures, as we detail below.

A number of models of distributed computation fill the temporal spectrum between asynchrony and synchrony, being grouped under the nomenclature of partially synchronous models [44]. These models typically *augment* the asynchronous model with information about the temporal behavior of processes and channels, as a way to circumvent the FLP impossibility [42]. But unlike in the synchronous model, the temporal information may be incomplete or may not apply throughout the entire computation, as formalized in 1988 by Dwork et al. when describing the first two versions of partial synchrony [44]. In one version, the system does behave synchronously but the upper bounds on processing and communication delays respected by the system are not known a priori by the processes. In the other version, the system can behave asynchronously (presenting arbitrary delays) for an arbitrarily long period of time, but it eventually starts to behave synchronously (with bounded delays). Processes do not know the instant at which this transition occurs, known as global stabilization time (GST), but they know the upper bounds on processing and communication delays that are valid from the GST. In both versions of the model, algorithms must be designed to preserve safety whatever the delays experienced in the system since processes never know which bounds actually apply to the delays at each instant of the computation. Liveness, in turn, may be subject to the correct inference of the upper bounds on delays or of their observance, guaranteed from GST onwards.

A distinct approach for augmenting the asynchronous model with timing information is elaborated in the timed asynchronous distributed system model [32, 35]. Formalized in 1999 by Cristian and Fetzer [35], the model starts from an asynchronous system, with no bounds on processing and communication delays, using an unreliable messaging service, where messages can be lost—while the FLP model assumes reliable communication. This base system is augmented with the assumption that processes have access to a reasonably precise source of time, in the form of local *hardware clocks*. The model does not require clocks to be synchronized but their drift rates from real-time are assumed to be bounded, which makes them particularly useful for measuring time intervals. The availability of a time reference supports the definition of *timed* services, whose specifications do not only include outputs and state transitions but also the expected maximum times for them to occur. The maximum time for a service to be performed can be stipulated by sampling the distribution of delays measured in experiments, then deriving probabilistic upper bounds for them. Whenever a stipulated timeout delay is not met, there is a *performance failure*. There are no bounds on the frequency or on the number of components affected by such timing failures—otherwise the system would not be asynchronous. However, as a liveness assumption, there must be time intervals of some minimum length in which the frequency of performance failures will be below a certain threshold. During such intervals most of the components behave "timely", and this transient synchrony ensures the termination of the algorithms—a concept similar to the GST time of partially synchronous models.

Finally, the quest for the minimal synchronism needed to circumvent the FLP result had a relevant, and possibly definitive, chapter when Chandra and Toueg introduced in 1996 the concept of unreliable failure detectors [24, 26]. A failure detector is a distributed *oracle* that provides hints about the state of the processes: correct or crashed. Oracles can make an unbounded number of mistakes, reporting as crashed a correct process (a false positive) or as correct a crashed process (a false negative). But to be useful, oracles must satisfy two properties that restrict the mistakes they can make, known as completeness and accuracy, which restrict respectively false negatives and false positives. The authors defined two completeness and four accuracy properties, thus generating eight classes of failure detectors. The strictest class is called P and specifies the perfect failure detectors, which never suspect a non-crashed process, but can only be implemented in synchronous systems. The next class $\Diamond P$ defines the eventually perfect failure detectors, which can be implemented in the partially synchronous models proposed by Dwork et al. [44]. But perhaps the most well-known class of failure detectors is $\Diamond S$, which has been proven to provide the least information about failures necessary and sufficient to solve consensus [24]. This is to say that consensus can only be solved in systems whose synchronism is enough to allow the implementation of a failure detector satisfying the properties of class $\Diamond S$.

The importance of Chandra and Toueg's results is attested by the almost unanimous adoption of unreliable failures detectors, or equivalent abstractions, in modern consensus implementations. Most of these implementations are derived, or at least inspired by two renowned algorithms: the Chandra and Toueg's consensus algorithm [26], which employs a $\Diamond S$ failure detector, and the Lamport's Paxos algorithm [64], which uses the equivalent abstraction of a Ω leader election. The two algorithms are intended for the asynchronous model and ensure the safety properties of consensus despite the synchronism presented by

the system. They also share some operational aspects: both proceed in rounds, each round being coordinated by a given process. A round succeeds when the value selected by the coordinator is accepted by a majority of processes, thus becoming the decision value. A round may not succeed when the coordinator crashes or becomes irresponsive, a situation that is (possibly mistakenly) identified by the failure detector or leader election oracles, and triggers the start of new rounds. Multiple rounds may then be running at the same time, and an inter-round consistency protocol ensures that processes in different rounds do not decide differently (safety). Moreover, to ensure that a value is eventually decided (liveness) a correct and "timely" process must start a round and not be suspected by the other processes. The guarantee that this scenario is observed stems from the properties that must be satisfied by the $\Diamond S$ failure detector or the Ω leader election oracles, which in turn depends on the system to present a minimum of stability and synchrony.

The starting point of our work is Paxos, when used as a multi-consensus algorithm for the implementation of total order broadcast and state machine replication [85]. The choice of Paxos is derived from the fact that it is the most implemented consensus algorithm, both in academy (e.g., [6, 15, 75, 76, 77, 84, 91]) and industry (e.g., [20, 23, 31, 53, 54, 74]). In addition, the system model adopted by Paxos is slightly more wide-ranging than the adopted by the Chandra and Toueg's algorithm, as it assumes unreliable communication channels and that crashed processes can later recover, rejoining the computation with the same identity. As a consequence, for example, the coordinator of a round in Paxos may start new rounds when it suspects that the current round has not succeeded due to message loss. Also, by considering the crash-recovery failure model processes have to store all relevant information (rounds started, values accepted, etc.) in persistent memory, so that they can remember the actions taken before they crashed. In short, the design of Paxos takes into account a number of practical aspects from real systems, which makes it more conducive to becoming a product than, for example, the Chandra and Toueg's algorithm. This does not make, however, the implementation of Paxos a simple task; on the contrary, a number of non-trivial design decisions must be taken to obtain a safe and efficient solution. We join in this effort (e.g., [6, 15, 23, 76, 84]), focusing on the impact of the synchrony on the performance of Paxos when used for the implementation of total order broadcast.

1.1 Total Order Broadcast

Having presented some of the most important distributed system models, in the light of the temporal aspect and the resolution of consensus, now we focus on the main subject of our thesis: total order broadcast algorithms. Consensus and fault-tolerant total order broadcast are equivalent, which means that in a distributed system in which consensus is solvable so is total order broadcast, for instance by reduction to consensus—and vice-versa. The reduction of consensus to total order broadcast is quite straightforward: since the same messages are delivered to all processes in the total same order, processes can decide for the first proposed value delivered to them. This reduction was formalized by Dolev et al. [42] when investigating the minimal synchronism needed to solve consensus.

They found that an asynchronous system augmented with broadcast communication and the guarantee that messages are delivered in the same total order (called synchronous message order by them) allows overcoming the FLP impossibility. The reduction in the opposite direction was formalized by Chandra and Toueg [26], as a way to extend the use of failure detectors to the resolution of total order broadcast—called atomic broadcast by them and in several other works. We present this reduction in detail in Section 4.2, but briefly, total order broadcast can be obtained through the execution of multiple instances of consensus, in which ordered sets of broadcast messages are proposed. Each instance is identified by a natural number, and the values decided in successive consensus instances determine the total order in which messages are delivered to all processes.

The reduction to consensus is not the only approach to achieve total order broadcast. In fact, among the about sixty total order broadcast algorithms surveyed by Defago et al. in 2004 [40], only fifteen fall into the destination agreement class, by using consensus to build total order. The authors also defined four other classes of algorithms that, at least in regular, fail-free operation, do not rely on consensus for message ordering. Total order in these algorithms is generated by the processes that initiate broadcasts—the senders, in the privilege-based and communication history classes—or by processes with the role of intercepting and assigning sequence numbers to the messages—in the fixed sequencer and moving sequencer classes. We briefly survey the five classes of algorithms in Section 2.6, outlining representative total order broadcasts that fit into each class. For now, we focus on a high-level comparison between consensus-based algorithms and other approaches to achieve total order broadcast, first in terms of fault-tolerance and then of performance. Note that Paxos, although not considered in the survey of Defago et al. [40], would fit into the destination agreement class, just as does the Chandra and Toueg's algorithm [26].

The main distinction between total order broadcasts based and not based on consensus is probably how they deal with the consequences of the FLP result. For instance, most of the algorithms cataloged by Defago et al. [40] are intended to the asynchronous model, in which fault-tolerant consensus, and so total order broadcast, is not trivially obtained. Nevertheless, not a few of them assume failure detection patterns whose implementation would require the system to present a considerable amount of synchronism. This raises the question of how would these algorithms operate when the system behaves in a truly asynchronous manner, so that the accuracy of failure detection becomes very poor. The same question applies, even to a lesser extent, to total order broadcasts that rely on group membership services [16, 17] to provide consistent views of the set of operational processes. It has been proven that group membership can not be achieved in asynchronous systems, even when processes erroneously suspected to have crashed can be forcibly removed from the group [25]. As a result, it is not really clear which properties algorithms that rely on group membership services are able to preserve when the system goes through periods of instability, leading to a succession of view changes. When relying on consensus, Paxos in particular, it is quite clear that safety properties are preserved despite the synchronism presented by the system and the mistakes made by the failure detection oracles. Liveness, in turn, depends on certain synchrony of the system, at least the synchronism enough for a correct process to impose on the others as the coordinator of the algorithm.

If on the one hand consensus-based total order broadcasts are potentially more robust,

in terms of tolerance to asynchrony and failures, on the other hand, their performance is admittedly low when compared to other approaches for message ordering. For example, in the analytical performance comparison of strategies for message ordering performed by Defago et al. [39], the destination agreement class was ranked as the worst, with the highest latency and the lowest throughput (broadcast completion rate) among the five classes of algorithms. This result does not necessarily apply to any consensus-based total order broadcast, once different communication patterns may be adopted, but it certainly reinforces what appears to be a common belief in the high cost of consensus when there are no failures [56]. The fact is that algorithms from other classes solve during regular, fail-free operation problems that would be equivalent to a weaker form of consensus, in which progress may be prevented by a single failure. These algorithms resort to stronger (and more costly) procedures, typically equivalent to consensus, only in case of (suspect of) failures, in order to restore the state and reconfigure the system. Put another way, algorithms based on consensus, such as Paxos, solve at each iteration a problem that is strictly stronger than the problems typically solved during regular operation by total order broadcasts that rely on group membership or reconfiguration services [69].

In terms of experimental evaluation of total order broadcasts the results we have or are aware of (e.g., [6, 15, 84, 76, 92]) show that "pure" consensus-based solutions present a rather poor performance. These are Paxos implementations that attempt to meet, as far as possible, the Lamport's original specification [64, 65, 67] while pointing out their practical limitations and proposing ways to overcome them. Most of these experimental evaluations are, in particular, part of works that devise optimizations and variations of Paxos, intended to improve the performance of the original algorithm. Among the Paxos variations, we can highlight Ring Paxos [76] and S-Paxos [15], included in the evaluation of total order broadcasts presented in Chapter 3, focusing on their latency distributions. Both are high-throughput algorithms, optimized for local-area networks, and developed in parallel with our research; they are based on a comprehensive performance analysis of Paxos, from which the authors draw the main elements of their algorithms. Ring Paxos addresses the high cost of the all-to-all communication pattern adopted by Paxos, due to the loss of the network reliability in the presence of multiple simultaneous senders, by organizing the processes in an overlay ring topology [76]. S-Paxos, in turn, addresses the high load applied by Paxos, as a leader-centric algorithm, to the process elected as the coordinator, by decoupling the dissemination from the ordering of messages [15].

Our research is also based on a comprehensive analysis of the performance of Paxos, considering the implementation maintained by our research group as a part of the Treplica toolkit [91]. Treplica also implements Fast Paxos [67], a variation of Paxos that allows to reduce, under favorable conditions, the cost of consensus in one communication step. In short, when conflicting proposals from different processes do not collide, Fast Paxos can decide a value in two communication delays, while Paxos requires three delays but does not suffer from collisions. Experimental evaluations of these two implementations [21, 92] revealed an interesting trend: as the protocols reach their saturation load, message loss and timing violations are responsible for reducing the efficiency of consensus instances. Our diagnosis for this effect is that it is a result of the essentially asynchronous design of Paxos and Fast Paxos, and the consequent lack of mechanisms for the coordination

of processes. Our hypothesis is then that the inherent synchronism of the system would enable the coordination of the processes, and thus a more effective use of the available resources. What is translated into betting on the synchrony presented by the system as a way of to improve the performance of total order broadcasts based on Paxos.

1.2 The Role of Synchrony

Taking as reference the survey by Defago et al. [40], the portion of total order broadcasts designed for the synchronous model, or that explicitly adopt timing assumptions, is rather small. This category includes algorithms that support the operation of real-time systems, with specialized software and hardware, such as TTP [59], MARS [61], and RTCAST [1]. And also more generic algorithms, which may or may not consider access to synchronized clocks, but invariably assume the existence of upper bounds on the communication and processing delays. These algorithms (e.g. [12, 14, 33, 48]) use the common notion of time that processes are assumed to have, or be able to construct, to generate a total order for the messages. Processes thus attach timestamps—whether physical, from synchronized clocks, or logical—to the broadcast messages, and other unique identifiers—usually the sender id—to sort messages with the same timestamp. As the system is synchronous, it is possible to establish a stabilization period Δ so that all messages with timestamp up to t are guaranteed to be received by all correct processes up to time $t + \Delta$. In the absence of failures, messages can then be delivered in total order as they become stable, so with a very predictable latency. Failures, in turn, may cause processes to build distinct views of the set of messages that have been broadcast over a certain interval, which are typically reconciled through the execution of a number of instances of consensus.

It is noteworthy that, when considering modern off-the-shelf distributed systems, the fundamental assumption common to these synchronous total order broadcasts is hardly fulfilled. In fact, deriving a practical stabilization period Δ that is guaranteed to apply throughout the execution for all correct components is mostly infeasible. An alternative approach, taken by some newer algorithms, is to relax the requirement of a maximum latency Δ while strengthening the clock requirements. For instance, the use of loosely synchronized physical clocks to generate total order in wide-area networks, backed by a group membership service, is proposed in [43]; the use of timestamps from GPS clocks, with conflicts caused by asynchronous behaviour handled using consensus, is presented in [100]; and Google Spanner [31] goes one step further, associating its correctness (safety included) to the existence of synchronized clocks with bounded skew. Thus, we can say that there is in the literature relatively recent examples of designs that come back to time-based ordering strategies, introduced in seminal works on replication and fault tolerance [63, 85], but without exploring in full the implications of synchrony.

We present a similar approach in this thesis, of building a time-based total order for the messages. But unlike existing solutions, we do not employ timestamps derived directly from synchronized physical clocks. Instead, our solutions are based on the abstraction of a global clock that periodically ticks at every process. Consecutive global clock ticks determine a discrete time base, with granularity considerably greater than synchronized

physical clocks, that provides timestamps for the broadcast messages. At the same time, clock ticks induce in the processes a lock-step operation, that is typical of synchronous systems [73]. Augmenting a distributed system, typically modeled as asynchronous, with synchronous assumptions, in the form of a global time base and a lock-step operation, is the core of our approach. In the remainder of the thesis we discuss how to make this approach feasible, and the benefits it brings to the performance of total order broadcasts.

1.3 Contributions

The following Chapters contain three papers that we produced as a part of the Doctorate. Chapter 2 presents our Time Hybrid Total Order Broadcast (THyTOB), a synchronous ordering protocol built atop of an asynchronous distributed system based on a broadcast network. THyTOB exploits the inherent synchrony of the broadcast network to generate, in the absence of process failures, a straightforward total order for the messages, resorting to a consensus algorithm, developed with Paxos in mind, only in the case of failure suspicion. The paper presents the design, implementation, and performance of THyTOB, as well as a comprehensive assessment of the synchrony actually observed in a distributed system based on a broadcast network; the experiments mention a key result on the use of synchrony in distributed systems published in 1999 by Cristian and Fetzer [35], but updated for modern systems. The contents of this chapter have been published in the 77^{th} volume, March 2015 issue, of the *Journal of Parallel and Distributed Computing*.

Chapter 3 contains the paper entitled *Chasing the tail of atomic broadcast protocols*, presented in the *IEEE 34th Symposium on Reliable Distributed Systems* that took place in Montreal, CA, during September 2015. It studies four high performance total order broadcast (also known as atomic broadcast) protocols representative of different classes of protocol design and characterizes their latency tail distribution under different workloads. Thus, it sheds light on an unexplored aspect of the performance of total order broadcast protocols: latency variability and the role of outliers in the latency distribution, that is the influence of the latency distribution tail. The assessed protocols were THyTOB, Spread (a well-known broadcast communication toolkit), S-Paxos, and Ring Paxos (Paxos variations, mentioned above). Our hypothesis, confirmed by the results, was that stricter flow and load control mechanisms would entail well-behaved latencies—the results show that it was particularly true in the case of THyTOB.

Chapter 4 presents On-Time Fast Paxos, which represents our final step in the integration of a synchronous, time-based ordering protocol with Fast Paxos, an asynchronous crash-recover consensus protocol. On this way, On-Time Fast Paxos is able to guarantee the original safety and liveness of Paxos and Fast Paxos, while exploiting the inherent synchrony presented by the system to disseminate and order the broadcast messages in a very efficient way. The result is probably the first total order broadcast implementation based on Fast Paxos that can provide high throughput while preserving a stable behavior, with controlled and quite predictable latencies. This paper is still unpublished[1], and we are working on a shorter version to be submitted to a journal.

[1] Available as TR IC-17-02 at `http://www.ic.unicamp.br/~reltech/2017/abstracts.html`

Chapter 2

Time Hybrid Total Order Broadcast: Exploiting the Inherent Synchrony of Broadcast Networks

Total order broadcast is a fundamental communication primitive for the construction of highly-available systems. Informally, the primitive guarantees that messages sent by a group of processes are delivered to all processes in the same order. This paper investigates the design and performance of a very simple synchronous total order broadcast that is built atop of an asynchronous distributed system based on a broadcast network. Our Time Hybrid Total Order Broadcast (THyTOB) explores the inherent synchrony of the broadcast network to build a total order for the messages, while ensuring safety under asynchrony and in the presence of process failures. We assess the performance of THyTOB in an Ethernet-based commodity cluster, and show that it is on a par with the performance of other well-known, and more complex total order broadcast protocols inherently designed for the asynchronous model.

2.1 Introduction

Informally, total order broadcasts allow processes to send messages with the guarantee that all processes deliver messages in the same order. In this paper, we investigate the design and performance of a very simple *synchronous* total order broadcast that is built atop of an *asynchronous* distributed system based on a broadcast network. We show that the exploration of the inherent synchrony of the network can lead to a THyTOB protocol with good performance and reliability.

Broadcast networks, e.g., Ethernet or Infiniband, are adopted by the vast majority of current datacenter clusters and high-performance computers [2]. However, even with the algorithmic [40] and practical [20, 23, 34, 86] importance of total order broadcast protocols, to the best of our knowledge, no one has tried to assess the performance of a synchronous total order protocol built atop broadcast networks. In this context, the main contributions of our work are: (i) a characterization of the inherent synchrony presented by broadcast networks; and (ii) the description and assessement of THyTOB, a novel

total order broadcast that uses the inherent synchrony of the network as the key feature to simplify its design and implementation.

The characterization of the inherent synchrony of a broadcast network is carried out through experiments that systematically show that asynchronous distributed systems can be used as a foundation to emulate synchronous computations. A simple protocol is used to generate rounds of synchronous communication atop of the asynchronous distributed system. The accuracy and efficiency of the rounds generated are measured under varying workloads and scales to ensure that they meet the synchronization requirements of THyTOB. The picture that emerges from the experiments allows us to conclude that it is possible to organize the computations of the asynchronous distributed system as a sequence of synchronous rounds.

The description of THyTOB—its specification, analysis of its progress and safety properties, and its performance evaluation—shows that it is on a par with the performance of other well-known total order broadcast protocols. THyTOB is latency optimal, requiring only two communication steps to deliver totally ordered messages. When set side by side with other implementations of total order broadcast, some based on Paxos [64], such as Treplica [91] or Ring-Paxos [76], other based on virtual synchrony [17], such as Spread [5] or LCR [49], THyTOB fares well both in terms of throughput and latency. Our study compares THyTOB to six other total order broadcast protocols: THyTOB ranks third in terms of throughput and first in terms of latency. In our comparison, the top performer in terms of throughput can sustain 950 Mb/s with an average latency of 4.6 ms while THyTOB can deliver 525 Mb/s but with a very precise latency of 2.3 ms. The results presented in the paper show that THyTOB fills an important gap in the throughput-latency spectrum of total order broadcast protocols designed for broadcast networks.

THyTOB adds to a trend towards the revision of the TCP/IP protocol stack for use in datacenters and clouds because of the benefits the revised protocols can bring to the performance and scalability of distributed applications. For instance, the synchronous nature of the datacenter network environment has allowed the implementation of a time-division multiple access (TDMA) MAC layer for commodity Ethernet that allows end hosts to dispense with TCP's congestion control [89]. The performance and simplicity of THyTOB makes it a good alternative to more complex protocols for several classes of cluster applications. For example, it can be used to implement an MPI Broadcast (`MPI_BCAST`) primitive equivalent to the one proposed by [52], but based on a much simpler and more efficient algorithm.

The design of THyTOB explores the fact that the processes and the broadcast network of asynchronous distributed systems have an inherent tendency to behave as synchronous distributed systems during reasonably long periods of time; Section 2.2 contains the definitions of both models of computation. Section 2.3 discusses in detail the design and implementation of THyTOB; special attention is given to the role of synchrony and asynchrony in the behavior of the protocol. Section 2.4 assesses the performance of THyTOB, and compares it with the performance of other well-known total order broadcasts. This section also discusses the different design principles that were used in each of the protocols and analyses the influence each of them had on the performance and fault-tolerance of the

protocols. From the perspective of THyTOB, the first three sections are self-contained because a complete understanding of the protocol can be obtained without looking into the experiments that demonstrate the inherent synchrony exhibited by broadcast networks. Nevertheless, the feasibility of the sub-systems that implement both models is verified experimentally in Section 2.5. Section 2.6 surveys the main classes of total order protocols designed so far and compares them with THyTOB; the section also comments on the relation of our time hybrid system and the concept of network synchronizers. Section 2.7 closes the paper; it summarizes our results and discuss briefly why we consider the principles that guide THyTOB's design an interesting contribution to the engineering of total order broadcast protocols, given the vast increase in the use of broadcast networks as key components of distributed systems.

2.2 Models of Computation

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

2.2.1 The Asynchronous Model

An asynchronous distributed system is composed of a fixed set of n processes connected by a broadcast network. Processes are asynchronous because there is no bound in the time they take to perform each of their computing 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 arbitrarily delayed, but we assume that they 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 do 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 guarantee that periodicity is maintained at every moment of the computation; but when considering sufficiently long periods of computation, the average interval of real time between successive invocations of the scheduled task is expected to be reasonably close to 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 non-faulty processes. Observe that this assumption does not impose bounds to the processing delays or network latencies: it only enables us to specify 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 are no bounds on the frequency with which performance failures can occur, or in 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 non-faulty processes are completed within δ_S. What this means is 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 number of performance failures has an upper bound.

Thus, we have complemented the well-known asynchronous crash-stop model with unreliable broadcast channels using two assumptions borrowed from the timed-asynchronous system model [35]: access to local clocks with bounded drift rates, and probabilistic maximum delays δ_S for the messages as a function of their size. The validity of these additional assumptions is restricted to periods of stability during which processes and channels exhibit a (predominantly) synchronous behavior.

