State Machine Replication: from Analytical Evaluation to High-Performance Paxos

THÈSE N^{O} 5410 (2012)

PRÉSENTÉE LE 21 SEPTEMBRE 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES RÉPARTIS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nuno Filipe DE SOUSA SANTOS

acceptée sur proposition du jury:

Prof. K. Aberer, président du jury
Prof. A. Schiper, directeur de thèse
Prof. D. Kostic, rapporteur
Dr D. Malkhi, rapporteur
Prof. F. Pedone, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Abstract

Since their invention more than half a century ago, computers have gone from being just an handful of expensive machines each filling an entire room, to being an integral part of almost every aspect of modern life. Nowadays computers are everywhere: in our planes, in our cars, on our desks, in our home appliances, and even in our pockets. This widespread adoption had a profound impact in our world and in our lives, so much that now we rely on them for many important aspects of everyday life, including work, communication, travel, entertainment, and even managing our money.

Given our increased reliance on computers, their continuous and correct operation has become essential for modern society. However, individual computers can fail due to a variety of causes and, if nothing is done about it, these failures can easily lead to a disruption of the service provided by computer system. The field of *fault tolerance* studies this problem, more precisely, it studies how to enable a computer system to continue operation in spite of the failure of individual components.

One of the most popular techniques of achieving fault tolerance is *software replication*, where a service is replicated on an ensemble of machines (*replicas*) such that if some of these machines fail, the others will continue providing the service. Software replication is widely used because of its generality (can be applied to most services) and its low cost (can use off-the-shelf hardware).

This thesis studies a form of software replication, namely, *state machine replication*, where the service is modeled as a deterministic state machine whose state transitions consist of the execution of client requests. Although state machine replication was first proposed almost 30 years ago, the proliferation of online services during the last years has led to a renewed interest. Online services must be highly available and for that they frequently rely on state machine replication as part of their fault tolerance mechanisms. However, the unprecedented scale of these services, which frequently have hundreds of thousands or even millions of users, leads to a new set performance requirements on state machine replication.

This thesis is organized in two parts. The goal of the first part is to study from a theoretical perspective the performance characteristics of the algorithms behind state machine replication and to propose improved variants of such algorithms. The second part looks at the problem from a practical perspective, proposing new techniques to achieve high-throughput and scalability.

In the first part, we start with an analytical analysis of the performance of two consensus algorithms, one leader-free (an adaptation of the fast round of Fast Paxos) and another leader-

based (an adaptation of classical Paxos). We express these algorithms in the Heard-Of round model and show that using this model it is fairly easy to determine analytically several interesting performance metrics.
We then study the performance of round models in general. Round models are perceived as inefficient because in their typical implementation, the real-time duration of rounds is proportional to the (pessimistic) timeouts used on the underlying system. This contrasts with the failure detector or the partial synchronous system models, where algorithms usually progress at the speed of message reception. We show that there is no inherent gap in performance between the models, by proposing a round implementation that during stable periods advances at the speed of message reception.
We conclude the first part by presenting a new leader election algorithm that chooses as leader a well-connected process, that is, a process whose time needed to perform a one-to-majority communication round is among the lowest in the system. This is useful mainly in systems where the latency between processes is not homogeneous, because the performance of leader-based algorithms is particularly sensitive to the performance and connectivity of the process acting as a leader.
The second part of the thesis studies different approaches to achieve high-throughput with state machine replication. To support the experimental work done in this part, we have developed JPaxos, a fully-featured implementation of Paxos in Java.
We start by looking at how to tune the batching and pipelining optimizations of Paxos; using an analytical model of the performance of Paxos we show how to derive good values for the bounds on the batch size and number of parallel instances. We then propose an architecture for implementing replicated state machines that is capable of leveraging multi-core CPUs to achieve very high-levels of performance. The final contribution of this thesis is based on the observation that most implementations of state machine replication have an unbalanced division of work among threads, with one replica, the leader, having a significantly higher workload than the other replicas. Naturally, the leader becomes the bottleneck of the system, while other replicas are only lightly loaded. We propose and evaluate S-Paxos, which evenly balances the workload among all replicas, and thus overcomes the leader bottleneck. The benefits are two-fold: S-Paxos achieves a higher throughput for a given number of replicas and its performance increases with the number of replicas (up to a reasonable number).

Keywords: Distributed algorithms, fault tolerance, consensus problem, partial synchrony, heard-of model, round model, round-based algorithms, quantitative analysis, swift algorithms, state machine replication, Paxos, leader election, decentralized algorithms, multi-core, high-throughput, scalability.

Résumé

Depuis leur invention il y a plus d'un demi-siècle, les ordinateurs, qui étaient alors une poignée de machines onéreuses chacune de la taille d'une pièce, sont devenus une partie intégrante de la vie moderne. Actuellement, les ordinateurs sont partout : dans nos avions, dans nos voitures, sur nos bureaux, dans nos appareils ménagers, et même dans nos poches. Cette adoption généralisée a eu un impact profond sur notre monde et sur nos vies. Aujourd'hui, nous comptons sur eux dans un nombre important d'aspects de la vie quotidienne, notamment au travail, mais aussi pour communiquer, voyager, se divertir, et même pour gérer notre argent.

Compte tenu de notre dépendance accrue par rapport aux ordinateurs, il est devenu essentiel pour les sociétés modernes qu'ils fonctionnent correctement. Cependant, les ordinateurs peuvent être victimes de défaillances pour différentes raisons et, si rien n'est fait à ce sujet, ces défaillances peuvent facilement conduire à une interruption du service fourni par le système informatique. Le domaine de la *tolérance aux fautes* étudie ce problème. Plus précisément, il étudie comment concevoir des systèmes informatiques qui continuent à fonctionner en dépit de la défaillance de composants individuels.

Une des approches de tolérance aux fautes les plus populaires est la réplication logicielle, où un service est répliqué sur un ensemble de machines (*répliques*) de telle sorte que, si certaines de ces machines tombent en panne, les autres continuent à fournir le service. La réplication logicielle est largement utilisée en raison de sa généralité (elle peut être appliquée à la plupart des services) et de son faible coût (elle peut utiliser du matériel informatique standard).

Cette thèse étudie une forme de réplication logicielle, à savoir la réplication de machine à états, où le service est modélisé comme une machine à états déterministe dont les transitions correspondent à l'exécution des requêtes des clients. Bien que la réplication de machines à états ait été proposée il y a plus de 30 ans, la prolifération récente des services en ligne a suscité un intérêt nouveau pour le sujet.

Les services en ligne doivent être hautement disponibles et pour cela la réplication de machines à états est souvent utilisée. Toutefois, l'ampleur sans précédent de ces services, qui ont souvent des centaines de milliers voire des millions d'utilisateurs, soulèvent de nouveaux défis par rapport aux performances de la réplication de machines à états.

Cette thèse est organisée en deux parties. La première partie étudie d'un point de vue théorique les performances des algorithmes de réplication de machines à états et propose des variantes améliorées de ces algorithmes. La seconde partie se focalise sur le problème d'un point de vue pratique, proposant de nouvelles techniques dans le but d'améliorer les performances et le passage à l'échelle du système.

La première partie commence par une étude analytique des performances de deux algorithmes de consensus, le premier sans leader (une adaptation des rondes *rapides* de Fast Paxos), le second avec leader (une adaptation de l'algorithme Paxos classique). Nous exprimons ces algorithmes dans le modèle de rondes "Heard-Of", et nous montrons qu'en utilisant ce modèle il est assez facile de déterminer analytiquement plusieurs indicateurs de performance intéressants.
Nous étudions ensuite les performances des modèles de rondes. Les modèles de rondes sont perçus comme inefficaces parce que dans leur implémentation typique la durée en temps réel des rondes est proportionnel aux délais d'attente (pessimistes) utilisés pour la réception des messages. Ceci contraste avec les modèles fondés sur les détecteurs de défaillances ou le modèle partiellement synchrone, où les algorithmes progressent habituellement à la vitesse de réception des messages. Nous montrons qu'il n'y a pas d'écart inhérent entre les performances de ces modèles, en proposant une implémentation fondée sur des rondes avec la propriété de progresser à la vitesse de réception de messages durant les périodes de stabilités.
Nous concluons la première partie en présentant un nouvel algorithme d'élection de leader qui choisit un processus bien connecté, c'est-à-dire, un processus dont le temps nécessaire pour effectuer un tour de communication avec une majorité des processus est parmi les plus bas dans le système. Ceci est surtout utile dans les systèmes où la latence entre les processus n'est pas homogène. Dans ce contexte, les performances des algorithmes fondés sur un leader est particulièrement sensible aux performances et la connectivité du leader.
La deuxième partie de la thèse considère plusieurs approches pour obtenir une réplication hautement performante de machines à états. Pour valider le travail expérimental réalisé dans cette partie, nous avons développé JPaxos, une implémentation de Paxos en Java.
Nous commençons par examiner comment configurer de façon optimale les optimisations de "batching" et de "pipelining", souvent utilisées dans les implémentations de Paxos. Nous proposons ensuite une architecture pour l'implémentation de machines à états répliquées, capable de tirer parti des processeurs multi-cœurs pour atteindre de très hautes performances.
La dernière contribution de cette thèse est basée sur l'observation que la plupart des implémentations de réplication de machines d'états ont une répartition asymétrique des tâches entre les processus, une réplique (le leader) ayant une charge de travail beaucoup plus importante que les autres répliques. Naturellement, le leader devient le goulot d'étranglement du système, tandis que les autres répliques ne sont que légèrement chargés. Nous proposons et évaluons S-Paxos, qui équilibre équitablement la charge de travail entre toutes répliques, et donc élimine le goulot d'étranglement au niveau du leader.

Mots-clés Algorithmes répartis, tolerance aux fautes, problème de consensus, système partiellement synchrone, modèle "heard-of", modèle en rondes, algorithmes en rondes, analyse quantitative, algorithmes "swift", réplication de machines d'état, Paxos, élection du leader, algorithmes décentralisés, multi-cœurs, haute performance, passage à l'échelle.

Acknowledgements

As anyone who completed a PhD can attest, doing a thesis is a challenging experience, requiring dedication, perseverance and countless hours of work spanning many years. Completing successfully such an endeavor would not be possible without the support of my colleagues, friends and family.
I would like to start by thanking my supervisor, Professor André Schiper, for having granted me this opportunity to work in the field of distributed systems. His guidance and unwavering trust, combined with his encouragement to pursue my own research interests, provided the perfect conditions to do a thesis, and allowed me to grow as a researcher and as a person. It was truly a privilege to complete this thesis under his supervision.
For taking time from their busy schedules to review my thesis, I wish to express my appreciation to members of the jury, Prof. Dejan Kostic, Dr. Dahlia Malkhi and Prof. Fernando Pedone, as well as to the president of the jury, Prof. Karl Aberer.
Modern scientific work is almost always done in collaboration with other people and this thesis was no exception. I had the pleasure of working and co-authoring papers with Martin Hutle, Fatemeh Borran, Zarko Milosevic and Martin Biely. My gratitude also goes to all the friends and colleagues from the LSR, for discussing ideas, for reviewing drafts of my papers, and for the occasional hike in the mountains or drink at the Sattelite. In particular, I would like to thank David Cavin, Sergio Mena, Olivier Rütti, Richard Ekwall, Cendrine Favez, Omid Shahmirzadi, Darco Petrovic and Thomas Ropars. Outside the LSR, I would like to thank Paweł T. Wojciechowski, Jan Kończak and Tomasz Żurkowski for their work on JPaxos. Many thanks as well to France Faille for taking care of all the administrative tasks.
Living in a foreign country comes with its own set of challenges but I was lucky to meet many fantastic people, both foreigners like me and locals, who made it a pleasure to live in Lausanne. I thank them for all great moments spent exploring the nature of this beautiful country.
Last but not least, I thank my whole family for their support. In particular, my deepest gratitude goes to my parents, who saw their only son moving to a country two thousand kilometers away to pursue his PhD. Nevertheless, they always gave me their unconditional support and encouragement, without which I could not have completed this thesis.

Some of the experiments presented in this thesis were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

I would like to acknowledge and thank the institutions that provided financial support for this thesis, including the Portuguese Foundation for Science and Technology (FCT) under grant SFRH/BD/17276/2004, the Swiss National Science Foundation under grant number 200021-111701, and the Hasler Foundation under grant number 2070.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context and motivation

Since their invention more than 60 years ago, computers have gone from being just an handful of expensive machines each filling an entire room, to being an integral part of almost every aspect of modern life. Nowadays computers are everywhere: in our planes, in our cars, on our desks, in our home appliances, and even in our pockets. This widespread adoption had a profound impact in our world and in our lives, so much that now we rely on them for many important aspects of everyday life, including work, communication, travel, entertainment, and even managing our money.

Given our increased reliance on computers, their continuous and correct operation has become more and more important. However computers fail. The causes of *faults* are varied, including faulty hardware, bit flips caused by ionizing radiation, overheating, human error or software bugs. Although hardware can be designed to decrease the likelihood of faults and software can be designed to very high levels of quality, this only decreases the probability of faults, so faults will happen sooner or later. When a fault occurs, if not handled correctly, it may cause a failure in the system, *i.e.*, the system stops operating or produces the wrong results. While some failures are just minor inconveniences (*e.g.*, not being able to check our email) others can have dramatic consequences, including the loss of life (*e.g.*, the failure of the flight control system of an airplane). But regardless of their severity, failures in computer systems are always undesirable.

The notion of *availability* is defined by the fraction of time that a given system is operating correctly. Availability depends on the frequency of crashes and on the time it takes to recover from a crash. There are several approaches at improving availability. For instance, the likelihood of crash can be decreased by designing better hardware and software (*fault avoidance*). However, this approach has high costs and can only decrease the likelihood of faults, it can never eliminate them completely. Another approach is to decrease the recovery time, so that even though the system may crash with the same frequency, its downtime will be smaller, thereby improving availability. This approach is also limited, as recovery of any system will always take some time. A third approach, which is the topic of this thesis, is called fault-tolerance.

A system is said to be *fault-tolerant* if it will continue operating in spite of the failure of some of its components. A popular way of achieving fault tolerance is software replication, where a service is provided by an ensemble of machines (*replicas*) such that if some of these machines fail, the others will continue providing the service. Software replication is widely used because of its generality (can be applied to most services) and its low cost (can use off-the-shelf hardware).

State Machine Replication *State machine replication* is a form of software replication, in particular of *active replication*, where the service is modeled as a deterministic state machine whose state transitions consist of the execution of client requests [Lam78, Sch90]. The service is then executed in every replica. The replication middleware controls the execution of requests, ensuring that all replicas execute the same sequence of requests, which together with the fact that the service is deterministic, ensures that their state remains consistent.

The problem of agreeing on a order for a sequence of requests is an example of the *atomic broadcast* (or total-order broadcast) problem. Atomic broadcast is defined for a set of processes, where each process can broadcast messages to the others, with the guarantee that all processes in this set will see the same sequence of messages. At the heart of atomic broadcast, and therefore of state machine replication, is the *consensus* problem, where a set of processes has to agree on a single value. The two problems are in fact closely related, meaning that a solution to one can be used to solve the other.

Consensus Consensus plays an important role not only in state machine replication, but also in fault tolerance in general. As such, it has been heavily studied for the last 30 years, both from a theoretical and practical perspective.

The difficulty of solving consensus depends both on the synchrony assumptions (process speed and bounds on message delivery time) and on the failure assumptions (type and number of failures). The synchrony assumptions go from a *asynchronous system*, where there are no bounds on process speed or message delivery time, all the way to the *synchronous system* where these bounds exist, are known, and hold from the beginning. The failure assumptions characterize the maximum number of faulty processes and the type of failures. These go from crash failures to arbitrary (byzantine) failures. In this thesis we discuss only crash failures.

One of the first results established that consensus is impossible to solve deterministically in the *asynchronous model* even if a single process may crash [FLP85]. In the other end of the spectrum, in a *synchronous system* consensus is solvable [Lyn96]. However, enforcing the strong assumptions of a synchronous system on a real system requires the use of large, conservative timeouts chosen for the worst case conditions, since violations of these assumptions may result in the consensus algorithm producing wrong results. Therefore, the research community has focused its attention on intermediate models, with weaker assumptions than the synchronous model but where consensus is still solvable. One such model is the *partially synchronous model* [DLS88], where the bounds on message delay and process speed are known but hold only eventually[1]. The *failure detector model* takes a different approach, by augmenting the asynchronous system with an oracle that provides (unreliable) information about the failures in the system [CT96]. There is a range of failure detectors that can be used to solve consensus, defined by different set of restrictions on how unreliable (or, conversely, how reliable) the information they provide can be.

Paxos State machine replication is nowadays widely used in the industry [Bur06, HKJR10, BBH$^+$11], and is the dominant mechanism to implement software replication with strong consistency. Among the protocols used for state machine replication Paxos is by far the most popular.

Paxos was first proposed by Lamport [Lam89] in 1988[2]. Paxos is defined for the partially synchronous model, requiring $n \geq 2f + 1$ processes to tolerate f faults. In Paxos, a distinguished process, the leader, assumes the responsibility of proposing an order for the requests received from the client. Once enough replicas accept the order proposed by the leader, the request is assigned a permanent order.

From a theoretical perspective, Paxos has optimal resilience [Lyn96] and achieves the minimum number of communication delays to order a request [Lam06]. And in practice, it can be implemented very efficiently and is amenable to a wide-range of optimizations that can greatly improve its performance.

Leader Election In order to ensure termination, Paxos requires that processes agree on a leader, which must be a correct process. This is the *leader election problem*, which has also been widely studied [ADGFT01, ADGFT03, ADGFT04, MOZ05, HMSZ06]. Leader election requires also a minimum amount of synchrony [ADGFT03], and therefore cannot be solved in the asynchronous system.

Challenges of software replication in the Internet age During the last few years Internet services have grown dramatically in popularity, reaching ever increasing user populations. Users have come to except both uninterrupted availability and fast response times, and services must be able to provide both these features in order to compete.

To achieve the required high availability, online services typically use software fault tolerance

[1]There are other formulations of the partially synchronous system, for instance, where the bounds hold but are unknown. See [DLS88]

[2]At the same time, Liskov independently proposed viewstamped replication [OL88], which is based on the same algorithmic principles as Paxos.

techniques, mainly under the form of replication. This trend has put additional requirements on the replication middleware, which now must not only provide fault tolerance, but must also provide high-throughput and low response time.

1.2 Contribution

The contribution of this thesis is centered on the Paxos protocol, with the first part focused mostly on theoretical aspects of consensus algorithms and leader election, and the second part focusing on the practical aspects of achieving high-performance with Paxos.

Quantitative analysis of consensus algorithms Although the solvability of consensus is now a well-understood problem, comparing different algorithms in terms of efficiency is still an open problem. We address this question for round-based consensus algorithms using communication predicates, on top of a partial synchronous system that alternates between good and bad periods (synchronous and non-synchronous periods). Communication predicates together with the detailed timing information of the underlying partially synchronous system provide a convenient and powerful framework for comparing different consensus algorithms and their implementations. This approach allows us to quantify the required length of a good period to solve a given number of consensus instances. With our results, we can observe several interesting issues, such as the number of rounds of an algorithm is not necessarily a good metric for its performance.

Swift algorithms for repeated consensus Round implementations have a reputation of being inefficient, since the real-time duration of rounds is usually proportional to the (pessimistic) timeouts used in the underlying system, regardless of the effective speed of the system. This contrasts with most of the algorithms for the partially synchronous or for the failure detector model which, in the absence of faults, progress naturally at the speed of the system. We formalize the notion of a *swift algorithm*, which captures this difference, by specifying what it means to “progress at the speed of the system”. We then present a round implementation that during stable periods advances when “enough” messages are received instead of waiting for timeouts, thereby allowing the algorithms running on top of it to be swift. An experimental evaluation confirms that indeed this round implementation closes the performance gap between the round model and the partially synchronous system or for the failure detector model.

Latency-aware leader election The performance of leader-based algorithms is particularly sensitive to the performance and connectivity of the process acting as a leader, especially in heterogeneous systems. The reason is that this class of algorithms consists mainly on communication rounds of the form leader-to-majority, whose duration equals the longest round-trip-time between the leader and the processes in this majority. Thereby, a poorly connected leader will significantly limit the speed at which the system makes progress by comparison with a leader that is well-connected with a majority of processes. We present and evaluate a latency-aware leader election algorithm that monitors the round-trip-time between processes and uses this information to choose as leader a process that is not only correct, but also has a one-to-majority round-trip time that is among the lowest in the system.

JPaxos - A research prototype of State Machine Replication As a basis for our research on state machine replication, we have developed a prototype implementation of Paxos in Java, called JPaxos. JPaxos is a fully-featured implementation of state machine replication, including snapshotting and recovery mechanisms. This work was done in collaboration with a group of researchers from Poznań University, and has been made available as an open-source project to anyone interested in research on state machine replication.

Tuning Paxos for High-Throughput with Batching and Pipelining The performance of Paxos can be greatly improved by using batching and pipelining. However, tuning these optimizations to achieve high-throughput is challenging, as their effectiveness depends on many parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. We address this question by presenting an analytical model of the performance of Paxos, and showing how it can be used to determine good configuration values for the batching and pipelining optimizations, in the sense of achieving high-throughput. We then validate the model, by presenting the results of experiments where we investigate the interaction of these two optimizations in LAN and WAN environments, and comparing these results with the prediction from the model. The results show that although batching by itself provides the largest gains in all scenarios, in most cases combining it with pipelining provides a very significant additional increase in throughput, not only on high-latency network but also in many low-latency networks.

Scaling State Machine Replication in multi-core CPUs The traditional architecture used by implementations of Replicated State Machines (RSM) does not fully exploit modern multi-core CPUs. This is increasingly the limiting factor in their performance, because network speeds are increasing much faster than the single-thread performance of CPUs. Thus, when deployed on Gigabit-class networks and exposed to a workload of small to medium size client requests, RSMs are often CPU-bound, as they are only able to utilize a few cores, even though many more may be available.

We revisit the traditional architecture of a RSM implementation, showing how it can be parallelized so that its performance scales with the number of cores in the nodes. We do so by applying several good practices of concurrent programming to the specific case of state machine replication, including staged execution, workload partitioning, actors, and non-blocking data structures. Furthermore, we implement and evaluate the architecture proposed using JPaxos.

The results show an almost linear scalability up to eight cores. More generally, in all our experiments we have consistently reached the limits of the network subsystem by using up to 12 cores, and do not observe any degradation when using up to 24 cores. Furthermore, the profiling results of our implementation show that even at peak throughput contention between threads is minimal, suggesting that the throughput would continue scaling given a faster network.

Scaling State Machine Replication with number of replicas As already mentioned, implementations of state machine replication are prevalently using variants of Paxos, which is a leader-based protocol. Typically these protocols are also leader-centric, in the sense that the leader replica performs a disproportionate amount of work, such that when the load on the system increases, the leader is the first running out of resources, becoming the bottleneck, even though other replicas may have plenty of available resources. A leader centric design also exhibits poor or no scalability, because as the number of replicas increases the load on the leader increases, while the resources of the additional replicas are left mostly unused.

We show that much of the work performed by the leader in a leader-centric protocol can in fact be evenly distributed among all the replicas, thereby leaving the leader only with minimal additional workload. This is done (i) by distributing among all replicas the work of handling client communication, (ii) by disseminating client requests among replicas in a distributed fashion, (iii) by performing the ordering protocol on ids. We derive S-Paxos, a variant of Paxos incorporating these ideas. Additionally, we implement and evaluate S-Paxos, using JPaxos as a starting point. Compared to leader-centric protocols, S-Paxos achieves significantly higher throughput for a given number of replicas. Another benefit is that the S-Paxos breaks the traditional trade-off between performance and fault tolerance, because adding more replicas to S-Paxos improves its performance (up to reasonable number of replicas).

1.3 Outline

Chapter 2 provides the background for the thesis, including definitions and problem statements.

The first part, consisting of Chapters 3 to 6, studies the problem of state machine replication from a mostly theoretical perspective. Chapter 3 introduces the HO model, which is the basis of the work in the following three chapters, Chapter 4 presents an analytical study of the performance of consensus algorithms expressed in a round model, focusing on the Paxos algorithm, Chapter 5 proposes a new round implementation that during stable periods advances at the speed of message reception, and Chapter 6 describes a new leader-election algorithm that takes network latency in consideration to choose a well-connected leader.

Chapters 7 to 10 form the second part of this thesis, which looks at State Machine Replication and Paxos from a practical perspective, focusing on high-throughput and scalability.

Chapter 7 presents the background, describing the Paxos protocol in more detail and the JPaxos implementation, and Chapter 8 looks into how to tune batching and pipelining for high-throughput. Chapters 9 and 10 show how to scale implementations of SMR with two orthogonal parameters: the number of cores in a CPU (vertical scalability) and the number of replicas in the system (horizontal scalability).

2 Definitions and Background

2.1 System Model

We consider a distributed system where a set of n processes, denoted $\Pi = \{p_1, \dots, p_n\}$ with $n > 2$, communicate by message passing. Processes do not have access to shared memory. Each pair of processes in the system is connected by a point-to-point communication channel. All messages sent in the network are unique and taken from a set $\mathscr{M}$. For each process $p \in \Pi$ there is a variable buffer_p, which contains all messages that have been sent to p but were not yet received by p. Processes proceed by making steps, where a step is either a receive step or a send step:

- In a *send step*, a single message can be sent to another process in the system, that is, when process p executes send$(\langle m \rangle, q)$, the tuple $\langle m, p \rangle$ is placed in buffer_q.
- In a *receive step*, some messages are received, that is, when process p executes receive(S), a set $S \subseteq \mathit{buffer}_p$ is removed from buffer_p, and delivered to p. Note that S may be empty.

In each step, processes can perform some computation. We further assume an abstract global discrete time, to which processes do not have access[1]. Without loss of generality, we assume that at each tick of this global clock, at least one process takes a step.

The description above is an abstract system model, which must be completed with other details before it can be used to solve distributed problems. These details fall into two categories, *failure modes* and *synchrony*, which we will present next.

2.2 Failure Model

As the topic of this thesis is fault tolerance, it is essential to have a good model of how processes and channels can fail. Since distributed systems can fail in many different ways, with failures varying in severity and in their cause, the failure model must have be flexible. This allows us to

[1]Not to be confused with the local clocks present in some of the models in this thesis, which give processes only a local time that may not match the global time.

choose the failure model that better describes a particular distributed system and to devise appropriate distributed algorithms. There are two main categories of failures, process and channel failures.

2.2.1 Process failures

The following are the main types of process failures:

Fail-stop In a fail-stop failure, the process executes the protocol correctly until the moment it fails, at which point it stops taking steps permanently: it stops sending or receiving messages and does not perform any further computation. This type of failures is also called by *crash* failures.

Omission failure This type of failure happens when a faulty process omits sending a message that it should have sent due to the protocol.

Byzantine This is the most general class of failures. A Byzantine process can behave arbitrarily, and is not obliged to follow the protocol: it can omit steps, alter messages, create spurious messages or perform any other type of malicious behavior.

This thesis focus on fail-stop failures, and therefore will not discuss omission or byzantine failures.

In Chapter 3, in the context of the HO Model, we will present another type of failure called *transmission failure*, which has the advantage of abstracting in a single notion the important aspects of fail-stop, omission and channel failures, thereby simplifying the development of distributed algorithms.

2.2.2 Communication channels

Communication channels (links) are defined by one or more of the following properties:

No creation If a process q receives a message m from another process p, then p sent m to q.

No duplication A process q receives a given message m at most once.

No loss If a correct process p sends a message m to a correct process q, then q eventually receives m.

Fair loss If a correct process p sends a message m infinitely often to a correct process q, then q receives m an infinite number of times. Note that a channel that satisfies the *No loss* property also satisfies *Fair loss*.

Depending on the properties they satisfy, channels can be reliable, quasi-reliable or fair-lossy. Informally, with reliable links a message sent to a correct process is always delivered, even if the sender fails, while with quasi-reliable links delivery is guaranteed only if both sender and receiver are correct. Finally, a fair-lossy link may lose messages but eventually some messages are delivered, while in a lossy link delivery is not at all guaranteed. Formally, they are defined as follows (from weaker to stronger):

Lossy Satisfies the properties *No creation* and *No duplication*.

Fair-lossy Satisfies the properties *No creation*, *No duplication* and *Fair loss*

Quasi-reliable A quasi-reliable link satisfies the *No creation*, *No duplication* and *No loss* properties.

Reliable A reliable link satisfies the *No creation*, *No duplication* and the following property: If p sends m to q and q is correct, then q eventually receives m.

2.3 Synchrony

The synchrony assumptions model the behavior of the system in terms of timing of events. These assumptions describe two aspects of the distributed system, namely the relative speed of processes and the time necessary to transmit a message on the network. The synchrony assumptions are important as they define the type of problems that can be solved in a particular system and the type of algorithms that can be used. Strong synchrony assumptions allow solving a larger range of problems than weaker synchrony assumptions. Additionally, while an algorithm that solves a problem in a model S will also work in a model S' stronger than S, the reverse is usually not true.

There are four main system models. The asynchronous system is the weakest, with no timing assumptions. On the opposite end of the spectrum, the synchronous system has fixed upper bounds on the duration of every event. In between, there are several intermediate models, that try to better approximate the behavior of the typical distributed system. There is the partially-synchronous model, which offers a limited form of timing guarantees, and the failure detector model, which augments the asynchronous system with an oracle that provides additional information to processes that help them overcome the limitations of the asynchronous system.

2.3.1 Asynchronous system

In the asynchronous system there are no bounds on the relative speed of processes or on the time needed to transmit a message. It is the most general model, as it makes no timing assumptions whatsoever. As such, any algorithm designed for the asynchronous system will work in a wide variety of distributed systems.

The drawback of the asynchronous system is that many important problems cannot be solved in it if processes are allowed to crash. Some noteworthy examples are consensus [FLP85], atomic broadcast(See Section 2.4.2), and terminating reliable broadcast [FRT99]. The reason is that without timing assumptions, it is impossible to distinguish between a slow process and a process that crashed. One problem that can be solved in the asynchronous system is reliable broadcast (Discussed in Section 2.4.1).

The pure asynchronous system is very restrictive, but fortunately it is not an accurate representation of most distributed systems, which behave in a timely fashion most of the time. The models described next make some timing assumptions, allowing them to be used to solve a wider variety of problems.

2.3.2 Synchronous system

In the synchronous system [Lyn96] there are known bounds on both the message transmission delay and the relative process speed (additionally, if local clocks are used, their drift rate must be bounded). More precisely:

Bound on message transmission delay If a message m is sent by process p to process q at time t, then q receives the message no later than time $t + \Delta$.

Bound on process relative speed of processes In any contiguous time interval I with duration Φ or greater, every correct process takes at least one step.

These conditions imply that messages take at most Δ time to be delivered and no correct process can run more than Φ times slower than another process.

The synchronous system can be used to solve a large number of distributed problems with fairly simple algorithms. However, it relies on strong assumptions that are hard to enforce in practice. In order to ensure that the bounds always hold, they must be chosen based on the worst case behavior of the system. Since real systems often have periods of instability where channels and processes become orders of magnitude slower than in the normal case, the resulting bounds are often very conservative. This often leads to poor performance. Worse, there is the risk of producing wrong results if the bounds were not chosen correctly. As explained next, the partially synchronous system and the failure detector model were introduced to address these limitations.

2.3.3 Partially Synchronous system

The partially synchronous system relaxes the assumptions of the synchronous system, by requiring that the timing bounds hold only eventually. This matches more closely real systems, while remaining strong enough to solve a large number of important problems. In particular, an algorithm for the partial synchronous system must be ready to cope with periods of asynchrony, which requires it to be always safe, *i.e.*, it cannot produce wrong results if the bounds do not hold. This allows choosing more aggressive bounds, without fear of violating correctness (although liveness can be negatively impacted if the bounds are too small).

There two variants of the partially synchronous model, depending on the assumptions on the bound Δ on message transmission delay and Φ on relative process speed [DLS88]:

Unknown bounds The bounds Δ and Φ hold from the beginning but are unknown (*i.e.*, Δ and Φ depend on the run).

Known bounds The bounds Δ and Φ are known but hold only after an unknown time called *Global Stabilization Time (GST)* (*i.e.*, GST varies from run to run). It is also assumed that channels may lose messages before GST but are quasi-reliable after GST.

2.3.4 Asynchronous system augmented with failure detectors

The idea of the failure detector model [CT96] is to augment the asynchronous system with an oracle that provides processes with information about the state of the system. By giving the right type of information, the processes are able to solve a wider range of problems.

In the failure detector model, each process p_i has a failure detector module FD_i that it can query at any time, to obtain the set of processes that FD_i currently suspects to have crashed. The information provided by the FD_i may not be correct, *i.e.*, the failure detectors are unreliable. For instance, each failure detector may suspect processes that have not crashed, and may not suspect processes that have crashed. Failure detectors can also change the list of suspected processes, either by removing or adding processes to the suspected set. Additionally, at any given point in time, the failure detector modules at two different processes may suspect a different set of processes.

If the output of failure detectors were unconstrained, they would not add anything to an asynchronous system, and the FLP result would still hold. Therefore, to be useful, the output of failure detectors must be constrained. The requirements are expressed in terms of two abstract properties, a *completeness* and an *accuracy* property.

Completeness defines which crashed processes are eventually suspected by whom. There are two variants:

- *Strong Completeness:* Eventually every process that crashes is permanently suspected by *every* correct process.
- *Weak Completeness:* Eventually every process that crashes is permanently suspected by *some* correct process.

Accuracy restricts the mistakes that failure detectors can make, *i.e.*, wrong suspicions of correct processes. There are four variants:

- *Strong accuracy:* No process is suspected before it crashes.
- *Weak accuracy:* Some correct process is never suspected.
- *Eventual strong accuracy:* There is a time after which correct processes are not suspected by any correct process.
- *Eventual weak accuracy:* There is a time after which some correct process is never suspected by any correct process.

A failure detector is defined by a pair of completeness and accuracy properties. In this thesis we will make use of the two classes of failure detectors, namely the $\Diamond\mathscr{P}$ and the $\Diamond\mathscr{W}$ classes. The *Eventually Perfect* failure detector ($\Diamond\mathscr{P}$) satisfies strong completeness and eventual strong accuracy, while the *Eventually weak* failure detector ($\Diamond\mathscr{W}$) satisfies weak completeness and eventual weak accuracy. In Chapter 6 we make use of the Leader Election Oracle Ω, which ensures that eventually all correct processes agree on a correct process, the so called leader. The Ω Leader Election Oracle is equivalent to the $\Diamond\mathscr{W}$ failure detector [Chu98], *i.e.*, given one it is possible to implement the other.

Note that although the failure detector model is asynchronous, implementing failure detectors requires a system with timing assumptions [ADGFT03, ADGFT04], *i.e.*, failure detectors encapsulate timing assumptions.

2.4 Problems

2.4.1 Reliable Broadcast

Often when a group of processes is working together to solve a problem, they need a mechanism for a process to send a message to all the other processes in the group. However, in the presence of faults, a regular network broadcast does not ensure that a message is delivered to all, since the sender may crash halfway through the broadcast or some messages may be lost.

Reliable broadcast provides stronger properties than regular broadcast, by ensuring that a message is either delivered to all processes or to none. It is defined by two primitives, *rbcast* and *rdeliver* that satisfy the following properties:

Validity If a correct process executes rbcast(m), then all correct processes rdeliver m.

Agreement If a correct process rdelivers a message m, then all correct processes eventually rdeliver m.

Integrity For any message m, every correct process rdelivers m at most once, and only if m was previously rbcast.

The properties above allow runs where a message m is partially rdelivered, as faulty processes are allowed to rdelivered m without correct processes having to do the same. *Uniform reliable broadcast* is a variant of reliable broadcast that forbids this scenario. It is defined by the same Validity and Integrity properties as non-uniform reliable broadcast, with the Agreement property replaced by the following property:

Uniform Agreement If a process (correct or not) rdelivers a message m, then all correct processes eventually rdeliver m.

2.4.2 Atomic Broadcast

Reliable broadcast is defined for isolated messages and as such, does not enforce any order among messages delivered by different processes or even by the same process. Atomic broadcast is an extension of reliable broadcast that ensures that messages are delivered by all processes in the same order. It is defined by the primitives *abcast* and *adeliver*, which satisfy the properties of uniform reliable broadcast in addition to the following order property:

Uniform Total order If some correct process adelivers m before m', then every correct process delivers m' only after it has adelivered m.

Like with reliable broadcast, there are uniform and non-uniform variants of atomic broadcast. In the rest of this thesis, Atomic Broadcast will be used to refer to the uniform variant. Chapter 7 discusses Atomic Broadcast in more detail in the context of State Machine Replication.

2.4.3 Consensus

Consensus is one of the key problems in fault-tolerant distributed computing. The problem is related to replication and appears when implementing atomic broadcast, group membership, or similar services. Consensus is defined over a set of processes Π, where each process $p_i \in \Pi$ has an initial value v_i: all processes must agree on a common value that is the initial value of one of the processes.

Formally, the consensus problem is defined by the primitives *propose(v_i*, which is used by processes to propose their initial value, and *decide(v_i)*, which is used by processes to decide on a value. These primitives satisfy the following properties:

Termination Every correct process eventually decides.

Validity If a process decides v, then v is the initial value of some process.

Agreement Two correct processes cannot decide differently.

The properties above allow a faulty process to decide a value different than the one decided by correct processes. The uniform variant of consensus prevents this situation. It replaces the agreement property above by the following property:

Uniform agreement Two processes (correct or not) cannot decide differently.

Consensus cannot be solved deterministically in an asynchronous system with faults, as established by the FLP impossibility result [FLP85]. Later it was shown that consensus can be solved in a partially synchronous system with a majority of correct processes [DLS88]. Roughly speaking, a partially synchronous system may initially be asynchronous, but eventually becomes synchronous; links may be initially lossy, but eventually become reliable. The failure detector model was introduced a few years later [CT96]. The model is defined as an asynchronous system "augmented" with a device called failure detector, defined by some completeness and accuracy properties (see [CT96] for details). Over the years the failure detector model has become very popular.

HO Round Model Part I

3 The Heard-Of Round Model

The consensus algorithms in Chapters 4 and 5 are expressed in the *Heard-Of* (*HO* for short) Round Model [CBS09].

The HO model is a variant of the round-based computational model that was introduced in [DLS88], as a convenient computational model on top of a partially synchronous system model. The round-based model in [DLS88] was later extended by Gafni [Gaf98] with the notion of communication predicates. The HO model resulted from the combination of communication predicates with the transmission fault model of [SW89]. In [CBS09] the authors show that the resulting round-based model unifies all benign faults (*i.e.*, handles faults, being static or dynamic, permanent or transient, in a unified way). We use here the notations from [CBS09].

3.1 Model and Definitions

A computation in the HO model evolves in rounds, where in each round every process sends a message to every other process (possibly a *null* message), receives a subset of the messages sent by the other processes, and finally performs a local computation on the set of received messages.

HO Algorithm An algorithm in the HO model is called a *HO algorithm* and consists of a tuple $\mathscr{A} = \langle S_p^r, T_p^r \rangle$, where S_p^r and T_p^r are the *sending* and the *transition* functions, respectively. Let s_p denote the current state of process p. For each round r and each process p, the sending function $S_p^r(s_p)$ determines a vector of messages to be sent, one message for each process (*null* if there is no message for this process). At the end of a round r, process p makes a state transition according to $T_p^r(\vec{\mu}, s_p)$, where $\vec{\mu}$ is the partial vector of messages received in round r. Rounds are communication closed: a message sent in round r to q and not received by q in round r is lost.

Hear-Of Sets and Communication Predicates We denote by $HO(p,r)$ the set of processes from which p receives a message at round r (including itself): $HO(p,r)$ is the *Heard-Of* set of p in round r. If $q \notin HO(p,r)$, then the message sent by q to p in round r was subject to a *transmission failure*. The exact cause of the transmission failure (message loss, message delayed or process crash) is not specified, as it is of no importance for the computation.

Communication predicates are expressed over the sets $\big(HO(p,r)\big)_{p\in\Pi, r>0}$. Communication predicates restrict transmission failures, for example, the predicate $\forall p, \forall r : |HO(p,r)| > n/2$ ensures that every process receives at least $n/2$ messages in every round.

HO Machine Let $\mathscr{A}$ be an HO algorithm and $\mathscr{P}$ be a communication predicate. The tuple $\langle \mathscr{A}, \mathscr{P} \rangle$ is called an *HO machine*, and can be used to solve a distributed problem over the set of processes.

Coordinated HO Machines A *coordinated HO machine* (CHO) is an extension of an HO machine that includes the notion of coordinator. This allows the specification of coordinator-based algorithms, by giving predicates not only over the HO sets but also over the current coordinator. In a CHO machine, *Coord*(p,r) denotes the process that p considers to be the coordinator at round r, henceforth called the coordinator of p in round r. The functions S_p^r and T_p^r take the current coordinator as an additional parameter, reflecting the fact that the messages to be sent and the state transitions depend also on the coordinator.

Consensus algorithms, including coordinator-based algorithms, typically consist of a sequence of one or more rounds that are repeatedly executed. This sequence of one or more rounds is called a *phase*. Typically, the coordinator is changed only at the beginning of a phase. This is the case of all the coordinated algorithms we consider here; therefore we will use the notation *Coord*(p,ϕ) to refer to the coordinator of process p during all the rounds of phase ϕ.

Remark The algorithms and predicates given in Chapters 4 and 5 ensure a decision of a majority of processes (two-thirds majority in case of OTR). However, with a small modification, all processes that are eventually reachable will decide: since our agreement property is a uniform property (there are no “faulty” processes that are exempted from agreement), processes — once they have decided — can simply communicate their decision to all other processes.

3.2 Solving consensus in the HO model

Chapters 4 and 5 study the question of how an HO machine $\langle \mathscr{A}, \mathscr{P} \rangle$, where $\mathscr{P}$ is some predicate, can be implemented in a “classical” message-passing model. Figure 3.1 illustrates how these parts work together in a system. The top layer, the HO Algorithm $\mathscr{A}$, is defined solely in terms of the sending function S_p^r and transition function T_p^r, and assumes some commu-

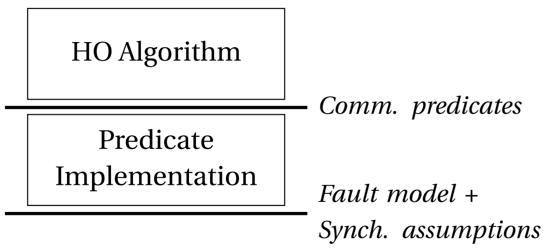


Figure 3.1: The HO and the predicate implementation layers

Algorithm 3.1 The *OneThirdRule* algorithm [CBS09].

```
Initialization:
    $x_p \leftarrow v_p$ /* $v_p$ is the initial value of $p$ */

Round $r$:
    $S_p^r$:
        send $\langle x_p \rangle$ to all processes

    $T_p^r$:
        if $|HO(p,r)| > 2n/3$ then
            $x_p :=$ the (smallest) most frequently received value
        if more than $2n/3$ values received are equal to $\overline{x}$ then
            DECIDE($\overline{x}$)
```

nication predicate $\mathscr{P}$. The communication predicate $\mathscr{P}$ is implemented by the *Predicate Implementation* layer, which builds on top of the system model. These two layers are independent, apart from the interface defined by the communication predicate. This enforces a clear separation between the high-level computational model of the HO Algorithm and the low-level system model and allows each layer to be developed independently.

Note that for coordinator-based algorithms, the predicate implementation layer is also responsible for electing the coordinator. This is in contrast to failure detector based solutions, in which the failure detectors (or the leader election oracle) are provided by some external service. Such a service typically uses heartbeat messages. No such external service is used here. The difficulty is, during a good period, to elect a common coordinator, resynchronize the processes and exchange the necessary messages to ensure the predicate, within a time as short as possible, and using as few messages as possible.

3.3 The OneThirdRule (OTR) consensus algorithm

Algorithm 3.1 is the code for the OneThirdRule (OTR) consensus algorithm expressed in the HO model. This algorithm appeared first in [CBS09], as an example of a simple consensus algorithm for the HO model. OTR is used both in Chapters 4 and 5.

OTR has similarities with a fast round of the Fast Paxos algorithm [Lam06]. Every round of OTR has the same sending and transition function.

In OTR processes keep an estimate (x_p) of the decision value, send it to all other processes in every round (line 5), and decide when they detect that “enough” processes have the same estimate (line 10).

Decision can be reached in one round if all initial values are identical, otherwise decision can be reached in two rounds. Liveness requires two distinct rounds (not necessarily consecutive) that satisfy the following predicate:

$$\mathscr{P}_u(r) :: \ \exists \Pi_0 \subseteq \Pi \ s.t. \ |\Pi_0| > 2n/3, \forall p \in \Pi_0 : HO(p,r) = \Pi_0$$

Informally, this predicate ensures that a "large enough" set of processes all receive the same set of messages. Such a round is called *uniform* with cardinality $2n/3$; we use the term *uniform round* when the cardinality is clear from the context. In the following, we will use $\mathscr{P}_{otr}$ to denote the predicate that ensures the existence of two distinct rounds (not necessarily consecutive) that satisfy $\mathscr{P}_u()$. In Chapters 4 and 5 we study several implementations of $\mathscr{P}_{otr}$.

3.4 Conclusion

This chapter introduced the background material used in Chapters 4 and 5, namely the HO model and the OTR consensus algorithm.

4 Quantitative Analysis of Consensus Algorithms

Although the solvability of consensus is now a well-understood problem, comparing different algorithms in terms of efficiency is still an open problem. This chapter addresses this question for round-based consensus algorithms using communication predicates, on top of a partial synchronous system that alternates between good and bad periods (synchronous and non-synchronous periods). Communication predicates together with the detailed timing information of the underlying partially synchronous system provide a convenient and powerful framework for comparing different consensus algorithms and their implementations. This approach allows us to quantify the required length of a good period to solve a given number of consensus instances. With our results, we can observe several interesting issues, such as the number of rounds of an algorithm is not necessarily a good metric for its performance.

Publication: Fatemeh Borran, Martin Hutle, Nuno Santos and André Schiper. Quantitative Analysis of Consensus Algorithms. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 9(2):1545-5971, March/April 2012

Joint work with Fatemeh Borran.

4.1 Introduction

As established by the FLP impossibility result [FLP85], consensus cannot be solved deterministically in an asynchronous system with faults.

Over the years researchers have proposed several models where consensus can be solved deterministically. One such model is the partially synchronous system, where consensus can be solved with a majority of correct processes [DLS88]. Roughly speaking, a partially synchronous system may initially be asynchronous, but eventually becomes synchronous; links may be initially lossy, but eventually become reliable. Another such model is the failure detector model, which was introduced a few years later [CT96]. This model is defined as an asynchronous system "augmented" with a device called failure detector, defined by some completeness and accuracy properties (see [CT96] for details). Over the years the failure detector model has become very popular.

Although the solvability of consensus is now a well understood problem, the efficiency of consensus algorithms has not received the same attention and remains a poorly understood problem. The problem of analytical quantitative evaluation of consensus algorithms was initially addressed in [DLS88], in the context of a partially synchronous system. For the algorithms proposed in the paper, the authors compute upper bounds for the time needed after GST (Global Stabilization Time), *i.e.*, the time at which the system becomes synchronous, for all correct processes to decide. Unfortunately, the results in [DLS88] cannot be applied directly to modern consensus algorithms, as the algorithms in [DLS88] were mostly ignored by the research community, which instead has focused on Paxos-like algorithms [Lam98]. Moreover, rounds in [DLS88] are implemented on top of a clock synchronization algorithm, and while the algorithms are polynomial in the constants n, Δ and Φ, the authors made no effort to optimize these constants. In our work we show how to implement rounds without resorting to clock synchronization algorithms, which leads to more efficient implementations.

After this early work, analytical performance evaluation of consensus algorithms did not receive much attention for a while. This is probably due to the advent of failure detectors, which led to consider an asynchronous system as the underlying model, and to ignore timing analysis. One of the first papers to re-initiate analytical performance study of consensus algorithms in non-synchronous systems is [Sch97]. The paper considers failure detectors, and uses as metric the minimum number of communication steps for deciding in a "nice" run *i.e.*, a run with no crashes and no false suspicions).

Later, [DGK07], [KS06], and [AGGT08] study the performance of consensus algorithms expressed in a round-based computational model. The performance metric is the number of rounds needed for processes to decide once the system has become synchronous. However, as pointed out in [KS06], the efficiency expressed in terms of number of rounds does not predict the time it takes to decide after the system stabilizes. This observation is not followed in [KS06] by any analysis, even though the authors note that this is an interesting subject for further studies. Such a timing analysis is done in [DGL05] for a modified version of Paxos. The authors show that, with their modified Paxos algorithm, consensus can be solved in $O(\delta)$ after the system stabilizes (actually 17δ), where δ is the upper bound on message delivery time after stability is reached (δ includes the time needed to process a message after reception). Timing analysis is also done in [PLL97] for Paxos, but only for an execution started during a good period, and leader election outside of the Paxos algorithm (it uses a failure detector implementation in which every process sends periodically messages to all).

Contribution In this chapter we go beyond this work and propose a more detailed timing analysis of some consensus algorithms. We consider a partially synchronous system that alternates between periods of synchrony and periods of asynchrony (which includes the original definition), and compare, for various consensus algorithms, the window of synchrony that allows processes to decide. Such a timing analysis requires a model with time, which explains our choice of the partially synchronous model. Moreover, in order to decouple the timing analysis from irrelevant details of consensus algorithms, we do our analysis in the HO Round model which was presented in Chapter 3. Such a modular approach allows us not only to reuse the same timing analysis for different consensus algorithms, but also to compare various round implementations for the same round-based consensus algorithm.

For each consensus algorithm and different round implementations, we express the minimal period of synchrony, also called good period, that allows the algorithm to solve x instances of consensus as $\mathit{initialization} + x \cdot \mathit{per\text{-}consensus}$. In a good period, the *initialization* time is a one time duration that allows processes to synchronize to a specific round of the consensus algorithm; while *per-consensus* is the recurring duration for solving one instance of consensus. This allows us to highlight two extreme cases: "short" and "long" periods of synchrony. If the period of synchrony is long, the *initialization* cost is amortized over all instances of consensus, and can thus be ignored. This is not the case if the period of synchrony is short.

One important observation from our results is that the number of rounds of an algorithm is not necessarily a good metric for its performance. This justifies the detailed performance analysis done in this chapter. Our results also allow us to quantify the influence of the clock precision. We show that a large clock skew, as it is the case when using *e.g.*, step counting, has only limited influence for algorithms that try to resynchronize in every round, but can become unacceptable for algorithms that resynchronize less often.

Roadmap This chapter is structured as follows. In Section 4.2 we present the HO algorithms and the communication predicates analyzed in the rest of the chapter. After defining our system model for our implementations in Section 4.3, we give an abstract algorithm in Section 4.4 that serves as a generic implementation for many predicates, and which is used by all our implementations. After that, in Sections 4.5 to 4.7 we describe how our predicates can be implemented using different strategies, which lead to different lengths of the good period that is necessary to guarantee the predicate, and to different numbers of messages. We finally analyze our results in Section 4.8 and conclude the chapter in Section 4.9. To improve the presentation of this chapter, some of the proofs were moved to Appendix A.

4.2 Algorithms and Predicate Implementations Analyzed

In this chapter, we analyze three consensus algorithms that are safe by design (*i.e.*, they never violate the integrity or agreement properties of consensus) despite benign (non-Byzantine) faults, but require some predicate to ensure liveness. For each algorithm, we give one or more implementations for the corresponding communication predicates, which differ in terms of efficiency. Table 4.1 summarizes the algorithms and predicates.

Table 4.1: Algorithms and predicate implementations studied.

Alg.	Rounds	Resilience	Predicate Implementations
OTR	2	$3f+1$	$2 \times \mathscr{P}_u$
LV-3	3	$2f+1$	$3 \times \mathscr{P}_u$, Phase Sync, Piggybacking
LV-4	4	$2f+1$	$4 \times \mathscr{P}_u$, Coord Sync

Next we present the three HO algorithms studied in this chapter. The predicate implementations are presented and analyzed in Sections 4.5 to 4.7.

4.2.1 OneThirdRule (OTR)

The first algorithm analyzed is the OneThirdRule algorithm, which was already described in Section 3.3 (Algorithm 3.1). Recall that OTR does not use a coordinator, is always safe and requires the predicate $\mathscr{P}_{otr}$ for liveness. As we will see below, contrary to OTR, the other two algorithms considered in this chapter are both coordinator-based algorithms.

4.2.2 Variant of Paxos: LastVoting in four rounds (LV-4)

The second consensus algorithm we consider is a variant of Paxos [Lam98], called LastVoting [CBS09] (see Algorithm 4.1). It is a variant of Paxos in the sense that the algorithm is expressed here in a round model, which is not the way Paxos has been expressed [Lam98]. LastVoting is coordinator based, and each phase of LastVoting consists of four rounds $4\phi-3$ to 4ϕ, where ϕ denotes the current phase. Roughly speaking, round $4\phi-3$ corresponds to phase $1b$ of Paxos, round $4\phi-2$ to phase $2a$, and round $4\phi-1$ to phase $2b$. Phase $1a$ of Paxos is hidden in the implementation of round 4ϕ, for leader election. The termination property of consensus is guaranteed by the existence of a phase ϕ such that following predicate holds:

$$\begin{aligned}\mathscr{P}_{lv4}(\phi) :: \exists \Pi_0 \subseteq \Pi \ s.t.\ |\Pi_0| > n/2, \exists c \in \Pi, \forall p \in \Pi_0 :\\ &Coord(p,\phi) = c \ \wedge \\ &|HO(c,4\phi-3)| > n/2 \ \wedge \ c \in HO(p,4\phi-2) \ \wedge \\ &\Pi_0 \subseteq HO(c,4\phi-1) \ \wedge \ c \in HO(p,4\phi).\end{aligned}$$

which ensures, loosely speaking, agreement on the coordinator c during one phase ϕ, and communication between c and a majority of processes during phase ϕ.

4.2.3 Variant of Paxos: LastVoting in three rounds (LV-3)

LastVoting in three rounds is a well-known variant of Paxos in which the last two rounds of a phase are aggregated in a single round [Lam98], as shown by Algorithm 4.2: in round 3ϕ all processes send their ack message directly to all other processes, instead of via the coordinator.