2.2.2 The Synchronous Model

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. It is a logical mechanism that generates ticks based on the processes' local physical 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 a *failure-free* synchronous round is governed by the following rules:

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

(ii) All messages the processes broadcast in round r are received by all processes within the same round r. Thus, as in [73], 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 [44] and [73] 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; a time hybrid system. Unfortunately, the total isolation of failures that occurs in asynchronous systems 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, leading to the introduction of performance failures in the synchronous model.

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 at 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 within a round, 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 performance evaluation.

2.3 THyTOB: The Protocol

This section presents THyTOB, an uniform total order broadcast protocol designed for the synchronous model of computation (Section 2.2.2). A client is any program whose processes are capable of obtaining the service provided by a protocol. Formally, THyTOB is defined through the service provided by its two primitives: *broadcast(m)* and *deliver(m)*, where m is a message. When a process of the client invokes *broadcast(m)*, we say it *broadcasts* m, that is, it requests the total order broadcast protocol to send m to all processes of the client. Similarly, when a process of the client invokes *deliver(m)*, we say it delivers m, that is, the message m, that has been totally ordered by the protocol, is delivered to the invoker of *deliver(m)*. 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 in a straightforward way. We first describe THyTOB considering a synchronous computation without failures, then we present the procedures to tolerate performance failures, and process crash failures, respectively, as two consecutive sections.

2.3.1 The Basic Protocol

To illustrate how 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 to all processes a message with sequence number i to be totally ordered. 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 are 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 (referring to all messages with sequence number i) in round r.

As p has received n messages tagged with sequence number i in round r, it is enabled to broadcast another message in the next round, with sequence number $i + 1$. This is actually the progress condition of the THyTOB, it sets the reception of the messages previously broadcast as a prior requirement to the broadcast of new messages:

Condition 1. *A process is enabled to broadcast a message with sequence 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 specifically learns that the remaining processes have also succeeded in receiving the messages i in a previous round. The set of messages p received in round r then becomes stable, and can be safely delivered 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. Two rounds is the minimal latency for uniform total order broadcast by reduction to uniform consensus that also requires at least two communication steps to be solved [56]. The fulfillment of the *uniform agreement* property requires the design of a second condition to guide the protocol:

Condition 2. *A process is enabled to deliver the messages with sequence number i after successfully having received all messages with sequence number $i + 1$ within a previous round.*

In order to turn these two conditions into the total order broadcast protocol, we have to detail the operation of two key mechanisms. First, whenever Condition 1 is true, every

process does broadcast a new message. If the process does not have new messages to broadcast, it broadcasts a *null* message, that is not delivered by the processes but is required to guarantee Condition 2. Second, whenever Condition 2 becomes true for the first time for a given message sequence number, the process delivers the set of messages that has become stable. As a means to ensure *total order* delivery, every process applies the same deterministic ordering function to the messages before delivering them. As a result, the first time that a process succeeds in receiving the messages i, it delivers the messages tagged $i-1$ using the total order determined by the ordering function, then it broadcasts a new (possibly *null*) message with sequence number $i+1$.

Correctness of the protocol 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 participating of a round receive the same set of messages, composed by all messages that were broadcast in that round. The processes apply the same ordering function to the messages received in each round, therefore they build the same sequence of messages tagged $i \in \{1, 2, 3, \ldots\}$ at the end of every successful round. The trivial total order achieved through the concatenation of such sequences of messages is the foundation of THyTOB. Uniformity is ensured by postponing the delivery of (already ordered) messages, as specified by Condition 2.

2.3.2 Performance Failures

In this section we discuss how the protocol handles performance failures. Let us recall that a process is said to *succeed in a round* when: (i) it receives n messages, one from each process; and (ii) all messages received have the same sequence number. Here, we use two scenarios to characterize the impact of performance failures upon these two conditions, respectively. Observe that, from now, we make no distinction between messages and the sequence numbers they carry, so messages i means messages tagged i.

In the first scenario, suppose a round r in which no process succeeds because no process has received the n messages expected for the round. This scenario may occur, for example, when a process p misses 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 in the next round, p broadcasts 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 may succeed in receiving the messages i, since they will continue to broadcast their messages i until that happens.

In the second scenario, consider 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 due to performance failures—synchronization failures, latency bounds missed, message loss, etc.—only some of them receive the n messages broadcast. As a consequence, in the round that 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 leads to an impasse: there are rounds where,

even in the absence of performance failures, the processes can not succeed because they are desynchronized.

The procedure to circumvent the impasse consists in making the processes that are "ahead"—the ones that succeeded in the round r illustrated above—to *revisit* smaller sequence numbers, in order to enable the processes "behind" to resynchronize themselves with those that are ahead. More specifically, a process that broadcasts a message i in a round but receives a message $j < i$ from some process, has to revisit the sequence number j. Processes that revisit j behave like processes that never succeeded in receiving the messages j: they continue to broadcast their messages j until they eventually succeed in receiving the messages j. Once this happens, the processes that came back will broadcast their messages $j + 1$, but they will not deliver the messages $j - 1$ again. Note that the success of this procedure requires all processes that were "ahead" to revisit j, and all processes have to succeed in receiving the messages j in the same round, otherwise they recur to the second scenario. These predicates are actually sufficient to circumvent the impasse, since the enabling conditions of THyTOB ensure that the processes that come back to smaller sequence numbers can only have revisited the immediate predecessor of 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 a process p that has broadcast a message i in round r. From Condition 1, p has already received all the messages with sequence number $i - 1$ in a round previous to r. Let r^* be the smallest round in which this occurred, that is, in which 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^* every process 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^*. Observe that from round r^* there is no possibility of a process to broadcast messages $j < i-1$. Therefore, since $r > r^*$ (from Condition 1, as stated before), 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, 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 when all processes succeed in the round. Otherwise, if only part of the processes has succeed in the round, we have the second scenario, that can be only be bypassed when the processes "ahead" revisit the sequence number i, that is, when the system goes into the state where only messages tagged with i are seen in a round. Consequently, the progress of THyTOB depends on failure-free rounds, so that all processes succeed in receiving the messages i. Some of them could have revisited 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 messages tagged $i - 1$ and broadcast its message identified by $i + 1$ for the first time.

2.3.3 Pseudocode

In this section we present the pseudocode of THyTOB (Algorithm 1) that represents an implementation of the conditions (Section 2.3.1) that dictate the general behavior of the protocol and the changes necessary to make it resilient to performance failures (Section 2.3.2).

The THyTOB's pseudocode is structured as two **upon-do** clauses with the meaning that **upon** the occurrence of the *event/condition* stated within **upon-do** the actions that follow the **do** are executed. Lines 1 to 5 show the initialization of the protocol. Lines 6 to 16 show the main procedure of the protocol, executed by every non-faulty process at round r, it 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 totally order them (Line 10) as the ith sequence of messages;

(ii) it delivers the sequence $i - 1$ of messages, the last stable set of messages it has assembled (Line 11);

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

Otherwise, when the process succeeds in receiving the messages tagged 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, that necessarily revisited 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 $i + 1$ again.

When the round is not successful for the process, it does not perform any of the actions discussed above, it simply repeats the broadcast of the last message (stored in the variable `current` (Line 16)). However, if some message received during the round had a sequence number j that was smaller than its current sequence number (Line 14), the process has to revisit j (Line 15). In this case, the broadcast (Line 16) is going to contain the messages associated with the sequence number j. As stated by Lemma 1, the process can only revisit the sequence number that is the immediate predecessor of the highest sequence number it has used, kept in the variable `last` that is updated only when a new message is broadcast (Line 9). Thus, when the process revisits a sequence number, the value of the variable `current` becomes equal to `last - 1`, and this specifically prevents the process from delivering the messages tagged $j - 1$ again (Line 8).