Algorithm 4.1 LV-4: *LastVoting in four rounds* [CBS09].

```
Initialization:
    $x_p := v_p \in V$ /* $v_p$ is the initial value of $p$ */
    $vote_p \in V \cup \{?\}$, initially ?
    $commit_p$ a Boolean, initially false
    $ready_p$ a Boolean, initially false
    $ts_p \in \mathbb{N}$, initially 0

Round $r = 4\phi - 3$:
    $S_p^r$:
        send $\langle x_p, ts_p \rangle$ to $Coord(p,\phi)$

    $T_p^r$:
        if $p = Coord(p,\phi)$ and number of $\langle v,\theta \rangle$ received $> n/2$ then
            let $\overline{\theta}$ be the largest $\theta$ from $\langle -,\theta \rangle$ received
            $vote_p :=$ one $\overline{x}$ such that $\langle \overline{x},\overline{\theta} \rangle$ is received
            $commit_p :=$ true

Round $r = 4\phi - 2$:
    $S_p^r$:
        if $p = Coord(p,\phi)$ and $commit_p$ then
            send $\langle vote_p \rangle$ to all processes

    $T_p^r$:
        if received $\langle v \rangle$ from $Coord(p,\phi)$ then
            $x_p := v$
            $ts_p := \phi$

Round $r = 4\phi - 1$:
    $S_p^r$:
        if $ts_p = \phi$ then
            send $\langle ack \rangle$ to $Coord(p,\phi)$

    $T_p^r$:
        if $p = Coord(p,\phi)$ and number of $\langle ack \rangle$ received $> n/2$ then
            $ready_p :=$ true

Round $r = 4\phi$:
    $S_p^r$:
        if $p = Coord(p,\phi)$ and $ready_p$ then
            send $\langle vote_p \rangle$ to all processes

    $T_p^r$:
        if received $\langle v \rangle$ from $Coord(p,\phi)$ then
            DECIDE($v$)
        $commit_p :=$ false
        $ready_p :=$ false
```

LV-3 terminates in a phase ϕ satisfying the following predicate:

$$\begin{aligned}\mathscr{P}_{lv3}(\phi) :: \exists \Pi_0 \subseteq \Pi \ s.t.\ |\Pi_0| > n/2, \exists c \in \Pi, \forall p \in \Pi_0 :\\ Coord(p,\phi) = c \ \wedge \ |HO(c, 3\phi - 2)| > n/2 \ \wedge\\ c \in HO(p, 3\phi - 1) \ \wedge \ \Pi_0 \subseteq HO(p, 3\phi).\end{aligned}$$

Algorithm 4.2 LV-3: *LastVoting in three rounds* [CBS06].

```
Initialization:
  $x_p := v_p \in V$ /* $v_p$ is the initial value of $p$ */
  $vote_p \in V \cup \{?\}$, initially ?
  $commit_p$ a Boolean, initially false
  $ts_p \in \mathbb{N}$, initially 0

Round $3\phi - 2$: identical to round $4\phi - 3$ of Algorithm 4.1.

Round $3\phi - 1$: identical to round $4\phi - 2$ of Algorithm 4.1.

Round $r = 3\phi$:
  $S_p^r$:
    if $ts_p = \phi$ then
      send $\langle ack, x_p \rangle$ to all processes

  $T_p^r$:
    if $\exists v$ such that number of $\langle ack, v \rangle$ received $> n/2$ then
      DECIDE($v$)
    $commit_p :=$ false
```

4.3 System Model and Definitions

We describe now the system model for the implementation of the predicate layer. We consider a similar model as in [DLS88], with some modifications to reflect good periods of bounded length. Further, we use clocks instead of a bound on the maximum speed of processes, a more general approach, as we explain later.

Let $\Pi = \{p_1, \ldots, p_n\}$ be the set of processes, with $n > 2$. Processes are connected by a communication network, modeled for each $p \in \Pi$ by a variable $buffer_p$, which contains all messages that have been sent to p but were not yet received by p. Processes proceed by making steps, where a step is either a receive step or a send step:

- In a *send step*, a single message can be sent to another process in the system, that is, when process p executes send($\langle m \rangle, q$), the tuple $\langle m, p \rangle$ is placed in $buffer_q$.
- In a *receive step*, some messages are received, that is, when process p executes receive(S), a set $S \subseteq buffer_p$ is removed from $buffer_p$, and delivered to p. Note that S may be empty.

In each step, some computation can be done, and the local clock $C_p(t)$ of process p at real time t can be read. We assume that local clocks are monotonically non-decreasing at any time. Real time and the local clock take values from $\mathbb{R}$.

Definition 4.1 (Δ-timely message)**.** *A message m sent at time t by some process to a process p is called* Δ-timely*, if it is received at the latest by the first receive step of p at or after time $t + \Delta$.*

A link between processes p and q is said to be Δ-timely in an interval I if every message sent by p to q at a time $t \in I$ is a Δ-timely message, provided that $t + \Delta \in I$.

Process synchrony is ensured by making steps at a minimum rate. Note that in contrast to [DLS88], there is no restriction on the maximum speed of processes, since we will use clocks instead of step counting in order to measure time.

Definition 4.2 (Φ-synchronous process in interval I)**.** *A process is said to be Φ-synchronous in interval I if there is a bound Φ such that in any sub-interval of I of length Φ, p takes at least one step.*

Definition 4.3 (local (α, β) bounded-drift clock in interval I)**.** *A local clock $C_p(t)$ has a bounded drift in a time interval I, if there are a priori known constants α and β with $0 < \alpha \leq \beta$, so that for any two times t_1, $t_2 \in I$ s.t. $0 < t_1 < t_2$:*

$$\frac{C_p(t_2) - C_p(t_1)}{t_2 - t_1} \in [\alpha, \beta]. \tag{4.1}$$

Note that our clock definition is very general, since it encompasses other definitions like the classical bounded-drift clocks ($\alpha = 1 - \rho$, $\beta = 1 + \rho$), whereas the values $\alpha = 1/\Phi$, $\beta = 1$ are obtained asymptotically if step-counting is used for measuring time (this would require an upper bound on the frequency of steps, of course).

Definition 4.4 (good period)**.** *Let $\Pi_0 \subseteq \Pi$ be a set of processes. An interval I is a* good period *for Π_0, if there are a priori known bounds $\Phi, \Delta \in \mathbb{N}$, and $\alpha, \beta \in \mathbb{R}$, with $\Phi > 0$, $0 < \alpha \leq \beta$, such that (i) in I all processes in Π_0 are Φ-synchronous and have a local (α, β)-bounded-drift clock in I, (ii) no process that is not in Π_0 makes a step, (iii) all links between processes in Π_0 are Δ-timely, and (iv) no messages from processes not in Π_0 are received by a process in Π_0.*

A *k-good period* is a good period for some arbitrary Π_0 with $|\Pi_0| \geq k$. In the sequel, when k is clear from the context we will use only the term *good period.*

Note that we do not specify why processes outside Π_0 do not make steps; they might have crashed, be just temporarily unavailable, or be mute for any other reason. Therefore the notion of *correct* or *faulty* process is not suitable in our context; however, with respect to some Π_0-good period, we say a process is *up* in this good period iff it is in Π_0, else it is *down.*

Due to clock drift, a timeout measured by a process does not necessarily match the elapsed real time. Nevertheless, during good periods the clock drift is bounded, which allows us to bound the time measured by a process within a real time envelope. The following Lemma 4.1 shows the relation between real time and process time, which will be used in the rest of the chapter to set process timeouts:

Lemma 4.1. *In a good period, a time interval of length $\tau_C = \beta\tau_L$, measured by some process p, corresponds to a real time interval of length in $[\tau_L, \tau_U]$, with $\tau_U = \frac{\beta}{\alpha}\tau_L$.*

We will keep the notation of τ_L, τ_C, and τ_U consistent within the chapter to denote these different kind of durations.

Algorithm 4.3 Generic algorithm of the predicate layer

```
1:  $Rcv_p \leftarrow \emptyset$ /* set of messages received */
2:  $r_p \leftarrow 1$ /* round number */
3:  $s_p \leftarrow init_p$ /* state of the process $p$ */
4:  $coord_p \leftarrow \perp$ /* coordinator of process $p$ */
5:  $t_p \leftarrow C_p()$ /* timer */
6:  while true do
7:    $coord_p \leftarrow ElectCoord(p, r_p, C_p() - t_p, coord_p, Rcv_p)$
8:    if $\neg SkipRound(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ then
9:      $msgs \leftarrow S_p^{r_p}(s_p, coord_p)$
10:     for all $q \in Dest(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ do
11:       if $p = q$ then
12:         $Rcv_p \leftarrow Rcv_p \cup \{\langle msgs[p], p, r_p \rangle\}$ /* local delivery */
13:       else
14:         send($\langle msgs[q], r_p \rangle$, $q$)
15:     $t_p \leftarrow C_p()$
16:     while $\neg NextRound(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ do
17:       receive($S$)
18:       for all messages $\langle\langle x, r \rangle, q\rangle \in S$ do
19:         $Rcv_p \leftarrow Rcv_p \cup \{\langle x, q, r \rangle\}$
20:   $s_p \leftarrow T_p^{r_p}(\{\langle x, q \rangle \mid \langle x, q, r_p \rangle \in Rcv_p\}, s_p, coord_p)$
21:   $r_p \leftarrow r_p + 1$
```

4.4 The Generic Protocol

We give in this section a generic algorithm for the predicate layer, which is parametrized by four abstract functions. The instantiation of these functions will allow us to devise three different algorithms for the predicate layer that differ mainly by the message pattern and the way the coordinator is elected. The first method is called *Full Synchronization* (Section 4.5); it ensures uniform rounds, thereby allowing the implementation of all predicates considered in this chapter, including $\mathscr{P}_{otr}$ (Defined in Section 3.3). *Phase Synchronization* (Section 4.6) is an optimized implementation for $\mathscr{P}_{lv3}$, where round synchronization takes place only once per phase. Finally, *Synchronization by a Coordinator* (Section 4.7), where synchronization uses only messages from coordinator process(es), is specialized for $\mathscr{P}_{lv4}$.

The generic Algorithm 4.3 works as follows. One iteration of the **while** loop (line 6) corresponds to one round: the sending function is called at line 9 and the transition function is called at line 20. Messages are sent at line 14: the abstract function *Dest* (line 10) specifies the set of processes to which a message is sent in the current round. Message reception occurs at line 17. The receive statement is executed repeatedly until *NextRound* returns `true` (line 16). This typically happens when a timer has expired or when a message from some higher round is received. Note also that some rounds may be totally skipped (no message sent, no message received): this happens whenever the function *SkipRound* (line 8) returns `true`, which typically occurs if process p in round r_p receives a message from some round $r' > r_p$. In this case, p skips all rounds from r_p to $r' - 1$. Finally, function *ElectCoord* specifies how a coordinator for each round is determined.

Parametrization 4.1 A generic parametrization using full synchronization; where $ho(r) := \{q \mid \langle -, q, r\rangle \in Rcv\}$.

$$NextRound(p, r, \tau, coord, Rcv) := \bigvee \begin{cases} \exists \langle -, -, r'\rangle \in Rcv\colon\ r' > r \\ \tau \geq \tau_C \end{cases}$$

$$SkipRound(p, r, \tau, coord, Rcv) := \exists \langle -, -, r'\rangle \in Rcv\colon\ r' > r$$

$$Dest(p, r, \tau, coord, Rcv) := \Pi$$

$$ElectCoord(p, r, \tau, coord, Rcv) := \begin{cases} \min(\Pi) & : \quad r = 1 \\ \min(ho(r-1)) & : \quad ho(r-1) \neq \emptyset \\ coord & : \quad \text{else} \end{cases}$$

4.5 Full Synchronization

Our first implementation is given as Parametrization 4.1 for the generic algorithm. When a process starts a round, it sends a message to all. It then waits until either it receives a message from a higher round or the round timeout elapses. This implementation ensures uniform rounds in a good period, which “almost” ensures $\mathscr{P}_{lv4}$ (for LV-4) and $\mathscr{P}_{lv3}$ (for LV-3); only the election of the coordinator is missing. However, a uniform round r allows the election of a unique coordinator for round $r+1$ (the coordinator can be determined through a deterministic function on the HO sets of round r). Thus, uniformity of round $4\phi_0$ and of the four rounds of phase $\phi_0 + 1$ allows us to ensure $\mathscr{P}_{lv4}$, and uniformity of round $3\phi_0$ and of the three rounds of phase $\phi_0 + 1$ allows us to ensure $\mathscr{P}_{lv3}$. More generally, $2y$ consecutive uniform rounds ensure y instances of $\mathscr{P}_{otr}$, in the worst case $4y+4$ consecutive uniform rounds ensure y instances of $\mathscr{P}_{lv4}$, and $3y+3$ consecutive uniform rounds ensure y instances of $\mathscr{P}_{lv3}$.

With Parametrization 4.1, every process sends a message to every other process in all rounds (see function *Dest*). While sending to all in all rounds seems natural for $\mathscr{P}_{otr}$, where every process must hear from every alive process, this induces some overhead for $\mathscr{P}_{lv3}$ and $\mathscr{P}_{lv4}$, since these predicates only require one-to-all or all-to-one patterns on some of their rounds. This is shown in Figure 4.1 for predicate $\mathscr{P}_{lv3}$.

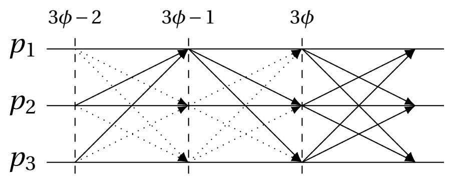


Figure 4.1: Message pattern of Full Synchronization. The full lines represent messages required by the $\mathscr{P}_{lv3}$ predicate and dotted lines represent the additional messages sent by the Full Synchronization predicate implementation (Parametrization 4.1).

In Section 4.6 and 4.7, we provide implementations for $\mathscr{P}_{lv3}$ and $\mathscr{P}_{lv4}$ with lower message complexity. The reminder of the section describes *Full Synchronization* in more detail, computes the timeout τ_C, and the length of a good period.

4.5.1 Outline of Full Synchronization

As shown by function *NextRound* in Parametrization 4.1, there are two ways for a process p to leave round r: (i) by receiving a message from a higher round $r' > r$, or (ii) by expiration of a timeout. In both cases, there is at least one process whose timeout for round r expires; this makes the protocol driven by timeout. In case (i), the process goes directly to round r'. Note that the function *SkipRound* and the first condition of *NextRound* play together to achieve this. In case (ii) the timeout τ_C is chosen to ensure uniformity, *i.e.*, $\mathscr{P}_u()$, in a good period (see Lemma 4.2 below). As shown by the function *ElectCoord*, the coordinator for some round r is the smallest process (min) in the HO set of round $r-1$ (whenever this HO set is non empty). Note the definition of the macro $ho(r)$ given in the caption of Parametrization 4.1; we will also use this notation in the following sections. This ensures a unique coordinator in good periods where rounds are uniform. For non-coordinated predicates, like $\mathscr{P}_u()$, no coordinator is needed and the function *ElectCoord* can be ignored.

4.5.2 Round Timeout

We first assume that a good period, which starts at some time t_g, holds forever, and show that the timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$ ensures $\mathscr{P}_u()$ for processes that are up in this good period. In the next subsection we compute the length of a good period that is sufficient to ensure $\mathscr{P}_u()$.

Lemma 4.2 (Timeout τ_C)**.** *Consider Parametrization 4.1 with the timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$. Assume that a k-good period starts at time t_g and holds forever, and that round r_0 is the highest round started by any process in Π_0 by time t_g. Then every new round $r > r_0$ started after time t_g is uniform ($\mathscr{P}_u(r)$) with cardinality k.*

Proof. We show that in round r, for every process $p \in \Pi_0$, we have: (i) p receives a message from all processes in Π_0, but (ii) not from any process not in Π_0.

We start with (ii). Assume that p received a round r message from a process q that is not in Π_0. By the definition of a good period, p could not have received this message after t_g. If p had received this message before t_g, then p would have advanced to round r immediately, which contradicts our assumption that no process in Π_0 has entered round r by t_g.

To prove (i), assume some process p_1 is the first to finish sending its round r messages at time $t_s > t_g$ (see Figure 4.2). These messages are ready for reception at each process in Π_0 (p_2 in Figure 4.2), the latest at $t_s + \Delta$, since messages are Δ-timely. These messages are received in the next receive step, which occurs the latest after $n-1$ send steps (in the case the process was just starting executing send steps). Since a step takes up to Φ time, p_1's message is received by all processes in Π_0 the latest at $t_s + \Delta + n\Phi$. Each process that receives this message jumps to round r, if not already there, and thus, by time $t_s + \Delta + (2n-1)\Phi$ has performed $n-1$ send steps and has sent its round r message to all. This message is ready for reception by the latest

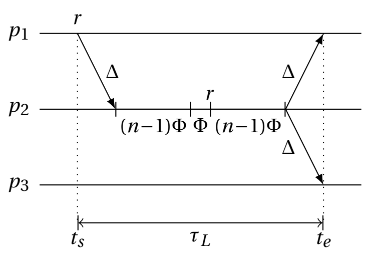


Figure 4.2: Full synchronization: Lemma 4.2.

at time $t_e = t_s + 2\Delta + (2n-1)\Phi$.

The timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$, together with Lemma 4.1, ensure that no timeout of length τ_C started at time t_s expires before t_e. So when the timeout expires, all messages for round r are either received or ready to be received. Before calling the transition function for round r, a receive step is performed; thus in round r every process in Π_0 receives a message from every process in Π_0. □

4.5.3 Length of a good period

The next theorem computes the required duration of a good period in order to ensure x consecutive uniform rounds. The special case $x = 0$ gives the initialization time, which for $\mathcal{P}_{otr}$ is the time from the start of the good period until the beginning of the first uniform round, which is $\frac{\beta}{\alpha}\Big(2\Delta + (2n-1)\Phi\Big) + 2n\Phi + \Delta$. Also from the formula, we can compute the time required for each uniform round after stabilization, which is $\frac{\beta}{\alpha}\Big(2\Delta + (2n-1)\Phi\Big) + n\Phi$.

Theorem 4.1. *In any good period of length*

$$(x+1)\left[\frac{\beta}{\alpha}\Big(2\Delta + (2n-1)\Phi\Big) + n\Phi\right] + \Delta + n\Phi$$

the generic algorithm with Parametrization 4.1 ensures x consecutive rounds that fulfill $\mathcal{P}_u()$.

Proof. Assume a good period starts at time t_g and at this time process p_1 has the highest round number r among the processes in Π_0. We distinguish two cases: (i) t_g is during these $n-1$ send steps (not shown in Figure 4.3) of round $r = 3\phi$. (ii) t_g after these send steps (see Figure 4.3). It can be shown that case (ii) is worse than case (i) in terms of length of the good period, thus we consider case (ii). Round $r+1$ is the first round that all processes in Π_0 start after t_g. According to Lemma 4.2, round $r+1$, $r+2$, etc. are uniform if the good period is long enough. We compute the maximum time it takes for any process p_2 to complete round $r+x$. As shown by Figure 4.3, p_2 starts round $r+1$ at latest at time $t_g + \tau_U + 2n\Phi + \Delta$ (end of *"initialization"* in Figure 4.3). This expression is obtained as follows: by the definition of p_1, no message of a round larger than r is received before p_1's timer expires, and $\tau_U = \frac{\beta}{\alpha}\tau_L$ is the time elapsed for a timeout $\tau_C = \tau_L\beta$; when the timeout expires, p_1 executes a receive step (ϕ),

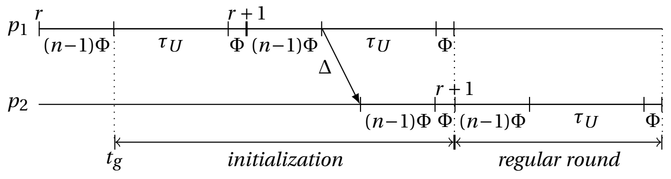


Figure 4.3: Full synchronization: Theorem 4.1.

moves to round $r+1$, executes $n-1$ send steps ($(n-1)\Phi$); in the worst case the message to p_2 is sent in the last of these send steps; Δ later the message is ready for reception on p_2; at this time p_2 may be executing n send steps ($(n-1)\Phi$) before the reception step (Φ) in which p_1's message is finally received; at this point p_2 moves to round $r+1$.

We now show that case (i) leads to a shorter good period. Here, by time $t_g + (n-2)\Phi$, at least one message of round r was sent by p_1. By time $t_g + (n-2)\Phi + \Delta + n\Phi$, this message is received by some process, and at the latest $(n-1)\Phi$ time later, this process has sent its round r messages to all. Thus, after time $t_g + 2\Delta + (4n-4)\Phi$, every process has performed its send steps for round r. Consequently, doing now the same analysis as in case (ii), it cannot be the case anymore that a process is performing send steps when the message for round $r+1$ is ready for reception. This leads to the fact that also in this case p_2 starts round $r+1$ not after time $t_g + \tau_U + \Delta + 2n\Phi$.

Process p_2 needs at most $n\Phi + \tau_U$ to complete round $r+1$ (see *"regular round"* in Figure 4.3): $n-1$ send steps ($(n-1)\Phi$), timeout τ_U (in the worst case no message of a larger round is received), one receive step (Φ).

Summing up the duration of "initialization" and of x "regular rounds" leads to $(x+1)[\tau_U + n\Phi] + \Delta + n\Phi$. Replacing τ_U with $\frac{\beta}{\alpha}\tau_L$, and τ_L with $2\Delta + (2n-1)\Phi$ (see Lemma 4.2) establishes the result. □

As mentioned at the beginning of Section 4.5, in the worst case for y instances of predicate $\mathscr{P}_{otr}$ we need $2y$ uniform rounds, for y instances of $\mathscr{P}_{lv3}$ we need $3y+3$ uniform rounds, and for y instances of $\mathscr{P}_{lv4}$ we need $4y+4$ uniform rounds. It follows that the initialization time of LV-3 Full Sync corresponds to the initialization time to get uniform rounds, plus the duration of three uniform rounds. After initialization, each instance of LV-3 Full Sync requires three uniform rounds. Applying a similar reasoning to LV-4, we have:

Corollary 4.1. *Let* $\vartheta = \frac{\beta}{\alpha}\left(2\Delta + (2n-1)\Phi\right) + n\Phi$*. The initialization time of LV-3 Full Sync (resp. LV-4 Full Sync) is* $4\vartheta + \Delta + n\Phi$ *(resp.* $5\vartheta + \Delta + n\Phi$*). After initialization, the duration of one instance of LV-3 Full Sync (resp. LV-4 Full Sync) is* 3ϑ *(resp.* 4ϑ*).*

4.6 Phase Synchronization

Full synchronization sends extra messages with respect to the "natural message pattern" induced by the predicates $\mathscr{P}_{lv3}$ and $\mathscr{P}_{lv4}$. In this section, we give an implementation for $\mathscr{P}_{lv3}$

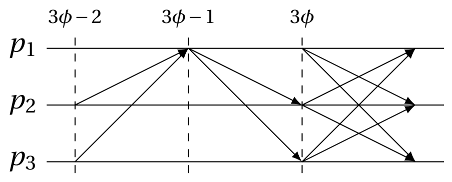


Figure 4.4: Message pattern of Phase Synchronization

Parametrization 4.2 LV-3 using phase synchronization; where $ho(r) := \{q \mid \langle -, q, r\rangle \in Rcv\}$

$$NextRound(p,r,\tau,coord,Rcv) := \bigvee \begin{cases} \exists\langle -,-,r'\rangle \in Rcv\colon\ r' > r \\ r \mod 3 = 1 \ \wedge\ (\tau \geq \tau_{C1} \ \vee\ |ho(r)| > n/2) \\ r \mod 3 = 2 \ \wedge\ \tau \geq \tau_{C2} \\ r \mod 3 = 0 \ \wedge\ \tau \geq \tau_{C3} \end{cases}$$

$$SkipRound(p,r,\tau,coord,Rcv) := \exists\langle -,-,r'\rangle \in Rcv\colon\ r' > r$$

$$Dest(p,r,\tau,coord,Rcv) := \begin{cases} coord & : \ r \mod 3 = 1 \\ \Pi & : \ r \mod 3 = 0 \ \vee\ (r \mod 3 = 2 \ \wedge\ p = coord) \\ \emptyset & : \ \text{else} \end{cases}$$

$$ElectCoord(p,r,\tau,coord,Rcv) := \begin{cases} \min(\Pi) & : \ r = 1 \\ \min(ho(r-1)) & : \ r \mod 3 = 1 \ \wedge\ ho(r-1) \neq \emptyset \\ coord & : \ \text{else} \end{cases}$$

that uses only the "natural" messages, which are illustrated in Figure 4.4.

The "natural" message pattern depicted in Figure 4.4 is generated by the function *Dest* in Parametrization 4.2. The key idea is to do process synchronization and coordinator election only at round 3ϕ of every phase ϕ, taking advantage of the all-to-all nature of this round of $\mathscr{P}_{lv3}$. The coordinator elected in round 3ϕ is then used for the following phase, that is, for phase $\phi + 1$.

Round 3ϕ of phase ϕ is identical to a round in the full synchronization case: all processes wait for the timeout before electing a new coordinator and moving to the first round of the next phase. Rounds $3\phi - 1$ and 3ϕ terminate always by expiration of the timeout as before, while round $3\phi - 2$ (line 2 in *NextRound*) may be terminated by the reception of messages.

For further details on Phase Synchronization, see [BHSS12]. This paper also includes an alternative implementation of $\mathscr{P}_{lv3}$ that uses piggybacking to reduce the total duration of a phase. In the following we will focus instead on the implementation of $\mathscr{P}_{lv4}$, which uses many of the same ideas as in Phase Synchronization.

4.7 Synchronization by a coordinator

The predicate for LV-4 can potentially be implemented with a lower message complexity than LV-3 and OTR, since $\mathscr{P}_{lv4}$ requires $4n$ messages (all rounds are one-to-all or all-to-one), while $\mathscr{P}_{lv3}$ requires $2n + n^2$ and $\mathscr{P}_{otr}$ requires $2n^2$. However, if we use full synchronization

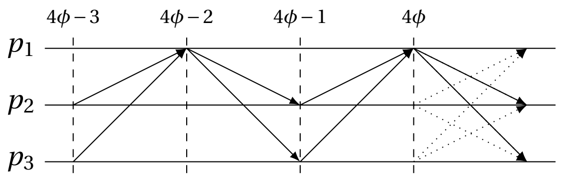


Figure 4.5: Message pattern of Synchronization by Coordinator

to implement $\mathscr{P}_{lv4}$, the resulting implementation will have $O(n^2)$ message complexity, and will be worse than LV-3 in terms of message complexity or length of the good period, because LV-4 requires one more round per phase. In this section we give another implementation that achieves a message complexity of $O(n)$ instead of $O(n^2)$ per regular phase during a good period, at the cost of a slightly larger length of the good period.

Contrary to $\mathscr{P}_{lv3}$, the predicate $\mathscr{P}_{lv4}$ does have any round all-to-all round that can be used to synchronize processes and elect a coordinator for "free", *i.e.*, without sending additional messages. Without such a round, these tasks have to be performed by sending additional messages in some round. As with $\mathscr{P}_{lv3}$ we do this only once per phase, in the last round of a phase. This leads to the message pattern depicted in Figure 4.5.

The messages represented by a full line are required by $\mathscr{P}_{lv4}$, while the messages represented by a dotted line in round 4ϕ are only for synchronization and election of a coordinator. As we explain below, once the system stabilizes and the processes synchronize, the additional messages are no longer sent, bringing down the number of per-phase messages to $4n$.

Section 4.7.1 describes the implementation in more detail. The values for timeouts are presented in Section 4.7.2 and the length of a good period is proved in Section 4.7.3.

4.7.1 Outline of Synchronization by a Coordinator

Our algorithm requires only n messages in round 4ϕ during a good period. This optimization is based on the following two observations: (i) to choose a coordinator, it is enough for the HO sets to be non-empty, as long as the round is uniform, and (ii) to synchronize to the same round in a good period it is enough if all processes receive a message from the process with the highest round.

Based on these observations, it is easy to see that in addition to the coordinator, only the processes that entered round 4ϕ by timeout need to send a message to all (line 3 of *Dest* in Parametrization 4.3). Otherwise, if a process p receives a round 4ϕ message while in a lower round, p can advance to round 4ϕ silently, since there is at least another process that sent a message to all in round 4ϕ and, therefore, can be chosen as coordinator. This strategy results in a message complexity of cn for the election in round 4ϕ, where c is the number of processes that compete to become coordinator. After the first election in a regular phase during a good period we have $c = 1$, since all processes will receive a round 4ϕ message from the coordinator

Parametrization 4.3 LV-4 using synchronization by coordinator; where $ho(r) := \{q \mid \langle -, q, r\rangle \in Rcv\}$

$$NextRound(p,r,\tau,coord,Rcv) := \bigvee \begin{cases} \exists \langle -,-,r'\rangle \in Rcv\colon\ r' > r \\ r \mod 4 = 0 \ \wedge\ \tau \geq \tau_{C4} \\ r \mod 4 = 1 \ \wedge\ (\tau \geq \tau_{C1} \ \vee\ |ho(r)| > n/2) \\ r \mod 4 = 2 \\ r \mod 4 = 3 \ \wedge\ (\tau \geq \tau_{C3} \ \vee\ |ho(r)| > n/2) \end{cases}$$

$$SkipRound(p,r,\tau,coord,Rcv) := \bigvee \begin{cases} \exists \langle -,-,r'\rangle \in Rcv\colon\ r' > r \\ r \mod 4 = 2 \ \wedge\ (p = coord \ \wedge\ |ho(r-1)| \leq n/2) \\ r \mod 4 = 3 \ \wedge\ coord \notin ho(r-1) \end{cases}$$

$$Dest(p,r,\tau,coord,Rcv) := \begin{cases} coord & \text{for } r \mod 4 \in \{1,3\} \\ \Pi & \text{for } r \mod 4 = 2 \wedge p = coord \\ \Pi & \text{for } r \mod 4 = 0 \wedge ho(r) = \emptyset \\ \emptyset & \text{else} \end{cases}$$

$$ElectCoord(p,r,\tau,coord,Rcv) := \begin{cases} \min(\Pi) & \text{for } r = 1 \\ \min(ho(r-1)) & \text{for } r \mod 4 = 1 \ \wedge\ ho(r-1) \neq \emptyset \\ coord & \text{else} \end{cases}$$

while in round $4\phi - 1$.