2.3.4 Process Crash Failures

The progress of THyTOB depends on the successive construction of global views of the system, so if a process crashes the protocol cannot progress. Also, as already discussed in

```
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 (M)⟩ do
7:     if |M| = n and ∀m ∈ M : m.seq = current then
8:         if current = last then
9:             last ← last + 1
10:            sequence[current] ← sortMessages(M)
11:            deliver(sequence[current − 1])
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: Pseudocode of the THyTOB protocol.

Section 2.3.2, when asynchrony (performance failures) affects some or all of the processes over a period of time, rounds do not succeed and THyTOB's progress is hindered. In this section we present the procedure THyTOB uses to handle the presence of crashed processes or long periods of asynchrony. For the first time in the paper, we have to address the consequences of dealing with a time hybrid system: THyTOB is synchronous but implemented atop an asynchronous distributed system. The handling of process crashes and/or asynchrony requires the use of a failure detector and of an asynchronous implementation of uniform consensus.

The failure detector is run in parallel with the main task of THyTOB (Algorithm 1) and is able to indicate, albeit unreliably, the occurrence of process crashes and/or of asynchronicity. In practice, what the failure detector does is to monitor the performance of THyTOB so that when it deteriorates significantly, it triggers a recovery procedure based on uniform consensus. A period of time with low throughput or a certain number of unsuccessful rounds can be used by the failure detector as a threshold to decide when the performance has worsened enough to trigger the recovery of the system. The thresholds' value can be determined either by an adaptive procedure or through the use of a static value. Because the distributed system is asynchronous, as defined in Section 2.2.1, crashed processes cannot be deterministically distinguished from slow processes [46], thus the failure detector will always be unreliable. So, it is not possible to use the information given by the failure detector to take THyTOB back into a consistent state from where it can resume its normal operation. The only way now to recover the system is by relying on an asynchronous consensus algorithm, that is safe under asynchrony and indulgent towards the failure detector mistakes [50], but ensures liveness under minimal conditions of synchrony, given that up to $f < n/2$ processes fail by crashing [24, 42].

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 [24, 50]: (a) if a process decides on some value, then this value was proposed by some process; (b) two processes do not decide different values; and (c) every correct process eventually decides on some value. Total order broadcast and consensus are equivalent problems [24], 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 the failure detector signals to a process, say p, that progress is hindered, p switches from the main task of THyTOB to the consensus implementation. Then, it initiates consensus *instances* in order to decide whether the messages already broadcast by THyTOB can be safely delivered. Consider i as the highest sequence number used by p—the value of variable `last` in Algorithm 1—when it switched to the consensus algorithm, then the scenario that captures the state of the system is as follows:

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

(ii) From Condition 2, p 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 p has never broadcast its message $i+1$, no process can have delivered the sequence of messages i.

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

The algorithm executed by a process when it suspects that THyTOB may have stalled, or when it detects that other process has already started the recovery procedure is the following:

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

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

Step 3. In consensus instance `last - 1` the process *proposes* the value `sequence[last - 1]`;

Step 4. In consensus instances from `last` the process can *propose* any value received from other process, or the special value *stop*;

Step 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 switches from the THyTOB protocol, forcing the remaining processes—which will not succeed anymore in the synchronous 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 Steps 4 and 5 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 sequences $j \geq i$ could not have been delivered by any process in THyTOB, so that the process can accept any value proposed for these instances. When the first *stop* command is included in the prefix of decided instances, the process learns that there are no more pending messages to be delivered, and the procedure is halted.

At 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 transition to the execution of any other fault-tolerant asynchronous total order broadcast protocol. When the system eventually restores its synchronism and/or all (possibly crashed) processes recover, the processes of the system can transition back to THyTOB. The second alternative consists in selecting, once the recovery procedure is concluded, a new set of non-faulty processes to start a new execution of THyTOB. A group membership service [17] or special instances of a consensus protocol can be used to reconfigure the system to include a new set of processes [69]. 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 the crashed or misbehaving processes that are preventing the progress of THyTOB.

2.4 THyTOB: Performance Evaluation

This section presents the experiments carried out to assess the performance of the THyTOB protocol. The experiments evaluate scenarios in which there are no process failures, as

they are expected to be rare events in our target system. By contrast, performance failures occur relatively often, because of the system asynchrony that becomes particularly evident under higher network loads. In particular, the results show that the performance of THyTOB is quite *predictable*, reinforcing our assumption that an asynchronous distributed system presents a synchronous behavior most of the time, given that the load applied by the protocol can be conditioned.

First, in Section 2.4.1, we present the experimental setup that was used in the experiments. Next, the description of the experiments, including the analysis of the results, is developed in stages. Sections 2.4.2 and 2.4.3 contain results related, respectively, to throughput and latency. Throughput measures the number of broadcasts that the protocol can complete per unit of time. Latency measures the time required to complete a single broadcast without contention. Section 2.4.4 assesses the performance of the protocol as a function of the message size. Section 2.4.5 shows what happens to performance when the scale of the system, that is, the number of processes, is increased. The last stage of the performance evaluation, Section 2.4.6, contains a comparison of the performance of THyTOB with the performance of other interesting total order broadcasts, that were designed for the same computing environment considered by THyTOB.

2.4.1 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 600 lines of code, as a subsystem of Treplica [91]. Treplica is a replication toolkit, implementing a consensus-based total order broadcast protocol, which includes a failure-detection module, persistence and communication layers, totaling about 4000 lines of code. Processes communicate through UDP (for unreliable datagram transport) and the network-level multicast primitive provided by the IP protocol over Ethernet. THyTOB implementation includes a protocol that implements the round-based model of computation described in Section 2.2.2. The description of this protocol, including its implementation and the main characteristics of the rounds it generates, has been left to Section 2.5.

The load for the experiments was generated by a simple client program that broadcasts a message of S bytes whenever a deliver is executed. The parameters of the experiments were: the size S of the message in bytes, the reference duration Δ for the synchronous rounds, measured in microseconds ($1\mu s = 10^{-6}s$), and the number n of processes. Experiments lasted for 10 minutes, and were repeated at least five times. For the next two sections, namely Sections 2.4.2, and 2.4.3, the parameters used in the experiments were: $n = 5$ and $S = 10$ KB. Additionally, the results discussed in the two sections come from the same set of experiments.

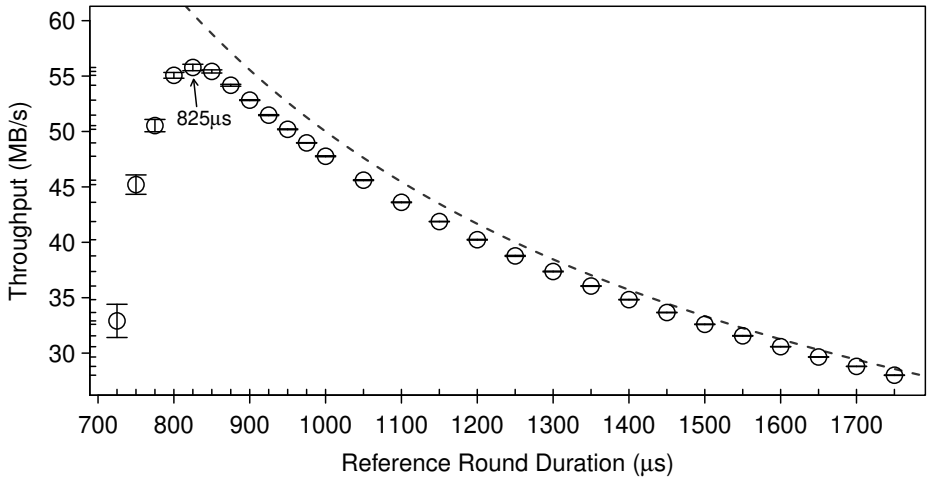

Figure 2.1: Throughput as a function of the reference duration Δ, with 5 processes and messages of 10 KB. The dashed curve is the optimal throughput for the experiments as a function of Δ, given by $(5 * 10000/\Delta)$ MB/s.

2.4.2 Throughput

Figure 2.1 presents the throughput achieved by THyTOB as a function of Δ. In the Figure, the circles and bars represent, respectively, the averages and the standard deviations of the mean throughput measured for each of the 5 executions of the system. The dashed line represents the optimal throughput for each experiment, i.e. the throughput that should have been achieved in the absence of performance failures, when all processes succeed in every round. Note that the larger the Δ selected for the rounds, the closer the throughput is to the optimal throughput expected for the protocol, with the implication that a much larger proportion of rounds resulted in successful broadcasts. This behavior is expected, since with the increased duration of the rounds, the occurrence of variations in the processing delays or in the communication latencies are less likely to result in performance failures, and therefore less likely to prevent rounds to be successful.

At the rightmost circle in Figure 2.1, with rounds of $1750\mu s$, THyTOB achieves 28.01 ± 0.02 MB/s of throughput, the value is equivalent to 98% of the optimal throughput for the experiment, that is 28.57 MB/s. In this case, we say that THyTOB had an efficiency of 98%, meaning that the protocol does not complete broadcasts in only 2% of the rounds executed. Efficiency is therefore defined as the ratio (percentage) of the total number of rounds that were successfully completed by the protocol. When we gradually reduce the duration of a round 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 throughput increases with the reduction of Δ, reaching its maximum value 55.78 ± 0.29 MB/s at $\Delta = 825\mu s$. From this point on, small reductions in Δ cause large efficiency drops (88% with $\Delta = 800\mu s$, 78% with $\Delta = 775\mu s$), which are no longer offset by the increase of the optimal throughput. As a result, at the leftmost circle in Figure 2.1, with $\Delta = 750\mu s$, the measured throughput was 33.20 ± 1.42 MB/s, equivalent to an efficiency of only 48%. Observe that not only the average throughput decreased

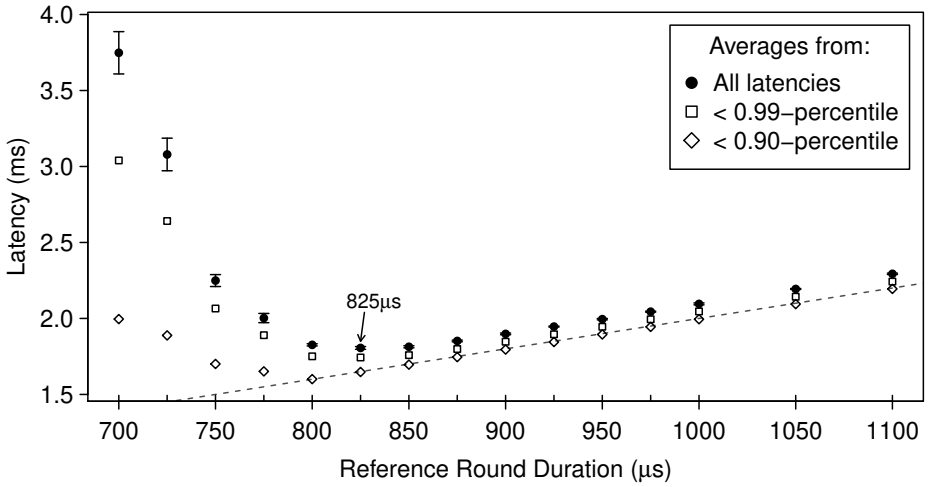

Figure 2.2: Latency as a function of the reference duration Δ, with 5 processes and messages of 10 KB. The dashed curve is the optimal latency for the experiments in function of Δ, given by $(2 * \Delta/1000)$ ms. Observe that the lowest average latency was achieved for $\Delta = 825\mu s$, the same duration for the rounds in which the throughput was maximal.

but also the standard deviation increased, a consequence of the instability of the system when the rounds become too short to support the load generated by the protocol.

2.4.3 Latency

Figure 2.2 shows the average latencies measured during the experiments. The black dots and the bars represent the average and the standard deviations for the mean latencies measured in five executions of the protocol. The dashed curve is the optimal latency for the experiments, it is a function of 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 design of the protocol implies that throughput and latency are positively correlated: as the processes can only broadcast new messages when they complete broadcasts, the more frequently they deliver messages, the smaller is the latency achieved. Thus, as one would expected, the best latency for this configuration was achieved at $\Delta = 825\mu s$, which is the inflection point for both curves of throughput and latency. The latency for this point was 1.80 ± 0.01 ms, which is only 9.4% above the optimal value 1.65 ms.

The white symbols plotted in Figure 2.2 show the average latencies obtained when the higher 1% (squares) and the higher 10% (diamonds) latencies measured are ignored in the computation of the averages. Note that from $\Delta = 800\mu s$ onward the three averages (black dots, white squares, and white diamonds) get closer to each other and also get increasingly closer to the line that represents the optimal latency achievable in the experiments. For instance, with $\Delta = 825\mu s$ the 90-percentile latency was 2.08Δ and the 99-percentile was 5.89Δ, with the meaning that at least 90% of messages were delivered at the optimal latency of two communication rounds, and 90% of the remaining messages had latencies of at most 6 rounds. These results, in conjunction with the fact that messages are delivered in 92% of the rounds (Section 2.4.2), allow us to conclude that the execution of the protocol is

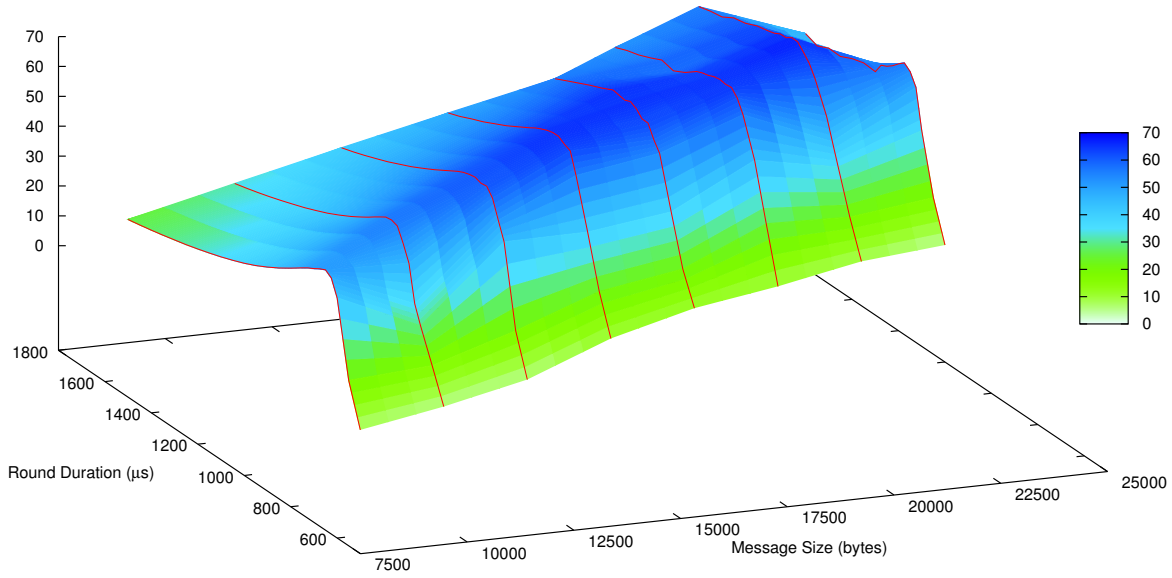

Figure 2.3: Throughput as a function of message size S and the duration of the rounds Δ, with $n = 5$ processes.

Message Size (S)	7.5 KB	10 KB	12.5 KB	15 KB	17.5 KB	20 KB	22.5 KB	25 KB
Round Duration (Δ)	$650\mu s$	$825\mu s$	$925\mu s$	$1025\mu s$	$1175\mu s$	$1250\mu s$	$1375\mu s$	$1375\mu s$
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 2.1: Best performance of THyTOB with 5 processes and several message sizes.

mainly composed by long uninterrupted sequences of successful rounds, in which processes deliver n messages per round with a latency of two rounds. Such sequences are interrupted by a few unsuccessful rounds, resulting in latencies of 3 to 5 rounds, and only rarely by longer periods of instability, that are also controlled: from $\Delta = 800\mu s$ to $1750\mu s$ the 99.9-percentile latencies are in the range 12-14 ms.

2.4.4 Message Size

In order to check whether the protocol was robust for a wider range of the parameter S, the message size, and Δ, the reference duration of the rounds, we carried out a set of experiments where the two parameters varied: S ranged from 7.5 KB to 25 KB, and Δ ranged from $550\mu s$ to $1800\mu s$. The behavior of the protocol with these parameters is analogous to the behavior of the protocol observed in the two previous sections. Specifically, for every message size there is an optimal reference duration Δ_{opt} for which the throughput achieved is maximal and the average latency is minimal (Table 2.1 and Figure 2.3).

The first result that can be observed in Figure 2.3 is that the shape of the curves for each configuration (plotted as lines) are similar to the shape of Figure 2.1. The throughput increases with the reduction in Δ, accompanied by a slight reduction of efficiency, until

Number of Processes	$n = 5$	$n = 10$	$n = 15$
Message Size (S)	15 KB	7.5 KB	5.0 KB
Round Duration (Δ)	$1025\mu s$	$1150\mu s$	$1300\mu s$
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 2.2: Performance of THyTOB as a function of the number of processes, when 75 KB of messages were broadcast per round.

Δ_{opt} is reached; around Δ_{opt} the system is saturated and the best throughputs for each configuration are observed (darker regions); then small reductions in Δ cause abrupt drops in the efficiency, and the throughput decreases to almost zero (lighter 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 throughput achievable. This behavior is explained by the increasing workload applied to the protocol 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 2.1.

A second result that can be observed in Figure 2.3 is the curve of throughput as a function of the message size; there is an increase in the value of the throughput until it reaches 67.0 ± 0.7 MB/s, with messages of 22.5 KB. From this point onward the efficiency drops fast as the message size increases, 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. When smaller messages are used, a reduction in the workload increases efficiency, it 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 it would be necessary to further reduce the duration for the rounds, but for shorter rounds the system can hardly comply with the deadlines it has to meet to sustain the rounds, so that the performance in this configuration is clearly limited by its reduced optimal throughput.

2.4.5 Process Scale

In all experiments presented so far the system was composed of five processes. In order to assess the scalability of THyTOB we have created the following experiment. First, we have selected from a set of configurations tested, a scale, $n = 5$, message size, $S = 15$ KB, and network workload, 75 KB/s. Next, with network workload and message size fixed, we have executed the system at three different scales: 5, 10 and 15 processes; the results of the experiments are summarized in Table 2.2.

As the scale increases, there is a slow decrease in the throughput observed, and a

proportional increase in the mean latency: it drops approximately 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 expected.

2.4.6 THyTOB versus other protocols

Table 2.3 compares the performance of THyTOB to that of six other total order broadcast protocols: LCR [49], Ring-Paxos [76], Zab [54], Spread [5], Treplica [91], and the protocol presented in [6] that we refer hereafter as Paxos4SB. There are published (and quite detailed) performance results for these protocols, obtained in experimental settings very similar to the one used in the experiments with THyTOB, namely Ethernet-based commodity clusters, with multicore machines interconnected by a Gigabit network switch. For this reason we selected these protocols, and the results presented in Table 2.3 are the best we found for each of them, published either by their own authors, or in previous performance comparison studies [49, 76].

The performance data for Ring-Paxos, LCR and Spread were taken from the paper that describes the former [76], for Zab we considered results of experiments when nothing is written to disk [54], and so for Treplica and for Paxos4SB, for which we considered the experiment with best throughput from the presented in [6]. All results in Table 2.3 refers to failure-free experiments with five processes, excepting Spread and Paxos4SB, from which we present their best results published, with three and twelve processes. Except for Zab, all these protocols (including THyTOB) batch small messages into bigger messages, so the message size column in Table 2.3 refers to the maximum batch size—which is optimized to best throughput. THyTOB appears three times in Table 2.3, in configurations with different message sizes: 22.5 KB, with the highest throughput achieved; 15 KB, with the best compromise between latency and throughput; and 10 KB, for which the efficiency of the protocol was maximal. The efficiency and the optimal duration for the rounds for these experiments can be found in Table 2.1. Note that the concept of rounds adopted by THyTOB comes from the synchronous model, and there is no equivalent for this concept in the other protocols—which are asynchronous, in the sense that processes have no access to a (not even logical) global source of time.

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 [63]. LCR achieves an almost-optimal throughput, corresponding to 95% of the network capacity, at cost of latencies that increase 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

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**
Zab	200 Mb/s	5.2ms	1 KB	Weak	Star	TCP	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 2.3: Comparison of uniform total order broadcasts, based on the best performance results published for each protocol. For Zab [54], Paxos4SB [6], and Treplica [91] we considered experiments when nothing is written to disk; results for LCR, Ring-Paxos and Spread were taken from [76]. The experimental setups reported by these works are very similar to the one in which THyTOB was assessed: Ethernet-based commodity clusters equipped with 1000 Megabit/s network switches.

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 smaller than LCR, but its latency is also smaller, and almost constant with the number of processes. Ring-Paxos tolerates message loss and process failures using instances of Paxos [64]—the consensus protocol of which Ring-Paxos is an optimization—, that are also responsible for reforming the ring.

In the lower portion of Table 2.3 we have two well-known commercial total order broadcasts: Zab, which is part of ZooKeeper, a highly-available coordination service developed by Yahoo[1], and Spread, which is one of the most-used group communication systems; then we have two consensus-based total order broadcasts implementing Paxos [64]: Paxos4SB and Treplica—the latter developed by our research group.

Zab supports a primary-backup scheme, in which a primary process executes the operations of the client, and propagates the incremental state changes to backup processes. In the absence of failures, Zab behaves like a two-phase commit protocol, and when a majority of processes suspect of the primary, a Paxos-based synchronization phase is started in order to establish a new primary and to recover the replicas' state. Zab adopts a star topology centered in the primary process, which is connected to $n - 1$ backup processes via TCP. As the primary have to send $n - 1$ copies of each message, Zab's throughput is clearly limited by the outgoing network bandwidth of the primary. The total order broadcast implementation of Spread is based on Totem [7], a ring-based protocol that allows a single process to broadcast messages at time. This privilege circulates the

[1]ZooKeeper is now maintained by Apache Fondation, and it is available on `http://zookeeper.apache.org`.

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 as 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 [56], and by the network instability observed when several processes broadcast messages concurrently and uncoordinately [76].

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 smaller 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 $n + 2$ 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.

2.5 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 reliable 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 2.2. The synchronous model organizes the distributed computation in rounds using a logical global clock (Section 2.2.2). 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 2.2.1).

2.5.1 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 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, up to that time, the 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, totalizing 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 handed over to the client, in our case THyTOB, for 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 using 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 $e' < e$ or $j < i$ this message is from a previous 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 [29, 44]. Finally, if $e' > e$ or $j > i$ 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.

2.5.2 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 broad-

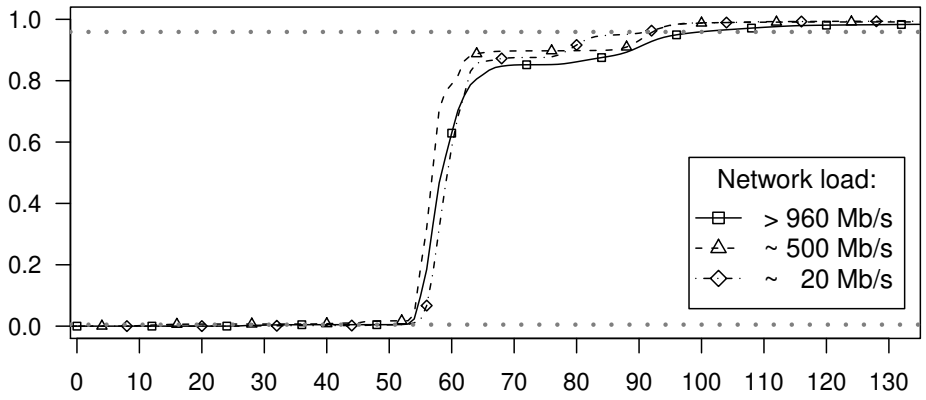

Figure 2.4: CDF for the processing delays, measured in μs, in the broadcast of 1.5 million ticks in three scenarios of load.

cast 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 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 in 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 has been elected as the synchronizer at the instant t_0 of its own 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 to the synchronizer's clock. The processing delay pd_i for tick i is then defined as $pd_i = ts_i - t_i$. The distribution of processing delays can be used to characterize the contribution of the system (namely, the processor, the operating system and the Java virtual machine) to the accuracy of the synchronizer.

Figure 2.4 presents the Cumulative Distribution Function (CDF) for the processing delays pd_i measured in five executions with five different synchronizers, that broadcast 300 thousand ticks in three scenarios of network load. For network workloads of 20, 500 and 1000 Mb/s, respectively, 98.4%, 97.1% and 95.5% of the processing delays were measured to be between 50 and 100μs, and more than 80% of them were within the range from 50 to 65μs. This 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 and precision, even with the tight period of 500μs and rate of 2000 ticks per second. Despite the very predictable behavior, respectively, 0.47%, 0.70% and 1.34% of the processing delays sampled were above 500μs, meaning that these ticks were broadcast after the time scheduled for the generation of the next tick. The ticks generated as a function of these much larger pd_is produced very short tick intervals, smaller than 100μs. This variability of the pd_is was observed in small percentages for larger Δs and for

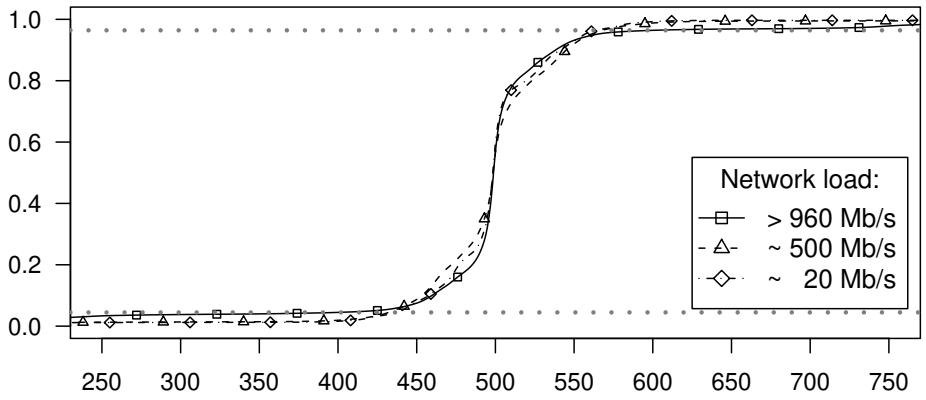

Figure 2.5: CDF for the duration of 1.5 million rounds, measured in μs by five processes, with reference value $\Delta = 500\mu s$.

different network workloads. The existence of these outliers indicate that asynchrony can be observed in the background of a predominantly synchronous behavior (Figure 2.4).

2.5.3 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 for 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 2.5 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 smaller 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 a duration of 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 [76]. 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 2.2.1, 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.

2.5.4 Latency Bound

The experiments conducted in this section consisted of five executions in which different synchronizers 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 smaller than 0.15%, 0.25% and 0.45% in executions with five, ten and fifteen processes.

Figure 2.6 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 than 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 approximately to the 98%-percentile of the distributions. Finally, in

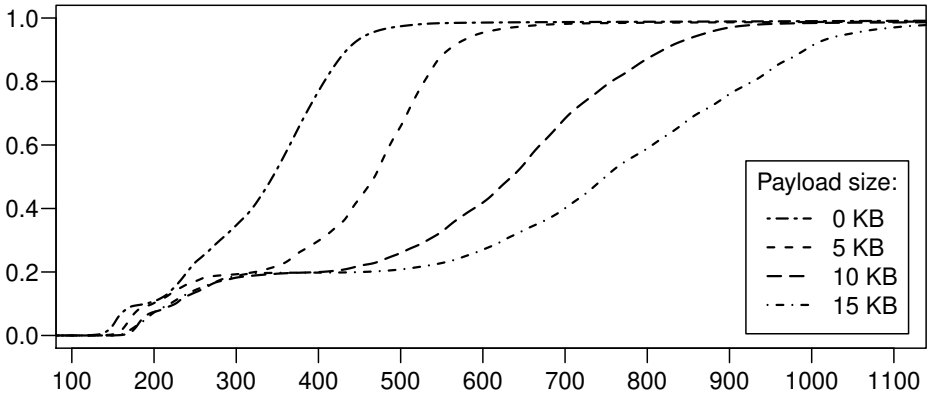

Figure 2.6: CDF for the RTT, given in μs, of messages broadcast by five processes, in rounds with reference duration $\Delta = 1000\mu s$.

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 2.4 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 2.4: Efficiency of the rounds as a function of 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 $950\mu s$ (densities above 0.01%), with ten processes 97.6% of them were up to $982\mu s$, and with fifteen processes 97.1% of them were up to $1084\mu s$. 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 2.5.

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 2.5: 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 2.4, 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.

2.6 Related Work

There exists a considerable amount of literature on total order broadcast, and many algorithms have been proposed to solve this problem. In Sections from 2.6.1 to 2.6.5 we briefly survey the classes of total order broadcast algorithms that have been distinguished in the literature [40]: 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 2.4.6), and also describe total order broadcasts that share structural aspects with THyTOB. Then, in Section 2.6.6 we compare the synchronizer used to organize computations into synchronous rounds in our time hybrid system (Section 2.5.1) with the concept of network synchronizers found in the literature.

2.6.1 Fixed Sequencer

In fixed sequencer algorithms, a distinguished process is elected as sequencer and it is responsible for ordering the messages. The role of sequencer is unique, and the sequencer is only replaced in the case of its failure. The sequencer may become a bottleneck because it has to receive all messages to be broadcast, and also the acknowledgments (acks) from all processes. For this reason, most fixed sequencer protocols [11, 18, 55] do not require all processes to send acks back to the sequencer, or to allow processes to deliver messages before receiving acks from other processes. As a result, algorithms that fall into this class are rarely uniform [39], with the consequence that processes suspected of being faulty (even incorrectly) may violate the total order.

However, the strategy of restricting the broadcast of messages to a distinguished process, combined with the adoption of a ring topology that allows the the sequencer to handle the acknowledgments in an efficient way, has allowed Ring-Paxos [76] (Section 2.4.6) to achieve high throughput in broadcast networks.

2.6.2 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 [36, 57] are optimizations of the Chang and Maxemchuk's protocol [27]. 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 [76].

2.6.3 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 [7] (presented in Section 2.4.6) and On-Demand [4], 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 [39]. However, as broadcasts are coordinated, they theoretically should achieve the best throughputs, especially when considering broadcast networks [39, 76].

As synchronous privilege-based total order broadcasts, we highlight the family of algorithms proposed by Gopal and Toueg [48]. The algorithms operate in synchronous rounds, and a single process, designated as the *transmitter*, has the privilege of broadcasting messages at the beginning of each round[2]. The remaining processes listen to the transmitter, and deliver their messages, in the order they have been received, when the messages become stable. Upon detecting the transmitter's failure, the processes start some consensus instances (which also proceed in synchronous rounds) in order to agree on the last messages the transmitter broadcast. The number of rounds needed to detect either the stability of a message, or the transmitter's failure, depends on the failure model adopted. In particular, when considering $f < n/2$ general omission failures, the message a non-faulty transmitter broadcasts in round r will be acknowledged in round $r + 1$, and delivered at the end of round $r + 2$. Therefore, in the absence of failures, processes deliver

[2] As pointed out by [40], it is possible to rotate this privilege by designating a different transmitter for each round r.

a message per round, with three rounds of latency, while THyTOB delivers n messages per round with latency of two rounds. In addition, THyTOB does not consider upperbounds on the number of processes affected by omissions failures, and because of that its recovery procedure must rely on asynchronous consensus. Thus, although sharing strategies with Gopal and Toueg's algorithms—trivial ordering using synchronous rounds, backed-up by consensus instances—, THyTOB adopts a considerably less restrictive system model.

2.6.4 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 [49] (see Section 2.4.6). 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 on a group membership service, and requires perfect failure detection—the latter can only be implemented in systems with strong synchrony guarantees. Similarly to some communication history algorithms [3, 63], THyTOB does not natively tolerate process failures, and 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 [39]. In practice, 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.

2.6.5 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 2.3.4). In some algorithms the subject of the agreement is the sequence number for a message (e.g., [17]); in others is the acceptance of a message order proposed by some process (e.g., [8]); but in most algorithms the ith instance of consensus determines the ith sequence of messages to be delivered [24, 41, 64]. Protocols that fall into this class are known for their relative poor performance in the absence of failures [39], which was observed in Section 2.4.6 considering two implementa-

tions of Paxos [64]. 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 [24].

In addition, the lower bounds for latencies of total order broadcasts are derived from algorithms of this class, by reduction to uniform consensus [56, 68]. The general lower bound for instances in which there are no failures (or suspect of failures), and considering that several processes can propose values (respectively, broadcast messages) simultaneously, is of three communication delays—achieved for example by Paxos [64]. However, given that there are no conflicting proposals, it is possible to solve a consensus instance in only two communication delays [68]. Algorithms adopting this optimization must be able to detect conflicts, and may have to pay an extra cost of a regular consensus instance to solve them—like in Fast Paxos [67]. When considering consensus-based total order broadcasts, conflicting proposals result from messages received by processes in different orders. Thus, by assuming that messages are likely to be received by different processes in the same order it is possible to build optimistic total order broadcast protocols [79], which deliver most of the messages in two communication delays. It is also possible to reduce the number of proposal conflicts by allowing some predefined subsets of messages to be delivered by different processes in commuted orders, observation that gave rise to Generic Broadcast [80] and Generalized Consensus [66] protocols.

Observe that optimistic and generic protocols solve relaxed versions of uniform total order broadcast, as a specified in Section 2.3, in order to achieve the optimal latency of two communication delays. THyTOB does not need to relax the Uniform Total Order property to achieve this latency due to two reasons. First, it proceeds in synchronous rounds, model in which it is possible to solve uniform consensus in the absence of failures in two communication steps [56]. Second, every sequence of messages generated by THyTOB is unique: composed by n messages, one from each process, with the same sequence number, and ordered in a deterministic way. Thus, if sequences are mapped to proposals for the corresponding consensus instances, like occurs in the recovery procedure, there is no possibility of having conflicting proposals.

An interesting destination agreement algorithm, based on the concept of communication-closed rounds, is ATR [41]. 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 [24]—assumptions that are stricter than those considered by THyTOB. Unfortunately, we are unaware of implementations or follow-ups of this protocol.

2.6.6 Synchronizers

To the best of our knowledge, the concept of synchronizer was introduced by Awerbuch et al [9], as a technique to simulate synchronous round-based computations in asynchronous point-to-point networks. The synchronizer is a distributed algorithm that generates a sequence of "clock-pulses" at each process, which are responsible for delimiting the rounds. However, unlike the ticks generated by the rounds protocol, a new pulse is generated at a process only when is guaranteed that all messages sent to that process at the previous rounds have already been received and processed. As a result, the synchronizer allows synchronous distributed algorithms to operate in an asynchronous network, but with an overhead of acknowledgments and signaling messages required for the operation of the synchronizer itself. Finally, the asynchronous model considered by the synchronizer is similar to the defined by [46], and differs from the presented in Section 2.2.1 for assuming reliable channels and processes which do not have access to any source of time.

In [30] the concept of synchronizers was extended to a model the authors called asynchronous bounded delay networks. In this model processes have access to local clocks with bounded drift rates, and channels deliver messages with an upperbounded delay. Thus, the only difference of this model to the synchronous model presented in [73] is that processes do not have access to a global source of time, what is necessary to organize computations into synchronous rounds. The synchronizer proposed is a procedure, executed by every process, that with the appropriate use of timers is able to decide when a new synchronous round must be started, and of which round each message received belongs to. The guarantee that all messages sent in a round are received by their destinations before the end of that round relies on the latency bound, and on the proper choice the duration for the rounds, which are a factor (2 or 3 times) higher than the latency bound. Therefore, while the synchronizer for asynchronous networks adds an overhead in message complexity, this synchronizer increases the time complexity for the synchronous distributed algorithm that is simulated.

Using a different approach, but also intended to organize computations in non-synchronous systems into synchronous rounds, Dwork et al. [44] proposed a distributed clock protocol for partially synchronous systems. Differently from the rounds induced by synchronizers, in the basic round model defined in this work not all messages sent in a round are received by their destinations: messages can be lost or can arrive after the end of that round. But similarly to the round-based computation model described in Section 2.2.2, all messages delivered to a process at the end of each round have necessarily been sent in that round. The distributed clock protocol is then intended to determine the beginning of the rounds, by synchronizing the clock ticks periodically generated by each process. More specifically, each process keeps a logical clock and operates in cycles determined by its physical clock; at the beginning of each cycle the process sends the value of its logical clock to every process; if at the end of a cycle the process receives tick messages $j \geq i$ from a majority of processes, it updates its logical clock to i, and starts its round i. The major difference between this protocol and the rounds protocol described in Section 2.5.1 is in message complexity: while this protocol requires two cycles and a quadratic number of messages (two broadcasts per process) per round, the rounds protocol, for electing a

process as synchronizer, requires only one broadcast per round.

2.7 Conclusion

THyTOB's performance allows us to conclude that in distributed systems based on broadcast networks total order broadcast operations do not have to be costly in terms of communication rounds because of the inherent synchrony of the system. 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 asynchronous model.

THyTOB's design 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).

It is also true that THyTOB is only effective while the system remains synchronous and failure free. This observation allows us to consider that a reconfigurable combination of synchronous and asynchronous total order broadcast protocols, for example THyTOB and Paxos [64], can probably provide the best overall performance and resiliency for datacenter replicated applications: during the synchronous failure-free periods THyTOB is used, while Paxos is only used during the (hopefully infrequent) asynchronous or failure-prone periods. We have already a robust implementation of Paxos and Fast Paxos [21, 91] that is going to be integrated with THyTOB using a consensus-based reconfiguration protocol [69].

In summary, it seems that THyTOB establishes a solid platform for the exploration of promising design alternatives for the implementation of total order broadcast protocols. Future work is going to address the ways synchronicity and asynchronicity, different failure assumptions, and reconfiguration can be used to improve the overall performance of total order broadcasts.

Acknowledgments

This work was partially founded by FAPESP under grants 2010/14555-6 and 2011/23705-4, and CNPq grant 473340/2009-7.

Chapter 3

Chasing the tail of atomic broadcast protocols

Many applications today rely on multiple services, whose results are combined to form the application's response. In such contexts, the most unreliable service and the slowest service determine the application's reliability and response time, respectively. State-machine replication and atomic broadcast are fundamental abstractions to build highly available services. In this paper, we consider the latency variability of atomic broadcast protocols. This is important because atomic broadcast has a direct impact on the response time of services. We study four high performance atomic broadcast protocols representative of different classes of protocol design and characterize their latency tail distribution under different workloads. Next, we assess how key design features of each protocol can possibly be related to the observed latency tail distributions. Our observations hint at request batching as a simple yet effective way to shorten the latency tails of some of the studied protocols; an improvement within the reach of application implementers. Indeed, our observation is not only verified experimentally, it allows us to assess which of the protocol's key design principles favor the construction of latency predictable protocols.

3.1 Introduction

Atomic broadcast, also known as total order broadcast, is a fundamental group communication abstraction that lies at the core of different approaches to replication, such as state-machine replication [63, 85]. In state-machine replication, servers replicate the service's state and deterministically execute client requests such that the failure of one or more replicas does not prevent client requests from being executed against non-faulty replicas. State-machine replication requires that (i) every non-faulty replica receive all requests and (ii) no two replicas disagree on the order in which requests are received. These requirements are captured by atomic broadcast.

Atomic broadcast has been extensively studied by the scientific community and many protocols under various assumptions have been proposed [40]. From a performance perspective, improving throughput and reducing latency have been the two main driving forces for new protocols. Years of research combined with developments in communica-

tion technology have led to atomic broadcast protocols that can meet most application requirements with respect to throughput and latency. In this paper, we consider an angle that has been unexplored in the design space of atomic broadcast protocols, namely *latency variability*. Latency variability is important because in many applications today, the execution of a request triggers the execution of multiple services, whose results are aggregated to form the request's response. In such environments, the responsiveness of the application will be determined by the latency of the slowest service; thus, each service must be as predictably responsive as possible. Although state-machine replication improves the availability of a service, it does not eliminate variations in the service's latency.

Two components account for the latency variability of a replicated service: the *ordering of requests*, that is, the atomic broadcast protocol, and the *execution of requests*. Taming the latency variability of the execution of requests has received much attention in the literature recently [37, 72]. It turns out that state-machine replication naturally reduces this source of variability in latency because commands are executed by all replicas and clients consider a command executed upon receiving the first (and thus the fastest) response from the replicas [37]. Reducing latency variations in the ordering of requests, however, requires different strategies since replicas interact in complex ways, depending on the total order protocol used, and slowdowns in one replica during the ordering of messages can easily slow down other replicas.

In this context, this paper analyses four different total order protocols with two goals in mind: (i) characterizing their design and implementation decisions in the light of their latency variability (i.e., latency tail distribution), and (ii) finding and assessing mechanisms that can be used to reduce the latency variability of these protocols.

The remainder of the paper is structured as follows. Section 3.2 presents the atomic broadcast protocols considered in our study. Section 3.3 explains our experimental methodology. Section 3.4 describes the results of our experimental evaluation. Section 3.5 discusses the lessons learned. Sections 3.6 reviews related work and Section 3.7 concludes the paper.

3.2 Atomic broadcast protocols

In this section, we overview the four atomic broadcast protocols considered in this study. These protocols are representatives of different engineering approaches to the problem of designing and implementing atomic broadcast.

3.2.1 System model

We consider a distributed system composed of processes that communicate with each other exclusively through message passing and do not have access to a common source of time. We assume processes may fail by crashing, but do not behave maliciously. The system is asynchronous and extended with additional assumptions to circumvent the FLP impossibility result [46]. For brevity, we do not describe these extensions here; detailed descriptions can be found in the papers that introduce the protocols assessed.

Atomic broadcast is defined by the primitives *broadcast*(m) and *deliver*(m), with the following guarantees: (i) if a non-faulty process broadcasts message m, it eventually delivers m; (ii) if a process delivers m, then all non-faulty processes deliver m; and (iii) no two processes deliver any two messages in different orders.

3.2.2 Ring Paxos

Ring Paxos is a high-throughput atomic broadcast protocol derived from Paxos [64]. Ring Paxos implements Paxos by distributing processes in a logical uni-directional ring [76]. Establishing a ring among processes maximizes throughput as it enables a balanced usage of the available bandwidth. In brief, Ring Paxos works as follows. Clients submit their requests to the processes in the ring. The dissemination of client requests is decoupled from the ordering of requests, which is accomplished using consensus on message identifiers. Requests are forwarded along the ring until they reach the leader, which creates the message identifiers used to totally order the requests. The leader tags each request with a unique identifier and initiates a consensus instance using a subset of the processes in the ring (i.e., Paxos's acceptors). The result of each consensus instance is circulated in the ring until all processes receive it. The execution of successive consensus instances induces a total order on requests.

3.2.3 S-Paxos

S-Paxos is an atomic broadcast protocol based on Paxos whose main design concern is to relieve the leader's load [15]. S-Paxos offloads the Paxos leader by balancing the load among the processes that execute the protocol. Similarly to Ring Paxos, S-Paxos separates request dissemination from ordering. Clients send their requests to a replica. Replicas create unique identifiers for the requests and send both (uid, request) to all other replicas. Requests become stable when acknowledged by a majority of replicas. When a request becomes stable, the leader passes its uid to the ordering layer, which uses instances of consensus. Batching is used by both the dissemination and the ordering layers to optimize throughput; acknowledgements are piggybacked on the messages used to transport the batches. S-Paxos operates on batches of requests and batches of uids. All processes execute all requests but only the replica that originally received a request forwards the result to the client. S-Paxos strives to balance CPU and network resources, the cost for which is paid by the number of messages that must be exchanged before a request can be ordered.

3.2.4 Spread

Spread is a widely used toolkit that provides several process group communication services, including atomic broadcast. Spread's initial atomic broadcast protocol was based on Totem [7] and was recently modified to include the Accelerated Ring protocol [10]. Both protocols distribute the processes in a logical ring and use a token to ensure total order. The token circulates along the ring and only the process that possesses the token can broadcast messages. In Totem, the process that holds the token increments its

sequence number, broadcasts a message, and passes the token to the next participant. In the Accelerated Ring protocol, the token holder can pass the token to its successor before completing the broadcast of messages. In both the protocols, if the token holder needs to broadcast multiple messages, the sequence number in the token must be updated once per each message. Spread implements a sophisticated flow control mechanism combining a global message window and two per-process message windows ("personal" and "accelerated") [10]. These windows are managed by the protocol in order to implement a ring-based reliable communication channel. The practical consequence of the focus on efficient communication is a token that circulates with predictable latency along the ring.

3.2.5 THyTOB

THyTOB uses rounds to total order client requests [22]. The idea is to emulate synchronous rounds of communication atop of an asynchronous distributed system. At the beginning of each round, all processes broadcast messages (batches of requests). Each broadcast message includes its sender's identifier and a sequence number. Rounds are communication-closed, meaning that all messages broadcast in a round are received by all processes within that same round. However, since the underlying network is unreliable and asynchronous, messages may be lost or arrive after the end of the round in which they were broadcast. THyTOB guarantees progress only in synchronous rounds. A process is allowed to broadcast a message with sequence number $i + 1$ only if it has received messages with sequence number i from all the other processes. If this is not the case, processes retransmit messages in order to resynchronize themselves. The total order is established by the sequence numbers and senders' identifiers. Messages are only delivered after becoming stable, which requires another synchronous round. The second round ensures that the system state can be restored in case of process failures by relying on consensus instances, which are also employed to reconfigure the system. THyTOB exploits the fact that processes and the underlying network behave synchronously for reasonably long periods of time to build a trivial total order among the messages, while ensuring safety under asynchrony and in the presence of process failures.

3.2.6 Discussion

The four protocols compared represent interesting classes of design choices. The first class includes two implementations of (multi-)Paxos optimized for local-area networks. Ring Paxos was designed to maximize throughput. It uses a ring topology to organize the dissemination and ordering of requests. We wanted to assess whether its ring design has implications on its latency distribution. S-Paxos is also a protocol designed for maximizing throughput. Like Ring Paxos, S-Paxos separates request dissemination from ordering but uses a classic multicast communication strategy. S-Paxos implements a fairly sophisticated request batching mechanism that can in theory contribute to keep latency tails short. Both protocols are implemented based on reliable unicast communication (TCP/IP).

Spread, the most mature of the systems tested, represents a class of protocols based on a ring where a control token circulates. The possession of the token dictates the

broadcast, retransmission, and delivery of messages. It contains a careful congestion control mechanism, which may contribute to curb latency variations. The third class of protocols is based on the notion of synchronous rounds and represented by THyTOB. Its key design assumption regards network latency predictability (i.e., synchrony) that allows round-based broadcasts to be built and their use allow a very fine control of the load applied to the network. We hypothesize that well-behaved latency distributions should result from THyTOB's capacity to condition network load. Both Spread and THyTOB implementations are based on unreliable broadcast communication (UDP/IP).

3.3 Experimental methodology

This section details the key experimental aspects of our study: the metrics used to report results, the workloads and workload generator, and the computational environment.

3.3.1 Metrics and reported results

The metric used throughout the experiments is the latency (response time) experienced by requests submitted to the atomic broadcast protocol under test. Latency variability, the result of interest, is reported using the cumulative distribution function (CDF) of the latencies measured. The CDF describes the probability (shown in the y-axis in the graphs) that a random variable, in our case the latency, has a value less than or equal to a certain value (shown in the x-axis in the graphs, in milliseconds). For example, coordinate $(x = 10, y = 0.99)$ means that with probability 0.99 (99%) latency is less than or equal to 10 milliseconds. We report probabilities and latencies in linear-linear and in log-log scales. A probability of 0.99, for example, corresponds to the distribution's 99th percentile.

3.3.2 Workload generator

To conduct our experiments, we have built a simple workload generator inspired by the software architecture of S-Paxos [15], depicted in Figure 3.1. The workload generator has two components: a client node and a replica node.

The client node can generate different workloads by varying the number of client threads. Each thread starts its execution by randomly selecting and connecting to a replica node. The replica node contacted by the client assigns a unique id to the client thread. The client thread id, together with a request sequence number, uniquely identify all requests submitted. Client threads execute in a closed loop: the thread submits a request and waits for its response before it submits another request. If a request times out, the client tries to connect to other replicas, in a round-robin fashion; as soon as it is able to connect to a different replica, the client resends its last request.

The replica node has three components: a client proxy, a connector, and a service load emulator. The client proxy is responsible for managing connections with client threads, receiving requests from them, and sending responses back to them. The *client proxy* keeps the last response computed per client thread so that responses can be retransmitted when

clients (re-)connect to a replica. The *connector* allows the connection of any of the atomic broadcast protocols to the service load emulator. Client requests are delivered to and executed by the *service load emulator*. A service load emulator can implement any function considered a legitimate load for the replicated service node. In our setup, it implements a null service, that is, a function whose only purpose is to return a fixed-size response for each request it processes. The service load emulator processes requests sequentially, in the order they are delivered to it, with corresponding responses being relayed back to the client proxy. Prior to the processing of a request, a service load emulator checks the message's sequence number to ensure a request is executed at most once.

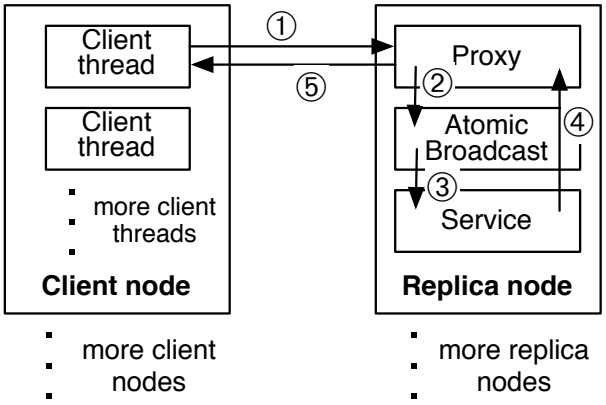

Figure 3.1: Architecture of a replicated service: (1) client submits a request to the client proxy of a replica, which (2) broadcast the request to all replicas; (3) upon delivering a client request, the service executes it, and (4) responds to the proxy, that acknowledges (5) the execution of the request to the client.

In order to submit all protocols tested to the same experimental conditions, all experiments use the same Java-based implementation of the workload generator. In the case of Spread, it is worth noting that the literature reports better performance results with a workload generator implemented in C [10]. The client proxy instantiates a pool of four threads. Each thread executes in an interruption-driven closed-loop that blocks in the absence of communication requests. The client proxies used by S-Paxos have been originally programmed so that sending events do not interrupt the thread if it has already entered its 10 millisecond sleep. As a consequence, communication between replicas and clients in S-Paxos occurs at regular intervals.

3.3.3 Workloads

We use two workloads to chase the tail distributions of atomic broadcast protocols, as described next. The workloads were generated using requests with three sizes: small (256 B), medium (2 KB) and large (32 KB).

The *peak workload* is the workload that takes the replicated service to 98–99% of its maximum throughput, for each of the protocols tested. We have determined the peak workload by observing when workload increments resulted in minor throughput variation but significant increase in response time, indicating that the atomic protocol under test

had reached its saturation point, that is, any increment in the number of threads used to generate the workload did not result in a significant increase in the throughput.

The *operational workload* is the workload under which the replicated service provides around 70% of the peak throughput, for each protocol tested. Since it was not always possible to find a number of clients that matches this percentage and ensures a fair distribution of clients per replica node, we report in the graphs the exact percentages of peak throughput obtained. The operational workload is motivated by the fact that production systems provision resources to operate within certain safety margins in order not to risk major response time fluctuations in the presence of workload peaks.

3.3.4 Computational environment