Although the algorithm has four rounds, it does not require four timeouts. This is because round $4\phi - 2$ can be message driven, *i.e.*, it terminates upon reception of a message from the coordinator (see line 4 in *NextRound*). Moreover, rounds $4\phi - 3$ and $4\phi - 1$ of the coordinator are also message driven (see lines 3 and 5 in *NextRound*).

To reduce the time needed to start a phase in which $\mathscr{P}_{lv4}$ might hold, our algorithm skips some rounds of phase ϕ if it detects that $\mathscr{P}_{lv4}(\phi)$ cannot hold. This can happen in two cases: (i) in round $4\phi - 2$ by the coordinator if it does not receive a majority of messages during round $4\phi - 1$ (*SkipRound*, line 2), and (ii) in round $4\phi - 1$ by any process if it does not receive a message from the coordinator in round $4\phi - 2$ (*SkipRound*, line 3).

4.7.2 Round Timeouts

Lemmas 4.3, 4.4 and 4.5 below establish results about timeouts τ_{C1}, τ_{C3} and τ_{C4}.

Lemma 4.3 (Timeout τ_{C1})**.** *Consider Parametrization 4.3 with* $\tau_{C1} = [3\Delta + 3n\Phi]\,\beta + \tau_{C4}\left(\frac{\beta}{\alpha} - 1\right)$*. Assume every process starts round* $4(\phi - 1)$ *in a* $\left(\frac{n+1}{2}\right)$*-good period and phase* ϕ *has a unique coordinator* c*. Then (i)* c *hears from a majority of processes in round* $4\phi - 3$*, and (ii) all processes in* Π_0 *hear from* c *in round* $4\phi - 2$*.*

Lemma 4.4 (Timeout τ_{C3})**.** *Consider Parametrization 4.3 with the timeout* $\tau_{C3} = [3\Delta + 2n\Phi]\beta$*. Assume every process starts round* $4(\phi - 1)$ *in a* $\left(\frac{n+1}{2}\right)$*-good period, and* ϕ *has a unique coordinator* c*. Then (i)* c *hears from a majority of processes in round* $4\phi - 1$*, and (ii) all processes in* Π_0 *hear from* c *in round* 4ϕ*.*

Lemma 4.5 (Timeout τ_{C4})**.** *Consider Parametrization 4.3 with the timeout* $\tau_{C4} = [2\Delta + (2n - 3)\Phi]\beta$*. Assume that a* k*-good period,* $k \geq 1$*, starts at time* t_g *and that round* r_0 *is the highest*

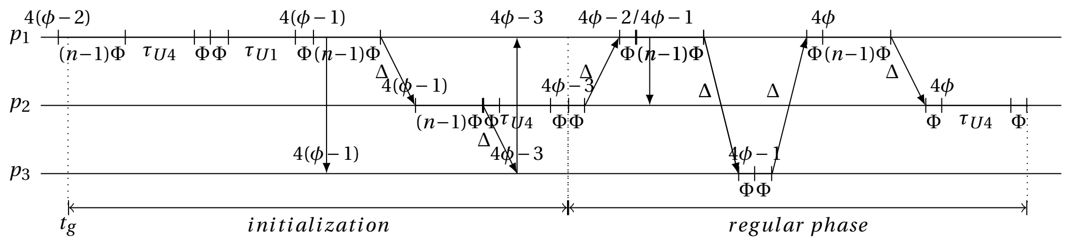


Figure 4.6: Synchronization by a Coordinator: Theorem 4.2.

round started by any process in Π_0 *by time* t_g. *Then every round* $4\phi > r_0$ *started after time* t_g *is uniform with non-zero cardinality.*

Note that the timeout τ_{C4} is different from the one in Lemma 4.2, since in some cases, processes do not send messages in this round.

Corollary 4.2. *Parametrization 4.3 with the timeout* $\tau_{C1} = [\Delta+(n+3)\Phi]\beta + [2\Delta+(2n-3)\Phi]\frac{\beta^2}{\alpha}$, $\tau_{C3} = [3\Delta+2n\Phi]\beta$, *and* $\tau_{C4} = [2\Delta+(2n-3)\Phi]\beta$ *ensures* $\mathscr{P}_{lv4}(\phi)$, *if round* $4(\phi-1)$ *starts in a* $\left(\frac{n+1}{2}\right)$*-good period.*

Proof. By Lemma 4.5, round $4(\phi-1)$ is uniform with non-zero cardinality. Thus by the definition of *ElectCoord*, every process in Π_0 has the same coordinator. Applying Lemma 4.3 to 4.5 and solving the equations for τ_{C1}, τ_{C2}, and τ_{C4} yields the result. □

4.7.3 Length of a good period

The next theorem computes the required duration of a good period in order to ensure y consecutive phases of $\mathscr{P}_{lv4}(\phi)$. The initialization time is given by setting $y = 0$, and the time per phase by the multiplication factor of y.

Theorem 4.2. *In any good period of length*

$$y\left[\left(2\Delta+(2n-3)\Phi\right)\frac{\beta}{\alpha}+4\Delta+(2n+5)\Phi\right]+ \\ +\left(2\Delta+(2n-3)\Phi\right)\frac{\beta^2}{\alpha^2}+\left(5\Delta+(5n-3)\Phi\right)\frac{\beta}{\alpha}+\Delta+(3n+1)\Phi$$

the generic algorithm with Parametrization 4.3 ensures y consecutive phases ϕ *that fulfill* $\mathscr{P}_{lv4}(\phi)$.

We present here an informal correctness argument for the length $6y\Delta+8\Delta$ (we ignore the Φ terms). The complete proof can be found in Appendix A.

Initialization period. It can be shown that the "initialization" period is the longest in the following case: (i) t_g starts just after the first send step of the highest round $4(\phi-2)$ reached by some process p, and (ii) round $4(\phi-2)$ is not uniform (only round $4(\phi-1)$ is uniform). In this case p will go through the full timeouts of round $4(\phi-2)$ and $4(\phi-1)-3$, which takes 5Δ. By this time, say t_0, if no other process has started round $4(\phi-1)$, process p will do so, skipping rounds $4(\phi-1)-2$ and $4(\phi-1)-1$ (see lines 2 and 3 of *SkipRound* in Parametrization 4.3). At latest at $t_0+\Delta$ the coordinator will start round $4(\phi-1)$; its round $4(\phi-1)$ message is ready for reception Δ later. Round $4(\phi-1)$ terminates by the expiration of a timeout (2Δ), see lines 2 of *NextRound* in Parametrization 4.3, which means that the coordinator message will be received only at $t_0+3\Delta$. So latest at $t_0+3\Delta$ all processes in Π_0 have finished round $4(\phi-1)$, which ends the initialization period and starts a regular phase. Thus, the initialization period lasts for 8Δ and a regular phase starts at $t_g+8\Delta$.

Regular phase. We show that the duration of a regular phase is 6Δ. Rename the regular phase to ϕ. Let the last process enter the regular phase ϕ at time t_r. Then by $t_r+\Delta$ the coordinator has a majority of round $4\phi-3$ messages, and 2Δ later the coordinator receives the round $4\phi-1$ messages from all processes in Π_0. By $t_r+4\Delta$ the coordinator message of round 4ϕ is ready for reception at all processes in Π_0. Round 4ϕ terminates by the expiration of a timeout (2Δ). So by $t_r+6\Delta$ all processes have decided.

4.8 Comparison

In this section we compare quantitatively the algorithms analyzed above. However, before doing so, we would like to clarify the scope of our results. Our analysis and conclusions are valid for the round-based algorithms above and can be easily adapted to other round-based algorithms. However, the results do not apply directly to failure detector based algorithms [CT96] or Paxos-like protocols [Lam98]. The key difference is that round-based protocols — like the ones presented in this chapter — are usually driven by timeouts, *i.e.*, these protocols require at least one process to expire its round timeout in every round, in order to proceed to the next round. In contrast, algorithms for the asynchronous or partially synchronous model usually proceed as fast as messages are sent and received, *i.e.*, they are message driven [CT96, Lam98].[1]

4.8.1 Impact of clock precision

First, we analyze the impact of the clock on τ_{good}, the duration of a good period that is sufficient to solve consensus. In order to simplify the comparison, we make the reasonable assumption $\Phi \ll \Delta$; this allows us to ignore the terms in Φ. The results are shown in Figure 4.7. The x-axis corresponds to β/α (see Sect. 4.3). Larger values of β/α correspond to larger variations in clock skew (worse clock *precision*); identical clock skew (including perfect clocks for which $\alpha=\beta=1$) correspond to $\beta/\alpha=1$.

[1] In Chapter 5 we show that round-based protocols are not inherently timeout-driven. In fact, it is possible to implement the round-structure in such a way that round-based protocols eventually start progressing as fast as messages are sent and received, without having to wait for timeouts to expire.

The y-axis corresponds to τ_{good}/Δ, which is the duration of a good period expressed using Δ as the time unit. Figure 4.7 shows that OTR is less sensitive to clock imprecision than the other

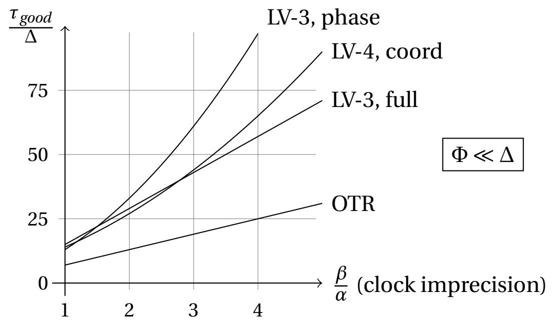


Figure 4.7: Duration of good period as a function of clock drift

algorithms. It shows another interesting result, namely that with perfect clocks, the different implementations of the *LastVoting* algorithm lead to almost the same result. This is no more the case with large clock imprecision (which occurs for instance when clocks are built from step counting and Φ large). We note also that the performance of algorithms that synchronize more often (like LV-3 with full synchronization) degrades less quickly with less precise clocks.

4.8.2 Analysis of the results for precise clocks

We do now a finer analysis of the different algorithms for the case $\beta/\alpha = 1$. We compare algorithms not only in terms of the duration of a good period, but also in terms of the duration of initialization after a good period starts.

Overview

Table 4.3 (at the end of the chapter) is an overview of the results obtained in this chapter for all three algorithms (OTR, LV-3 and LV-4). For OTR (predicate $\mathscr{P}_{otr}$), we have one single option to consider, namely two uniform rounds ($\mathscr{P}_u$). For LV-3 (predicate $\mathscr{P}_{lv3}$), we have three options: three uniform rounds (line 2), Phase Sync (line 3) and Piggybacking (line 4). Finally for LV-4 (predicate $\mathscr{P}_{lv4}$), we have two options: four uniform rounds (line 5) and Coord Sync (line 6), which is designed specifically for $\mathscr{P}_{lv4}$. For each option, Table 4.3 shows the initialization time, the time for each consensus after initialization, and the number of messages required for initialization and for each consensus. The time until the first decision after the beginning of a good period can be determined by summing the columns *initialization* and *per consensus*.

Line 1 (OTR) follows from the beginning of Section 4.5 (*e.g.*, Theorem 4.1 with $x = 0$ for the initialization time, time for two rounds for consensus). Line 2 (LV-3, $3 \times \mathscr{P}_u$) follows from Corollary 4.1. Line 3 (LV-3, Phase Synch) and line 4 (LV-3, Piggybacking) follow from the results in [BHSS12]. Line 5 (LV-4, $4 \times \mathscr{P}_u$) follows from Corollary 4.1 and, finally, line 6 (LV-4 Coord Synch) follows from the beginning of Section 4.7.3.

Table 4.2: Average round duration ($\beta/\alpha = 1$, $\Phi \ll \Delta$).

		Short good periods		Long good periods	
Alg.	# rounds	1st decision	$\Delta/rounds$	per-consensus	$\Delta/rounds$
OTR	2	7Δ	3.5	4Δ	2
LV-3 (Piggybacking)	3	12Δ	4	4Δ	1.3
LV-4 (Coord Sync)	4	14Δ	3.5	6Δ	1.5

Impact of the round implementation

We see from Table 4.3 that the performance of our algorithms, in terms of the length of a good period, varies significantly depending on the round implementation. For example, the generic implementations of $\mathscr{P}_{lv3}$ and $\mathscr{P}_{lv4}$, based on uniform rounds (lines 2 and 5), perform clearly worse than the implementations designed specifically for the corresponding predicates (lines 3, 4 and 6). More precisely, for $\Phi \ll \Delta$, the initialization time of $\mathscr{P}_{lv3}$ over uniform rounds is 9Δ and the decision time is 6Δ (line 2), while an improved implementation of this predicate (Piggybacking) achieves 8Δ and 4Δ respectively (line 4), while sending a smaller number of messages.

This shows that the round layer implementation plays a crucial role in the performance of round-based algorithms. Specifically, the simplest option (generic uniform rounds) does not lead to the most efficient solution. This is because algorithms like LV-3 and LV-4 do not require uniformity in all rounds. For these two algorithms, the first two rounds of a phase only require sending messages between a coordinator and all processes, which as shown can be implemented more efficiently (both in terms of length of a good period and message complexity) than generic uniform rounds. Moreover, looking at the rounds of a phase together instead of separately provides additional cross-round optimization opportunities.

A consequence of using round implementations tailored to the communication predicate is that the number of communication rounds of an algorithm is no longer a good metric of its performance. When using a generic round implementation (like $\mathscr{P}_u$), where every round is of the same duration, the performance can be easily estimated as (*round duration*) × (*#rounds of algorithm*). But this is no longer the case with optimized round implementations like Phase Sync, Piggybacking or Coord Sync, where the duration of a round differs between predicate implementations and between rounds of the same algorithm. This is confirmed by Table 4.2, which shows the average round duration ("$\Delta/rounds$") for each of our algorithms, considering the best round implementation for LV-3 and LV-4 (both in terms of length of a good period and message complexity). The table shows the results both for short good periods and long good periods. As we have mentioned in the introduction, a period of synchrony is "short" if it allows only a few decisions, in which case the initialization time is important. A period of synchrony is "long" if the number of decisions taken is large enough so that the initialization time is amortized over many consensus instances, and can be ignored. The results show some variation on the average duration for short good periods, ranging from 3.5Δ (OTR and LV-4) to 4Δ (LV-3), and a large variation for long good periods, from 1.3Δ (LV-3) to

2Δ (OTR).

Quantitative comparison of OTR, LV-3 and LV-4

Let us go back to Table 4.3 in order to compare the best implementations of our three consensus algorithms: OTR (only one implementation), LV-3 (Piggybacking) and of LV-4 (Coord Sync). We consider again the (reasonable) assumption $\Phi \ll \Delta$. OTR with $2 \times \mathscr{P}_u$ is the algorithm with the shortest initialization time, namely 3Δ, compared to 8Δ for LV-3 and LV-4. OTR has also the shortest time until the first decision (7Δ) and the shortest time per consensus after initialization (4Δ). However, this comes at the cost of requiring a greater number of replicas ($3f+1$ for OTR, compared to $2f+1$ for LV-3 and LV-4), and of a slightly higher message complexity ($2n^2$ for OTR, n^2+2n for LV-3 and $4n$ for LV-4). Among the algorithms that have a resilience of $2f+1$, LV-3 has the same initialization time as LV-4 (Coord Sync), but a lower per consensus time (4Δ instead of 6Δ). This is to be expected, as LV-3 and LV-4 differ on the last rounds: one all-to-all round for LV-3 *vs.* two rounds for LV-4 (all-to-coordinator and an coordinator-to-all). On the other hand, LV-4 (Coord Sync) is the algorithm with the lowest message complexity, with only $4n$ messages per consensus, while all other algorithms have at least one round with n^2 messages.

The analysis clearly shows that none of the algorithms is the best choice in every situation. In unstable networks, where the good periods are of short duration, OTR is the best algorithm, as it requires the shortest good period for the first decision. On stable networks, with long good periods, OTR and LV-3 are similar in terms of time per consensus. Since LV-3 is more resilient and has a slightly lower message complexity, it is a better choice. If the number of messages is important and the network is stable, then it is worth considering LV-4, as it requires only $4n$ messages.

4.9 Conclusion

In this chapter we have derived analytical performance results for several round-based consensus algorithms (OTR, LV-3, LV-4) in a system that alternates between good and bad periods. We have considered different implementations of rounds, and have computed for each algorithm (i) the time from the beginning of a good period until the first decision, and (ii) the time for each additional decision. The results show that the performance of round-based algorithms largely depends on the implementation of rounds. The results also show the number of rounds of an algorithm is not always a good metric for the performance of an algorithm. Finally we can observe trade-offs in resilience, minimum duration of a good period, decision time in the case of long good periods, and message complexity.

	Alg.	# rounds	Resilience	Pred. Impl.	length of good period		# messages	
					initialization	per consensus	initialization	per consensus
1	OTR	2	$3f+1$	$2\times\mathscr{P}_u$	$3\Delta+(4n-1)\Phi$	$4\Delta+(6n-2)\Phi$	n^2	$2n^2$
2	LV-3	3	$2f+1$	$3\times\mathscr{P}_u$	$9\Delta+(13n-4)\Phi$	$6\Delta+(9n-3)\Phi$	$4n^2$	$3n^2$
3	LV-3	3	$2f+1$	Phase Sync	$8\Delta+(12n-1)\Phi$	$5\Delta+(7n+2)\Phi$	$2n^2+2n$	n^2+2n
4	LV-3	3	$2f+1$	Piggybacking	$8\Delta+(12n-1)\Phi$	$4\Delta+(4n+1)\Phi$	$2n^2+2n$	n^2+2n
5	LV-4	4	$2f+1$	$4\times\mathscr{P}_u$	$11\Delta+(12n-5)\Phi$	$8\Delta+(9n-4)\Phi$	$5n^2$	$4n^2$
6	LV-4	4	$2f+1$	Coord Sync	$8\Delta+(10n-5)\Phi$	$6\Delta+(4n+2)\Phi$	$2n^2+3n$	$4n$

Table 4.3: Summary of results for $\beta/\alpha=1$.

5 Swift Algorithms for Repeated Consensus

We introduce the notion of a *swift algorithm*. Informally, an algorithm that solves the repeated consensus is swift if, in a partially synchronous run of this algorithm, eventually no timeout expires, *i.e.*, the algorithm execution proceeds with the actual speed of the system. This definition differs from other efficiency criteria for partially synchronous systems.

Furthermore, we show that the notion of swiftness explains why failure detector based algorithms are typically more efficient than round-based algorithms, since the former are naturally swift while the latter are naturally non-swift. We show that this is not an inherent difference between the models, and provide a round implementation that is swift, therefore performing similarly to failure detector algorithms while maintaining the advantages of the round model.

Publication: Fatemeh Borran, Martin Hutle, Nuno Santos and André Schiper. Swift Algorithms for Repeated Consensus. In *29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*, New Delhi, India, October 31 - November 3, 2010.

Joint work with Fatemeh Borran.

5.1 Introduction

Timeouts are often required to solve problems in distributed computing. Due to the FLP impossibility result [FLP85], there is a need of some minimal synchrony assumptions for solving the consensus problem, and timeouts are the dominant mechanism for algorithms to make use of synchrony assumptions.

Timeouts are often chosen conservatively, so that an algorithm is correct for a large number of real-life scenarios. However, timeouts should be used only to cope with faults, and not slow down the execution time in good cases. As an example, when implementing communication-closed synchronous rounds in a synchronous message passing system, after a process sent its messages for a certain round it usually waits for a timeout, before it terminates the round and sends its messages for the next round. However, in many runs of the algorithm, a process might have received all messages from other alive processes already long before that. It would be favorable to start the next round immediately after all messages from correct processes are received. This is, for example, the case for an algorithm that uses a $\Diamond\mathcal{P}$ failure detector (FD). Here, a process waits for a message from some process p until p is in the FD output. If p has crashed, this involves waiting for a timeout, but only once: later rounds profit from the fact that the failure detector "remembers" information about faults. We formally capture such a behavior by the definition of *swift*, which we define in the context of repeated consensus [DGDF$^+$08]. The main intuition behind our definition is that swift algorithms make progress at the speed of the system, and therefore, are more "efficient" than non-swift algorithms. A swift algorithm for a repeated problem is thus one in which eventually all instances of the problem are "efficient".

In more detail, for the definition of swift we look at partially synchronous runs, *i.e.*, runs where a bound Δ on the transmission delay eventually holds forever. Note that such a run exists also in an asynchronous system, and all runs of a synchronous systems are of course also partially synchronous. The definition is thus not limited to partially synchronous systems. For the good period of such a run, that is the partial run R in which bound Δ holds, we can define the actual transmission delay $\delta(R)$ as the maximum of all transmission delays in R. Such an actual transmission delay can be much smaller than the bound Δ. If in this case the execution time for each instance of the repeated consensus eventually depends only on $\delta(R)$ (in contrast to Δ), then the algorithm is said to be *swift*.

While intuitively swift algorithms progress at the speed of messages in good periods, and non-swift algorithms progress sometimes only by the expiration of timeouts, we refrained from calling these two classes of algorithms *message-driven* and *timeout driven*. This is because the term *message-driven* is used in [HW05, BW09] with a different meaning, namely to refer to the way events are generated at a process. If processes are allowed to measure time (*e.g.*, with clocks or step counting), then it is possible to construct message-driven algorithms (according to this definition) that are not swift. On the other hand, if processes use an adaptive timeout, then the algorithm can be swift despite timeout expiration. Thus these terms are not suitable to precisely characterize this class of algorithms.

Other notions of efficiency for distributed algorithms have been considered. The term *fast* has been used to refer to (consensus) algorithms that solve consensus with less communication steps in favorable cases [Lam06]. A favorable case corresponds usually to an execution without faults that is synchronous from the beginning. On the contrary, the definition of swift is related to the execution *time* of an algorithm in the context of *repeated* consensus. Furthermore, the definition of swift considers also runs with faults. The notion of fast is orthogonal to the notion of swift: it is possible to design both, fast algorithms that are swift and fast algorithms that are not swift. The same argument holds for *early terminating* algorithms [Lyn96].

Contribution This chapter makes the following two contributions. The first is the definition of swift algorithms that we just discussed. The second contribution is a new implementation of communication-closed rounds in a partially synchronous system with crash faults. This new implementation leads to swift round-based consensus algorithms, while previous round implementations, including those described in [DLS88, Gaf98] are not swift. This result is especially relevant in the context of comparing advantages and drawbacks of the failure detector approach [CT96] with the round-based approach [DLS88, CBS09] for solving agreement problems. Indeed, failure detector based algorithms, despite the usage of timeouts in the implementation of the failure detector algorithm, are naturally swift. On the other hand, round implementations in a partially synchronous model have some advantages over FD based implementations [HS07]. Our new solution thus combines the advantages of both approaches.

Roadmap The rest of the chapter is structured as follows. In Section 5.2, we specify our model and give a formal definition of *swift*. Then, in Section 5.3 we show a simple round-based consensus algorithm that is not swift, and in Section 5.4 we show that the same consensus algorithm expressed using a failure detector is swift. In Section 5.5 we present our main contribution: a new implementation of rounds that is swift. Section 5.6 validates the theoretical analysis with experimental results comparing the swift and non-swift implementations. Section 5.7 concludes the chapter. To improve the presentation of this chapter, some of the proofs were moved to Appendix B.

5.2 Definitions and model

We consider a system of n processes connected by a message-passing network. Among these n processes, at most f may crash. We attach an in-queue and an out-queue to each process, where for repeated consensus, the in-queue contains the consensus proposals, and the out-queue contains the consensus decisions. We denote with In_p (resp. Out_p) the in-queue (resp. out-queue) of process p. Processes execute an algorithm by taking steps, where a step can be one of the following:

- *Send step* $\langle p, \textsc{Send}, m\rangle$: process p sends a message m to another process;
- *Receive step* $\langle p, \textsc{Receive}, S\rangle$: process p receives a (possibly empty) set S of messages;
- *Input step* $\langle p, \textsc{In}, I\rangle$: process p reads a value from its in-queue;
- *Output step* $\langle p, \textsc{Out}, O\rangle$: process p outputs a value to its out-queue.

In each step a process also performs a state transition.

We assume an abstract global discrete time. Without loss of generality, at each time t at least one process makes a step. A single process can make at most one step at any time. Processes measure time by counting their own steps.

Channels satisfy validity and integrity:

- *Validity:* A message m that is received by q was previously sent by some process p to q;
- *Integrity:* A message m that is sent from p to q is received by q at most once.

Channels are reliable if additionally the following property holds:

- *Reliability:* If message m is sent from p to q and q performs an infinite number of receive steps, then eventually m is received by q.

We consider partially synchronous runs, defined by a bound Φ on the process relative speeds and a bound Δ on the transmission delay of messages [DLS88]. For a run R, we say that the *process speed bound* Φ holds in R if, in any partial run of R that contains Φ steps, every non-crashed process makes at least one step. Further, we say that the *transmission delay* Δ holds in R after some time t_0 if (i) any message sent by p to q at time $t \geq t_0$ is received the latest in the first receive step after $t + \Delta$; and (ii) every message sent before t_0 is received the latest in the first receive step after $t_0 + \Delta$.

Definition 5.1 (Partial synchrony)**.** *A run R is* (Δ, Φ)-partially synchronous *if there is a time GST (Global Stabilization Time) such that after GST the transmission delay bound Δ holds, the process speed bound Φ holds, and no process crashes after GST.*

We call the time interval (GST, ∞) the *good period of R*. We say a *system* is (Δ, Φ)*-partially synchronous* if every run R of the system fulfills Definition 5.1. To simplify the presentation, we assume $\Phi = 1$, and write Δ-partially synchronous for $(\Delta, 1)$-partially synchronous.

Definition 5.2 (Actual parameters)**.** *Let R' be a partial run. Then $\delta(R')$ denotes the maximum transmission delay of the partial run R', i.e., the smallest value $\overline{\delta}$ such that the transmission delay is bounded by $\overline{\delta}$ in the partial run R'.*

If R' is the good period of a Δ-partially synchronous system, then $\delta(R') \leq \Delta$. When R' is clear from the context, we simply write δ. The bound Δ may be known or unknown. For the algorithms in this chapter, we assume that Δ is known. However, δ is unknown, since it represents a performance metric of a single *run.*

5.2.1 Repeated consensus

We focus on the repeated consensus problem. The in-queue and out-queue are queues of pairs $\langle i, v \rangle$, where i is a consensus instance number and v a value. In the repeated consensus problem, for each instance i, the following holds:

- *Validity*: For every process p, if $\langle i, v \rangle \in Out_p$ then there exists some process q such that $\langle i, v \rangle \in In_q$.
- *Uniform agreement:* For all processes p, q, if $\langle i, v \rangle \in Out_p$ and $\langle i, v' \rangle \in Out_q$ then $v = v'$.
- *Termination:* For every correct process p there exists v such that $\langle i, v \rangle \in Out_p$.

5.2.2 Swift algorithms

Before giving a formal definition of swift, we need to formalize the notion of execution time of an instance of consensus.