All experiments were executed on a cluster with HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory. Nodes were connected to a 1 Gbps HP ProCurve 2910al-48G switch, with round-trip time of approximately 0.1 ms.

The replicated service was implemented by five processes, each one hosted in a different node. Experiments lasted 200 seconds. Requests submitted in the first 20 seconds were disregarded when computing the latency distributions, which then refer to 3 minutes of execution. The setups for the protocols, insofar as possible kept unchanged, were the following.

We used the Java version of Ring Paxos, based entirely on unicast communication (TCP/IP) [13]. The five nodes instantiated implemented all the three Paxos roles: proposer, learner, and acceptor. The quorum size was set to 3 acceptors. Proposers implemented a batching procedure in which a batch of client requests was broadcast when either: (i) its size reached 32 KB, or (ii) no new request was received within 0.5 ms.

For S-Paxos, we adopted the settings presented in [15] for experiments with the best performance. The batch sizes for dissemination and ordering layers were set to 8700 and 50 bytes; the maximum batch delays to 5 and 10 ms. Batches were broadcast when either their size exceeded the maximum size, or the corresponding maximum delays expired. Parallelism was limited to 30 pending consensus instances at a time.

Five Spread daemons were instantiated, belonging to the same Spread segment. Each replica connected to the daemon hosted in the same node, and all replicas joined the same group. The service type for the messages broadcast was set to safe, as it ensures that messages delivered by a daemon have been received by all daemons. Based on [10], sizes of accelerated, personal and global windows were set to 15, 20, and 160.

For THyTOB, the maximum batch size was set to 32 KB and the round duration to 2.6 ms. Processes broadcast a batch with up to the maximum size at the beginning of each round. With five processes, this lead to an "optimal" (target) throughput of about 500 Mbps, which on average is the best reported in [22], for various maximum batch sizes.

3.4 Experimental evaluation

In this section, we present and analyze the results of our experiments with the peak and the operational workloads, and discuss our findings from the perspective of the different protocol designs.

3.4.1 Peak workload

Table 3.1 summarizes the performance of the four protocols when subject to the peak workload. The latency distributions at peak workload are depicted in Figure 3.2. We present results for this workload using CDFs with linear-linear (left) and log-log (right) scales. CDFs with log-log scales emphasize the latency tail, which in many cases cannot be easily distinguished in the linear-linear CDFs.

For medium and large request sizes, all protocols reached high throughput, with Ring Paxos and S-Paxos sporting performance closer to the network capacity (1 Gbps). Spread and THyTOB presented lower and also less dispersed latencies, with mean latencies always below 10 ms. For small requests, we observed more variation among the protocols.

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
		Ring Paxos	
40 clients	32 KB	11.8 ± 4.0 ms	887 Mbps
160 clients	2 KB	11.6 ± 18.8 ms	568 Mbps
1280 clients	256 B	13.1 ± 33.3 ms	198 Mbps
		S-Paxos	
75 clients	32 KB	23.8 ± 5.4 ms	825 Mbps
600 clients	2 KB	13.3 ± 2.5 ms	736 Mbps
7680 clients	256 B	27.0 ± 6.0 ms	578 Mbps
		Spread	
20 clients	32 KB	8.2 ± 1.7 ms	643 Mbps
100 clients	2 KB	4.8 ± 0.7 ms	345 Mbps
400 clients	256 B	9.2 ± 1.4 ms	89 Mbps
		THyTOB	
15 clients	32 KB	7.8 ± 0.6 ms	502 Mbps
240 clients	2 KB	7.8 ± 0.5 ms	503 Mbps
1800 clients	256 B	8.0 ± 1.2 ms	461 Mbps

Table 3.1: Performance with peak workload.

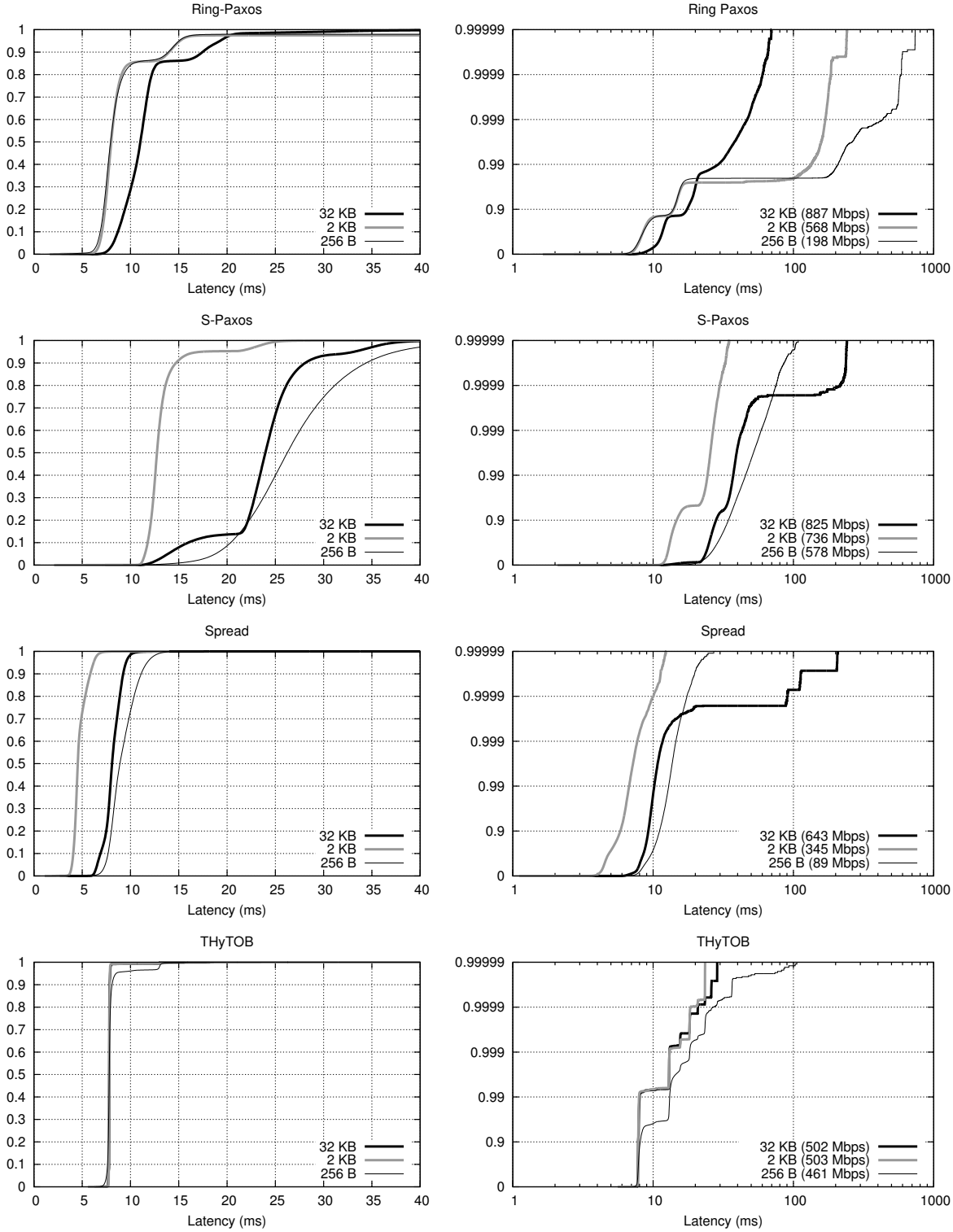

Figure 3.2: Overall latency distributions (linear-linear CDFs, on the left) and the corresponding latency tail distributions (log-log CDFs, on the right) for Ring Paxos, S-Paxos, Spread and THyTOB under peak workload.

Ring Paxos. Although not very noticeable on the overall latency distribution (CDF at the top left of Figure 3.2), Ring Paxos has the longest tails—i.e., the most skewed latency distributions—among the tested protocols, with tails becoming longer as the request sizes

get smaller. For example, from the 90th-percentile to the 99th-percentile (CDF at the top right of Figure 3.2, y-axis at 0.9 and 0.99) latency increases by more than 8 times with medium (2 KB) requests and by more than 13 times with small (256 B) requests. Large (32 KB) requests experience an increase of 67% from the 90th-percentile to the 99th-percentile. Longer tails imply, on average, latencies larger than the medians (around 8.0 ms for medium and small requests), and larger standard deviations.

S-Paxos. S-Paxos presents the higher latencies among the four protocols, and distinct tail distributions for each request size. For large requests, 99% of the latencies are from 12 and 37 ms, while more concentrated around the average, which coincided with the median. The 99.9th-percentile is about 80% bigger than the average latency, and a long tail is observed around the 99.99th-percentile, where latencies exceeded 215 ms. For medium requests, a better behavior is observed, with latencies much more concentrated: 99.99th-percentile around 30 ms, while the average was 13.3 ms. Finally, latencies for small requests are the highest and most scattered: average at 27.0 ms and 99.99th-percentile about three times higher, at 75 ms.

Spread. The shape of the tail distribution curves for Spread at peak workload resemble those for S-Paxos, although Spread presents significantly lower and less dispersed latency values. The largest tail is observed for large requests, although latency scatters only at high percentiles. For example, while from the 99.9th to the 99.99th-percentile latency increases by almost 7 times, from the 99th to the 99.9th-percentile it increases by only 15%. Both small and medium requests have smaller latency tails than large requests, with medium requests presenting the lowest average latency and standard deviation. It is worth noting, however, that in our experiments Spread saturated at considerably low loads for smaller requests. As a consequence, the throughput Spread reached for medium requests was 46% lower than for large requests; and for small requests it was 7 times lower, reaching only 89 Mbps.

THyTOB. THyTOB presented the most predictable and stable behavior among the protocols assessed, with little performance variation with different request sizes. To understand the latency distributions of THyTOB, recall that it operates in synchronous rounds, whose duration was set to 2.6 ms. Since in the best case requests needed three rounds to be delivered (two rounds for ordering and one for client-replica interaction), we can expect a lower bound of about 7.8 ms. When a round fails, due to message loss or timing faults, latencies of all ongoing requests are increased by an additional round. For medium and large requests, about 99.3% of requests fitted THyTOB's best case, and were delivered in three rounds. For small requests, 96% of requests fell in this category. We can also observe that tails become significant only at high percentiles.

3.4.2 Operational workload

We now observe the behavior of Ring-Paxos, S-Paxos, Spread, and THyTOB at the operational workload, i.e., at 70% of peak workload. We have selected this workload to assess whether the latency tails observed at peak workload are the result of the saturation of the servers, or are intrinsic to the protocols. The tail distributions (CDFs) are presented

in Figure 3.3, while the overall performance and the workload for these experiments are summarized in Table 3.4.2.

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
		Ring Paxos	
10 clients	32 KB	4.2 ± 1.7 ms	619 Mbps
120 clients	2 KB	4.9 ± 10.5 ms	400 Mbps
420 clients	256 B	6.1 ± 19.4 ms	140 Mbps
		S-Paxos	
25 clients	32 KB	11.4 ± 1.5 ms	576 Mbps
360 clients	2 KB	11.2 ± 0.5 ms	526 Mbps
3200 clients	256 B	15.3 ± 2.1 ms	428 Mbps
		Spread	
10 clients	32 KB	5.4 ± 0.5 ms	483 Mbps
30 clients	2 KB	2.0 ± 0.3 ms	241 Mbps
80 clients	256 B	2.6 ± 0.4 ms	64 Mbps
		THyTOB	
10 clients	32 KB	7.8 ± 0.3 ms	336 Mbps
170 clients	2 KB	7.8 ± 0.3 ms	357 Mbps
1280 clients	256 B	7.8 ± 0.8 ms	335 Mbps

Table 3.2: Performance with operational workload.

Ring Paxos. We note that the shape of the tail distributions at peak and operational workload are similar. The main difference, in particular for medium and small requests, is that the tails were displaced to higher percentiles: while under peak workload they affected about 3% of requests, under operational workload this percentage dropped to 0.75%. Further reducing the load leads to similar tails, although affecting even smaller portions of requests. This suggests that latency tails are intrinsic to the protocol.

S-Paxos. The most remarkable change when the load is reduced to 70% is the better behavior for large requests. The tail observed in peak performance is no longer noticeable, indicating that it is not inherent to the protocol, but due to queuing effects at high load. In addition, there is a "step" around the 98th-percentile, with a leap from about 13 ms to beyond 20 ms. This behavior results from a peculiarity of the protocol. As pointed out in Section 3.3, the client proxies of S-Paxos were originally programmed to communicate with clients periodically, every 10 ms. S-Paxos presumably assumes that most requests will be ordered within this period, which results in latencies starting at 10 ms, independently of the load. Moreover, requests for which this assumption fails, about 2% in this case, will

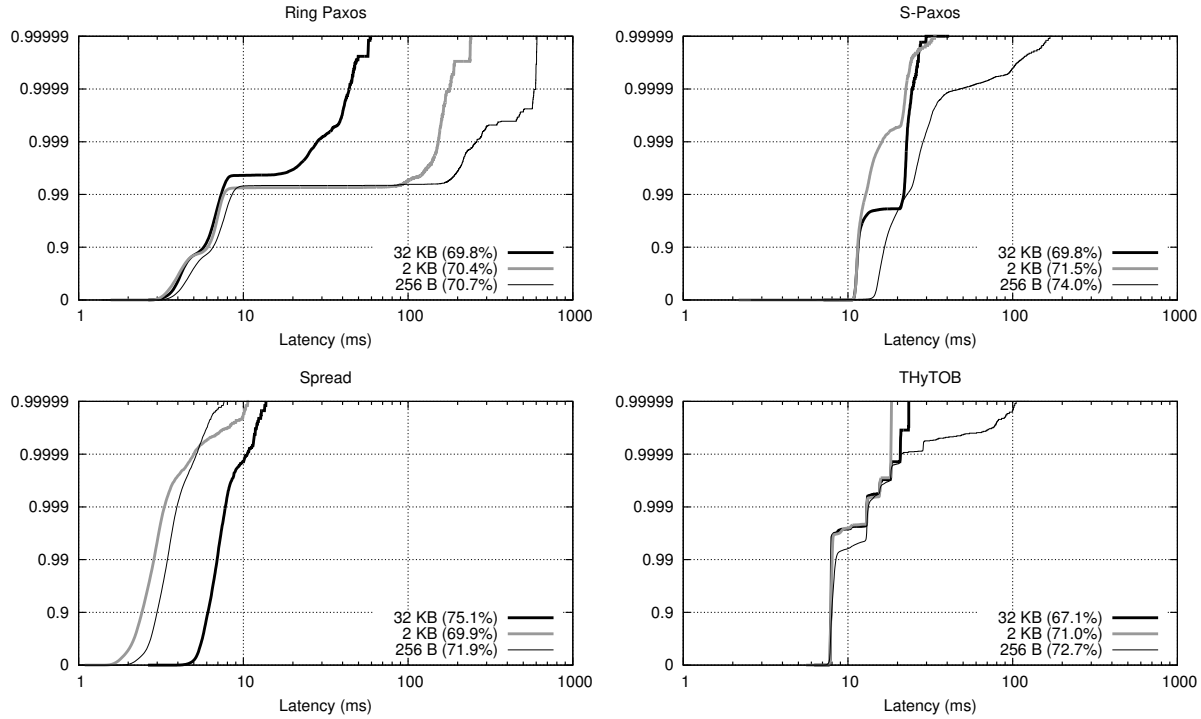

Figure 3.3: Tail distributions for Ring Paxos, S-Paxos, Spread and THyTOB at operational workload; actual percentages of peak throughput shown in parentheses.

only be replied in the next iteration of the respective client proxy threads; their latencies will then start at 20 ms. This same effect can also be observed for medium requests under peak workload (Figure 3.2, left side).

Spread. The most well-behaved latency distributions among those presented in Figure 3.3 belong to Spread. Latencies for the three request sizes did not exceed 14 ms, while the 99.99th-percentiles were 10.6 ms, 5.0 ms, and 5.2 ms for large, medium and small requests, respectively. We repute this behavior to the sophisticated flow control mechanisms adopted by Spread. They ensure a strict control of the load applied to the network, enabling the broadcast of requests to be completed, with high probability, within stipulated deadlines. However, it is worth noting that the workload applied, specially for small requests, is much lower than for the remaining protocols.

THyTOB. As for Ring Paxos, there is not significative change in the shape of the latency distribution when the load is reduced to 70%. There is an increase in the portion of requests that fit in the best case of the protocol, with latencies of three rounds: from 96% to about 99.4% for small requests, and from 99.3% to 99.8% for medium and large requests. For the latter two, in particular, latencies did not exceed nine rounds (23.4 ms). However, there is still a tail at very high percentiles for small requests. Apparently, the processing overhead induced by such small requests causes, in very few occasions, a long sequence of consecutive rounds to fail, affecting all ongoing requests.

3.4.3 Discussion

We could identify two aspects in the design of Ring Paxos that may explain the long tails observed for the protocol. First, there is no flow control in the proposers. A proposer can circulate requests in the ring at the same rate they are received from the clients. Without any flow control, proposers may overcharge their outgoing links, as they are responsible for circulating requests and protocol control messages along the ring, leading to an increase in latency variability. Second, the ring topology adopted by the protocol tightly couples the communication behavior of the processes. In effect, there is a single stream of messages circulating the ring, implying that every message has to be received, processed and forwarded by every process in the ring. When a process blocks because of background activities (i.e., garbage collection), it neither broadcasts new requests nor forwards messages received from its predecessor in the ring. Consequently, irregular delays experienced by a process are potentially propagated throughout the ring, affecting (i.e., delaying the deliver of) ongoing requests.

To verify these two hypotheses, we profiled the proposers of Ring Paxos and measured their request sending rates and delays. With increasing sending rates, average delays increase, reaching about 0.12 ms for large, 0.65 ms for medium, and 1.7 ms for small requests. The higher delays for smaller messages are due to the batching procedure, which requires more smaller messages to fill a batch. When analyzing their distributions, we could observe some abnormal sending delays, exceeding 40 ms for large, 150 ms for medium, and 500 ms for small requests. We were able to associate most of these outliers to garbage collection events, which were more frequent and had longer durations than those observed for smaller requests.

The abnormal delays measured in the proposers necessarily affect the overall request latencies, and in part explain the long tails observed for Ring Paxos. However, only these outliers alone cannot account for the magnitude and the frequency of the latency peaks measured by the clients. This lead us to suspect that other effects contribute to Ring Paxos's latency variability. In fact, when analyzing the peaks of latency as measured by the clients of different replicas, we observed that they were often correlated to each other in terms of the moment in which their executions occurred in the replicas (ring). This indicates that the occurrence of an abnormal delay at one process of the ring is actually propagated throughout the ring, a shortcoming that seems intrinsic to the topology.

The same behavior was not observed for S-Paxos, despite some similar design principles it shares with Ring Paxos: both protocols separate request dissemination from ordering to mitigate the bottleneck in the coordinator, and use TCP unicast to achieve throughput close to the network capacity. S-Paxos's behavior can be in part explained by the adoption of several flow control mechanisms, including a conservative batching policy, and limitation on the number of concurrent broadcast instances at both dissemination and ordering layers. But the main reason is probably the adoption of a fully connected communication graph topology plus majority-based mechanisms for the dissemination and ordering of requests. Since with this approach processes tend to be loosely coupled, the blocking of a single process is less likely to affect requests that were broadcast by other processes.

Another interesting aspect of S-Paxos's design is the effect of its intricate batching mechanisms. On the one hand, for medium requests they proved quite efficient in conditioning latency. Under both peak and operational performance, the 99.99th-percentiles for medium requests were approximately twice the average latencies. In particular, the mechanisms were effective in amortizing the load applied by replicas to the network. Thus, despite a throughput only 10.8% lower than that for large requests (which are not subject to batching at the dissemination layer), there was no significant effect of network contention under peak workload. On the other hand, latencies for small requests were considerably dominated by queuing delays: they were significantly higher and quite dispersed. The probable reason is the larger number of requests required to fill a batch. As a result, batches were more likely to be sent by timeout than by size. The latency variability was then caused by contention and can be seen as an overhead of batching.

Spread needs to establish a reliable communication channel among a group of processes atop an unreliable datagram service (UDP). For this end, the protocol relies on a complex flow control mechanism, orchestrated by a token that circulates a logical ring of processes. To improve performance, requests are disseminated using ip-multicast, an efficient broadcast primitive in local-area networks, but prone to message loss [76]. By restricting the number of concurrent broadcasts, Spread manages to minimize the shortcomings of this primitive. In fact, when monitoring the protocol, we noted that the message loss and retransmissions rates were negligible. This explains the good behavior of its latency distributions.

Another interesting design decision of Spread, not present in the other protocols, is to use a separate process, the Spread daemon, to handle the inter-process communication. Once the replica executes in a different process, the possible overheads of handling clients and executing requests are less likely to be propagated to the protocol, which contributes to its stability. However, the interaction between replicas and local daemons, accomplished through a TCP connection, may have curbed the performance of Spread in our experiments, specially with small requests. Thus, if on the one hand separating the protocol implementation from the replica execution can be useful in controlling latencies, on the other hand the communication between replica and daemon may limit performance.

THyTOB's key design feature is to emulate a synchronous communication service atop an asynchronous (and unreliable) broadcast network. It organizes the execution in synchronous rounds, by means of which it controls the load applied to the network. In fact, the duration of the rounds, as a function of the number of processes and maximum batch size, determines the protocol's maximum throughput and minimum latency. The longer the rounds, the lower the throughput, the higher the latency, and the lower the probability of a round to fail because of message loss or timing faults. This trade-off between achievable performance and latency predictability is at the core of the synchronous design of THyTOB. When considering that more than 99% of latencies, in particular for large and medium requests, fitted the protocol's best case, it is possible to affirm that this strategy can lead to quite controlled latencies, while providing good throughput.

3.5 Lessons learned

A remarkable aspect of the results of our experiments is the significative performance degradation—in terms of latency distribution and throughput—of the four protocols with small requests. Although affecting all protocols, two aspects allow us to distinguish two groups of protocols, in terms of performance with small requests. First, Ring Paxos and Spread presented a noticeable throughput degradation as requests got smaller, an effect that was not observed in the same proportion for S-Paxos and THyTOB. For instance, from large to medium requests, the peak throughput dropped by 46% for Spread, and by 36% for Ring Paxos; for S-Paxos it dropped by only 11%, and it remained fairly stable, almost constant, for THyTOB. Second, while the latency distribution of Ring Paxos became more scattered for medium requests, for this request size (2 KB) both S-Paxos and THyTOB presented the least latency variations.

The distinct behavior of these two groups of protocols can be explained by their *batching policies*. S-Paxos and THyTOB are protocols designed from scratch to operate on batches of requests, instead of single requests. Processes of both protocols wait for client requests and aggregate them into batches, which are then disseminated and ordered. Upon the delivery of a batch, the requests are retrieved, and then processed by the replicas. Ring Paxos and Spread do not expressly adopt batching of requests; instead, they consider alternative strategies to handle small requests. Ring Paxos is optimized for large requests; proposers try to aggregate small requests in order to send them together, but this is a "network batching", since requests are individually processed by all participants. Spread considers the size of pending requests to compute the message windows, which use "packets" as unit. Large requests may be split in several packets, while small requests can be aggregated into a single packet. Similarly, all requests are processed one by one, not in batches. In our experiments, these alternative strategies have proven ineffective to minimize the performance loss with small requests. This led us to consider the adoption of a batching policy at the replicas.

We conducted a second set of experiments to assess the performance of Ring Paxos and Spread with batched requests. To this end, we implemented a simple routine in the replicas, responsible for building batches of requests. Replicas broadcast a batch, composed by requests individually received from the clients, when either its size reaches 32 KB or the batching delay exceeds a maximum duration δ. The value of δ was computed so that if all replicas broadcast a full batch every δ units of time, the aggregate broadcast rate is equivalent to the maximum throughput of the protocol (a similar method is used in THyTOB to select the duration of rounds). For Ring Paxos, δ was set to 1.5 ms and for Spread it was set to 2.0 ms. The tail distributions for these protocols with batched requests, at peak and operational workload, are depicted in Figure 3.4, and the protocols performance is summarized in Table 3.3.

The best improvement in the overall performance when replicas batch client requests was observed in Ring Paxos. While in the original setup the latency distributions at peak throughput for the three request sizes were comparable up to around the 97th-percentile, with batched requests they are quite similar up to the 99.95th-percentile. In addition, for medium requests the peak throughput was 874 Mbps, 54% higher than in the original

Workload	*Request size*	*Latency (mean ± stdev)*	*Throughput*
Ring Paxos with batched requests			
		Peak workload	
40 clients	32 KB	11.9 ± 4.1 ms	883 Mbps
640 clients	2 KB	12.0 ± 4.0 ms	874 Mbps
2720 clients	256 B	10.6 ± 6.6 ms	521 Mbps
		Operational workload	
15 clients	32 KB	5.6 ± 2.2 ms	707 Mbps
160 clients	2 KB	4.3 ± 1.8 ms	611 Mbps
960 clients	256 B	5.5 ± 2.0 ms	356 Mbps
Spread with batched requests			
		Peak workload	
25 clients	32 KB	12.1 ± 0.9 ms	542 Mbps
320 clients	2 KB	10.3 ± 0.8 ms	509 Mbps
2560 clients	256 B	13.8 ± 1.9 ms	377 Mbps
		Operational workload	
10 clients	32 KB	5.8 ± 0.6 ms	451 Mbps
100 clients	2 KB	4.7 ± 0.8 ms	349 Mbps
960 clients	256 B	7.4 ± 1.2 ms	264 Mbps

Table 3.3: Performance with batched requests.

setup, and very close to the one achieved with large requests. Throughput also increased for small requests, by 2.6 times, while the average latency decreased by 2.5 ms. Since the performance for medium and small requests became more similar to the achieved for large requests, we can affirm that batching at replicas is more efficient to cope with smaller requests than the batching implemented at the proposers.

The reasons for this improvement were found by profiling the batching routine and, as done for the proposers, measuring the sending rates and delays. While the sending rates for small and medium requests increased, the average delays remained almost unchanged—1.6 ms and 0.6 ms, respectively—from the original experiments. However, abnormal sending delays, which in the original setup exceeded 150 ms for medium, and 500 ms for small requests, very few times exceeded 40 ms for all request sizes. The explanation for this difference is related to the number of requests a proposer has to handle, and the overhead of temporary information it stores for each of them. When batching is done at replicas, a single metadata instance is allocated for each batch of requests, instead of one instance for each client request in the original setup. Considering that metadata instances are sent together with the requests, and are potentially stored by all processes,

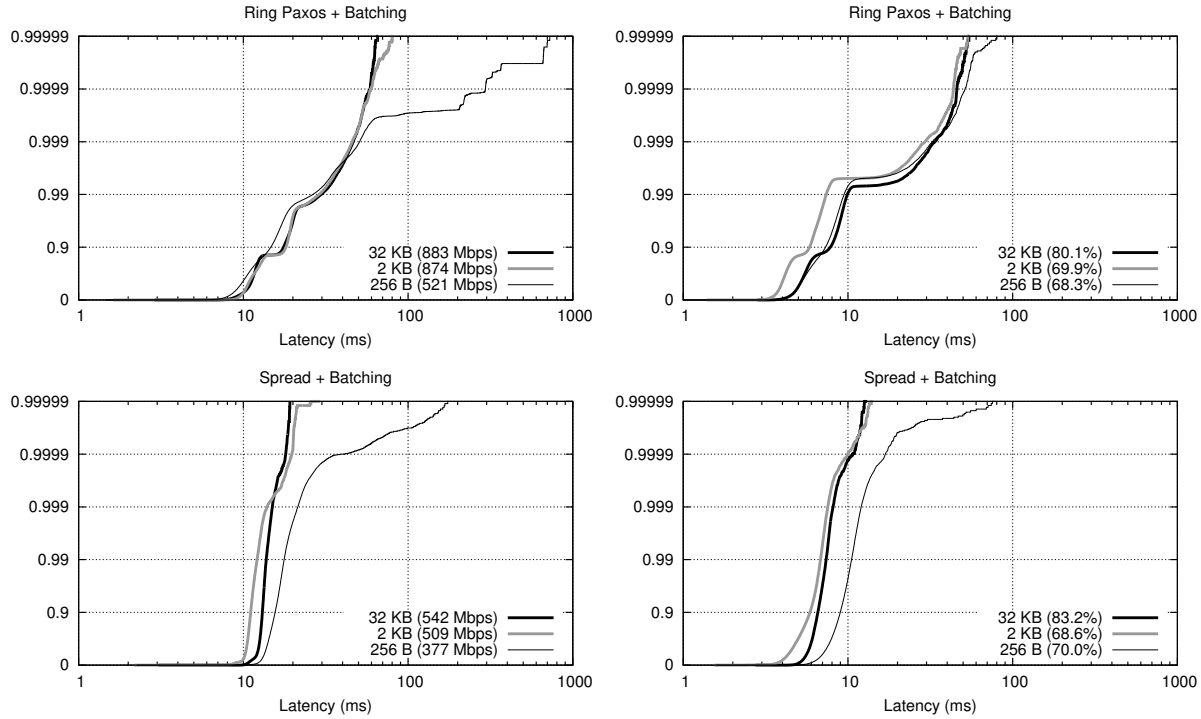

Figure 3.4: Tail distributions for Ring Paxos and Spread with batched requests under peak workload (on left) and at operational workload (on right).

this makes a big difference, specially for small requests. In fact, the time spent in garbage collection events at peak throughput for single small requests is up to 14 times higher than for batched small requests.

As a consequence, we also observed with batched requests a reduction on the range within which latencies are distributed, that is, on the length of the latency tail. While in the original setup of Ring Paxos at peak throughput the 99.9th-percentiles for medium and small requests were about 180 and 260 ms, with batched requests they were both around 30 ms. It is also true that tails at high percentiles are still observed, mainly with small requests. This possibly explains why the peak throughput for such requests is 40% lower than for the larger ones.

For Spread, the impact of batching at replicas was mostly on throughput for smaller requests. Compared to the original setup, throughput was 4.2 times higher for small, and 32% higher for medium requests. The increase in throughput was accompanied by an increase in average latency, by 4.6 ms and 5.5 ms, respectively. Latencies also increased by 3.9 ms for large requests, which caused the corresponding throughput to drop by 16%. This was the consequence of requests being broadcast only by the batching routine (a single thread) rather than by the client proxies (four threads), as done in the original setup. The increase in latency reveals how costly, in terms of delay, calls to the broadcast method provided by Spread, responsible for transferring requests to the daemon, are.

Spread saturated at different loads with batched requests, and this had some impact on the latency distributions. When taking into account the increase in the average latencies, the distributions for small and medium requests were more dispersed for batched than they were for single requests. In particular, for small requests long tails were observed

from the 99.99th-percentile. It is worth noting, however, that we are comparing here latencies from experiments that achieved different peak throughputs, and were subjected to different workloads: 400 clients with single requests versus 2560 clients with batched requests. The opposite behavior is observed for large requests, for which throughput was lower with batched requests. The latency distribution became more concentrated, and almost no tail was noticed, even at peak throughput. Thus, Spread is subject to a tradeoff involving the load it can sustain and the resulting latency distribution. The more the protocol amortizes the load applied, the better controlled latencies are, but also the lower the throughput is. Thus, mechanisms that attenuate the protocol contention, as batching does, can indeed improve throughput, at the cost of less predictable latencies.

li14tail

3.6 Related Work

The increased attention to the latency tail by companies such as Google and Amazon [37, 38] indicates the importance of end-to-end latency guarantees to distributed applications. This has raised interest in the scientific community, and several works have considered the problem (e.g., [94, 95, 96, 98]).

When a request enters a system it passes through many layers and stages, both locally on a single machine, and globally across machines. Moreover, current distributed applications are complex, composed of several services, each one operating independently. The latency a request experiences depends on each of these stages, layers, and services. Therefore, it is important that each component reduces the delays it imposes on the final response time of a request. A recent work [72] considers the effects of different layers on a request's lifetime in an individual machine, and makes suggestions to mitigate the latency tail, e.g., controlling background processes, scheduling tasks to specific threads, and using core-affinity.

In the context of atomic broadcast protocols, most early and recent work focused on maximizing throughput and minimizing latency, but not on latency variability. When the response time has been of interest, its average over periods of system operation has been considered, rather than its stability and equality across requests. An exception is Totem, Spread's atomic broadcast protocol, that had its latency distribution studied in some works [87, 88, 97]. The latency probability functions for Totem were determined analytically by [87], analysis that was experimentally verified in a subsequent work [88]. Finally, the end-to-end latency of remote methods invoked in a replicated CORBA server built atop Totem was analyzed in [97]. Our findings are consistent with the analysis presented in these studies: Spread provides controlled latency, as a result of carefully designed flow control mechanisms.

This paper expands on these previous studies as we consider the latency distribution of four different atomic broadcast protocols, including Spread's recent revamped implementation, based on the Accelerated Ring Protocol [10]. Moreover, we identify bottlenecks and propose ways to improve performance and reduce the latency tail of existing protocols.