Definition 5.3 (Execution time)**.** *Consider a run R of a repeated consensus algorithm. The execution time $\tau_i(R)$ of instance i of consensus is defined as follows. Let $t_{in} = \max\{t : \langle i, v\rangle$ is taken from In_p at some process p at time $t\}$, $t_{out} = \max\{t : \langle i, v\rangle$ is output to Out_p at some process p at time $t\}$. Then $\tau_i(R) = t_{out} - t_{in}$.*

Let $A(\Delta)$ denote algorithm A parametrized with Δ.[1]

Definition 5.4 (Swift algorithm)**.** *An algorithm $A(\Delta)$ that solves repeated consensus is* swift *if there are constants $k, c \in \mathbb{N}$ such that for every run R of $A(\Delta)$ that is Δ-partially synchronous with good period R', and includes an infinite number of instances, there exists i' such that for all instance $i \geq i'$, we have $\tau_i(R) \leq k\delta(R') + c$.*

Note that this definition does not refer to timeouts. Our definition only depends on the relation between system properties (*i.e.*, transmission delays) and algorithm properties (*i.e.*, execution time), and therefore avoids any reference to timeout expiration.

5.3 A non-swift round-based algorithm

We illustrate swiftness and non-swiftness using the OneThirdRule (OTR) consensus algorithm, which we have described in the context of the HO model in Chapter 3 (see Algorithm 3.1). Recall that OTR requires $f < n/3$ and is always safe. For liveness it requires two rounds in which the set Π_0 of alive processes (at least $2n/3$) receives all messages from processes in Π_0, and only from these processes. In the following, a round that satisfies this property is called *uniform.*

The implementation of the round structure is given by Algorithm 5.1. It is an extension of the implementation given in Chapter 4 (Algorithm 4.3 and Parametrization 4.1) with support for repeated instances of consensus.

Each iteration of the outermost loop is composed of three parts: *input & send* part, *receive* part and *comp. & output* part. In the *input & send* part, the process queries the input queue for new proposals (line 6), initializes new slots in the *state* vector for each new proposal (line 8), calls the send function of all active consensus instances (line 10), and sends the resulting messages (line 13). The process then starts the *receive* part, where it waits for messages until either the timeout *TO* expires (line 17) or it receives a message from a higher round (line 21). Finally, in the *comp. & output* part, the process calls the state transition function of each active

[1] For models with known bounds on transmission delays, Δ represent this knowledge. For models with unknown Δ, or asynchronous algorithms, we assume $A(\Delta)$ to be a constant function, *i.e.*, $A(\Delta)$ represents one single algorithm.

Algorithm 5.1 A non-swift round implementation (code of p)

```
1:  $r_p \leftarrow 1$ /* round number */
2:  $next\_r_p \leftarrow 1$
3:  $Rcv_p \leftarrow \emptyset$ /* set of received messages */
4:  $\forall i \in \mathbb{N}:\ state_p[i] \leftarrow \bot$ /* state of instance $i$ */

5:  while true do
        input & send
6:      $I \leftarrow input()$
7:      for all $\langle i, v\rangle \in I$ do
8:          $state_p[i] \leftarrow \langle v, \bot\rangle$
9:      for all $i:\ state_p[i] \neq \bot$ do
10:         $msgs[i] \leftarrow S_p^{r_p}(state_p[i])$
11:     for all $q \in \Pi$ do
12:         $M_q \leftarrow \{\langle i, msgs[i][q]\rangle :\ state_p[i] \neq \bot\}$
13:         send$\langle M_q, r_p, p\rangle$ to $q$
        receive
14:     $i_p \leftarrow 0$
15:     while $next\_r_p = r_p$ do
16:         $i_p \leftarrow i_p + 1$
17:         if $i_p \geq TO$ then
18:             $next\_r_p \leftarrow r_p + 1$
19:         receive($M$)
20:         $Rcv_p \leftarrow Rcv_p \cup M$
21:         $next\_r_p \leftarrow \max(\{r : \langle -, r, -\rangle \in Rcv_p\} \cup \{next\_r_p\})$
        comp. & output
22:     $O \leftarrow \emptyset$
23:     for all $i:\ state_p[i] \neq \bot$ do
24:         for all $r \in [r_p, next\_r_p - 1]$ do
25:             $\forall q \in \Pi:\ M_r[q] \leftarrow m$ if $\exists M\ \langle M, r, q\rangle \in Rcv_p$
                                 $\wedge \langle i, m\rangle \in M$, else $\bot$
26:             $state_p[i] \leftarrow T_p^r(state_p[i], M_r)$
27:         if the first time $state_p[i].decision \neq \bot$ then
28:             $O \leftarrow O \cup \langle i, state_p[i].decision\rangle$
29:     $output(O)$
30:     $r_p \leftarrow next\_r_p$
```

instance (line 26), and outputs any new decisions (line 29). Note that some rounds may be partially skipped (no message sent, no message received, only transition function executed): this happens whenever a message from a higher round is received.

5.3.1 Correctness proof

We now prove the correctness of the round implementation for $TO \geq 2\Delta + 2n + 5$.

Theorem 5.1. *Consider a run of Algorithm 5.1 with $TO \geq 2\Delta + (2n+5)$ and $n > 3f$. Let R be a Δ-partially synchronous run. Then every consensus instance that starts at t decides the latest at $max(GST, t) + 3TO + \Delta + 4n + 8$.*

Following lemmas together with the results of [CBS09] proves the theorem.

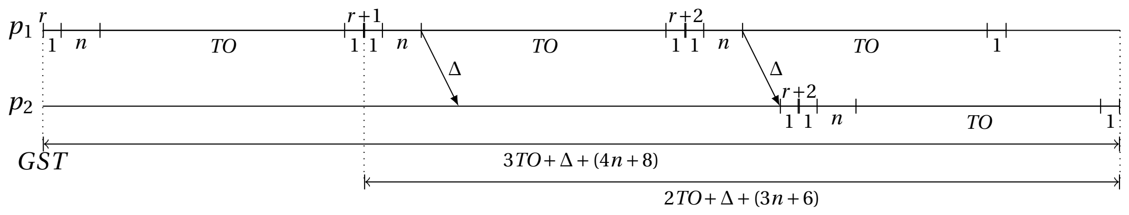


Figure 5.1: Non-swift rounds: Theorem 5.1 and Lemma 5.3

Lemma 5.1. *Consider Algorithm 5.1 with $TO \geq 2\Delta + (2n+5)$ and $n > 3f$. Let R be a Δ-partially synchronous run. Let t_r be the time the first process starts a new round r after GST. Then round r is uniform.*

Lemma 5.2. *Consider Algorithm 5.1. Let R be a Δ-partially synchronous run. Then by time $GST + TO + (n+2)$ at least one process has started a new round r_0.*

5.3.2 Maximum and minimum execution times

We now show that for each instance i of consensus started after *GST*, we have an execution time $\tau_i \leq 2TO + \delta + 3n + 6$. This defines the maximum execution time.

Lemma 5.3. *Consider Algorithm 5.1 with $TO \geq 2\Delta + (2n+5)$, $n > 3f$, and a Δ-partially synchronous run R. Let r_0 be the first new round that is started after GST. Then for all instances i started in a round $r \geq r_0$, we have an execution time $\tau_i \leq 2TO + \delta + (3n+6)$.*

Proof. (See Figure 5.1 for illustration.) Let i be an instance started in a round $r \geq r_0$ by a process p. Recall that Algorithm 5.1 needs at most two uniform rounds to decide. Since by Lemma 5.1 rounds r and $r+1$ are uniform, all processes decide instance i by round $r+1$ (*i.e.*, they output (i, x) at line 29, where x is the decision).

It remains to calculate the maximum time for rounds r and $r+1$. Let p be the first process to start round r at time t_r. Process p will finish round r the latest at $t_r + TO + (n+2)$, and start the send steps for round $r+1$, 1 step later. By time $t_r + TO + \delta + (2n+3)$, p's round $r+1$ messages are ready for reception at all processes. At this point, all processes have finished executing the send steps for round r (process p's round r messages forced them to advance) and are either executing receive steps for round r or have entered round $r+1$. Therefore, all processes will enter round $r+1$ at most 1 step after receiving p's round $r+1$ message. Round $r+1$ will take at most $TO + (n+2)$ time, so by time $t_r + 2TO + \delta + (3n+6)$ all processes have finished round $r+1$. □

Next we show that the implementation is not swift by computing the minimum execution time for each instance of consensus.

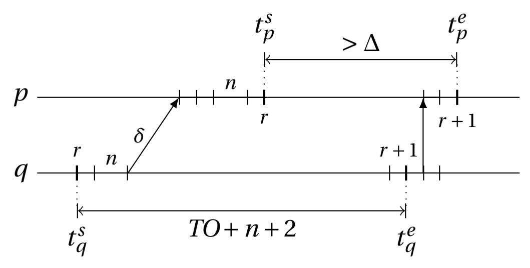


Figure 5.2: Non-swift rounds: Lemma 5.4

Lemma 5.4. *Consider Algorithm 5.1 with $TO \geq 2\Delta + 2n + 5$, $n > 3f$. Let R be a Δ-partially synchronous run. Let r_0 be the first new round that is started after GST. Then for all instances i started in a round $r \geq r_0$, we have an execution time $\tau_i > \Delta$.*

Proof. We prove the result by showing that, for every round $r \geq r_0$, every process p stays in round r for more than Δ time.

Let t_p^s and t_p^e be the time when p starts and finishes round r, respectively. Process p may finish round r either (i) by the expiration of its timeout (line 17), or (ii) by receiving a higher round message (line 21).

In case (i) we have $t_p^e - t_p^s = TO + (n+2) > \Delta$, that is the timeout, n send steps, one input step, and one output step. Thus p stays in round r more than Δ time.

For case (ii), we calculate the minimum duration of round r by determining the latest time t_p^s and the earliest time t_p^e when p could have started and ended round r, respectively (see Figure 5.2). Let q be the first process to finish round r at time t_q^e. Then the earliest that p may receive a round $r+1$ message is $t_q^e + 3$ (one input step by q at the start of round $r+1$, one send step, and one output step by p to finish round r). Hence, $t_p^e = t_q^e + 3$.

Let t_q^s be the time when q started round r. Process q sends a round r message to p the latest by $t_q^s + n + 1$ (if the message to p is sent in the last send step). By assumption $r \geq r_0$, so t_q^s is after *GST*. Therefore, p receives the message at most $\delta + n + 2$ later (δ is the maximum transmission delay in this run and, in the worst case, p is taking an output step when the message is received, so that in total it takes one output step, one input step, and n send steps, before the next receive step). After one final output step, p enters round r. This happens the latest by $t_q^s + \delta + 2n + 4$. Therefore $t_p^s = t_q^s + \delta + 2n + 4$.

The minimum duration of round r at p is $t_p^e - t_p^s = (t_q^e + 3) - (t_q^s + \delta + 2n + 4) = (t_q^e - t_q^s) - \delta - 2n - 1$. To calculate $t_q^e - t_q^s$, recall that q finishes round r by timeout and not by receiving a higher round message, because by assumption no other process started a round higher than r before q. Therefore, q stays in round r a total of $t_q^e - t_q^s = TO + n + 2$. Substituting $t_q^e - t_q^s$, we obtain $t_p^e - t_p^s = (TO + n + 2) - \delta - 2n - 1 \geq 2\Delta + n + 6 - \delta \geq \Delta$, which means that p stays in round r more than Δ time. $\square$

Algorithm 5.2 OTR with the failure detector $\Diamond\mathscr{P}$ (code of p)

```
State:
  $r_p \leftarrow 1$ /* round number */
  $x_p \in V$
  $decision_p \in V$

while true do
  send $\langle r_p, x_p \rangle$ to all processes
  wait until received values for round $r_p$ from all processes $q \notin \Diamond\mathscr{P}_p$
  if number of values received $> 2n/3$ then
    $x_p \leftarrow x$ smallest most often received value
    if more than $2n/3$ values received are equal to $v$ then
      $decision_p \leftarrow v$
  $r_p \leftarrow r_p + 1$
```

Since the execution time is proportional to the parameter Δ and independent of the effective transmission delay δ, the implementation is not swift:

Theorem 5.2. *The round implementation of Algorithm 5.1 is not swift.*

Proof. In case that $TO < 2\Delta + 2n + 5$, the algorithm is not live. Therefore we only consider $TO \geq 2\Delta + 2n + 5$. Assume by contradiction that the collection of algorithms $A(\Delta)$ given by Algorithm 5.1 is swift. Then, there exist $k, c \in \mathbb{N}$, such that in every Δ-partially synchronous run R with a good period R', there is an i_R such that, for all instances $i > i_R$, $\tau_i(R) < k\delta(R') + c$. For a contradiction, consider $A(k\delta(R') + c)$. By Lemma 5.4, for all instances started after GST, we have $\tau_i > \Delta = k\delta(R') + c$. A contradiction. □

5.4 A failure detector-based algorithm that is swift

We consider now the OTR algorithm expressed with the failure detector $\Diamond\mathscr{P}$ (Algorithm 5.2). Intuitively it is easy to see that repeated execution of this algorithm is swift. Indeed, some time after GST, the failure detector list contains exactly the faulty processes. At this point, by line 7, all correct processes wait only for messages from correct processes and, since $f < n/3$, the condition on line 8 is always true. Note that the failure detector model requires reliable links, contrary to the solution in the previous section.[2] In this section we assume that links are reliable.

Repeated execution of Algorithm 5.2 is expressed by Algorithm 5.3. The box in Algorithm 5.3 corresponds to line 7 of Algorithm 5.2. For simplicity, we have not shown in Algorithm 5.3 the (trivial) implementation of $\Diamond\mathscr{P}$. We assume that both Algorithm 5.3 and the implementation of $\Diamond\mathscr{P}$ run in the same partially synchronous system in the following way: in every even step Algorithm 5.3 is executed, in every odd step the implementation of $\Diamond\mathscr{P}$ is executed.

[2] Consider two correct processes p and q and line 7 executed by p. If the message sent by q is lost, and p's failure detector never suspects q, then p is blocked forever at line 7.

Algorithm 5.3 Multiple instances of Algorithm 5.2 (code of p)

```
Initialization:
    $r_p \leftarrow 1$
    $\forall i \in \mathbb{N}: x_p[i] \leftarrow \bot$
    $\forall i \in \mathbb{N}: decision_p[i] \leftarrow \bot$

while $true$ do
    $I \leftarrow input()$
    for all $\langle i, v \rangle \in I$ do
        $x_p[i] \leftarrow v$
    send $\langle r_p, x_p, p \rangle$ to all processes
    while not received $\langle r_p, x_q, q \rangle$ from all processes $q \notin \Diamond\mathcal{P}_p$ do
        $receive(M)$
        $Rcv \leftarrow Rcv \cup M$
    $O \leftarrow \emptyset$
    for all $i : x_p[i] \neq \bot$ and $decision_p[i] = \bot$ do
        if number of values received $\langle r_p, x', - \rangle > 2n/3$ then
            $x_p[i] \leftarrow$ smallest most often value $x'[i]$
            if more than $2n/3$ values $x'[i]$ are equal to $v$ then
                $decision_p[i] \leftarrow v$
                $O \leftarrow O \cup \{\langle i, v \rangle\}$
    $output(O)$
    $r_p \leftarrow r_p + 1$
```

The correctness of Algorithm 5.3 follows from the following lemma:

Lemma 5.5. *For Algorithm 5.3, there is eventually a round GSR so that for all rounds $r \geq GSR$, every correct process receives a message from every correct process in round r and receives no message from faulty processes.*

Proof. By the properties of $\Diamond\mathcal{P}$, there is a time where the FD is accurate and complete, *i.e.*, a process is suspected if and only if it is faulty. In every round that is started after this time, every correct process waits for a message from every correct process. □

Theorem 5.3 proves that Algorithm 5.3 is swift, by showing that eventually every instance of consensus decides in at most $3\delta + 6n + 6$.

Theorem 5.3. *For a run of Algorithm 5.3 with $n > 3f$ and an infinite number of instances of consensus, there is an instance i_0 such that for all $i > i_0$, we have $\tau_i \leq 3\delta + 6n + 6$.*

Proof. Let *GSR* be the round defined by Lemma 5.5. Since in every input step only a finite number of instances are read, there is an input step so that this step and all later input steps are in a round after *GSR*. Let i_0 be the largest consensus instance started in a round before *GSR* (instance i is started in the round in which the last process starts instance i). Consider an instance $i > i_0$. The maximum execution time of i corresponds to the maximum duration of two rounds. This follows from Lemma 5.5, which ensures that instance i decides in at most two rounds. It remains to calculate the maximum time for two rounds after *GSR*.

Let t be the first time a process, say p, starts round $r > GSR$. Since $r-1 \geq GSR$, p received round $r-1$ messages from all correct processes. This must have happened the latest by time $t-2$ in order to allow p to execute the output step of round $r-1$, and to enter round r at time t.[3] Therefore p executed the receive step of round $r-1$ at latest by time $t-4$, and all correct processes started the send steps for round $r-1$ at latest by time $t-4$; these send steps finished at latest by time $t-4+2n = t+2n-4$, and messages are received at latest by time $t+2n-4+\delta$. Adding the output step, all correct processes started round r the latest at $t' = t+2n-2+\delta$.

By $t'' = t' + 2n + 2 + \delta$ all round r messages are thus ready for reception, and received by $t'' + 2$. Again by $t'' + 2 + 2n + 2$ all round $r+1$ messages are sent, and thus round $r+1$ ends the latest at $t'' + 2 + 2n + 2 + \delta + 2 = t + 3\delta + 6n + 6$. □

Remark Failure detector based solutions require reliable links. This has the following implication. In contrast to partial round implementation of Section 5.3, no round is skipped, *i.e.*, processes send messages for all rounds, and wait for the messages from all unsuspected processes. This implies that, unlike the round implementation in the previous section, it is no more possible to bound the time from *GST* until the first decision. To see this, note that at *GST*, a process p might be in a round r that is arbitrarily smaller than the highest round number r_{max} at that time. Since other correct processes might wait in any round r', $r \leq r' \leq r_{max}$, for the round r message of process p, p cannot skip the sending step of all rounds between r and r_{max}. This takes an unbounded amount of time, as $r_{max} - r$ can be arbitrarily large. Note that the problem cannot be solved by packing all messages into a single one since, between the sending steps, process p has to perform receive steps (to receive messages from the other correct processes).

5.5 A new round implementation that is swift

We show now that the implementation of the round model can be made swift. Like in the failure detector approach, each process estimates a set of alive processes (the complementary of the set of suspected processes) and uses this set to terminate a round earlier after *GST*, namely, as soon as it receives all messages from the *Alive* set. Contrary to the failure detector approach, the algorithm tolerates message loss, by using a timeout that expires only before *GST*. Like in the round-based implementation, processes resynchronize after message loss by skipping rounds. Skipping rounds also allows the algorithm to decide in a bounded time after *GST*.

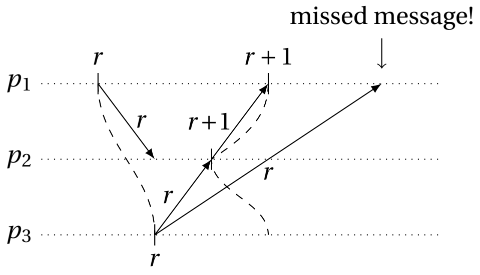


Figure 5.3: Issue to address in swift-rounds implementation

5.5.1 Issue to address

Combining the termination of a round upon reception of all messages from alive processes, and the round-skipping mechanism, requires some attention. The problem is illustrated in Figure 5.3. In this scenario, p_3's round r message is the last message needed by p_2 to have all round r messages. Let us assume that upon receiving this message, p_2 immediately sends its round $r+1$ message to all. In this case, process p_1 may receive the round $r+1$ message of p_2 before the round r message of p_3. If p_1 jumps to round $r+1$ upon receiving the first round $r+1$ message, it will miss p_3's round r message, thereby breaking uniformity on round r. This situation may repeat in every round, thus preventing the algorithm from deciding. We show now how we address this problem.

5.5.2 The full algorithm

The ideas described above are used in Algorithm 5.4, which is a round implementation that is swift. Algorithm 5.4 enhances Algorithm 5.1 as follows:

(i) Each process p maintains an estimation of the set of alive processes in $Alive_p$ (see line 13), and updates it every TO_A steps. TO_A is thus the timeout used to suspect faulty processes.
(ii) A process goes directly to the next round if it receives a message from all processes in its *Alive* set (lines 14-15). This is the key point to make the algorithm swift.
(iii) In any case, a process goes to the next round after *TO* time (lines 16-17). *TO* is thus the timeout for a round in bad periods.
(iv) When receiving a round message from the next round for the first time, the process waits for at most TO_D steps before going into this round (lines 21-22). For this and the last point, each process p maintains a variable $timeout_p$, initially set to *TO* (line 8) which is modified when a round $r+1$ message is received (line 22). This is used to address the problem described in Section 5.5.1.
(v) When receiving a message from a round higher than the next round (*i.e.*, larger than r_p+1), the process immediately goes to this round (lines 19-20). This ensures a fast resynchronization of the processes after a bad period.

[3]Note that we have to double the time for a step, since only every second step is of the asynchronous algorithm.

Algorithm 5.4 A swift round implementation (code of p)

```
1:  $r_p \leftarrow 1$ /* round number */
2:  $next\_r_p \leftarrow 1$
3:  $Rcv_p \leftarrow \emptyset$ /* set of received messages */
4:  $\forall i \in \mathbb{N}: state_p[i] \leftarrow \bot$ /* state for instance $i$ */

5:  while $true$ do
6:    input & send /* lines 6-13 of Algorithm 5.1 */
                                                              receive
7:    $i_p \leftarrow 0$;
8:    $timeout_p \leftarrow TO$
9:    while $next\_r_p = r_p$ do
10:     $i_p \leftarrow i_p + 1$
11:     receive($M$)
12:     $Rcv_p \leftarrow Rcv_p \cup M$
13:     $Alive_p \leftarrow$ {set of processes from whom
              there is a message within last $TO_A$ steps}
14:     if $\forall q \in Alive_p : \exists \langle M_q, r_p, q \rangle \in Rcv_p$ then
15:       $next\_r_p \leftarrow r_p + 1$
16:     if $i_p \geq timeout_p$ then
17:       $next\_r_p \leftarrow r_p + 1$
18:     $r \leftarrow max\{r : \langle -, r, - \rangle \in Rcv_p\}$
19:     if $r > r_p + 1$ then
20:       $next\_r_p \leftarrow r$
21:     if there is a message from round $r_p + 1$
        for the first time then
22:       $timeout_p \leftarrow \min\{i_p + TO_D, TO\}$
23:    comp. & output /* lines 22-29 of Algorithm 5.1 */
24:    $r_p \leftarrow next\_r_p$
```

We now show the correctness (Section 5.5.3) and swiftness (Section 5.5.4) of this solution.

5.5.3 Correctness

We now show that the OTR consensus algorithm when executed over the rounds implementation provided by Algorithm 5.4 solves repeated consensus in a partially synchronous system. As already discussed, OTR is always safe (with $n > 3f$). Before proving that the round implementation given by Algorithm 5.4 provides liveness, we show some properties of the algorithm—related to correctness—that hold after *GST*.

When the good period starts at *GST*, processes will synchronize to the same round using the following two mechanisms: (i) when a process receives a higher round message, it advances rounds either immediately (line 20), or within TO_D (lines 21-22), or when the original timeout *TO* expires; (ii) in any case, processes remain in a round at most *TO* time, starting a new round when this timeout expires (lines 16-17 and line 22). Therefore, shortly after *GST*, there will be a process p that starts a new round r that is higher than any round started by the other alive processes. When the other processes receive the round r message from p, they will advance to round r and send their own messages. These messages are then received by all alive processes, resulting in a uniform round.

As discussed in Section 5.5.1, a round $r+1$ message may be received before all round r messages (Figure 5.3). To address this issue, if a process p in round r receives a message from round $r+1$ for the first time and it has not received all the messages from its *Alive* set, it does not advance immediately. Instead, it waits either for an additional TO_D or until the end of the original timeout, whichever comes first. During the good period, all the remaining round r messages will be received before this revised timeout expires. To see why, notice that for a process to send a round $r+1$ message, it must have received all round r messages from the alive processes, so these messages will also be received by process p within at most $TO_D = \Delta + (n-1)$, namely $n-1$ send steps and Δ maximum transmission delay. In any case, all messages will be received before the original round timeout, so the process only has to wait for the minimum of TO_D or what is left of TO.

If a process p in round r receives a message from round $r+2$ or higher, it can conclude that the good period has not yet been started, so p advances immediately to round $r+2$. This holds for the following reason. Assume that the system is in a good period, and let some process q send a round $r+2$ messages; then either (i) q received all round $r+1$ messages, including p's message, which is not possible; or (ii) the timeout for round $r+1$ expires, which is not possible as the timeout is chosen in a way that processes have enough time to receive all round messages and messages are not lost in the good period. This shows a contradiction: the system cannot be in a good period.

Thus we can show:

Theorem 5.4. *Consider a run of Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partially synchronous run. Then every consensus instance that starts at t decides the latest at $max(t, GST) + TO_A + 2TO + TO_D + 3\Delta + (6n+15)$.*

The proof is based on the following two lemmas. The first establishes that eventually rounds are uniform (see Sect. 5.3):

Lemma 5.6 (Timeouts TO and TO_D)**.** *Consider Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$. Let R be a Δ-partially synchronous run, and t_r the time the first process starts a new round r after GST, such that all processes have the same Alive set after t_r. Then round r is uniform.*

The previous lemma requires all processes to have the same *Alive* set. Lemma 5.7 shows that this becomes true shortly after *GST*.

Lemma 5.7 (Timeout TO_A)**.** *Consider Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partially*

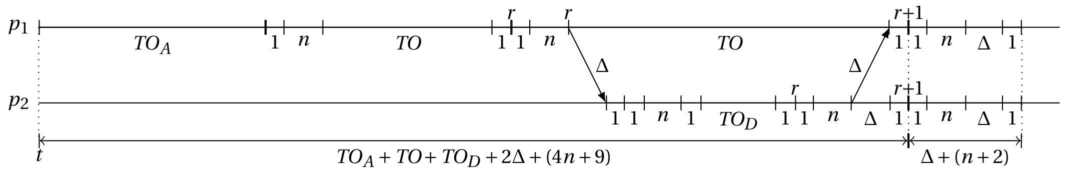


Figure 5.4: Swift rounds: Theorem 5.4

synchronous run. Let t_r be the time the first process starts a new round r after GST. Then by time $t_r + 2 + TO_A$ all processes have the same Alive set.

We now prove Theorem 5.4.

Proof. See Figure 5.4 for illustration. We distinguish two cases (1) $t < GST$, (2) $t \geq GST$. In case (1) by Lemma 5.6 a new round is started after GST the latest by time $GST + TO + n + 2$. From Lemma 5.7 all processes have the same *Alive* set by time $t_0 = GST + TO + TO_A + n + 4$. In case (2) by Lemma 5.7 all processes have the same *Alive* set by time $t_0 = t + TO_A + 2$, which is strictly smaller than $GST + TO + TO_A + n + 4$. From the code of the algorithm a process, *e.g.*, p_1, starts a new round r every $TO + (n+2)$ steps, *i.e.*, the latest by time $t_1 = t_0 + TO + n + 2$. All processes do so by time $t_2 = t_1 + TO_D + \Delta + (2n+5)$. This means that all processes start round r with the same *Alive* set. From Lemma 5.6, round r is uniform. Furthermore, all processes receive all round r messages from their *Alive* set, and end round r the latest by time $t_3 = t_2 + \Delta + (n+2)$ and start round $r+1$ at this time.

From the assumption, no process crashes after GST, therefore, the *Alive* set remains the same. In round $r+1$, all processes send their messages to all the latest by time $t_3 + (n+1)$. These messages can be received by all processes the latest by time $t_3 + (n+1) + \Delta$. From lines 14-15, all processes end round $r+1$ the latest by time $t_3 + (n+1) + \Delta + 1$ after an output step. This means that all processes decide the latest by this time which is equal to $t_0 + TO + (TO_D + 2\Delta + (4n+9)) + (\Delta + (n+2))$, or $max(GST, t) + TO_A + 2TO + TO_D + 3\Delta + (6n+15)$. □

5.5.4 Swiftness

In order to show that OTR together with the round implementation provided by Algorithm 5.4 is swift, we show that the execution time of a consensus instance depends only on δ and not on Δ.