3.7 Final remarks

Given the importance of atomic broadcast protocols to replicated services, this work offers a novel analysis of their behavior, focused on latency tail distribution instead of average latency or throughput. The results presented shed light on the design tradeoffs—topology, communication, role of synchrony, batching—made by four representative protocols to achieve a fair balance between throughput and the maintenance of short-tailed, predictable, response latency distributions. Among the design principles assessed, we have identified batching as having a key role to curb latency variability. While it is well-known that batching can boost throughput [47], it is remarkable that in some cases (i.e., Ring Paxos) it can also help shorten the latency tail of atomic broadcast.

Acknowledgments

This work was partially supported by FAPESP under grants 2011/23705-4 and 2013/21651-0, CNPq grant 473340/2009-7, and CAPES PVE grant 88881.062190/2014-01.

Chapter 4

On-time Fast Paxos

Informally, total order broadcast protocols allow processes to send messages with the guarantee that all processes eventually deliver them in the same order. In this paper, we investigate the efficiency and performance of On-Time Fast Paxos a synchronous total order broadcast protocol for the crash-recover failure model that is built atop a broadcast-based asynchronous distributed system. On-Time Fast Paxos combines an asynchronous consensus protocol with a synchronous communication protocol while guaranteeing the original safety and liveness properties of Fast Paxos. The synchronous communication protocol relies on virtual global time to create the synchronicity necessary to make Fast Paxos work at its theoretical optimum, two communication steps. Experimental results allow us to conclude that On-Time Fast Paxos performs very well in terms of both throughput, reaches 960mbps in a 1Gbps network, and latency, with an averages below 2ms. Finally, its novel hybrid design, asynchronous layer driven by a synchronous layer, shows that, time and synchrony be used to improve the performance of total order broadcasts.

4.1 Introduction

Total order broadcast is a fundamental component of modern fault-tolerant distributed systems, from single mission-critical clusters to wharehouse-scale computers, as it ensures that messages sent to a set of processes are delivered by them in the same total order. Total order broadcast is useful to enable portions of the hardware and software in these machines to work in cooperation to efficiently deliver services that affect the performance of many other key Internet applications. An elegant way to implement total order broadcast is through reduction to consensus [26]; an algorithm that allows a set of processes, some of which can fail, to choose a single value out of a set of proposed values. Fast Paxos [67] is an optimal consensus algorithm for the asynchronous crash-recover model of computation because it can solve consensus in two communication delays [68]. So Fast Paxos would theoretically be the ideal consensus algorithm upon which to implement total order broadcast. Unfortunately, in practice, Fast Paxos can only maintain optimality when the values proposed for consensus do not collide, that is, none of the proposed values obtain enough votes and therefore no value is chosen as the value of consensus. In terms of the implementation of total order broadcast, this occurs when there is no *a priori*

agreement among the processes as to the order in which messages should be delivered.

In the asynchronous model, the *a priori* order considered for the messages comes from the network, which would spontaneously tend to deliver broadcast messages in the same total order to the processes [81, 82]. This approach, however, has a number of practical limitations, which is attested by the lack of total order broadcast implementations based on Fast Paxos. In this paper we propose an alternative way to generate a *a priori* order for the messages from the use of *temporal order*, and use it to implement a synchronous variation of Fast Paxos, that we call On-Time Fast Paxos. To this end, we augment the asynchronous model with a *virtual global clock*, whose periodically generated clock ticks dictate the execution of the algorithm. Processes then seek agreement on the time, and not necessarily on the order, in which broadcast messages are received. The result is an original, very fast and latency conditioned total order broadcast algorithm.

The remaining of the paper is structured as follows. Section 4.2 contextualizes our research and points up our contribution to the area of total order broadcast protocols. Section 4.3 defines the model of computation assumed throughout the paper. The key algorithmic aspects of our On-Time Fast Paxos are covered from Section 4.4 to Section 4.7. Section 4.4 contains a summary of the original Fast Paxos; it has been included in the paper to make the explanation of our version of the algorithm self-contained. Section 4.5 defines On-Time Fast Paxos using a top-down approach, from the consensus layer down to the synchronous communication layer. Section 4.6 details the operation of the synchronous communication layer. Section 4.7 provides experimental evidence that the design and implementation of On-Time Fast Paxos represents a significative advance in the state of the art of total order broadcast solutions. Section 4.8 concludes the paper.

4.2 Total Order Broadcast and Consensus

Chandra and Toueg [26] have shown that total order broadcast and consensus are equivalent by building a total order broadcast algorithm that relies on repeated executions of an asynchronous consensus algorithm. The algorithm starts with the assignment of a unique identifier to each message or, more generally, each set of messages to be broadcast. The pair unique identifier and set of messages is then proposed using any consensus algorithm. The reduction uses multiple independent, and possibly concurrent, instances of consensus; each instance chooses a single set of messages, among the possibly various sets of messages proposed to receive a given identifier. The output of the total order broadcast algorithm is formed by the set of messages decided in successive instances of consensus, taken in the order established by their identifiers. The delivery order is thus determined by the identifiers chosen for the messages; messages with the same identifier are delivered in some pre-agreed deterministic order—considering, for instance, the message ids.

The critical step in the reduction of total order broadcast to consensus is the assignment of unique identifiers to messages. This step can be implemented in a centralized or in a distributed manner. The usual way to generate identifiers is to rely on a distinguished process, namely a sequencer, that intercepts all messages broadcast and tags them with a unique identifier. This process accumulates the role of coordinator of the consensus al-

gorithm, being responsible for selecting the value to propose in each instance. Consensus is therefore used to ratify the ordering for the messages generated by a sequencer, which is only replaced in case of failure. Examples of consensus algorithms that have been used to build this class of total order broadcasts are Chandra-Toueg's [26] and Paxos [64].

An alternative way to generate total order is to allow each process involved in the algorithm to propose identifiers for the messages. Identifiers are autonomously generated, and pairs set of messages and assigned identifier are proposed for consensus. Total order broadcasts opting for the distributed generation of identifiers do not require a sequencer because they tipically assume the existence of an *a priori* order for the messages. This means that, under favorable conditions, processes are likely to propose orders (assignments of identifiers to messages) that are identical, or at least not conflicting. Consensus in these circumstances becomes trivial [19], and is merely used to verify that the orders generated by each process are compatible, and only occasionally to solve conflicts. Examples of this approach are total order broadcasts based on prefix agreement [8], ordering oracles [82], and optimistic algorithms [81, 99], among them Fast Paxos.

The quest for algorithms that rely on a distributed procedure to generate total order is justified by the potentially lower communication cost provided by them, when compared with centralized algorithms. In fact, by eliminating the sequencer, the cost of total order broadcast is reduced to two communication delays, which is the best-case lower bound for consensus [28, 68]. Observe that centralized algorithms—such as Paxos or, equivalently, Fast Paxos operating in classic mode—can provide optimal latency for broadcasts initiated by the coordinator; a third communication delay however is necessary for messages broadcast by ordinary processes. In algorithms adopting a distributed approach, such as Fast Paxos, optimal latency can be achieved for broadcasts initiated by any process; possible exceptions occur when the *a priori* order assumption is not verified.

The nonobservance of the assumption that identifiers generated independently by the processes are order-compatible implies the proposal of conflicting values for consensus. Although conflicts are tolerated because safety is never violated, this situation is undesirable in terms of performance because liveness is hampered. For instance, when different identifiers are proposed for a message it is possible that either multiple or, worse, no identifiers are chosen for it. In this case, the resolution of conflicts may cost additional consensus instances to ensure that in the end an identifier is chosen for every message broadcast. Conversely, when the same identifier is assigned to distinct sets of messages, the corresponding instance of consensus is potentially affected as it may not admit a trivial solution; again additional communication may be required to ensure a successful completion of consensus. This scenario, common to all optimistic total order broadcasts is called *a collision* in Fast Paxos. Collisions thus denote failures to comply with the *a priori* order hypothesis, and constitute, at least in theory, the main adversary of total order broadcast implementations atop Fast Paxos.

The asynchronous model of computation does not inherenly allows the existence of an *a priori* order for messages broadcast concurrently in a distributed systems. However, in practice, it is often the case that messages broadcast, especially in local-area networks, are received in total order with high probability. This property, known as *spontaneous total order* [81, 82] or, in a different context, consistent delivery order [58, 60], is at the core of

asynchronous total order broadcasts atop Fast Paxos. By relying on its validity, processes tag messages so as to simply represent their reception order, proposing them in successive instances of consensus. More specifically, processes interpret each broadcast message as if it were the value proposed for the next, still unused instance of Fast Paxos [66]. This straightforward correspondence between messages and consensus values establishes an optimal consensus pace that is effective as long as the processes, specifically those with the role of acceptors in Fast Paxos, receive the same messages in the same total order.

When the spontaneous total order is not observed, either because part of the processes fail to receive some messages, or because messages are received by processes in different orders, the conflicting values proposed for Fast Paxos may cause collisions. At the cost of additional communication delays, Fast Paxos can solve collisions, and decide which messages to assign each identifier. However, the optimal consensus pace may take some time to be restored because the straightforward correspondence has been disrupted by the collision and may further trigger collisions, negatively impacting performance. For example, when processes receive the same messages in permuted orders, a whole range of identifiers may be affected by collisions, even if most of such messages are received in the same order. Another situation occurs when a given message is received only by a part of the processes: the sequence of messages, indexed by identifiers, observed by processes that received such messages, and those did not, becomes displaced by one position. Thus, practical implementations of total order broadcast atop Fast Paxos are very sensitive to frequent violations of spontaneous total order. For example, violations tend to be frequent when the processes that broadcast messages are the same that are responsible for ordering them, making implementations of Fast Paxos practically unfeasible.

In this paper we propose an alternative way to generate total order from the use of *temporal order* and its use to implement a synchronous variation of Fast Paxos, namely On-Time Fast Paxos. To this end, we augment the asynchronous model with a virtual global clock, and use the values returned by this clock as identifiers for the messages. Order is then determined by the reception time of the messages as defined by a global clock, and not necessarily by the order with which they are received by the processes.

4.3 Model of Computation

Our model of computation assumes an asynchronous distributed system augmented with global time.

4.3.1 Distributed System

A distributed system is a set of n processes $\mathcal{S} = \{P_1, \ldots, P_n\}$ that exchange messages via a network. An *event* is an occurrence of a process. Events can be internal or external. Internal events only influence the state of the process that has executed it. External events can be the sending or receiving of messages, and time events. $\mathcal{S}$ adheres to the crash-recover model of distributed computation supplemented with a virtual global clock. Processes fail by crashing, and may later recover. Before crashing, processes behave as instructed by the protocol that specify their correct behavior. Processes communicate

by unicast or broadcast message exchange, that is, we assume that there is a network primitive that allows a process to send a message to all process of $\mathcal{S}$ in a single operation. Communication channels are unreliable, so both unicast and broadcast messages can be lost, duplicated, reordered, or arbitrarly delayed, but can not be corrupted.

4.3.2 Virtual Global Time

The measurement of time is essentially a process of counting. Any repeating phenomena, natural or artificial, can be used to count. The occurrences of the repeating phenomena are events; events take no time. The time passed between two consecutive events is a period of time or a duration. A clock is a device which is based on some periodic oscillation (phenomena) that can be used to measure the progression of time. Each oscillation of the clock mechanism generates an event called a *clock tick*. The granularity of a clock is the period between two consecutive clock ticks. The granularity of a given clock, say C, can only be measured if there is another clock, say a reference clock RC, available with a finer granularity. In this case, RC can be used to implement C.

In our model, every process of $\mathcal{S}$ has access to local physical clock. The local clocks are not equal, but it is supposed that they differ at most by the clock precision Π, and that they have granularity g. Local clocks use the network to exchange timekeeping information that allows them to synchronize with each other. The clock synchronization protocol governing the ensemble of local clocks is not synchronized to an external clock. Moreover, as $\mathcal{S}$ is asynchronous, no assumption can be made about the communication or process latencies, but nevertheless, it is assumed that during *good* conditions of execution synchronization is possible. This means that for all local clocks, the differences between their readings is smaller than a given ϵ, and the variability for interprocess message delays is such that it is possible to establish the maximum error in the estimation of the delay taken for a round-trip message exchange among the processes of $\mathcal{S}$. These assumptions are sufficient [62, 51] to allow the implementation of a virtual global clock.

The virtual global clock is used (i) to define the order of events and (ii) to coordinate processes so that they can execute an action at the same approximate real time instant. It implements an abstraction of time represented by an strictly monotonic increasing integer. The virtual global clock can be queried to obtain the current time and it can be advanced. Its granularity Δ is the period of oscillation of the clock, that is, the time period elapsed between two consecutive virtual global clock ticks. Processes can register with the virtual global clock to receive, in the form of software interruptions, the clock ticks it generates. The integer label of a clock tick is the time of its occurrence, so clock tick i is the event (instant) that marks the beginning of the time period (or time granule) i.

4.4 Fast Paxos in a Nutshell

This section presents a brief overview of Fast Paxos, a consensus algorithm; a complete description can be found in [67]. The consensus problem demands a set of processes to choose a single value. Fast Paxos is easier to explain in terms of three sets of agents that play complementary roles in the resolution of consensus: *proposers* are agents that propose

values, *acceptors* ensure that a single value is chosen, and *learners* learn the value that has been chosen. Agents are implemented by processes, a single process can implement multiple agents playing different roles.

The consensus algorithm proceeds in rounds, aka as *ballots*, in which values are put to a vote. Each ballot is owned by a process, the coordinator of a ballot, and can be either a fast ballot or a classic ballot. Ballots are identified by positive integers, the ballot numbers. A ballot number identifies the coordinator of a ballot, and indicates whether the ballot is fast or classic. Ballots are not necessarily started in numerical order, some ballot numbers can be skipped, and multiple ballots may be executed concurrently. Each ballot progresses through two phases, with the following steps:

- a coordinator starts a ballot by choosing a ballot number b, larger than any ballot number it is aware of, and sending b to all acceptors in a PHASE 1A message;
- an acceptor only reacts to the reception of a ballot b PHASE 1A message if it has not participated of any ballot $b' \geq b$—otherwise, it ignores the request. The acceptor confirms its participation in the ballot b by sending a PHASE 1B message to the coordinator. If the acceptor participated in the Phase 2 of a previous ballot, it attaches to the PHASE 1B message the last vote it has cast: the highest ballot number b^* in which the acceptor voted, and the value v it voted for in ballot b^*;
- Phase 2 of is started when the coordinator receives PHASE 1B messages from a quorum of acceptors. It is a requirement to not have yet started Phase 2 of ballot b, nor started any ballot $b' > b$. The action taken by the coordinator depends on whether acceptors reported to have cast votes in previous ballots. If they have, the coordinator is forced to propose in Phase 2 one of the values reported in Phase 1. To propose a value v the coordinator sends v in a PHASE 2A message; the rule for selecting a value for Phase 2 is described in the following. If no acceptor reported to have cast votes in previous ballots, the coordinator is free to propose any value in Phase 2. From this point, the operation is different for classic and fast ballots. If b is a classic ballot, the coordinator uses the PHASE 2A message to propose any value received from the proposers. If b is a fast ballot, this role is transfered to the acceptors, by sending to them a special Phase 2 message called ANY message.
- the reception of the ANY message enables an acceptor to operate in fast mode in ballot b. The acceptor then treats a value v received from a proposer as if it was a PHASE 2A message from the coordinator proposing v for ballot b. The operation is the same as for PHASE 2A messages of classic ballots, as described below.
- upon receiving a ballot b PHASE 2A message an acceptor verifies whether it can participate in Phase 2 of that ballot: if it has not participated of any ballot $b' > b$, nor cast votes in any ballot $b' \geq b$. If so, the acceptor casts a vote for the value v carried by the PHASE 2A message. The acceptance of v in ballot b is registered, and informed to the learners by sending a PHASE 2B message to them.

A value v is decided when a quorum of acceptors vote for v in the Phase 2 of a ballot. This situation is identified by the learners, which monitor the `Phase 2B` messages sent by

acceptors. Once a value is decided in a ballot, the algorithm ensures that no other value can be proposed by the coordinator of any subsequent ballot, as detailed below. Moreover, it is assumed agents remember the actions they performed. In particular, coordinators have to remember the last ballot they started, and acceptors the last ballot they have taken part in and the last vote they have cast. Since Fast Paxos agents may crash and recover, this information must be stored in stable memory.

Fast Paxos ensure that ballots do not choose different values. This guarantee relies on the role for selecting values for Phase 2 and on requirements for the intersection of quorums. For selecting the Phase 2 value we consider the simplified rule for Fast Paxos proposed by [90]. As in the original rule [67], the coordinator ranks the received votes by ballot number and selects the values associated to the highest ballot b^* reported. If the selected values contain a single element, which should always happen when b^* is a classic ballot, this value must be proposed again. Otherwise, if multiple values were selected, b^* is necessarly a fast ballot and the value most often present among the selected ones must be proposed.

Quorums are minimal sets of processes whose participation in a ballot is a condition for it to proceed. Each type of ballot has a set of quorums associated with it, classic quorums for classic ballots and fast quorums for fast ballots. Quorums are used to guarantee that if a value was or might ever be chosen in a ballot, it is going to be reported on the Phase 1 of any successive ballot. The requirements for that are [67]: (i) any two quorums must have non-empty intersection, and (ii) any two fast quorums and any classic or fast quorum have a non-empty intersection. An optimal classic quorum can then contain a simple majority of processes, i.e. $\lfloor N/2 \rfloor + 1$ processes in a system with N processes. In this case, the corresponding fast quorum must contain $\lceil 3N/4 \rceil$ processes. For an optimal fast quorum, fast and classic quorums must have the same size of $\lfloor 2N/3 \rfloor + 1$ processes [90].

The option for fast or classic ballots is made by the coordinator, based on its perception of the state of the algorithm. If the coordinator believes that no value has been yet proposed, it starts a fast ballot. If it was indeed the case, no previous votes will be reported by the acceptors, which are then instructed to operate in fast mode. This is the expected behavior for Fast Paxos, and allows proposed values to be chosen, without the intervention of the coordinator, in two communication steps. Classic ballots are reserved for ballots where the coordinator suspects that an active ballot may not succeed in choosing a value. In this scenario, acceptors are expected to report previous votes, from which the coordinator selects the value it proposes for consensus. The algorithm operates in recovery mode, and can benefit from the potentially smaller quorums required by classic ballots.

Ballots may not succeed for several reasons, such as the failure of their coordinators, message loss, or the concurrence of higher-numbered ballots. To detect these situations, often indistinguishable from each other, processes monitor the various algorithm messages and resort to failure detectors. To deal with these situations that tend to hamper progress, different recovery mechanisms can be implemented, but ultimately the intervention of a coordinator may be necessary to start a new ballot. To avoid competition, the role of starting ballots is restricted to a single coordinator, elected among all agents, and replaced only when it crashes. Fast Paxos is safe in presence of multiple coordinators, so correctness

is not compromised by mistakes of the election mechanism. However, the eventual election of a correct process as the only active coordinator is a progress requirement.

In addition to the reasons described below, fast ballots may also not succeed due to *collisions*. A collision occurs when acceptors accept, in the Phase 2 of a fast ballot, values from different proposers, so that no value is accepted by a quorum of acceptors. From the learners' point of view, the votes are distributed among several values, and none of them can reach enough votes to get chosen. To solve a collision it is necessary to start a new ballot, incurring the cost of operating in recovery mode in a failure-free scenario. Thus, altough some optimizations are possible, recovery from collisions is very costly to the algorithm. In fact, depending on how often collisions happen, there is can be no advantage operating in fast mode.

The procedure presented above refers to the operation of a single *instance* of consensus. Fast Paxos supports the execution of multiple consensus instances, identified by positive integers. Instances are independent, in terms of values proposed and chosen in each of them, but share the same set of agents and the same coordinator. The coordinator, once elected, selects a fast ballot number and tries to activate it in *all* consensus instances. This is done by sending a single `Phase 1A` message to all acceptors. At this initial moment, each acceptor only informs the coordinator the last (highest-numbered) consensus instance in which it has participated. Based on the reports of a quorum of acceptors, the coordinador identifies the last instance in which values have been proposed, the last active instance. All subsequent instances are inactive, which means that no value can have been chosen for them. The coordinator then sends a `Any` message enabling acceptors to operate in fast mode for all instances above the last active one. Next, acceptors report their previous votes for the active instances, which are handled individually by the coordinator. Hopefully, most of those instances will have already been decided, so that Phase 1 needs only to be executed for few of them.

4.5 On-Time Fast Paxos

On-Time Fast Paxos is a total order broadcast algorithm that combines two protocols: a synchronous communication protocol and an asynchronous consensus protocol. The synchronous communication protocol is responsible for (i) the on-time dissemination of messages, and (ii) the use of a virtual global clock to generate timestamps that establish a timeline for the broadcast messages and are used to uniquely identify them. The unique identifiers assigned to the broadcast messages and their timeline are then used to establish a total order. Subsequently, this total order is ratifyed by the asynchronous consensus protocol, essentially an implementation of Fast Paxos, which main function is to render fault tolerance to the synchronous communication protocol.

Akin to other total order broadcasts based on (Fast) Paxos, in our algorithm clients first interact with proposers that are, in their turn, responsible for initiating broadcasts. Delivery of totally ordered messages is the responsibility of learners, with acceptors playing the role of building and rendering persistent the total order. In a nutshell, On-Time Fast Paxos differs from other existing asynchronous total order broadcasts based on Fast

Paxos by its use of a synchronous communication protocol to condition the behavior of the consensus agents. The critical aspect of our design is determined by how the agents of the asynchronous consensus protocol interact with the components of the synchronous communication protocol to materialize our total order broadcast algorithm. In the next sections we detail these interactions, starting with a description of the procedure required to activate On-Time Fast Paxos, and proceeding through the normal operation and the exceptional operation: collision handling, lost messages, and flow control.

4.5.1 Activation