The main properties of the algorithm related to the swiftness, which hold after *GST*, are the following. First, the *Alive* set becomes accurate the latest by $GST + TO + TO_A + n + 4$ (line 13). This follows from Lemma 5.7, with t_r being at latest $GST + TO + n + 2$. Then, once the *Alive* set is accurate after *GST*, it no more changes and therefore no further timeout expires. Finally, all processes finish rounds as soon as all messages from alive processes are received and advance round by lines 14-15, rendering the algorithm swift.

Theorem 5.5. *Consider Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partially synchronous run. Then every consensus instance that is started after $GST + X$ with $X = TO_A + 3(TO + n + 2) + 2$, has an execution time of $\tau_i \leq 3\delta + 3n + 5$.*

Proof. Let i_0 be a new consensus instance started at time $t_s > GST + X$. Such an instance exists, because the input queue contains an infinite number of elements and each input step reads a finite number of instances. Let p be the process that started instance i_0 (last process doing an input step for i_0) and r_i the round where it was started.

We will first show that rounds $r \geq r_i - 1$ are uniform. By lines 8, 16 and 22 of Algorithm 5.4, processes remain on a round for at most $TO + n + 2$ time. Therefore, the first round started after GST starts the latest at $t_0 = GST + TO + n + 2$. By Lemma 5.7, the latest at $t_1 = t_0 + TO_A + 2$ all processes have the same *Alive* set, and by Lemma 5.6, all rounds started after t_1 are uniform. Therefore the first uniform round, r', starts the latest at time $t_2 = t_1 + TO + n + 2$, and $r' + 1$ the latest by $t_3 = t_1 + 2(TO + n + 2)$. Expanding this expression, we obtain $t_3 = GST + TO_A + 3(TO + n + 2) + 2 = GST + X$. Since r_i started at time $t_s \geq GST + X$, rounds $r \geq r_i - 1$ are uniform.

Using Theorem 5.4 we can conclude that instance i_0 is decided by round $r_i + 1$. We are now ready to compute the maximum execution time of i_0. By definition, we have $\tau_i = t_e - t_s$, where t_e is the time when the last process performs an output step for round i_0 (and t_s is previously defined). To determine the upper bound on τ_i, we'll compute the smallest and the largest values for times t_s and t_e, respectively. Since by assumption t_s happens in round r_i, then t_s is smallest if p is the first process starting the round. The largest value for t_e is the time of the output step of the last process finishing round $r_i + 1$. Next we compute t_e.

Since round $r_i - 1$ is uniform, process p received all round $r_i - 1$ messages before advancing to round r_i, hence the latest by $t_s - 2$ all alive processes had sent their round $r_i - 1$ message to p. By $t_s + n - 2$ all round $r_i - 1$ messages were sent, and δ time later received. Thus, by $t_s + \delta + 2n$ all processes entered round r_i and finished sending all messages. δ time later all round r_i messages are received and by time $t_s + 2\delta + 2n + 1$ all processes started round $r_i + 1$. By a similar reasoning, by time $t_s + 3\delta + 3n + 3$, all processes finished round $r_i + 1$. Hence, instance i_0 ends at time $t_e = t_s + 3\delta + 3n + 3$, and we have $\tau_i = 3\delta + 3n + 3$. □

5.6 Experimental results

In this section we present the results of an experimental study, comparing the three algorithms presented previously. The main questions we want to answer are (i) how much improvement can be obtained in a round-based algorithm using a swift round implementation, and (ii) are swift round implementations competitive with implementations that use failure detectors.

Experimental setup We performed our experiments both in an emulated network and directly in a physical network (a cluster). The emulated network allowed us to test the behavior of the algorithms with different transmission delays and message loss rates, while the physical network shows what to expect in a cluster environment.

In all experiments, processes were started with 1 second of delay between each other. This prevents the spontaneous synchronization that often occurs when processes start at approximately the same time, and thus exercises the ability of the algorithms to resynchronize the processes.

The metric considered is the decision time for each consensus instance. Processes run instances sequentially, starting the next one either when they decide, or when they learn the decision by receiving a message from a higher instance. Each data point shown on the plots below was obtained from a 10 minutes run. We then calculated the average decision time, ignoring the first 10% of the run. For each data point, we show the 95% confidence intervals.

Implementing $\Diamond\mathscr{P}$ and reliable channels for the failure detector algorithm We implemented $\Diamond\mathscr{P}$ by having each process send heartbeats to all every η time. A process p suspects q if it does not receive any heartbeat for more than τ time. We also implemented reliable channels using message acknowledgments and retransmission. We decided not to use TCP, because our initial experiments using TCP resulted in very poor performance under high message loss conditions. TCP is designed to interpret message loss as an indication of congestion, and therefore it reacts by increasing the retransmission time. On a typical TCP implementation, the interval between retransmissions may reach several minutes, which in practice blocks the algorithms running on top of it.[4]

Notation In the following, δ_{net} denotes the one-way transmission delay of the physical network, δ_{emu} the delay emulated by ModelNet, and δ_{eff} the effective one-way transmission delay between two processes. In the experiments run directly on the physical network, $\delta_{eff} = \delta_{net}$. However, when using ModelNet, $\delta_{eff} = 2\delta_{net} + \delta_{emu}$, since each packet is transmitted two times on the physical network (see Section 5.6.1). Finally, note that contrary to δ defined previously, δ_{net} is not a bound. Instead, it is a random variable, reflecting the non-deterministic behavior of a physical network.

In the following, NS-OTR, S-OTR, and FD-OTR denote respectively the non-swift OTR (OTR + Algorithm 5.1), the swift OTR (OTR + Algorithm 5.4), and OTR with FD (Algorithm 5.3 + $\Diamond\mathscr{P}$).

[4] A few months after completing the work described here, we used the same cluster for other experiments after it had undergone a kernel upgrade to Linux version (from 2.6.11 to 2.6.26). In this new setup, TCP dealt gracefully with high levels of congestion, suggesting that the problems we found in the experiments presented in this chapter were due to a poor implementation of the TCP stack in kernel 2.6.11.

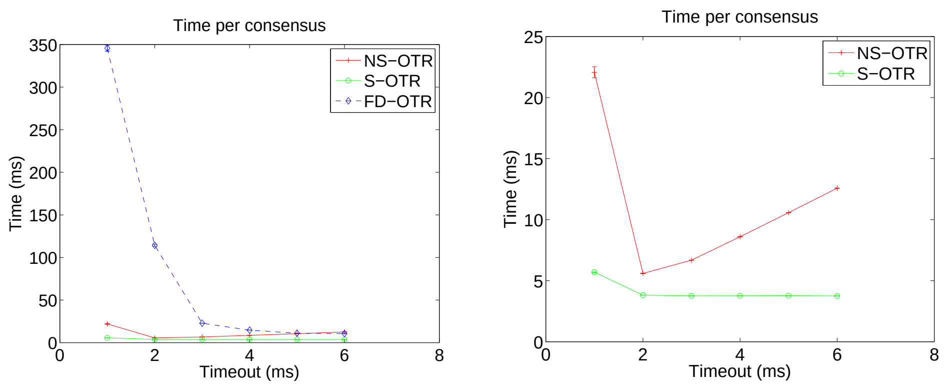


Figure 5.5: Experimental results in ModelNet with $\delta_{eff} \approx 0.3$ms ($\delta_{emu} = 0$, $2\delta_{net} \approx 0.3$). The figure on the right repeats NS-OTR and S-OTR from the left, with a different time scale.

5.6.1 Emulated network

We used ModelNet [VYW+02] to emulate the network topology used in the experiments. ModelNet uses two types of nodes: a *core node* that applies the traffic policies, and one or more *edge nodes* that run the application being tested. The edge nodes redirect all traffic sent by the processes to the core node, which applies the traffic policy (*e.g.*, delay, loss and maximum bandwidth) and then transmits the packet to the intended receiver. We varied the emulated delay and loss rate, while leaving the emulated bandwidth set to 1Gbps, which effectively means that the processes are allowed to use the full bandwidth of the underlying physical network. We used two physical machines for all experiments run on ModelNet. All four replicas were running on a dual Pentium 4 at 3.6GHz with 1GB RAM running Linux 2.6.11, while the core node was a Pentium Pro at 200MHz with 70MB of RAM. The machines were connected by a full duplex 100Mbits Ethernet, and had a ping time of approximately 0.3ms. Hence, $\delta_{eff} \approx 0.3 + \delta_{emu}$.

Varying the timeout

In the first set of experiments, we fixed the emulated transmission delay while varying the timeout *TO* used by the algorithms. Figure 5.5 shows the results for $\delta_{emu} = 0$ms and Figure 5.6 for $\delta_{emu} = 40$ms. The x axis indicates the timeout *TO* used by the algorithms to terminate a round.[5] For the tests with $\delta_{emu} = 40$ms, the failure detector was configured with $\eta = TO/2$ and $\tau = TO$. The rationale is that TO is the time an algorithm should wait before declaring a failure and taking corrective measures, *e.g.*, advancing rounds or suspecting a process. With $\delta_{emu} = 0$ms, following the same policy would result in the network being overloaded with heartbeats, so we opted for $\eta = TO$ and $\tau = 2TO$.

The results clearly validate the main motivation behind this work, in that S-OTR performs at

[5]Equivalent to 2Δ on the NS-OTR and 3Δ for the S-OTR.

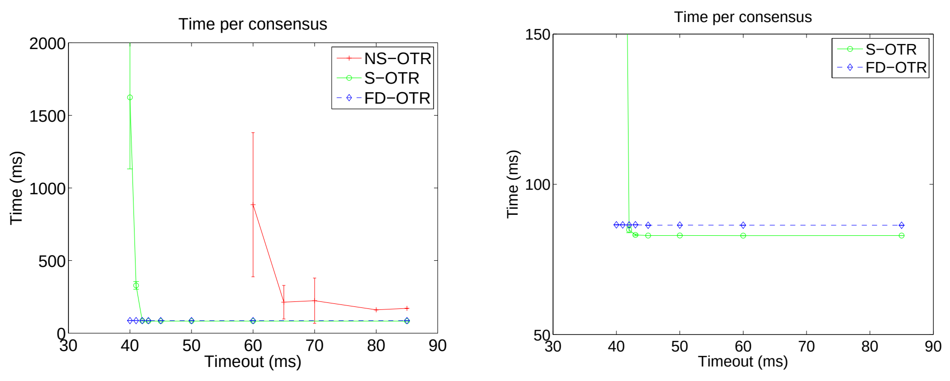


Figure 5.6: Performance on ModelNet with $\delta_{eff} \approx 40.3$ms ($\delta_{emu} = 40$, $2\delta_{net} \approx 0.3$). The figure on the right repeats S-OTR and FD-OTR from the left, with a different time scale.

the speed of the network, being independent from the timeout.

With $\delta_{emu} = 0$ms (Figure 5.5-left), FD-OTR performs poorly with low timeouts. This is caused by the additional messages sent by the failure detector and the reliable channels implementation, which slow down the processes and congest the network. For higher timeouts, this overhead becomes less significant and the algorithm starts performing similarly to the other implementations. When looking only at NS-OTR vs S-OTR (Figure 5.5-right), it is clear that the decision time of NS-OTR increases linearly with the timeout, while S-OTR is constant. Furthermore, even with a 2ms timeout where it achieves the best results, NS-OTR performs worse than S-OTR, because no fixed timeout can approximate perfectly the time that it takes for a process to receive all messages (it fluctuates from round to round).

With $\delta_{emu} = 40$ms (Figure 5.6-left), NS-OTR performs poorly with timeouts lower than 80ms. For timeouts lower than 60ms, the algorithm took hundreds of rounds for each decision, so we did not show the results as they were not statistically significant. Notice that $80 \approx 2\delta_{eff}$, which matches the results from the analytical analysis, where a round must last $TO = 2\Delta$ in order to ensure decision. The swift version S-OTR is more tolerant to a non-optimal timeout, being able to synchronize even with timeouts slightly above 40ms. This is because processes finish rounds early, after receiving all messages, allowing the processes that are behind to slowly catch-up with the ones in the lead.

FD-OTR is also independent of the timeout, producing the optimal performance regardless of the values used for the underlying failure detector. Recall that in the absence of message loss, the values chosen for the failure detector (*i.e.*, $\tau = 2\eta$) prevent false suspicions, so FD-OTR can proceed at the speed of the network. The overhead of the implementation of failure detectors and reliable channels is smaller in this scenario, as shown in Figure 5.6-right, where FD-OTR performs only slightly worse than S-OTR.

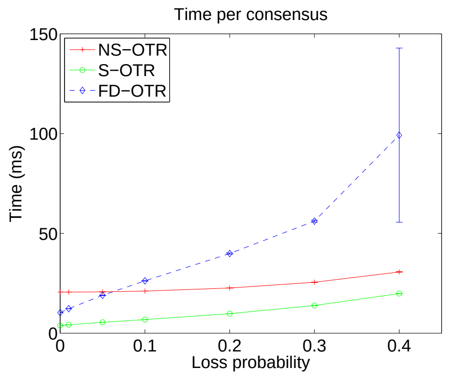


Figure 5.7: Experimental results with message loss in ModelNet: $\delta_{eff} \approx$ 0.3ms ($2\delta_{net} \approx 0.3$, $\delta_{emu} = 0$)

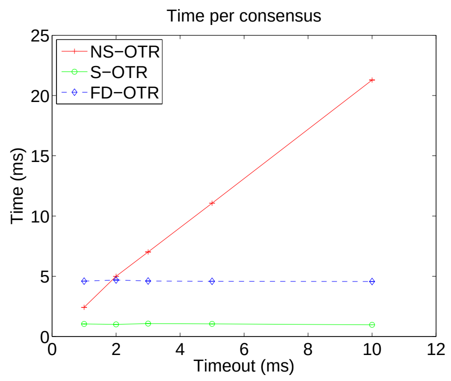


Figure 5.8: Experimental results in the cluster

Message loss

Figure 5.7 shows the behavior of the algorithms in networks with message loss. The experiment was run on ModelNet with $\delta_{emu} = 0$. Both the swift and the non-swift versions were configured with a timeout of 10ms. The failure detector was configured with η = 10ms and τ = 25ms, so that it tolerates 2 or 3 lost heartbeats before (wrongly) suspecting a process. The reliable channels implementation retransmits a message every 25ms.

Both NS-OTR and S-OTR are very resilient to message loss. Even with 40% messages loss, the average decision time is only a few milliseconds more than with no message loss. This is because the algorithms make progress as soon as a single process receives three messages ($2n/3$), *i.e.*, two messages from other processes since its own message is always delivered. FD-OTR performs worse because it waits for messages from all processes that are not suspected, so that a single message loss in a round is enough to delay progress (suspecting a process requires more than a single message loss).

S-OTR outperforms both NS-OTR and FD-OTR in the presence of message loss. In particular, the performance of FD-OTR degrades significantly with message loss, caused by the overhead of the retransmissions to simulate reliable links.

5.6.2 Physical network (cluster)

For the tests with the physical network, we used a cluster of Dual Pentium 4 at 3.00GHz with 1GB memory running Linux 2.6.11 and connected by a 1Gbits Ethernet. Each process run on a separate node, and the ping time between two nodes was between 0.1 and 0.2ms. The failure detector was configured with $\eta = TO$ and $\tau = 2TO$.

Figure 5.8 shows that on the cluster even a timeout of 1 ms is enough for OTR to terminate. S-OTR always outperforms the two other algorithms. Compared to NS-OTR, even with a

	NS-OTR	FD-OTR	**S-OTR**
Link	lossy	reliable	**lossy**
Exec. time	$4\Delta + \delta + O(1)$	$3\delta + O(1)$	$\mathbf{3\delta + O(1)}$
Swift	no	yes	**yes**

Table 5.1: Repeated consensus algorithms analyzed

1ms timeout, S-OTR performs better. Lowering the timeout of NS-OTR may improve its performance, but with such small timeouts the algorithm becomes sensible to the normal variability of the system, which is caused by non-deterministic factors like OS scheduling and background activity, either on the hosts or on the network. This will cause rounds to finish without receiving all required messages, leading to unstable performance. The timeout of S-OTR can be set to a conservative value, making the algorithm immune to non-deterministic factors, while still providing optimal performance.

FD-OTR suffers again from the overhead of the implementation of the failure detector and reliable channels, resulting in worse performance than S-OTR.

5.7 Discussion

Table 5.1 summarizes the results of this chapter. We have analyzed the efficiency of algorithms for solving repeated consensus in two models: the round-based model (which can be implemented on top of a partially synchronous system), and the asynchronous system augmented with failure detectors. Efficiency refers here to *swiftness*, a new notion that captures the fact that an algorithm, once the system has stabilized, progresses at the speed of the messages. Our new round-based implementation combines the advantages of failure detector solutions (swiftness) and round-based model (lossy links). This weak link assumption makes round-based algorithm easy to adapt to the crash-recovery model with stable storage [HS07].

We have illustrated the new round-based implementation on a specific consensus algorithm (OTR). This does not mean that the new solution is limited to OTR. It applies to any consensus algorithm expressed in the round model, in particular to the *LastVoting* algorithm [CBS09], a round-based variant of Paxos [Lam98] that requires only $n > 2f$.

6 Latency-aware Leader Election

Experimental studies have shown that electing a leader based on measurements of the underlying communication network can be beneficial. We use this approach to study the problem of electing a leader that is not only eventually correct (as captured by the Ω failure detector abstraction), but also optimal with respect to the transmission delays to its peers. We give the definitions of this problem and a suitable model, thus allowing us to make an analytical analysis of the problem, which is in contrast to previous work on that topic.

Publication: Nuno Santos, Martin Hutle and André Schiper. Latency-aware Leader Election. In *2009 ACM symposium on Applied Computing (SAC'09)*, Honolulu, Hawaii, USA, March 9-12, 2009

6.1 Introduction

Leader election is an important service for fault tolerant systems. Such a service is typically used in several consensus algorithms, the so-called "leader-based" consensus algorithms. In such algorithms the leader can be determined either by a static rule (*e.g.*, rotating coordinator), or can be computed dynamically. In the latter case the leader is selected by a distributed leader election algorithm. Leader election is also sometimes called "implementation of the Ω failure detector". A lot of work on the topic has been published in the recent years, *e.g.*, [ADGFT01, ADGFT03, ADGFT04, MOZ05, HMSZ06]. One of the goals addressed in the work mentioned above was to weaken as much as possible the synchrony assumptions required for leader election. Another goal was to have an algorithm with two important properties: (i) stability and (ii) low message complexity. Stability means that the current leader is never demoted in favor of a new leader without reason. Low message complexity refers to the messages needed to monitor the leader. If the system consists of n processes, we would like ideally the monitoring to require only $O(n)$ messages. All these approaches ignore message latencies.

If the latencies of the different links in the system differ significantly, it might make sense to elect a leader that has links with low latency to the other processes in the system. This is typically the case for leader-based consensus algorithms in which the communication pattern is 1 to n, *i.e.*, communication is only between the leader and the other processes. Indeed in this case, choosing a leader with low latencies to other processes increases the efficiency of the leader-based consensus algorithm.

Taking Paxos as an example, the algorithm progresses either when the leader receives a majority of messages or the other processes receive a message from the leader, all of which will complete faster if the leader is the process with the lowest overall latency. To give a concrete example, consider three replicas, p_1 and p_2 in the same LAN and p_3 in a remote LAN. The best performance is obtained if the leader is either p_1 or p_2, as this allows a majority of replicas (two in this case) including the leader, to communicate quickly.

The idea of taking link latencies into account for leader election has been considered in [SBCF03, SB05, SNBJP05]. In [SBCF03], an adaptive consensus algorithm is given. The initial coordinator is chosen to be a "fast" process based on measurements of the time each process takes to answer when it is a coordinator. In [SB05], the notion of "process order" is used to convey information about the link latencies. A Process Order oracle running at each process measures the round trip time from the local process to every other process and computes the majority-RTT, *i.e.*, the time it takes for the local process to receive answers from a majority of processes, including itself. This information is then aggregated across all processes, and the consensus algorithm is used for processes to agree on a single consistent order. The result can be used to select successive coordinators by a consensus algorithm based on $\Diamond S$ and the rotating coordinator paradigm (instead of using the process id). A consensus algorithm based on Ω can also use the process order within the implementation of Ω to select a "fast" leader. However, the paper does not describe the implementation of Ω. In [SNBJP05], a predictor is used for estimating the best process, but the stochastic model in which this predictor is supposed to work is not specified.

Contribution The contribution of this chapter is to extend the above ideas by considering a latency-based system model, which is parameterized with a constant ε. We assume that during a stable period the latency of every link deviates only by ε from some (unknown) fixed value (where the value can be different for every link). The model allows an analytical analysis of the leader election algorithm (in contrast to the above best-effort approaches), and allows us to show that the leader election algorithm guarantees that during a stable period the leader latency to a majority of processes is optimal within 12ε. The optimal latency "to a majority" is related to the fact that, with benign faults, leader based consensus algorithms progress whenever the leader receives replies from a majority.

Roadmap We begin by presenting the system model and defining the problem of latency-aware leader election in Section 6.2. Section 6.3 presents our novel algorithm for leader election based on latencies, and Section 6.4 proves its correctness. In Section 6.5 we present a summary of an experimental evaluation comparing our latency-aware oracle with a conventional oracle. We conclude with Section 6.6. To improve the presentation of this chapter, some of the proofs were moved to Appendix C.

6.2 System Model and Problem Definition

6.2.1 Model

We consider a model with n processes Π, which communicate over a fully connected network via message passing. Up to $f < n/2$ processes may fail by crashing. We assume each process has a local clock with no skew[1] and that computational steps take no time. The timing behavior of the link from process i to process j is described by a function $\delta_{ij}(t)$, that is, if a message is sent from i to j at time t, it is received at time $t + \delta_{ij}(t)$. If at time t process i has crashed, we define $\delta_{ij}(t) := \infty$. Since a process sends at most one message to another process at a certain time, this function is well-defined. If no message is sent at a time t, $\delta_{ij}(t)$ has no meaning, but to simplify presentation we assume that it has the same value as the last time i sent a message to j.

We define further the function $rtt_{ij}(t) := \delta_{ij}(t) + \delta_{ji}(t + \delta_{ij}(t))$, which intuitively models the round trip delay of a message if the peer process algorithm responds immediately to this message. Additionally, $RTT_i(t)$ denotes the vector of $rtt_{ij}(t)$ for a process i, sorted by ascending order of values. Finally, we define $majrtt_i(t) := RTT_i(t)[\lfloor n/2 \rfloor + 1]$, which models the time required for a two-way message exchange between i and a majority of processes.

We assume that the transmission delays in our system eventually stabilize. In more detail, we assume that there is a time GST, so that for all times $t \geq GST$ the following holds:

- All faulty processes have crashed.
- There is an a priori known value Δ, s.t. for each pair of correct processes i and j:

$$\delta_{ij}(t) \leq \Delta$$

- All message transmission delays remain within a window of 2ε:

$$\forall i, j \; \exists \bar{\delta}_{ij} : \left| \delta_{ij}(t) - \bar{\delta}_{ij} \right| \leq \varepsilon$$

For a given run, for each link, fix one $\overline{\delta}_{ij}$ that satisfies the third condition. Then, in compliance with the definitions above, we can define

[1]The results can be adapted for clocks with skew.

- **RTT of link between *i* and *j***: $\overline{rtt}_{ij} := \overline{\delta}_{ij} + \overline{\delta}_{ji}$
- **RTT of process *i***: $\overline{RTT}_i$ as the vector of $\overline{rtt}_{ij}$ for process i, sorted by ascending order of values
- **Majority-RTT**: $\overline{majrtt_i} := \overline{RTT}_i[\lfloor n/2 \rfloor + 1]$

We now show that after GST, the round-trip time between two processes i and j remains within a window of 4ε centered around $\overline{rtt}_{ij}$:

Lemma 6.1. *For $t > GST$, we have $\left|rtt_{ij}(t) - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ and $\left|majrtt_i(t) - \overline{majrtt_i}\right| \leq 2\varepsilon$.*

Proof. From the definitions we get:

$$\begin{aligned}\left|rtt_{ij}(t) - \overline{rtt}_{ij}\right| &= \left|\delta_{ij}(t) + \delta_{ji}(t + \delta_{ij}(t)) - \overline{\delta}_{ij} - \overline{\delta}_{ji}\right| \\ &\leq \left|\delta_{ij}(t) - \overline{\delta}_{ij}\right| + \left|\delta_{ji}(t + \delta_{ij}(t)) - \overline{\delta}_{ji}\right| \\ &\leq 2\varepsilon\end{aligned}$$

For the second result, let $p_1, p_2, \ldots, p_n$ be the ordered list of processes corresponding to $\overline{RTT}_i$, that is, the processes ordered by increasing $\overline{rtt}_{ij}$ from i; let $q_1, q_2, \ldots, q_n$ be the similar list for $RTT_i(t)$, and let $m = \lfloor n/2 \rfloor + 1$. By definition, $\overline{rtt}_{ip_1} + 2\varepsilon \leq \overline{rtt}_{ip_2} + 2\varepsilon \leq \ldots \leq \overline{rtt}_{ip_m} + 2\varepsilon$. Since after GST, $rtt_{ij}(t) \leq \overline{rtt}_{ij} + 2\varepsilon$, we have $rtt_{ip_k}(t) \leq \overline{rtt}_{ip_m} + 2\varepsilon$, for all k, $1 \leq k \leq m$, and thus $majrtt_i(t) \leq \overline{majrtt_i} + 2\varepsilon$. A similar argument can be used to show that $majrtt_i(t) \geq \overline{majrtt_i} - 2\varepsilon$, thus proving the result. □

6.2.2 Optimal Leader Election

The classical definition of leader election in the literature is the following:

Definition 6.1 (Leader Election Problem)**.** *There exists a correct process ℓ and a time t after which, for every alive process p, $leader_p = \ell$.*

We need to extend this definition with the notion of optimality. Intuitively, the optimal leader is the process with the lowest majority-RTT. However, since in our model the message latencies may change even after GST (although within a bounded interval), the process with the lowest majority-RTT might also change frequently. Thus, a simplistic definition of optimal leader requiring strict optimality could result in frequent leadership changes even after GST, which would likely lead to poor performance of the system. Additionally, it is not evident that such a strict definition can even be implemented in practice, as the optimal process could change from the time the measurements are taken until the time the leader is elected. However, as we show later, we can guarantee that the elected leader is within a small interval from optimality, which is enough for a practical system. That is, it is only worth to demote a correct leader p in favor of another process q if this would provide "enough gains" to justify the cost of a leader change. If the differences are small, then it is better to maintain the current leader. The

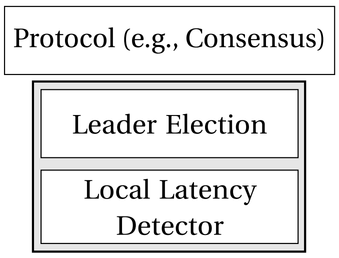


Figure 6.1: Structure of the latency-aware leader election oracle.

following definitions capture this idea of near-optimality, with σ representing the sensitiveness of the algorithm to latency changes (*e.g.*, formalizes the notion of "enough gains" to change leader).

Definition 6.2 (σ-delay-optimal process at time t)**.** *A process p is σ*-delay-optimal *at time t if there is no other process q that has a majority latency $majrtt_q(t)$ at time t that is smaller than $majrtt_p(t) - \sigma$.*

Note that there might be several σ-delay-optimal processes. The goal is to elect a leader among these.

Definition 6.3 (σ-Optimal Leader)**.** *There exists a correct process ℓ and a time t after which, for every alive process p, $leader_p = \ell$ and ℓ is σ-delay-optimal for all $t' \geq t$.*

6.3 σ-Optimal Leader election

We consider a two-layer implementation of our oracle as depicted in Figure 6.1. The local latency detector layer provides to the leader election algorithm a vector with the estimation of the RTT times from the local process to all other processes in the system. This vector is then taken in consideration to elect an optimal leader.

It is important to notice that the leader election algorithm uses the information of the latency detector only to optimize the choice of leader, and not to detect faulty leaders. The latter is done by sending heartbeat messages from the current leader to every other process. The reason for this approach is that measuring the latencies of the system is quite costly: it requires $O(n^2)$ messages in general, while a communication efficient implementation of the leader election algorithm requires only $O(n)$ links to carry the periodic heartbeat messages. Separating these two issues, allows implementations to control the trade-off between the number of messages sent and the reaction time to latency changes: faster reaction times can be obtained at a cost of increased network load by measuring the latencies more frequently. Implementations can therefore be tuned for each specific case, taking in consideration the underlying network and the QoS requirements.

6.3.1 The Local Latency Detector

The lowest layer of our system is the *local latency detector*. A local latency detector module periodically outputs an n-dimensional vector L_i at each process i, where $L_i[j]$ represents the estimate of the round-trip time between process i and process j. In contrast to the approach taken by [SB05], there is no global oracle providing global latency information to every process. Instead, every process gets only a local estimation of the round-trip delays to its peers. Note that, since even after GST round-trip delays may change, the output of the latency detector has only limited accuracy.

The implementation of such an oracle is done by sending every η time ping-pong messages between all processes to measure round-trip times. The sampling period η is a configuration parameter that controls the trade-off between network load (*i.e.*, the number of messages per time period) and speed of adaptation of the algorithm to changes in the latency of the links. Then, from the system model we get the following property:

Lemma 6.2 (local latency detector)**.** *Consider a latency detector that is implemented by every process sending pings to all every η time. Then we have*

$$\forall t > GST + \eta + 2\Delta, \forall i, j : |L_i[j] - \overline{rtt}_{ij}(t)| \leq 2\varepsilon.$$

Proof. After GST, the duration rtt of a round-trip which started at some time t is bounded by $|rtt - \overline{rtt}_{ij}(t)| \leq 2\varepsilon$. Since each round-trip takes at most 2Δ and measurements are done every time η, then the latest by $GST + \eta + 2\Delta$ the output of the detector is bounded. □

Note that GST and $\overline{rtt}_{ij}(t)$ are unknown system model parameters, ε is a known system model parameter, and η is an implementation-dependent parameter of the latency detector.

6.3.2 Electing a leader

The leader election algorithm is given as Algorithm 6.1 and Algorithm 6.2. The algorithm works by first electing a correct leader, with disregard for optimality. Then the leader aggregates the latency information collected by all processes and, based on this information, decides if it should grant leadership to another process with better latency. Below we explain these three ideas separately, although in the algorithm they work closely together. The proof of correctness is given in Section 6.4.

Leader Election

The leader election part of the algorithm is based on an algorithm from [ADGFT01]. The algorithm is organized as a sequence of rounds, with the leader of round r being process $r \bmod n$. Processes remain in round r as long as they believe process $r \bmod n$ should remain the leader, that is, it is correct and optimal. The leader p sends ALIVE messages to all processes

Algorithm 6.1 Delay-aware leader election.

```
 1: **Initialization:**
 2:    $\forall q:\ localrtt_p[q] \leftarrow \infty$
 3:    $r_p \leftarrow 1$
 4:    $StartRound(r_p)$

 5: **procedure** $StartRound(r, \tau)$
 6:    $r_p = r$
 7:    $leader = r_p \bmod n$
 8:    reset $timer$ to $\tau$
 9:    send $(\text{START}, r_p)$ to $leader$
10:    **if** $p = leader$ **then**
11:       $SendAlives()$

12: **procedure** $SendAlives()$
13:    $\forall p, q:\ rttMatrix_p[p][q] \leftarrow \infty$
14:    reset $timer$ to 0
15:    send $\langle \text{ALIVE}, r_p \rangle$ to all processes except $p$

16: **upon** receive $\langle \text{REPORT}, r, rtt \rangle$ from $q$ with $r = r_p$
17:    $rttMatrix_p[q] = rtt$

18: **upon** receive $\langle \text{ALIVE}, r \rangle$ from $q$
19:    **if** $r < r_p$ **then**
20:       send $\langle \text{START}, r_p \rangle$ to $q$
21:    **if** $r = r_p$ **then**
22:       reset $timer$
23:       send $\langle \text{REPORT}, r_p, localrtt_p \rangle$ to $leader$
24:    **if** $r > r_p$ **then**
25:       $StartRound(r, 0)$

26: **upon** receive $\langle \text{START}, r \rangle$ with $r > r_p$
27:    $StartRound(r, 0)$

28: **upon** $timer = 2\Delta$
29:    **if** $p = leader$ **then**
30:       $rttMatrix_p[p] \leftarrow localrtt_p$
31:       $newLeader \leftarrow SelectLeader(rttMatrix_p, leader)$
32:       **if** $newLeader = leader$ **then**
33:          $SendAlives()$
34:       **else**
35:          $r \leftarrow$ smallest $k > r_p$ $s.t.$ $k \bmod n = p$
36:          $StartRound(r, \Delta)$

37: **upon** $timer = 3\Delta$
38:    $StartRound(r_p + 1, \Delta)$

39: **upon** new $rttVector$ from latency detector
40:    $localrtt_p \leftarrow rttVector$
```

every 2Δ time. If a process does not receive any message from the leader for more than 3Δ time, it suspects the leader to have failed and advances to the next round $r+1$ (lines 37–38). It then sends a $\langle \text{START}, r_p \rangle$ message with $r_p = r+1$ to the leader of round $r+1$ (line 9). Upon

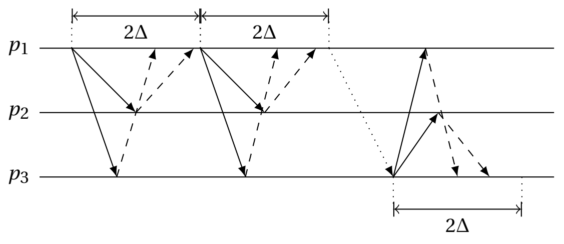


Figure 6.2: Message pattern of the election algorithm. Solid arrows represent ALIVE messages, dashed arrows represent REPORT messages, and dotted arrows represent START messages.

receiving this message, the new leader advances to the new round (lines 26–27) and sends ALIVE messages to all (line 11). This will force all other processes to advance to round $r+1$.

The selection of an optimal leader is done in lines 31, 35–36, where the current leader calls *SelectLeader*. This function encapsulates the election of an optimal leader, based on *rttMatrix*. If it returns a process different than the current leader, then the current leader advances to a higher round, forcing the election of the new process. *SelectLeader* is independent from the rest of the algorithm, and is explained in Section 6.3.2.

Aggregating RTT information

In order to elect a leader the local knowledge about latencies, *i.e.*, the output of the latency detector, is not sufficient. Instead, it is necessary to do some form of aggregation to obtain a more complete picture of the latencies in the system. In [SB05], the authors implement a global oracle that aggregates the latency information and ensures a consistent view of this latency across all processes, by proposing it as part of the higher level consensus protocol. Here we take a different approach, by aggregating the latency information only at the current leader. Our approach has the advantage of not requiring a consistent view of the latency information across all processes, instead using only the mechanisms that are already part of the leader election algorithm.

Every 2Δ time, the leader sends ALIVE messages (lines 15, 31–33) and resets the latency matrix *rttMatrix* (line 13). The other processes reply with a REPORT message (line 23). The leader uses these messages to rebuild *rttMatrix* (line 17). This way, 2Δ after sending ALIVE, the *rttMatrix* matrix of the leader contains estimates that are no older than 2Δ. Lemma 6.2 shows that if the ALIVE messages are sent after $GST+\eta+2\Delta$, then the *rttMatrix* is such that $\left|rttMatrix[i][j]-\overline{rtt}_{ij}\right| \leq 2\varepsilon$.

The REPORT message is sent after every ALIVE received, even if the local rtt vector did not change. Rebuilding the *rttMatrix* from scratch every 2Δ ensures that if some process i crashes, after the next ALIVE cycle, for all j, we have $rttMatrix[i][j]=\infty$. Note that if old values were kept until updated by a REPORT message, *rttMatrix*$[i]$ would never be updated.

Figure 6.2 illustrates the algorithm. Initially p_1 is the leader. After the first cycle, the *rttMatrix*

Algorithm 6.2 Choosing a leader.

```
procedure SelectLeader($L$, $leader$)
    for all lines $i \in \{1, \dots, n\}$ do
        sort $L[i]$ by increasing latencies
    $curRtt \leftarrow L[leader][\lfloor n/2 \rfloor + 1]$
    $minRtt \leftarrow \min_{i \in \{1,\dots,n\}} \{L[i][\lfloor n/2 \rfloor + 1]\}$
    if $minRtt < curRtt - 4\varepsilon$ then
        return $\min\{i \in \{1, \dots, n\} \mid L[i][\lfloor n/2 \rfloor + 1] = minRtt\}$
    else
        return $leader$
```

built with the ALIVE messages from the other processes still indicates that p_1 is the most suitable process to be the leader, so p_1 sends a new cycle of ALIVE messages. In the meanwhile, new measurements were taken by some process and the *rttMatrix* built by p_1 at the end of the second cycle shows that p_3 is now better suited to be the leader. So p_1 advances to the next round where p_3 is the leader and sends a START message to p_3. Upon receiving it, p_3 advances rounds, assumes leadership and starts sending ALIVE messages.

Selecting a σ-optimal leader

The protocol explained so far ensures that in a good period (i) no faulty process is elected as leader and (ii) the current leader periodically calls *SelectLeader* to verify whether it is still σ-optimal.

We describe now *SelectLeader*, see Algorithm 6.2. Note that from Section 6.3.2, the leader gets a matrix L for which Lemma 6.2 holds. *SelectLeader* starts by ordering each line of the matrix by increasing latencies. As a result, for all i, element $(i, \lfloor n/2 \rfloor + 1)$ of the matrix gives an estimate of the RTT of a majority of processes to process i.

The key point here is that a new leader is elected only if its majority-RTT is 4ε better than the current one (lines 6–7). Since by Lemma 6.1 the majority-RTTs of processes vary at most by $\pm 2\varepsilon$ after GST, this prevents the role of leader from oscillating between two processes. Moreover, as shown in the proofs, the selection rule ensures that if the leader changes, the new leader is a process with a lower $\overline{majrtt}$. Thus, even if the leader changes after GST, this happens only a finite number of times (the $\overline{majrtt}$ are fixed), and thus eventually a single correct process is leader forever. This process is an 12ε-optimal leader:

Proposition 6.1. *Algorithms 6.1 and 6.2 solve the σ-optimal leader election problem with $\sigma = 12\varepsilon$.*

6.4 Proof of correctness

We first prove that after $GST + \eta + 2\Delta$ the function *SelectLeader* returns only processes whose $\overline{majrtt}$ is within 8ϵ of the optimal. Then, we prove that after GST once a leader is elected the leadership will change only to another process that has a lower $\overline{majrtt}$. Therefore, the number of possible changes is limited and the system eventually elects a leader forever that is within 12ϵ of the optimal.

For the proofs below, let p_m denote the process with minimal $\overline{majrtt}_{p_m}$; if there are several such processes, the one with the lowest id.

Definition 6.4 (Optimal set). *Let $\mathcal{O} \subseteq \Pi$ be the set of optimal processes, i.e., $\mathcal{O} = \{p \in \Pi \mid \overline{majrtt}_p \leq \overline{majrtt}_{p_m} + 8\epsilon\}$*

Since for $t > GST, |majrtt_{p_m}(t) - \overline{majrtt}_{p_m}| \leq 2\epsilon$, we have:

Lemma 6.3. *Every process from $\mathcal{O}$ is 12ϵ-optimal at any time after GST.*

Lemma 6.4. *The set $\mathcal{O}$ is non-empty and contains only correct processes.*

Proof. Trivially, the set is non-empty because $p_m \in \mathcal{O}$. By definition, if p has crashed, then for all process j, $\overline{\delta}_{pj} = \infty$ and thus $\overline{majrtt}_p = \infty$. Therefore, $p \notin \mathcal{O}$. □

Lemma 6.5. *Let M a matrix with $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\epsilon$ for all $i, j \in \Pi$. Then SelectLeader(M, ℓ) returns a process p such that: (i) $p \in \mathcal{O}$ and (ii) if $p \neq \ell$ then $\overline{majrtt}_p < \overline{majrtt}_\ell$.*

Lemma 6.6. *After time $GST + \eta + 4\Delta$, if SelectLeader(M, ℓ) is called, $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\epsilon$ for all $i, j \in \Pi$.*

From (ii) of Lemma 6.5 and Lemma 6.6, and the fact that we have a finite number of processes, we get:

Corollary 6.1. *After $GST + \eta + 4\Delta$, the leader changes only a finite number of times.*

Next we show that if no new round is started after GST, then the optimal leader is elected. The proof of Proposition 6.1 discusses the other case, when new rounds are started after GST.

Lemma 6.7. *If there is a time t after which the maximum round reached by any process does not increase, then eventually a process from $\mathcal{O}$ is elected as leader.*

The following lemma shows that after GST, once there is a leader ℓ, the leader will only change as a result of ℓ calling *SelectLeader*. Together with Lemmas 6.5 and 6.6 this shows that the leader only changes to processes with better performance.

Lemma 6.8. *Let ℓ be a process that at time $t > GST$, is in round r_ℓ for which ℓ is the leader. Let t' be the time when ℓ sends the next* ALIVE *message. If all alive processes are in a round not higher than r_ℓ when they receive the* ALIVE *message from ℓ, then (i) no process $q \neq \ell$ advances to a round $r' > r_\ell$ while ℓ remains in round r_ℓ, and (ii) ℓ only advances to a new round if SelectLeader$_\ell(M, \ell)$ returns a process different than ℓ.*

Lemma 6.9. *Assume there is a process p and a round r_p such that p is the first process starting round r_p at time t_0, $t_0 > GST + 4\Delta$. Then before time $t_0 + 2\Delta$ no process will advance to a round $r' > r_p$.*

Proposition 6.2. *Algorithms 6.1 and 6.2 solve the σ-optimal leader election problem with $\sigma = 12\varepsilon$.*

Proof. We prove that eventually a permanent leader $\ell \in \mathcal{O}$ is elected. Then, by Corollary 6.3 it comes that ℓ is 12ε-optimal.

Let $t_s = GST + \eta + 4\Delta$. If after t_s no new round is started, then by Lemma 6.7, a process $\ell \in \mathcal{O}$ is eventually elected. We now prove the other case, where some new round is started after t_s. Assume there is a process p and a round r_p such that p is the first process starting round r_p after t_s at time t_0. We consider two cases: (i) p is the leader for round r_p and (ii) p is not the leader for round r_p.

Starting by (i), p sends ALIVE messages to all processes at time t_0. These messages are received the latest by time $t_0 + \Delta$. By Lemma 6.9 no process is in a higher round at this time. We can now use Lemma 6.8 to conclude that no process advances to a new round unless a call to *SelectLeader* made by p returns a process different than p. Since $t_0 \geq GST + \eta + 4\Delta$, by Lemmas 6.5 and 6.6 we know that all future invocations of *SelectLeader* will return a process $\ell \in \mathcal{O}$. If $p \notin \mathcal{O}$ then the first invocation of *SelectLeader* by p will return a process $\ell \in \mathcal{O}$, making p pass leadership to ℓ. Otherwise, p may or may not remain leader forever, but if the leader changes, it will always be to a process in $\mathcal{O}$. Either way, by Corollary 6.1, the leader can only change a finite number of times after t_0 and eventually a σ-optimal process is elected permanently.

In case (ii), p sends a START message to the leader q of round r_p and waits for 2Δ (Line 8 sets the timer to Δ and the timer expires at 3Δ for non-leader processes). Process q may either be alive or crashed. If q is alive, then by time $t_0 + \Delta$, q receives the message and advances to round r_p. At the same time it sends ALIVE messages to all, which arrive by time $t_0 + 2\Delta$. By Lemma 6.9, no process is in a higher round at this time. We are now in the same conditions as in case (i), so the same reasoning applies. If q is crashed, p will timeout at time $t_0 + 2\Delta$ and advance to round $r_p + 1$. Since by Lemma 6.9, no process other than p advances to a round higher than r_p before $t_0 + 2\Delta$, p is the first in round $r_p + 1$. We are again on the conditions of this Lemma, so we can apply the same reasoning to conclude that p will try to contact the leader for round $r_p + 1$. Since there are a maximum of f crashed processes, eventually p will reach a round where the leader q is alive and we can apply the same reasoning as above to conclude that a σ-delay-optimal leader is elected. □

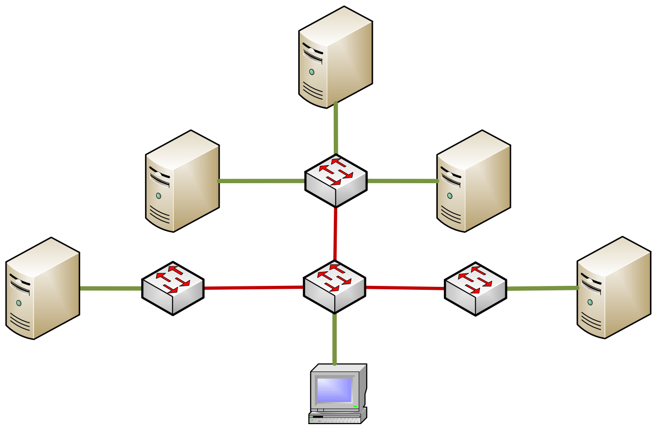


Figure 6.3: Topology used for experiments in ModelNet.

6.5 Experimental evaluation

The following experimental evaluation[2] compares the latency-aware leader election algorithm proposed in this chapter with the algorithm in [ADGFT01], which is a traditional election algorithm that ignores link latencies. Since the algorithm in [ADGFT01] was used as the basis for our work, the two algorithms differ only on the ability to adapt to the latencies in the network.

Although latency-awareness may be useful in a network with symmetric latencies, as is typically the case of a LAN[3], it is mainly targeted at situations where the link latencies are asymmetric, like in a WAN. Therefore, we have used ModelNet [VYW$^+$02] to emulate a topology with asymmetric links representative of WAN settings.

Figure 6.3 shows the topology used for the experiments. We use a total of five replicas, with three of them having 'good' connections among each other (marked green), while the other two have comparatively 'bad' connections (marked red) to each other and to the first group. In addition to latency, we also simulate message loss since this is a relatively frequent occurrence in a WAN. This simulates, for instance, a situation where three replicas are in the same data center and the other two are in remote data centers, so that the quality of the connections between the replicas in the data center is better than the one to the external replicas. We have performed experiments with varying levels of message loss and latency, as shown in Table 6.1.

In the topology considered in this experiment, we expect the system to perform better if the leader is chosen among the group of fast replicas, as these three replicas are capable of

[2] The work presented in this section was done by Benjamin Donzé under the supervision of the author of this thesis, as part of a semester project during the Master in Computer Science at the EPFL. We include here only a summary of the results. The complete evaluation can be found in [Don10].

[3]Our election algorithm will tend to elect 'fast' processes, since a process that is slower than the others (maybe because of running an external workload) will take longer to answer to pings and thus exhibit a higher RTT.

	Experiment	**0a**	**0b**	**1a**	**1b**	**2a**	**2b**
Green links	Loss (%)	0	0	0.1	0.1	0.1	0.1
	Latency (ms)	1	1	1	1	1	1
Red links	Loss (%)	0	0	0.1	1.1	0.5	0.5
	Latency (ms)	30	60	30	60	30	60

Table 6.1: Emulated network parameters

deciding without waiting for the messages from the other two replicas[4].

We present here two sets of experiments, each measuring a different value:

- **Majority RTT**: Time needed for the leader to send a message to all processes and obtain replies from a majority of processes.
- **Client response time**: Time elapsed since the client sends a request to the replicated service until it receives the reply.

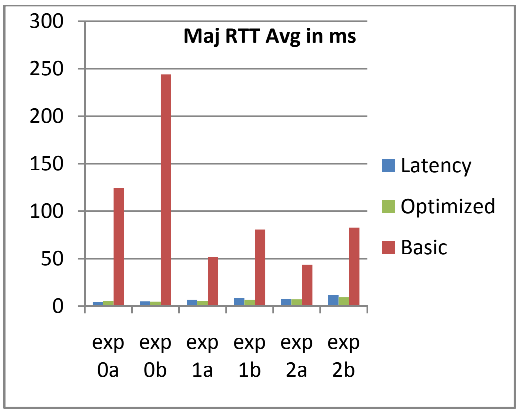
We report the average of all the measurements done during the experiments.

For the first set of experiments we have run the leader election algorithm by itself, while for the second set we ran it as part of a replicated system consisting of a (null) service replicated using the implementation of Paxos described in Chapter 7 and accessed by a single client. In the latter case, the client is required to send the requests directly to the leader. This creates a difficulty in choosing the location of the client, since the response time includes not only the time needed to order a request, but also the round-trip time between client and leader. Therefore, if the client is closer to some replica than others, this will skew the results in favor of this replica. To avoid these scenarios, we defined a topology such that the client has the same latency to all replicas. The client sends requests sequentially, waiting two seconds after receiving a reply before sending the next request.

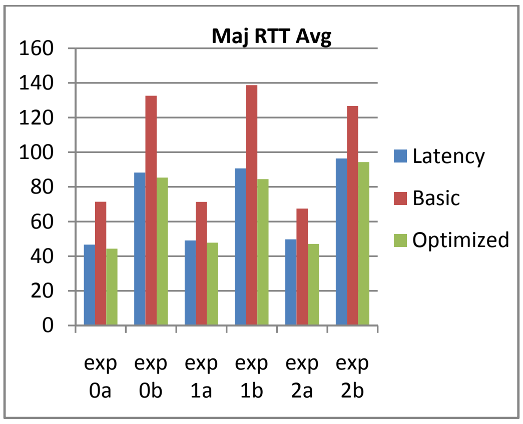
In the experiments measuring the Majority RTT of the leader, we have also tested the system in the presence of crashes. Crashes are simulated by having each process 'crash' itself periodically according to a probability distribution. Each process randomly schedules a crash using a Gaussian distribution of mean 6 minutes and standard deviation of 1 minute. When a process crashes it remains crashed during 1 minute, after which it restarts and schedules again a crash with the same distribution. It also resets the state of the leader oracle. The experiments simulating crashes were run for five hours, instead of the one hour used for all other experiments.

In all experiments, the leader election algorithm and the latency detector are configured with the following parameters:

[4] As the other two replicas cannot be allowed to lag behind too much, the main group may eventually have to stop processing new requests until the slow replicas catch-up. However, a 'fast' quorum is well suited to deal with bursty traffic. Additionally, if the 'slow' links have enough bandwidth for all the traffic generated by the protocol, then the lag will remain constant.



(a) No crashes



(b) Crashes

Figure 6.4: Majority RTT average of leader process

- $\Delta = 500$ ms: Thus an alive message is sent every 1s and a leader is suspected to have crashed if no alive message arrive during 1.5s
- $\epsilon = 15$ ms: A 'better' leader is selected if its *majrtt* is at least 60ms smaller (4ϵ) than the one of the current leader. This parameter is only used by the latency-aware algorithm.
- Ping-period = 1000 ms: The latency detector sends a ping message to other processes every second.

We report the average three series of results: *Latency* refers to the latency-aware algorithm, *Basic* to the algorithm from [ADGFT01] which is used as a control, and *Optimized* to a variant of the latency-aware algorithm that reduces sensitivity to message loss.

The *Optimized* version addresses a problem we found earlier in the experimental evaluation, which showed that in the presence of message loss, the latency-aware algorithm has a tendency to demote a correct optimal leader frequently. This happens because if a ping-pong message is lost, the latency detector will wrongly report a RTT value for the concerned process equal to infinite. This may lead to the following case: the leader (p_1) obtains a local RTT vector equal to $(0, 2, \infty, 60, 60)$ and another "good" process p reports a RTT vector equal to $(2, 0, 2, 60, 60)$. In that case the leader believe it needs 60 ms to contact a majority of processes, while process p can contact a majority of processes within 2 seconds. The current leader will thus force the election of the process p as the leader. To minimize wrong demotions of optimal leaders, in the *Optimized* variant the latency-detector reports ∞ for a link only when it does not receive an answer to the ping messages sent in that link for two consecutive rounds, contrary to the *Latency* version which report ∞ is it fails to receive even a single answer.

6.5.1 Leader Majority-RTT

Before looking at the experimental results for the majority RTT in Figure 6.4, it is worth to first determine what is the expected majority RTT depending on the location of the leader. If the leader is among the group of three replicas with good connectivity, then a round of communication with a majority of replicas takes four transmission delays on green links (*i.e.*, 4ms). However, if the leader is in one of the other replicas, it requires crossing green links four times and red links also four times. This corresponds to 128ms or 248ms, for red links with 30ms and 60ms latencies, respectively.

In the experiments without crashes (Figure 6.4a), the latency-aware leader oracles outperform the basic algorithm by a large margin. The latency-aware oracles have an average majority RTT near or below 10ms, which is what is expected when the leader is in the well-connected majority. In contrast, the results for the basic oracle always exceed 50ms, even reaching 250ms in one experiment. These results deserve a closer look, as at first it looks puzzling that the experiments without message loss (0a and 0b) show worse results than the experiments with message loss (1a, 1b, 2a and 2b) for the basic protocol. This is because, in order to illustrate the worst-case scenario, we have set the system in a way the first elected leader is not delay-optimal. In the experiments without message loss, as the results show, the latency-aware oracles quickly realize that the leader is non-optimal and reelects an optimal leader, while the basic algorithm keeps this “bad” leader during the whole experiment. This explains the majority RTT of 130ms and 250ms in experiments 0a and 0b with the basic protocol. However, when there is message loss, the leader role will move around randomly, and will more often be on the fast group than on the slow group. This lowers the average RTT for the basic protocol.

In the experiments with crashes (Figure 6.4b) the gains of the latency-aware oracles are smaller, although still significant. This is because when one of the “good” replicas is crashed there is no more a fast connected majority of processes. In that case any replica can be designated as the leader without breaking the rules of the latency-aware algorithm.

6.5.2 Client response time

Figure 6.5 shows the client response time. The latency-aware oracles outperform the basic oracle in every experiment.

The results also show that the optimized variant is effective at improving the performance in the presence of message loss. In experiments 0a and 0b, as expected the optimization does not provide any gains since there is no message loss.

Contrary to the majority RTT, the client response never drops below 50ms. This is because the client messages must always pass through at least a red link, which creates a lower bound for the response time of at least two times the latency of the red link.

We can also observe that the average response time increases with the loss probability. This is

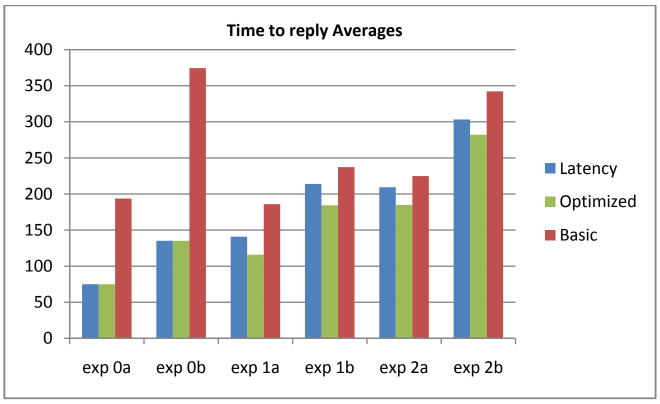


Figure 6.5: Client reply time. No crashes.

because of the additional leader changes (wrong suspicions). When the client sends a request to a process which is no more the leader, this process will reply with a message containing the identity of the current leader, and the client will then resend its request to the valid leader. These cases increase the average response time significantly.

6.6 Conclusion

In this chapter, we have given a model where the latencies of all links between correct processes eventually stabilize, so that they are within an interval of $\pm\varepsilon$ around some fixed value. We have given and proved correct an algorithm that elects as leader a process that is 12ε-optimal after the global stabilization time.