As we have seen in section 4.4, Fast Paxos uniquely identifies instances of consensus by positive integers, and their operation in fast mode must be enabled by the coordinator. Fast Paxos has been designed for the asynchronous model, On-Time Fast Paxos honors the original asynchronous design of Fast Paxos while making their agents work on-time, that is, synchronously. It is the responsibility of the synchronous communication protocol to offer mechanisms that can be used by the consensus protocol to achieve total order on time. So the first problem the design of On-Time Fast Paxos has to solve is how to activate the protocol, that is, how to bootstrap a correspondence between Fast Paxos instance identifiers and the identifiers used by the synchronous communication protocol. On-Time Fast Paxos manages a mapping between the timestamps generated by the virtual global clock and instances of consensus. The map is straightforward: virtual global clock timestamps are used to uniquely identify instances of Fast Paxos. The second problem to be solved is how to enable the concerted, paced, operation of the agents of Fast Paxos; this is achieved by synchronously delivering sets of messages to the agents.

The activation of On-Time Fast Paxos is initiated by its coordinator, which as soon as elected selects a fast ballot number b and sends it to the acceptors using a special Phase 1a message that does not refer to any particular instance of consensus. If b is a high enough ballot number, each acceptor informs the coordinator the last (higher-numbered) instance of consensus for which it has cast a vote. Upon receiving replies from a quorum of acceptors, the coordinator can establish the activation point [93]: the first (lower-numbered) instance of consensus i from which no value may have been decided in ballots previous to b. It then enables the fast operation mode starting from instance i by sending a ballot b Any message to all processes. When received by a proposer, the Any message instructs it to send broadcast messages directly to the acceptors, through the synchronous communication protocol. For an acceptor, in its turn, it enables the acceptance of sets of proposer's messages delivered by the synchronous communication protocol as values proposed for instances of consensus. Acceptance of consensus instances is enabled for instances greater than or equal to i, as indicated by the Any message, or, equivalently, for broadcast messages timestamped from i onwards. Virtual clock timestamps and instances of consensus are placed in a one-to-one correspondence by advancing the virtual global clock to a future time equal to the value established by the consensus activation procedure as the point of activation.

4.5.2 Normal Operation

During the normal operation On-Time Fast Paxos proceeds through the following steps. To broadcast a message m, a proposer uses the synchronous communication protocol to send m to all destinations. Along with m, the proposer sends its process identifier (*pid*) and a counter, whose concatenation form the message identifier. The synchronous layer timestamps every message it transports with the current global time. At the receivers, this is especially important for the acceptors, messages received through the synchronous layer are grouped according to their timestamps. Let i be the timestamp of the messages in the set M_i, received from clock tick i to immediately before clock tick $i + 1$; at the clock tick $i + 1$, the pair $\langle i, M_i \rangle$ is delivered to the consensus protocol.

An acceptor interprets the pair $\langle i, M_i \rangle$ received from the synchronous protocol as if it was a PHASE 2A message that proposes M_i as the value for the instance i of Fast Paxos; the message is assumed to refer to the last (high-numbered) fast ballot b activated by the coordinator for instance i. During normal operation, it is expected that no ballot $b' > b$ has been started, and no value has been accepted in the instance i—in particular, due to the mapping of consensus instances to the timestamps produced by the virtual global clock that are monotonically increasing. The acceptor then endorses the assignment of the unique identifier i to the set of messages M_i, proposed by the synchronous layer, by casting a vote and sending it to the learners in a ballot b PHASE 2B message.

Learners and the coordinator of Past Paxos also receive the proposer's messages that are broadcast using the synchronous communication layer. This allows the PHASE 2B messages sent by acceptors to carry only the message's ids, instead of the full payload. Upon receiving identical PHASE 2B messages from a quorum of acceptors, and learning the ids of the proposer's messages decided for that instance, the learner retrieves from the synchronous layer the corresponding full messages (payloads). The set of proposer's messages retrieved, taken in a pre-agreed order of their ids, form a sequence of messages; the sequences of messages formed for every instance of consensus, taken in the natural order of their identifiers, define the total order for the messages. Observe that learners only keep a cache of proposer's messages, which are permanently stored by the acceptors, and can, if necessary, be retransmitted by them, as detailed in section 4.5.5.

4.5.3 Collisions

When acceptors disagree on the set of messages to which a given timestamp is affixed, the corresponding instance of consensus may end in a collision [67]. The situation can be detected by the coordinator, that by also playing the role of learner listens to PHASE 2B messages. Upon detecting a collision, the coordinator starts a classic ballot to enforce the decision of a single set of messages chosen from the distinct sets of messages proposed in the conflicting fast ballot. Assume the collision occurs in a fast ballot b owned by the current coordinator, which also owns the subsequent classic ballot $b + 1$. When this is the case, PHASE 2B messages of ballot b can be interpreted as if they were the votes reported by the acceptors in PHASE 1B messages of ballot $b+1$. This allows the starting of the collision-recovery ballot $b + 1$ from Phase 2, thus saving the two communication steps required for Phase 1: a procedure known as coordinated recovery [67].

Recall that acceptors' PHASE 2B messages do not carry full proposer's messages, but only message's ids. The same applies to PHASE 2A messages sent by the coordinator in collision-recovery ballots. The point is that acceptors may even send only references to their votes in PHASE 2B messages, but can not accept PHASE 2A messages referencing proposer's messages they have not yet received. The protocol must then intercept each PHASE 2A message, extract the set of message's ids proposed therein, and retrieve the corresponding proposer's messages, so that it can offer a set of full messages to the acceptor. Hopefully, in most cases the acceptor has already received the referenced proposer's messages—even if associated with different timestamps. Otherwise, the retransmission of the missing proposer's messages is requested, and their reception becomes a condition for the acceptor to process PHASE 2A message. Thus, in addition to the intervention of a coordinator in the consensus layer, collisions may require the exchange of proposer's messages in order to resynchronize the processes in the communication layer.

4.5.4 Tempo

From the activation onwards the tempo of the consensus algorithm, that is, the pace at which new instances of consensus are created, are determined by the clock ticks of the virtual global clock. Optimistically, once initiated most instances will end with no collisions and no need for retransmissions, in a single communication delay. Some instances may take longer, requiring extra communication steps for the algorithm to reach a decision, or for the synchronous exchange of the proposer's messages. On average it is expected that all agents of the consensus algorithm work at approximately the same tempo, implying that learners, acceptors, and proposers are subject to stable workloads. The funcion of the tempo mechanism is to ensure maximize the periods of the agent's stable work.

Instances that take too long to complete represent probably the main cause of tempo variability because of unsuccessful ballots or lost messages, that, in their turn, are caused by periods of asynchronicity of the distributed system. Late instances form *gaps* in the sequence of completed consensus instances. The existence of a gap delays the delivery of already totally ordered messages that succeed the gap until it is closed. Delivery delays clearly penalize the latency of those messages, while blocking the end of the corresponding total order broadcasts and subsequent release of the resources associated with them. In summary, instances that take too long to complete generate a backlog of tasks that propagate work back along the agents—learners do not deliver messages, acceptors have to retransmit messages and try to advance via asynchronous Paxos ballots, proposers pace may further overload acceptors—in a domino effect with clear negative impact on performance. It is impossible to guarantee workload and latency stability in an asynchronous distributed system. So, to try to cope with unfavourable setups on-time fast paxos resorts to lowering the proposer's tempo.

We can not prevent atypical delays from affecting instances of consensus, but we can restrict the start of new instances in order to restrain the backlog.

This is achieved by controlling the generation of ticks by the instances of the global clock—which are in part operated by software. Initially, consider that upon being activated an instance of the global clock is enabled to generate a given number of clock

ticks. When the number of clock ticks an instance was enabled to generate is reached, the instance is disabled. This means that in the processes governed by that clock instance ticks are no longer generated, and so new instances are no longer initiated. In order to remain in operation, a clock instance should be opportunely *wound*, that is, enabled to generate an additional number of clock ticks.

4.5.5 Message Loss

The synchronous communication layer connects all processes and allows proposers to send messages directly to all destinations, both acceptors and learners. Proposer's messages carry application messages, possibly multiple of them, and can therefore be particularly large: if Phase 2 messages had to include several proposer's messages, this step would easily become a bottleneck. By sending only message's ids in Phase 2 messages, however, it becomes possible for a process to receive a message's id from the consensus layer but not the corresponding full proposer's message from the synchronous layer. This requires a simple protocol for retrieving proposer's messages from message's ids, which possibly includes asking for retransmissions from other processes.

The primary source of messages circulating on the synchronous communication layer are the proposers. However, due to their possibly transient participation in the protocol they are not good candidates for retransmitting any messages possibly lost during the original broadcast. Acceptors are better suited to the function of message retransmission and storage as Fast Paxos requires the reception of messages by at least a quorum of acceptors. So, in our solution missing messages are handled by acceptors. Any process can request the retransmission of a set of proposer's messages by sending their ids to all acceptors. If the requested messages were, or might have been, decided in a consensus instance, then necessarily a quorum of acceptors has already received them, and at least one non-failed acceptor will be able to fulfill the retransmission request.

Observed that in regular operation, several acceptors will be able to retransmit each requested proposer's message. To avoid the retransmission of multiple copies of the same message, the space of message's ids is partitioned among acceptors, and each acceptor only retransmits messages with ids in a certain range. In addition, retransmissions are sent to all processes, and an acceptor does not retransmit a message if has already done so recently. On the other side, if retransmissions are not received after a maximum delay, the request is resent with increased counter. The request counter shifts message's ids so that to address different acceptors. This avoids situations in which the target acceptor for a given id is irresponsive, or has not received the corresponding message.

4.6 Synchronous Communication Protocol

In this section we detail the operation of an essential component of On-Time Fast Paxos: the synchronous communication protocol. As referred in Section 4.5 it is responsible for the on time dissemination of broadcast messages. This role comprises: (i) the transport of messages, whose broadcast is initiated by proposers, to their destinations, acceptors and learners; and (ii) the assignment of timestamps to the broadcast messages, which in

turn determine their delivery order. This is a question of establishing a preliminary order for the broadcast messages, which is then submitted to Fast Paxos for ratification.

The synchronous communication protocol thus combines the transport of broadcast messages with timing functions. Its operation is governed by a virtual global clock, that periodically generates clock ticks at every process. A simple implementation of a virtual global clock, that does not rely on specialized hardware or real-time software, is presented in Section 4.6.1. Section 4.6.2 then progressively construct a synchronous dissemination protocol, governed by the virtual global clock, for the broadcast of messages.

4.6.1 Virtual Global Clock

Modern computers keep track of the passage of time using a battery-operated CMOS clock circuit driven by a quartz resonator. This simple physical clock does not have the precision or accuracy for real-time processing, but is sufficient to implement a fairly accurate virtual global clock [51]. The resonators of the local clocks are not equal, but it is supposed that they differ at most by the clock precision Π. Furthermore, the granularity g of the virtual global clock is defined as its period of oscillation, expressed as a multiple of the frequency of oscillation generated by the resonators of the local clocks.

A simple way to implement a virtual global clock is to have a set of time daemons, one per process, organized in a master-slave hierarchy. One of the time daemons is elected to play the role of *synchronizer*, the process responsible for the implementation of the virtual global clock. The synchronizer has granularity Δ, meaning that a synchronization message m is broadcast to every process of the system every Δ unities of its local clock. Every synchronization message carries a timestamp $\mathsf{m.t}$, a positive integer incremented by the synchronizer at each broadcast. It represents a discrete instant of the global time base the virtual global clock is supposed to implement. Note that the synchronizer can be activated, enabling the broadcast of a given number of synchronization messages, its timestamp can be advanced to given value, or can be deactivated if necessary.

The slave time daemons listen to the synchronization messages periodically broascast by the synchronizer. At every message reception a slave, say s compares its clock value $\mathsf{s.c}$ with the timestamp $\mathsf{m.t}$ received from the synchronizer. If $\mathsf{m.t} > \mathsf{s.c}$ then clock tick $\mathsf{m.t}$ is generated and the clock value is updated $\mathsf{s.c} \leftarrow \mathsf{m.t}$. Otherwise the message is just discarded and the value of the global clock at that slave remains unchanged. Thus, the virtual global clock synchronization protocol does not fully implement any distributed time synchronization algorithm, it implements a mechanism that is sufficient to generate a timeline where global clock ticks, represented as integers, are about Δ apart.

The virtual global clock is fault tolerant The virtual clock global tolerates the failure of the synchronizer which is, in this case, replaced by a new synchronizer elected among the remaining correct slaves. The detection of the synchronizer's failure and the subsequent election of the new synchronizer is carried out using a mechanism similar to the one implemented in Ω failure detector [71]. In the case of the virtual global clock, the synchronization messages serve an additional purpose, they double as *heartbeat* messages of an Ω failure detector. Piggybacked onto the synchronization messages is the following

information: an epoch number and the process identifier of the current synchronizer.

Any slave daemon that suspects that the master daemon may have failed becomes a candidate and tries to elect itself as the new synchronizer. The candidate generates a new epoch number and takes the role of synchronizer using as the initial timestamp of the new epoch the current value of its own clock. In the presence of concurrent candidates to the role of synchronizer, the candidate with the largest epoch number is elected. The remaining candidates are demoted to slaves as soon as they receive a synchronization message containing an epoch number larger than their own. Time daemons conform to a synchronization message from a different epoch provided it is greater than the current epoch; and ignore synchronization messages with epochs smaller than the current epoch. As a consequence, the virtual global clock eventually converges toward a state where a single synchronizer exists and all the remaining daemons behave as slaves.

Precision and Accuracy of the Global Clock The quality of the virtual global clock obtained by this approach, centered around the figure of a synchronizer, is determined by three factors related to the synchronism presented by the system. First, the oscillation period of the physical clock read by the synchronizer, that defines the granularity of the virtual global clock. This factor is measured by the drift rate of the synchronizer's clock when compared to a reference clock or to the other processes' local clocks. Second, the accuracy with which the synchronizer broadcasts synchronization messages in relation to the instants of its physical clock scheduled for such. Together with the clock drift rate, the accuracy of the synchronizer is decisive for the broadcast of synchronization messages to occur at intervals close to the desired granularity Δ. Third, the variability of latencies for the delivery of synchronization messages to every process. This variability ultimately defines the precision of the virtual global clock, that is, the duration of the real-time interval during which different processes generate the same clock tick.

We will not delve into the analysis and experimental evaluation of the three factors mentioned, in particular because we already presented in a previous work [22]. For now, it is enough to say that the performance achieved by On-Time Fast Paxos is considerably due to the accuracy and precision of a synchronizer-based virtual global clock.

4.6.2 Synchronous Message Dissemination

Having presented a simple protocol to implement the virtual global clock, we now discuss how the on-time dissemination of messages occur, and how it is related to the assignment of timestamps to the broadcast messages. Recall that the virtual global clock mechanism is responsible for the periodic generation of clock ticks at every process. Clock ticks are ideally generated with periods close to the configured granularity Δ, and all non-crashed processes are expected to generate the same timeline. The global virtual clock therefore allows to induce in the processes the lock-step operation of a synchronous system.

Considering first processes that are destinations of the broacast messages, acceptors in particular, the lock-step operation consists on the interleaving of message receiving and message processing phases. Messages received are not immediately processed but buffered until the next clock tick, when all buffered messages are processed together. As a result

of this processing phase relative, to say, time instant i the synchronous communication protocol delivers a pair $\langle i, M_i \rangle$, where M_i is *a priori* the set of messages received from clock tick i to immediately before clock tick $i + 1$, as discussed in Section 4.5.2. Observe that so far we have made no assumptions about the behavior of proposers.

Asynchronous and synchronous proposers On the side of processes that initiate broadcasts, the proposers, there are two possible modes of operation. The original Fast Paxos proposers are asynchronous, in the sense that whenever they receive a broadcast request, the message is immediately submitted to the acceptors. Proposers are allowed then to access the network at any time, without any external form of contention. This is the case, for example, when proposers are considered clients, external to the total order algorithm. An alternative approach is, since the system is equiped with a global clock, to make proposers synchronous. A synchronous proposer does not immediately submits a message whose broadcast is requested by a client; instead, the message is enqueued to be submitted, together with other messages in the same situation, at the next clock tick. The broadcast of messages is then conditioned by the pace dictated by the virtual global clock, so that each proposer can broacast a single batch of messages per clock tick.

An experimental argument to support the adoption of synchronous proposers is presented in Section 4.7.2. A more comprehensive argument concerns the ability to control the load applied by proposers to the network through the broadcast of messages. In fact, as proposers are synchronous, it is possible to limit the network usage by restricting the number of bytes each proposer is allowed to broadcast at every clock tick. It is assumed that the number of concurrent proposers is known, and so the granularity of the global clock, allowing to accurately predict the maximum load applied to the network. As this could be a very restrictive policy, we relax it by defining an *optimal batch size* with which proposers try to comply when considering the average message size per clock tick.

A synchronous transport Assuming that proposers have access to the virtual global clock, how could they participate more actively on the generation of the total order? An alternative would be to transfer the role of message ordering to the proposers, which can assign the timestamp of each message at the time of its broadcast. A proposer can then read its clock value at the broadcast time and add to it a number κ of time units that depends on the clock granularity Δ and the broadcast message size. An acceptor a that receives a message m from a proposer, compares the current value of its clock $\mathsf{a.c}$ with the message timestamp $\mathsf{m.t}$. If $\mathsf{m.t} < \mathsf{a.c}$ the message was early received, and its delivery is postponed until instant $\mathsf{m.t}$. Otherwise, if $\mathsf{m.t} > \mathsf{a.c}$ the message was received late, and for the sake of synchronous principles must be discarded. This is actually the major limitation of the proposer-oriented ordering approach, when applied to a system that is not stricly synchronous. If a proposer chooses κ wrongly, in particular a very low value, a broadcast message may end up being discarded by most of the acceptors.

We opted for a solution that represents a compromise between the proposer-oriented and the acceptor-oriented ordering approaches. Proposers do assign timestamps for the messages they broadcast but acceptors do not discard messages received late, since we do not assume strict upper bounds for message transmission. The timestamps assigned have

then the role of establishing a minimum latency κ for the delivery of broadcast messages. Early messages have their delivery postponed to the instant designated by the proposer, while the other messages are assigned, as final timestamp, the instant at which they are received by the acceptors. As detailed in Section 4.7.4, the adoption of this approach allowed a significant reduction in the rate of collisions and a consequent improvement on the performance of On-Time Fast Paxos. The main reason for the improvement is the reduction of time uncertainty as to the global time instant at which different acceptors receive certain messages that are likely to be received around a clock tick—before for some, after for others.

The approach based on a minimum latency parameter allows the deployment of a process that colocates two Paxos agents usually never placed together in Fast Paxos: a proposer and an acceptor. This in only possible because our synchronous approach defines a minimum latency for the reception and processing of the messages processed by the acceptor.

4.7 Experimental Evaluation

This section contains the experimental evaluation of On-Time Fast Paxos. First, it specifies the cluster of computers and network used in the experiments. Second, it shows that synchronous broadcasts can be used to uplift the performance of protocols in contrast with asynchronous broadcasts. Third, it presents and discusses a progression of total order broadcast protocol designs that lead to On-Time Fast Paxos. All designs are discussed in the light of the experimental results.

4.7.1 Computing Environment

The experiments reported in this Section were performed in a cluster with Supermicro X8DTL machines equipped with two quad-core Intel Xeon E5620 processors running at 2.4 GHz, and with 12GB of main memory. Machines ran a 64-bit Gnu/Linux Debian 8.0, and were connected to a 3Com 4200G Gigabit Ethernet switch, with round-trip time of approximately 0.2ms. The algorithms were implemented in Java atop of the Treplica toolkit [91], and experiments were carried out using the OpenJDK 1.7 Server VM.

4.7.2 Asynchronous versus Synchronous Broadcast

In order to place asynchronous broadcast in contrast with synchronous broadcast, the following experiment has been devised. A set of processes, in the role of clients of the ip-multicast network (Ethernet), broadcast 8kB messages to generate increasingly larger workloads (mbps) with the goal of using the full bandwidth of the network (1Gbps). The total workload grows linearly with each process being responsible for the generation of an equal fraction of it. Identical workloads—the throughput, measured in messages or bits per second—are generated by two different means: asynchronously and synchronously. The asynchronous workload is generated by allowing processes to execute their broadcasts

at any time. The synchronous workload is generated by requiring processes to broadcast messages periodically at the same time, with the support of the virtual global clock.

Figure 4.1 Asynchronous (left) shows what happens to message loss as a function of the workload. A single process, purple line, is able to reach full bandwidth usage without any message loss. Message loss reaches approximately 0.10% for a workload of 650 mbps (10,000 8kB-messages/s) generated by two concurrent processes. If four and eight broadcasters are used to generate the workload, then message loss reaches, respectively, 0.28% and 0.36%. For a workload of 750 mbps message loss grows by almost an order of magnitude at every scale; the loss rates for two, four, and eigh broadcasters are, respectively, 1.0% (10 times higher), 2.2% (8 times higher), and 2.5% (7 times higher). As the workload grows, the asymptotic behavior of message loss becomes evident, it grows exponentially. It is worth observing that message loss, even at small rates, have a negative impact on the performance of any broadcast-based protocol, more specifically of Fast Paxos, because they imply message retransmissions, and message retransmissions imply extra work by the protocol. The message loss effect observed in Figure 4.1 Asynchronous has previously been been observed and used as a motivation to develop a variant of Paxos [76].

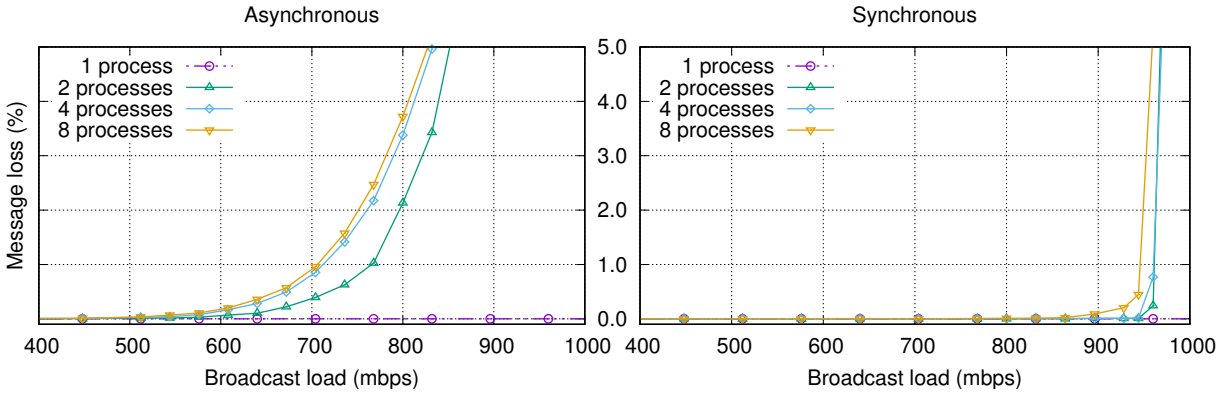

Figure 4.1: Rates of message loss measured when asynchronous (left) and synchronous (right) processes ip-multicast 8kB messages in a Ethernet-based local area network.

By coordinating the processes and restricting the ip-multicast of messages exact the same instants of time, the results are considerably improved. Figure 4.1 Synchronous shows that message loss remains below 0.10% up to an workload of approximately 900 mbps for scales 2, 4, and 8 processes. If a single process (purple line) is used to generate the workload then, obviously, there is no communication contention and the full 1Gpbs bandwidth is used. As the workload reaches approximately 960 mbps, message loss rates becomes relevant. In the graph it is clearly noticeable that an inflection point has been reached for the configurations of broadcasters with 2, 4, and 8 processes within a very small workload interval, from aproximately 945 mbps to 960 mbps, message losses depart from 0.0% to reach 0.24%, 0.77%, and 5.4%, respectively. From this point on, message loss increases rapidly. The lesson learned from the comparison of the message losses for the asynchronous and synchronous workloads is clear: synchronous broadcasts make better use of the bandwidth available. In fact, as detailed next, when proposers are synchronous,

On-Time Fast Paxos can sustain throughputs up to 930 mbps.

4.7.3 Fast Paxos: Synchronous Siblings

In the next section we analyse the performance of four total order broadcasts based on Fast Paxos in order to evaluate the impact of

different designs of the communication substrate. The four implementations thus share the same asynchronous consensus protocol

but differ in the way broadcast messages are disseminated, and mapped to instances of consensus. The first implementation corresponds to the traditional, asynchronous way of obtaining total order atop Fast Paxos, where acceptors allocate messages to successive instances of consensus, so as to represent their reception order. The other three are implementations of our On-Time Fast Paxos, in which broadcast messages are mapped to the instance of consensus corresponding to the instant of global time at which they are received. In the three implementations acceptors operate atop the synchronous communication protocol, which proposes at each clock tick a (possibly empty) set of messages for consensus. They differ then by the way proposers operate to initiate the broadcast of messages.

	Agent Synchrony	
Protocol	Proposer	Acceptor
Original Fast Paxos (FP)	asynchronous	asynchronous
Timed Acceptor Fast Paxos (TA)	asynchronous	synchronous
Timed Acceptor-Proposer Fast Paxos (TP)	synchronous	synchronous
On-Time Fast Paxos (OT)	synchronous	κ-synchronous

Table 4.1: Synchrony and Fast Paxos

In the first synchronous protocol based on Fast Paxos, that we henceforth call Timed Acceptor Fast Paxos (TA), proposers operate asynchronously, that is, in the same way as in the Original Fast Paxos (FP). Namely, proposers are asynchronous broadcasters that can ip-multicast messages at any time. This approach has advantage of being more generic, since proposers do not necessarily need to be part of the protocol, but also disadvantages in terms of the reliability of communication, as presented in Section 4.7.2. In the Timed Acceptor-Proposer Fast Paxos (TP) protocol, in turn, proposers are also synchronous (see Section 4.6.2), with operation dictated by the virtual global clock. So at each clock tick proposers may ip-multicast a batch of messages which broadcast was requested during the inter-ticks interval, if any. Finally, the On-Time Fast Paxos (OT) protocol is the most synchronous of the three, as proposers not only are synchronous broadcasters but also participate in the mapping of messages to instances of consensus. As detailed in Section 4.6.2, proposers may stipulate a minimum latency κ for the delivery of the messages they broadcast, so that the temporal information provided by the virtual global clock not only to pace their operation, but also to generate ordering.

4.7.4 Results and Discussion

In the experiments presented in this Section a fixed setup for the agents was used for the four protocols. The number of acceptors was set to four, the least replication factor for which Fast Paxos tolerates failures. The size of fast and classic quorums were then set to three acceptors, so that to tolerate the failure of one acceptor. The coordinator was allocated to a fifth process, which in the case of the timed protocols accumulate the role of synchronizer (the master of the virtual global clock protocol, see Section 4.6.1). The workload is generated by other four processes, which play the roles of proposers and learners. Having proposer and learner co-allocated enables a more accurate measurement of latencies, since the broadcast and delivery of messages occur in the same process. A client thread in each of these processes requests the broadcast of 1 kB random messages to the associated proposer at exponentially distributed intervals. The experiments had duration of 60 seconds, and the data collected in the first 30 seconds, when the broadcast rate is progressively increased up to the target rate, is disregarded. In the experiments in which proposers are synchronous broadcasters, the optimal batch size (see Section 4.6.2) was set to 14 kB. Finally, in the experiments with timed agents the granularity of the virtual global clock was set to $\Delta = 500\mu s$, that is, 2000 clock ticks per second.

The summary of the data collected during the experiments is depicted in Figures 4.2, 4.3, and 4.4. The x-axis of the graphs present the average throughput measured for each configuration (essentially the target broadcast rate) by the learners. Each point depicted in the graphs is generated from the data of at least 5 executions of the experiment with a given configuration. The graphs present relevant aspects of total order broadcasts based on Fast Paxos. Figure 4.2 summarizes the performance of the four implementations, under the form of the average latency as a function of the achieved throughput. In Figure 4.3 we depict the rate of collisions, relative to the number of consensus instances initiated, measured by the coordinator. As we observed two very different behaviors, with low rates less than 0.5% and high rates that can exceed 80%, we use logarithmic scale on the y-axis. Figure 4.4 depicts the average CPU usage in the acceptors, relative to the capacity of a single core (machines have 8 cores), as a function of the rate of messages (or bits) they are able to order per time unit. It is a relevant metric when we consider, in particular, that performance can be severely hampered by an excessive CPU usage.

The first relevant aspect observed in Figure 4.2 is that all three timed protocols based on Fast Paxos achieve high throughput. On-Time Fast Paxos slightly exceeds 930 mbps, the TP protocol hits 930 mbps, and the TA protocol can reach 850 mpbs of throughput, with corresponding average latencies in the range 3.0 to 3.5 ms. Original Fast Paxos, in turn, can only maintain a reasonable latency behavior up to about 265 mpbs, when the average latency increases very rapidly. It is also true that under low load, up to 200 mbps Original Fast Paxos presents latencies substantially lower than the timed protocols: with no contention, FP latency is tipically below 0.4 ms; with timed acceptors, TA and TP have latencies up to 3 times higher (around 2Δ); finally, with a minimum latency $\kappa = 2$ On-Time Fast Paxos has latency about 4 times higher than FP, slightly above 3Δ. It is interesting to note that, at least up to 300 mbps, TA and TP protocols have very similar latencies. Recall that TP differs from TA because it conditions not only acceptors but also

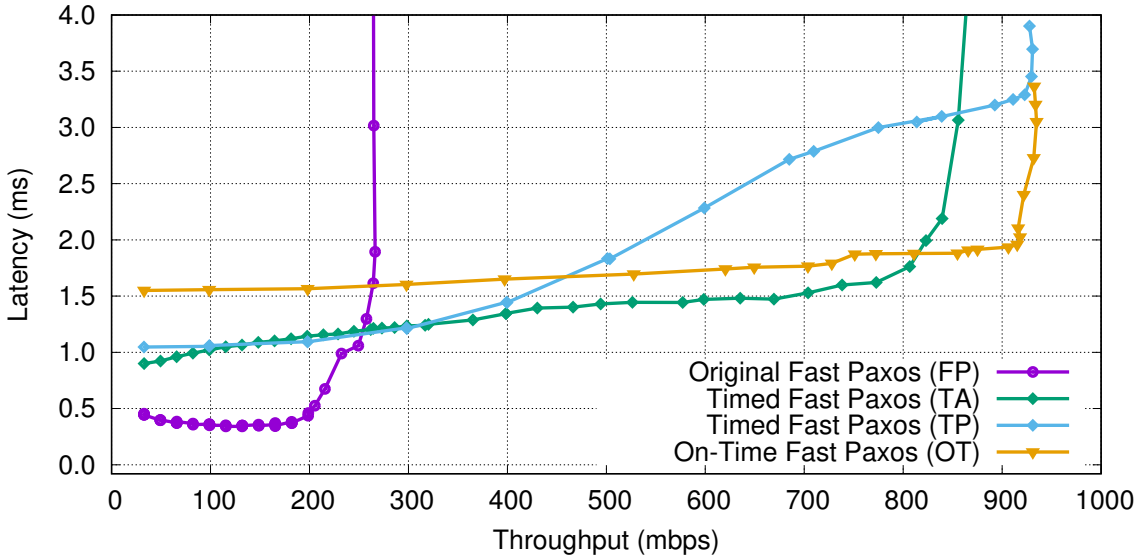

Figure 4.2: Performance of the four total order broadcasts based on Fast Paxos with 4 proposers and 4 acceptors, in the form of average latency in function of throughput.

proposers; so, in theory, the latency of TP (expected to be around 2Δ) would be higher than that of TA (with average expected to be up to 1.5Δ). This is not the case and the probable reason is the higher collision rates of TA when compared with TP, especially at these low loads, where the difference is more than one order of magnitude.

To better understand what happens to the timed protocols TA and TP in the central range of load, between about 300 mpbs and 800 mpbs, is interesting to move to Figure 4.3. Note, first, that the execution of the two protocols, TA in particular, are dominated by collisions: at 300 mpbs of load the collision rate for TA was about 57% and 7% for TP; at 700 mbps, rates reach their picks of 88% for TA and 64% for TP. In the case of TA protocol, the very high collision rates is a result of the high rate of relative small messages the proposers broadcast: between 19 and 48 per clock tick ($500\mu s$), on average. Just one such message received at different clock ticks by proposers is enough to cause, not only one but probably two, collisions. As TP protocol restricts broadcasts to discrete instants of time, the number of messages per clock tick can not exceed the number of proposers, but the average size of the batches of messages broadcast increases with increasing load. The point is that depending on the broadcast message size, the network load, and the number of concurrent proposers, the expected latency may be close to multiples of the clock granularity Δ. This triggers a scenario where part of the acceptors receives a given message just before a clock tick, while others a little later the same clock tick. It is a special situation in which minor latency variations may trigger collisions.

The adoption in On-Time Fast Paxos of a minimum latency for broadcast messages, related to the clock granularity, stems from the observation of this situation in the TP protocol. As a result, collisions affect less than 0.2% of the instances of consensus in the OT protocol up to 915 mbps; an impressive improvement from a simple modification of the communication protocol. However, it is worth noting that despite the significant collision rates observed for the TA and TP protocols, they manage to achieve high throughput,

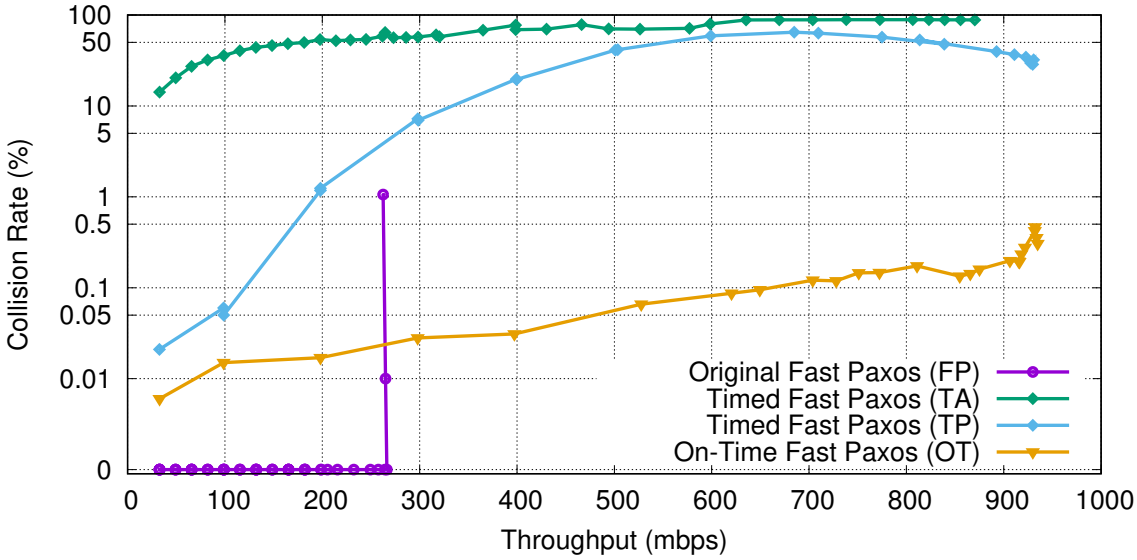

Figure 4.3: The rate of collisions, relative to the number of instances of consensus started, in log scale, as a function of the throughput achieved with 4 proposers and 4 acceptors.

while providing increasing but still fairly reasonable latencies. This is possible because, despite the broadcast load, the number of instances of consensus started per time unit remains unchanged: one per clock tick. Thus, there is time to recover instances affected by collisions, without actually compromising the effectiveness of newly started instances. But as observed in Figure 4.2, there is an important impact on latency as a result of such collision rates, in particular for TP protocol due to its contention mechanisms.

Protocol	Throughput	Av. Latency	Collisions Rate	CPU usage
FP	198.72 mpbs	0.44 ms	0.000%	222%
TA	806.55 mpbs	1.76 ms	88.8%	189%
TP	910.98 mbps	3.25 ms	36.7%	134%
OT	915.17 mpbs	1.96 ms	0.195%	120%

Table 4.2: Performance for configurations with maximum throughput/latency ratio.

We summarize the performance of the four protocols assessed in Table 4.2, by selecting the configuration with the best trade-off between throughput and average latency for each of them. In this critical configuration, the ratio throughput/latency is maximal, meaning that subsequent increases in throughput come with the cost of a considerable increase in latency. Examples of this behavior are observed for Original Fast Paxos (FP), considered in the last configuration with latency below 0.5 ms, and for the TA protocol, considered before the inflection of latencies noticed from throughput 800 mbps. It is also relevant the very high throughput achieved by On-Time Fast Paxos, of about 91% of the network capacity, while sporting a reasonably low latency, below 2.0 ms. The TP protocol reaches similar throughput but with 66% higher latency, which is nevertheless impressive given the cost of dealing with collisons in 36% of the instances of consensus executed.

Figure 4.4 allows us to make an argument for the relatively poor performance observed

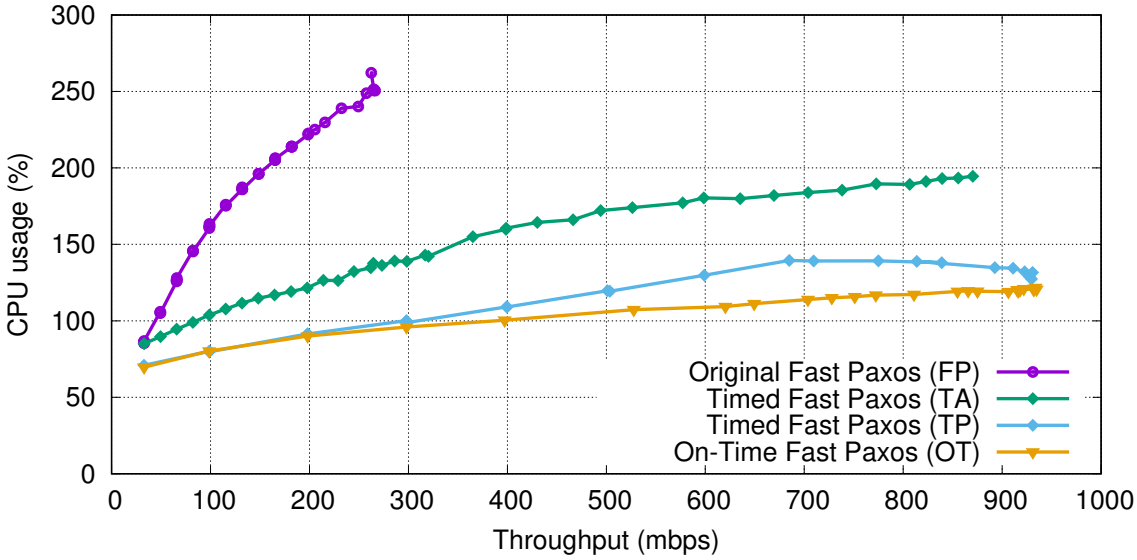

Figure 4.4: Average CPU usage relative to a single core measured for 4 acceptors as a function of the rate of messages (bits) from 4 proposers they can order per time unit.

for the Original Fast Paxos. Observe that out of the four protocols, FP has from the outset a high CPU usage, and it grows very fast (slope of purple curve) as the workload is increased. Figure 4.3 can be used in conjunction with Figure 4.4 to observe that there is a point after which the collision rates jump to very high values. What happens is that the treatment of collisions requires extra CPU usage, that in its turn, increases the chances of new collisions, as the acceptors become increasingly overloaded, thus more asynchronous. At the limit, the cascading effect of increased CPU usage coupled with increased collision rates stalls the acceptors as they become very busy trying to process the ever increasing backlog of pending consensus instances and the reception and ordering of new proposals.

4.8 Conclusion

This paper has presented On-Time Fast Paxos, a synchronous total order broadcast that results from the principled application of time and synchrony to design an algorithm that paces the operation of Fast Paxos, an asynchronous consensus protocol for the crash-recover failure model. The paper has briefly contextualized the use of consensus to implement total order broadcast and summarized the Fast Paxos algorithm. On-Time Fast Paxos has been described using a top-down approach. The topmost layer of On-Time Fast Paxos consists of a Fast Paxos modified to follow the tempo of timed broadcasts generated by the synchronous communication protocol; the lowest layer of On-Time Fast Paxos. The core of the protocol lies in the mechanisms for the activation, operation, tempo maintenance, and message loss that had to be designed and implemented to allow the on-time generation of total order broadcasts using Fast Paxos. Also key to the design of the protocol is the use of a virtual global clock and synchronous communication layer capable of dynamically adapting themselves to the requirements of the timely execution of consensus instances.

The experimental design created to assess the characteristics of On-Time Fast Paxos considers two sets of experimentos. The first set has been designed to compare the performance of asynchronous versus synchronous ip-multicasts (broadcasts) over an Ethernet network; probably the most common local area network used in the world. The results show that message loss is much better behaved when processes make synchronous broadcasts. This is key to our argument that conditioned synchronous broadcasts can be used to implement fast performing total order broadcasts. The second set of experiments assess the behavior of On-Time Fast Paxos through the following metrics: throughput (mbps) versus latency (ms), throughput (mbps) versus collision rates (% of message loss), and throughput (mbps) versus CPU usage (% of a single processor). Four variations of total order broadcast protocols are assessed: the Original Fast Paxos (FP), a Timed Acceptor Fast Paxos (TA), a Timed Acceptor-Proposer Fast Paxos (TP), and On-Time Fast Paxos (OT). What sets these versions of total order broadcast apart is the use of time and synchrony in ther implementations, FP is asynchronous, then application of synchrony grows in the following order: TA, TP, and OT. In summary, what the results show is that OT has the best throughput (915mbps), smallest collision rate (0.195%) and smallest CPU usage (less then half of the CPU usage of FP, our benchmark protocol), with a latency similar to the raw round-trip latency of the network, approximately 2ms. The figures obtained allow us to conclude On-Time Fast Paxos is indeed very efficient vindicating the use of time and synchrony in the design of total order broadcasts.

Chapter 5

Conclusion

Usually, total order broadcasts have been designed under worst-case models of computation, that is, assuming full asynchrony and a bounded number of crash, omission or arbitrary failures. The thesis of this work is that a hybrid model of computation can be used to engineer fast total order broadcast protocols: asynchrony underpins safety, synchrony supports progress. The trail of evidences for the thesis has been laid out in three papers. The first paper investigates the design and performance of a very simple synchronous total order broadcast that is built atop of an asynchronous distributed system based on a broadcast network. Our Time Hybrid Total Order Broadcast (THyTOB) explores the inherent synchrony of the broadcast network to build a total order for the messages, while ensuring safety under asynchrony—modeled as performance failures, with unbounded frequency—and in the presence of crash-stop failures. The second paper of the trail compares four different total order broadcast protocols with two main goals in mind: (i) characterizing their design and implementation decisions in the light of their latency variability (i.e., latency tail distribution), and (ii) finding mechanisms that can be used to reduce the latency variability of these protocols, among them synchrony. The third and final paper of the trail investigates the efficiency and performance of On-Time Fast Paxos, a synchronous total order broadcast protocol for the crash-recover failure model that is built atop an asynchronous distributed system. On-Time Fast Paxos combines a synchronous communication protocol with an asynchronous consensus protocol, while guaranteeing the original safety and liveness properties of Fast Paxos. The synchronous communication protocol relies on a virtual global clock to generate the synchronicity necessary to make Fast Paxos work at its theoretical optimum, of two communication steps. On-Time Fast Paxos's hybrid design therefore shows that time and synchrony can fundamentally be used to improve the performance of total order broadcasts.

In conclusion, the set of results presented in this doctoral dissertation form a coherent exploration of the role of synchrony in the engineering of total order broadcasts derived from (Fast) Paxos, allowing us to defend the thesis that synchrony plays an intrinsic, fundamental, aspect in the construction of fast total order broadcast protocols. This gives rise to a number of open questions, that deserve to be subject of future research. For instance, are the achieved results extensible to more heterogeneous environments, such as wide area networks and cloud infrastructures? Or what is the impact of adopting specialized hardware and software, with stricter real time guarantees, in the performance of the synchronous total order broadcast protocols proposed in this thesis?

Bibliography

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. Rtcast: lightweight multicast for real-time process groups. In *Proceedings Real-Time Technology and Applications*, pages 250–259, Jun 1996.

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

[3] 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.

[4] Guillermo A. Alvarez, Flaviu Cristian, and Shivakant Mishra. Asynchronous atomic broadcast. *Distributed System Engineering Journal*, 33, 1991.

[5] 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.

[6] Yair Amir and Jonathan Kirsch. Paxos for system builders. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*, New York, September 2008.

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

[8] Emmanuelle Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FTCS '97)*, pages 292–301, June 1997.

[9] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[10] Amy Babay. The accelerated ring protocol: Ordered multicast for modern data center. Master's thesis, The Johns Hopkins University, Baltimore, Maryland, USA, May 2014.

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

[12] Ziv Bar-Joseph, Idit Keidar, and Nancy Lynch. Early-delivery dynamic atomic broadcast. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC '02)*, volume 2508 of *Lecture Notes in Computer Science (LNCS)*, pages 1–16, Toulouse, France, October 2002. Springer Berlin Heidelberg.

[13] Samuel Benz. Unicast Multi-Ring Paxos. Master's thesis, Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland, June 2013.

[14] Piotr Berman and Anupam A. Bharali. Quick atomic broadcast. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG '93)*, volume 725 of *Lecture Notes in Computer Science (LNCS)*, pages 189–203, Lausanne, Switzerland, September 1993. Springer Berlin Heidelberg.

[15] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS '12)*, pages 111–120, Irvine, California, USA, October 2012.

[16] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles - SOSP'87*, pages 123–138, New York, USA, 1987. ACM Press.

[17] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

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

[19] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technologies (PaCT '01)*, pages 42–50, London, UK, September 2001. Springer-Verlag.

[20] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[21] Luiz Eduardo Buzato, Gustavo Maciel Dias Vieira, and Willy Zwaenepoel. Dynamic content web applications: Crash, failover, and recovery analysis. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)*, pages 229–238, Estoril, Lisbon, Portugal, June 2009.

[22] Daniel Cason and Luiz Eduardo Buzato. Time hybrid total order broadcast: Exploiting the inherent synchrony of broadcast networks. *Journal of Parallel and Distributed Computing*, 77(0):26–40, March 2015.

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

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

[25] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 322–330, New York, NY, USA, 1996. ACM.

[26] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

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

[28] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *Journal of Algorithms*, 51(1):15–37, April 2004.

[29] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71, April 2009.

[30] Ching-Tsun Chou, Israel Cidon, Inder S. Gopal, and Shmuel Zaks. Synchronizing asynchronous bounded delay networks. *IEEE Transactions on Communications*, 38(2):144–147, February 1990.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, August 2013.

[32] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[33] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, April 1995.

[34] 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.

[35] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[36] Flaviu Cristian and Shivakant Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems (ISADS '95)*, pages 215–221, 1995.

[37] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[38] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, 2007.

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

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

[41] Carole Delporte-Gallet and Hugues Fauconnier. Real-time fault-tolerant atomic broadcast. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, pages 48–55. IEEE Computer Society, October 1999.

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

[43] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*, pages 343–354, Atlanta, Georgia, June 2014. IEEE Computer Society.

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

[45] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International Conference on Fundamentals of Computation Theory*, volume 158, pages 127–140, Borgholm, Sweden, August 1983. Springer-Verlag.

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

[47] Toy Friedman and Robbert Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 233–242, Washington, DC, USA, August 1997. IEEE Computer Society.

[48] Ajei Gopal and Sam Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, volume 392, pages 110–123. Springer Berlin, Heidelberg, Nice, France, September 1989.

[49] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, July 2010.

[50] Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *The Computer Journal*, 50(1):53–67, January 2007.

[51] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by TEMPO in berkeley UNIX 4.3BSD. *IEEE Transactions of Software Engineering*, 15(7):847–853, July 1989.

[52] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. A practically constant-time MPI broadcast algorithm for large-scale infiniband clusters with multicast. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, June 2007.

[53] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, April 2007.

[54] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN '11)*, pages 245–256, Hong Kong, CH, June 2011.

[55] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the Amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 436–447, Los Alamitos, CA, USA, May 1996. IEEE Computer Society.

[56] 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, June 2001.

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

[58] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[59] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *Micro, IEEE*, 9(1):25 –40, feb 1989.

[60] Hermann Kopetz, G. Grünsteidl, and J. Reisinger. *Dependable Computing for Critical Applications*, chapter Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System, pages 411–429. Springer Vienna, 1991.

[61] Hermann Kopetz and G. Grünsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524 –533, jun 1993.

[62] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, C-36(8):933 –940, aug. 1987.

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

[64] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[65] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[66] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[67] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[68] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, June 2006.

[69] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[70] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[71] Mikel Larrea and Cristian Martín. Quiescent leader election in crash-recovery systems. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, pages 325–330, Shanghai, China, November 2009. IEEE Computer Society.

[72] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 9:1–9:14, Seattle, WA, USA, 2014.

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

[74] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI '04)*, page 8. USENIX Association, December 2004.

[75] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The performance of Paxos in the cloud. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*, pages 41–50, Nara, Japan, October 2014. IEEE Computer Society.

[76] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems Networks (DSN '10)*, pages 527–536, Chicago, USA, June 2010.

[77] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[78] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234, April 1980.

[79] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, volume 1499, chapter Lecture Notes in Computer Science (LNCS), pages 318–332. Springer, Andros, Greece, September 1998.

[80] Fernando Pedone and André Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, volume 1693, pages 94–106, Bratislava, Slovak Republic, 1999. Springer Berlin Heidelberg.

[81] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[82] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing (EDCC '02)*, volume 2485 of *Lecture Notes in Computer Science (LNCS)*, pages 44–61, Toulouse, France, October 2002. Springer, Berlin, Heidelberg.

[83] Michel Raynal. Consensus in synchronous systems: A concise guided tour. In *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC '02)*, pages 221–228. IEEE Computer Society, December 2002.

[84] Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN '12)*, volume 7129 of *Lecture Notes in Computer Science (LNCS)*, pages 153–167. Springer, Berlin, Heidelberg, 2012.

[85] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[86] 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)*, volume 5, pages 224–237, Saint Malo, France, October 1997. ACM.

[87] Efstratios Thomopoulos, Louise E. Moser, and Peter M. Melliar-Smith. Analyzing the latency of the Totem multicast protocols. In *Proceedings of the 6th International Conference on Computer Communications and Networks (IC3N '97)*, pages 42–50, September 1997.

[88] Efstratios Thomopoulos, Louise E. Moser, and Peter M. Melliar-Smith. Latency analysis of the Totem single-ring protocol. *IEEE/ACM Transactions on Networking*, 9(5):669–680, October 2001.

[89] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, pages 225–238. ACM, 2012.

[90] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. On the coordinator's rule for Fast Paxos. *Information Processing Letters*, 107:183–187, August 2008.

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

[92] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. The performance of Paxos and Fast Paxos. In *Proceedings of the 27th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC '09)*, pages 291–304, Recife, Brazil, May 2009.

[93] Gustavo Maciel Dias Vieira, Islene Calciolari Garcia, and Luiz Eduardo Buzato. Seamless Paxos coordinators. *Cluster Computing*, 17(2):463–473, June 2014.

[94] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*, pages 14:1–14:14, San Jose, California, October 2012. ACM.

[95] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. Lightning in the cloud: A study of very short bottlenecks on n-tierweb application performance. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems (TRIOS '14)*, page 9. USENIX Association, 2014.

[96] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

[97] Wenbing Zhao, Louise E. Moser, and Peter M. Melliar-Smith. End-to-end latency of a fault-tolerant CORBA infrastructure. In *Proceedings of the fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02)*, pages 189–198, April 2002.

[98] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. PriorityMeister: Tail latency QoS for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, pages 29:1–29:14, New York, NY, USA, 2014. ACM.

[99] Piotr Zieliński. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *Proceedings of the 19th International Symposium on Distributed Computing (DISC '05)*, volume 3724 of *Lecture Notes in Computer Science (LNCS)*, pages 369–383. Springer, Berlin, Heidelberg, Cracow, Poland, September 2005.

[100] Piotr Zieliński. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, April 2008.