From a practical point of view, for algorithms like Paxos [Lam98], a leader needs to be the same only for a bounded time interval to solve consensus. In such a system, the parameter ε trades stability against optimality: with a small value of ε, the algorithm adapts to smaller changes in the network and thus selects processes that are closer to the optimal, risking, however, frequent leader changes. A large value of ε on the other hand favors long-lived leaders over optimality.

High-performance State Machine Replication Part II

7 JPaxos - A research prototype of State Machine Replication

The second part of this thesis studies State Machine Replication (SMR) from a mostly practical/experimental perspective, investigating several approaches to improving its performance. Towards this goal, we have implemented a fully-featured prototype of State Machine Replication based on the Paxos protocol in the Java language.

This chapter provides the background for the second part of the thesis. We start with an overview of the State Machine Replication approach to building fault-tolerant distributed systems, followed by a brief description of Paxos. We then discuss in general terms the architecture and implementation of JPaxos, and conclude with a overview of the research contributions presented in the following chapters.

7.1 State Machine Replication

State Machine Replication is a popular technique to make a service fault-tolerant [Lam98, Sch90]. In the SMR approach, the service is modeled as a deterministic state machine, with the state transition consisting of the execution of client requests. Being deterministic means that the state of the service at any point in time, is determined only by the initial state and the sequence of commands executed until that time. Given such a service, the SMR approach replicates the service in several machines. Since the service is deterministic, consistency among the replicas can be ensured by having each replica execute client requests in the same order.

Since many services are by nature deterministic, State Machine Replication is a very general approach that can be applied to a wide variety of services. It is also very effective at ensuring uninterrupted operation in the presence of (a limited number of) crashes, since correct replicas can usually continue serving client requests without any interruption (contrary to primary-based approaches which exhibit some downtime during fail-over). For these reasons, the state machine replication approach has been widely considered by both the theoretical [Lam98, Lam06, LMZ10] and systems research community [BBH+11, RST11a] and is also used in several real-world systems [Bur06, HKJR10, MMN$^+$04].

7.1.1 State Machine Replication and Atomic Broadcast

The complexity of implementing SMR lies at the protocol used to ensure that requests are executed in the same order by all replicas. This is essentially the Atomic Broadcast problem, which was presented in Section 2.4.2, with the client requests being the messages ordered by the protocol.

There is a fair amount of literature on different atomic broadcast protocols and their implementations. In their survey Défago et al. [DSU04] identify five classes of atomic broadcast algorithms. Based around the question "Who builds the order?" they classify algorithms into fixed-sequencer, moving-sequencer, privilege-based, destination-agreement and communication-history protocols.

In the *fixed-sequencer* category one process is elected to become the sequencer, which will then be responsible for imposing a unique order on messages. Clients send the request directly or indirectly to the sequencer, which then assigns them an order and broadcasts the chosen order to all other processes. Normally, in the absence of faults, the role of the sequencer does not change. This is the most popular approach to implement SMR, with the canonical example being the Paxos [Lam98] protocol.

The *moving-sequencer* class is a variant of the sequencer strategy, where the role of the sequencer is transfered periodically between processes. The goal is to balance among all processes the load of sequencing requests, thereby avoiding the bottleneck that usually occurs with fixed-sequencer algorithms at the sequencer process. In Chapter 10 we show that a fixed-sequencer protocol does not necessarily has to suffer from the leader bottleneck.

In *privilege-based* protocols the order is established by the sender when it broadcasts the message. To enforce a unique order, at any given time there is a single process that is allowed to broadcast messages, with this privilege circulating among processes so that all have a chance of sending messages. The privilege to broadcast can be either represented by a token or, in synchronous systems, by pre-assigned time slots. This class of algorithms differs from the moving-sequencer in that a process has to wait for the token before sending a message, while on moving-sequencer algorithms processes can send a message at any time by relaying it to the current sequencer.

In *communication-history* algorithms senders can broadcast messages at any time, tagging each message with a logical or physical time stamp. The order is established locally at each process, after the message is received, by delaying the delivery of the message until the process has enough information to assign a final order to the message. Logical clocks are a common way of establishing order in these algorithms [Lam78].

Finally, in *destination-agreement* the processes that receive the message agree explicitly on the order by which they are going to deliver the messages. The most popular example of such algorithms is the reduction of atomic broadcast to consensus in [CT96], where the destination processes execute consensus to agree on the next group of messages to be delivered (messages within each group are ordered using a deterministic function on messages ids).

In the rest of this thesis we study the Paxos [Lam98] protocol, which belongs to the fixed-sequencer group. There are some variants of Paxos that fall in the moving-sequencer category [MJM08], which are also related to the work presented in here. The other approaches are not commonly used to implement State Machine Replication.

7.1.2 Paxos

In recent years the Paxos [Lam98] algorithm and its variations have become the de-facto industry standard in state machine replication [Bur06, HKJR10, MMN+04, RST11a]. According to the terminology presented above, Paxos can be seen as a sequencer-based atomic broadcast protocol, where the sequencer orders requests received from the clients. In the Paxos terminology, the sequencer is called *leader*. Although Paxos is usually described in terms of the roles of proposer, acceptor and learner, this distinction is not relevant for the work in this thesis so we ignore it and assume that every process is at the same time proposer, acceptor and learner. Moreover, we will refer to a replica which is not the leader as *follower*.

Paxos is designed for the partially synchronous systems with benign faults, and requires $n \geq 2f + 1$ replicas to tolerate f crash failures. It also requires a leader election oracle, like the one proposed in Chapter 6.

The algorithmic ideas of Paxos are very similar to the ones underlying the LastVoting 3 and LastVoting 4 algorithms analyzed in Chapter 4 in the context of the HO round model. Here we present Paxos as described originally, *i.e.*, for the partially synchronous system.

In Paxos, the replica elected as leader is responsible for coordinating the other replicas with the goal of ordering the requests received from the clients. It does this by executing a series of consensus instances, each of them trying to assign one or more requests to a particular sequence number.

The consensus protocol at the heart of Paxos, called *Synod*, consists of two phases. Assume that the leader is trying to assign a value (client request) to sequence number i. In Phase 1 the leader starts by choosing from a private pool of numbers a proposal number k higher than any other proposal number it has seen so far. This proposal number is needed because multiple processes may attempt to assign different values to the same sequence number i concurrently (*i.e.*, when several processes consider themselves leader at the same time). By requiring each such attempt, *i.e.*, consensus instance, to have a monotonically increasing proposal number, it is possible to solve these conflicts by always giving preference to the attempt with the highest proposal number.

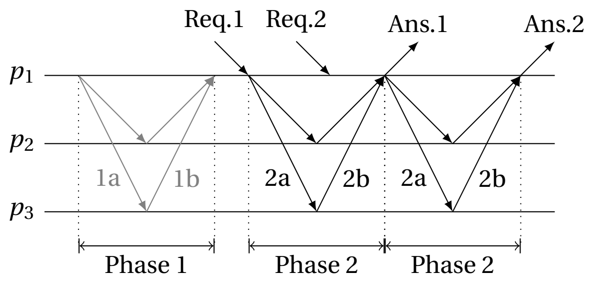


Figure 7.1: Message pattern of MultiPaxos

After choosing a proposal number, the leader sends a $\langle \textsc{Phase1A}, k, i\rangle$ message to all replicas. The other replicas reply with a $\langle \textsc{Phase1B}, k, i, [k', x]\rangle$ message if they have not yet seen any proposal higher than k (otherwise they may simply ignore the message). Additionally, if they have already participated in a consensus instance for the same sequence number i but with a lower proposal number, they include in the $\textsc{Phase1A}$ message the pair $\langle k', x\rangle$ where k' is the highest proposal number they have received for sequence number i and x the associated value (if any). Once the leader receives a majority of Phase 1B messages, it advances to Phase 2, where it tries to get a majority of replicas to accept a value for sequence number i. If some of the Phase 1B messages contained a value, then it uses as proposal the value associated with the highest proposal number received. Otherwise, the leader is free to propose any value. It then sends a $\langle \textsc{Phase2A}, k, i, x\rangle$ message to all replicas. The replicas accept this proposal if in the meantime they did not receive any proposal number higher than k, answering with a $\langle \textsc{Phase2B}, k, i\rangle$ message. Once the leader receives a majority of positive replies, it decides the value, *i.e.*, permanently assigns sequence number i to the request contained in the value decided. The leader must then inform the other replicas of the decision. Alternatively, the replicas may send the Phase 2b messages to all, which allows every replica to decide by itself after receiving a majority of messages.[1]

The Paxos protocol as described above is rarely used in practice. Instead, implementations typically use a variant called *MultiPaxos*, which improves on the algorithm described above by reducing from four to two the number of communication delays needed to order a request in the common case. The key observation behind MultiPaxos is that when a process is elected leader, it can execute Phase 1 for an infinite number of instances, which then allows it to execute only Phase 2 for every subsequent instance (until it is replaced by another leader). Figure 7.1 illustrates the message pattern of MultiPaxos. In the following, we use the term *instance* as an abbreviation for *one instance of Phase 2*.

Batching and pipelining Two optimizations that greatly improve the performance of Paxos are batching and pipelining. *Batching* consists of grouping several client requests in the same ballot, while *pipelining* consists of executing several instances concurrently. Chapter 8

[1]In terms of the algorithms presented in Chapter 4, these two variants correspond to the LV4 and LV3 algorithms, respectively.

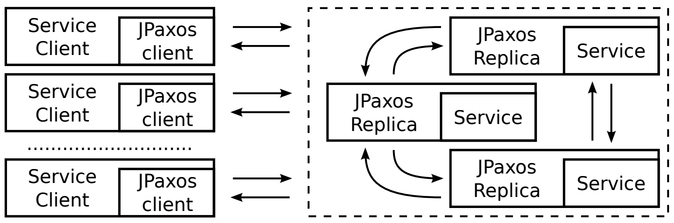


Figure 7.2: A service replicated in three replicas accessed by several clients. Arrows indicate communication flows.

discusses these optimizations in detail, showing how to tune them to achieve high-throughput.

7.2 JPaxos - Implementation of State Machine Replication based on Paxos

JPaxos[2] was developed in the context of this thesis as a tool for research into state machine replication.[3] It is a complete implementation of Paxos written in Java, with focus on modularity and extensibility.

The rest of this chapter briefly describes JPaxos: the high-level architecture is presented in Section 7.2.1, the storage and data-structures are discussed in Section 7.2.2, communication in Section 7.2.3, and the life cycle of requests in Section 7.2.4. The threading architecture and the extension of JPaxos for high-scalability are part of the research contribution of this thesis and are discussed in Chapters 9 and 10. The technical report [SKŻ+11] provides additional details.

7.2.1 Architecture

JPaxos consists of a Replica and a Client library. The *Replica* library executes the Paxos ordering protocol and calls the service state machine, while the *Client* library allows client applications to access the service. JPaxos also defines a *Service* interface that must be implemented by the service being replicated. Figure 7.2 shows a typical deployment.

Client

The `Client` library mediates the interaction between the application and the replicated service. It hides from the application most of the complexity of accessing a replicated service, including the discovery the alive replicas, request transmission and reception, and failure handling.

[2] Source code available at http://lsrwww.epfl.ch/page-55735-en.html
[3] JPaxos is joint work with Paweł T. Wojciechowski, Jan Kończak and Tomasz Żurkowski from Poznań University.

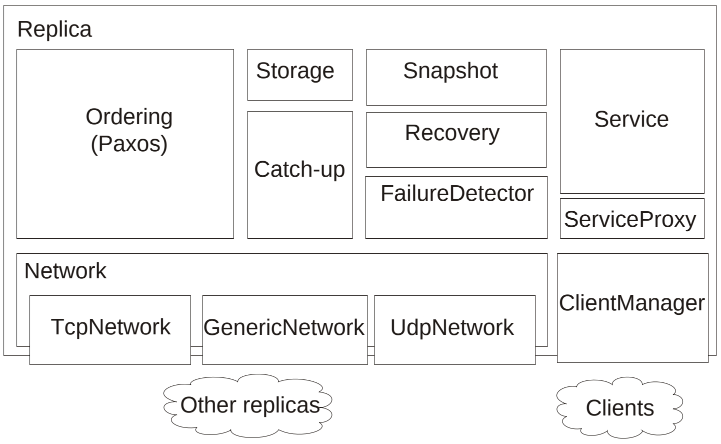


Figure 7.3: Block diagram of JPaxos modules

Service

The service implementation is provided by the application and, as already mentioned, must be deterministic. It must also implement several lifetime management methods including creating a snapshot of the state and recovering the state from a snapshot.

Replica

The Replica library orders requests using the Paxos protocol. Additionally, it implements many other modules responsible for managing the state and lifetime of the service, and interacts with the clients. Figure 7.3 shows the modules of the Replica library. Below we briefly describe each module.

- **ClientManager** This module manages the communication between clients and the replica. In particular, it listens for new TCP connections from the clients, receives the client requests, forwards them to the other modules for execution, and finally sends the answers back to the client.
- **Paxos** This module runs the ordering protocol on client requests.
- **Storage** Manages the state of the Paxos protocol, including the replicated log with the state of all instances started and auxiliary variables, like view number. The state is centralized in this module in order to simplify its management.
- **CatchUp** During execution some replicas might miss the decision of some instances, due to message loss or a crash, in which case they need to fill up the resulting gaps in the command sequence in order to continue executing requests. The CatchUp module is responsible for discovering the decisions of missing instances, by retrieving them from other replicas. This service is also used for recovery after a crash; after the service state

Variable	Description
log	the Paxos Log
view	current view
firstUncommitted	first instance not decided yet
snapshot	the most recent snapshot

Table 7.1: State kept by Storage module

is reinitialized from a snapshot, the CatchUp service brings up-to-date the state of the replica with any new commands that may have been decided since the snapshot was taken.

- **Snapshot** This module performs periodic snapshots of the state of the service to stable storage.
- **Recovery** Responsible for implementing recovery after a crash. This module supports several recovery algorithms, with different trade-offs between resilience and performance. During normal execution, and depending on the algorithm chosen, this module logs to stable storage information about the progress of the algorithm and adds some additional fields to the messages sent between replicas. During recovery, this module restores the state of the service from a snapshot or from the state of other replicas.
- **ServiceProxy** Interposes between the Replica and the Service provided by the user, and performs the bookkeeping required to manage snapshotting.
- **Network** This module manages the connections between replicas. It abstracts the details of network communication, exporting to other modules an interface with several `send` primitives and callbacks for receiving messages.

7.2.2 Storage and data structures

The protocol state kept by JPaxos consists mainly of the replicated log and a few auxiliary variables describing the current state of the process. Since these data structures are accessed by almost all other modules (including the ordering protocol, snapshotting and catchup), we have aggregated them in the Storage module. This significantly simplifies the management of the shared state. Table 7.1 shows the state kept in the *Storage* module.

The Paxos Log

Each replica contains its own copy of the log, with the replication algorithm being responsible for updating the log consistently in all replicas. The log contains a series of entries, each describing what the replica knows about the state of a particular consensus instance. Table 7.2 shows the information that is kept on the log for each consensus instance. An instance is in the UNKNOWN state if the replica did not start this instance yet, in the KNOWN state if the

Variable	Description
view	a view of the last received message related with this instance.
value	the value which is held by this instance.
state	one of UNKNOWN, KNOWN or DECIDED.
accepts	set of known replicas which accepted the (view, value) pair.

Table 7.2: State of a consensus instance

instance was started but not yet decided and finally it changes to the DECIDED state when the final decision is known.

Paxos is often presented using an infinite log, which simplifies its description considerably. But in practice the size of the log must be bounded. JPaxos achieves this by snapshotting periodically the service state, and keeping in the log only the instances that had not been executed at the time of the snapshot. Snapshotting and recovery are discussed in [SKŻ+11].

Auxiliary Storage state

The other variables kept by the Storage module are used to track the current state of the protocol execution: *view* is the current view of the process, *firstUncommitted* is the id of the lowest instance that was not yet decided, and *snapshot* is the last snapshot taken by the local replica.

7.2.3 Communication

Client-replicas communication

JPaxos uses persistent TCP connections between clients and replicas. Requests must be sent directly to the leader. The first time the client sends a request to the service, the client library connects to a replica chosen randomly since it does not know who is the leader. If a replica does not answer, the client tries the replica with the next id (modulo n). Follower replicas reply to connection requests from clients with a REDIRECT reply containing the id of their current leader. After connecting to the leader, the client library sends the request and waits for the answer. If the connection fails or the answer does not arrive within a certain time, the it assumes that the replica failed, and tries to reconnect to another replica, retransmitting the request when it succeeds.

Communication between replicas

For the communication with other replicas, the Network module supports TCP, UDP, and an hybrid mode combining UDP and TCP (GenericNetwork). This flexibility is necessary because of the trade-offs between the different communication protocols in the context of state machine replication; there is no single protocol that performs optimally in all deployment

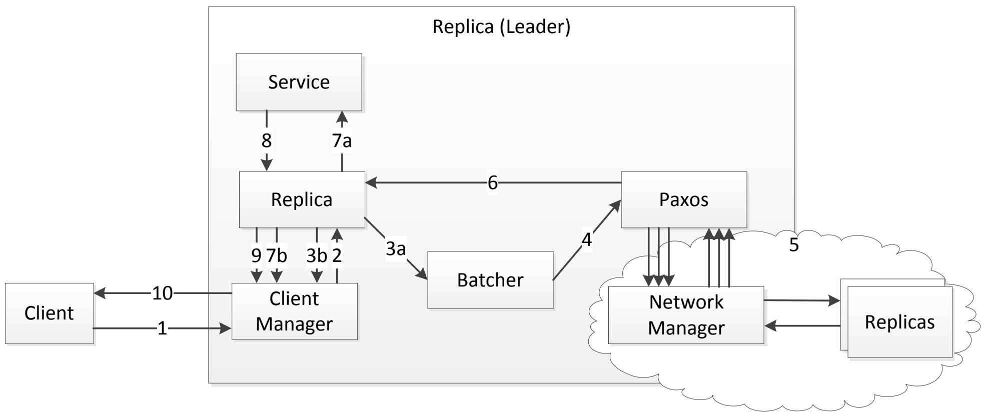


Figure 7.4: Request handling in JPaxos. 1) Client sends request, 2) request read and forwarded to Replica module, 3a) request added to batch queue (new request) or 3b) send cached answer (repeated request), 4) propose request as part of a batch, 5) order batch using Paxos, 6) after being ordered, batch is given to Replica for execution, 7a) Replica executes request in service if the request is new, or 7b) answers with cached reply if request is repeated, 8), 9) and 10) answer is sent back to client.

scenarios. Since JPaxos is a research project, supporting multiple protocols opens the door to conducting research into this topic.

TCP connections between the replicas are persistent. If a connection fails, the replicas try to reestablish it periodically, as the other replica may recover or it may have been only a connectivity problem. JPaxos retransmits the last message sent to a replica if the previous connection failed and the message must still be delivered (*i.e.*, if the corresponding consensus instances was not yet decided).

When using UDP, JPaxos retransmits some types of messages until either they are delivered or they become obsolete. This is the case of the Phase 1a and 2a messages, which are retransmitted to the replicas that have not yet answered with the corresponding Phase 1b or 2b message. Retransmission stops when the leader decides the corresponding instance.

The GenericNetwork uses UDP for small messages and TCP for large messages, where the threshold between small and large is a configuration parameter. This module addresses the message size limitation of UDP[4]. The GenericNetwork avoids the need to implement message fragmentation by delegating that work to TCP when the message does not fit in a UDP packet.

7.2.4 Request Handling

Figure 7.4 shows the life-cycle of a request in JPaxos. Clients send requests in a closed-loop, waiting for the answer of the previous request before transmitting the next (1). When a replica

[4] The theoretical limit is 64KB, but in practice the limit can be lower, depending on the underlying network.

receives a request (2), it first checks if the request was already executed, in which case it answers with the cached reply (3b and 10). If the request is new, it is dispatched for ordering and execution. Before being proposed, the request is handed out to the incoming queue of the Batcher module (3a), where it waits until it is included in a batch (See Chapter 8). The batch is then passed to the Paxos module (4), which then proposes and eventually decides on a order for the batch (5). Once ordered, the batch is given back to the Replica (6), which extracts the requests contained within. The individual requests in a batch are ordered in relation to each using their request id. The replica must once again check if the request has already been executed (which can happen when the client retransmits a request due to a connection failure or leader crash). If the request was not executed previously, it is executed in the service (7a) and the answer is sent back to the client (9 and 10). If the request was already executed, then the leader either answers with the cached reply or drops the request. The leader drops the request if it knows of a more recent request from the same client, as this indicates that the client already has the answer for the old request.

7.3 Contributions

Chapters 8 to 10 present our contributions in the context of State Machine Replication, with each focusing on a different approach to improving its performance. Chapter 8 shows how to tune the batching and pipelining optimizations of Paxos for optimal throughput. Chapter 9 looks at how multi-core CPUs can be used to improve the performance of the Paxos protocol, by proposing and evaluating a generic threading architecture whose performance scales with the number of cores. Finally, Chapter 10 proposes and evaluates S-Paxos, a high-throughput and scalable (up to reasonable values of *n*) variant of Paxos. S-Paxos achieves this result by balancing the workload in the system equally well across all replicas, thereby fully utilizing the resources of all replicas and eliminating the leader bottleneck that limits the throughput and scalability of other Paxos-like protocols.

8 Tuning Paxos for High-Throughput with Batching and Pipelining

The performance of Paxos can be greatly improved by using batching and pipelining. However, tuning these optimizations to achieve high-throughput is challenging, as their effectiveness depends on many parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. We address this question by presenting an analytical model of the performance of Paxos, and showing how it can be used to determine configuration values for the batching and pipelining optimizations that result in high-throughput. We then validate the model, by presenting the results of experiments where we investigate the interaction of these two optimizations both in LAN and WAN environments, and comparing these results with the prediction from the model. The results confirm the accuracy of the model, with the predicted values being usually very close to the ones that provide the highest performance in the experiments. Furthermore, both the model and the experiments give useful insights into the relative effectiveness of batching and pipelining. They show that although batching by itself provides the largest gains in all scenarios, in most cases combining it with pipelining provides a very significant additional increase in throughput, not only on high-latency network but also in many low-latency networks.

Publication: Nuno Santos and André Schiper. Tuning Paxos for high-throughput with batching and pipelining In *13rd International Conference on Distributed Computing and Networking (ICDCN 2012)*, Hong Kong, China, January 3-6, 2012

8.1 Introduction

In the simplest Paxos variant, henceforth called baseline Paxos, the leader orders one client request at a time. In general, this is very inefficient for two reasons. First, since ordering one request takes at least one network round-trip between the leader and the replicas, the throughput is bounded by $\frac{1}{2L}$, where L is the network latency. This dependency on the latency is undesirable, as it severely limits the throughput, especially in moderate to high latency networks. Second, if the request size is small, the fixed costs of executing an instance of the ordering protocol can become the dominant factor and quickly overload the CPU of the replicas.

In this chapter, we study two well-known optimizations to baseline Paxos that address these limitations: batching and pipelining. *Batching* consists of packing several requests in a single instance of the ordering protocol. The main benefit is amortizing the fixed per-instance costs over several requests, which results in a smaller per-request overhead and, usually, in higher throughput. *Pipelining* [Lam98] is an extension of baseline Paxos where the leader initiates new instances of the ordering protocol before the previous ones have completed. This optimization is particularly effective when the network latency is high as compared to the speed of the nodes, as it allows the leader to process additional instances of the ordering protocol during the time that would otherwise be spent idle waiting for the network.

Batching and pipelining are used by most replicated state machine implementations, as they usually provide performance gains of one to two orders of magnitude. Nevertheless, to achieve the highest throughput, they must be carefully tuned. With batching, the batch size controls the trade-off between throughput and response latency. With pipelining, the number of instances executed in parallel must be limited to avoid overloading the system resources, either the CPU of the replicas or the network, which could significantly degrade the performance. Moreover, the optimal choice for the bounds on the batch size and number of parallel instances depends on the properties of the system and of the application, mainly on process speed, bandwidth, latency, and size of client requests.

Contributions We begin by studying analytically the problem of finding the combinations of batch size and number of parallel instances that maximize throughput for a given system and workload. This relationship is expressed as a function $w = f(S_{batch})$, where S_{batch} is a batch size and w is a number of parallel instances (also denoted by window size). This result can be used to tune batching and pipelining, for instance, by setting the bounds on the batch and window size to one of the optimal combinations, so that given enough load the system reaches maximum throughput. To obtain the relation above, we developed an analytical model for Paxos, which predicts several performance metrics, including the instance and request throughput, the CPU and network utilization of an instance and of a request, and the wall-clock duration of an instance.

We then present the results of an experimental study comparing batching and pipelining in two settings, one representing a WAN and the other a cluster. We evaluate the gains obtained by using each of the optimizations alone and by combining them. The results show that although batching by itself provides the largest gains in all scenarios, in most cases combining it with pipelining provides a very significant additional increase in throughput, not only on high-latency network but also in many low-latency networks.

Finally, we compare these results with the prediction of our model, showing that the model is effective at predicting several performance metrics, including the throughput and optimal window size for a given batch size.

Roadmap The rest of the chapter is organized as follows. Section 8.2 presents the related work, and Section 8.3 provides the background for our work, describing in more detail the batching and pipelining optimizations in Paxos, Section 8.4 presents our analytical model of Paxos and shows how it can be used to tune these two optimizations, Section 8.5 presents the experimental evaluation of these two optimizations on LAN and WAN environments, and Section 8.6 discusses our results and concludes the chapter.

8.2 Related Work

The two optimizations to Paxos studied in this chapter are particular applications of general techniques widely used in distributed systems. Batching is an example of message aggregation, which has been previously studied as a way of reducing the fixed per-packet overhead by spreading it over a large number of data or messages, see [FR95, BCP$^+$03, CGH$^+$04, FH06]. Batching is widely used, with TCP's Nagle algorithm [Nag84] being a notable example. Pipelining is a general optimization technique, where several requests are processed in parallel to improve the utilization of resources that are only partially used by each request. One of the main examples of this technique is HTTP pipelining [PM95]. In our work, we look at these two optimizations in the context of state machine replication protocols, studying how to combine them in Paxos.

Most implementations of replicated state machines use batching and pipelining to improve performance, but as far as we are aware, there is no detailed study on combining these two optimizations. In [FR95], the authors use simulations to study the impact of batching on several group communication protocols. The authors conclude that batching provides one to two orders of magnitude gains both in latency and throughput. A more recent work [BCP$^+$03] proposes an adaptive batching policy also for group communication systems. In both cases the authors look only at batching. We show that pipelining should also be considered, as in some scenarios batching by itself is not enough for optimal performance.

Batching has been studied as a general technique by [CGH$^+$04] and [FH06]. In [CGH$^+$04] the authors present a detailed analytical study, quantifying the effects of batching on reliable message transmission protocols. One of the main difficulties of batching is deciding when to stop waiting for additional data and form a batch. This problem was studied in [FH06], where the authors propose two adaptive batching policies. The techniques proposed in these papers can easily be adapted to improve the batching policy used in our work, which was intentionally kept simple as it was not our main focus.

There are a few experimental studies showing the gains of batching in replicated state machines. One such example is [AK08], which describes an implementation of Paxos that uses batching to minimize the overhead of stable storage.

There has been much work on other optimizations for improving the performance of Paxos-like protocols. LCR [GLPQ10] is an atomic broadcast protocol based on a ring topology and vector clocks which is optimized for high throughput. Ring-Paxos [MPSP10] combines several techniques, to maximize network utilization, like IP multicast, ring topology, and a minimal quorum of acceptors. These two papers consider only a LAN environment and, therefore, use techniques that are only available in a LAN (IP multicast) or that are effective only if network latency is low (ring-like organization). We make no such assumptions in our work, which applies both to WAN and LAN environments.

Protocols like Mencius [MJM08] take a different approach at improving the throughput of baseline Paxos, by allowing every replica to coordinate different instances of the ordering phase concurrently. These so-called multi-coordinated protocols can be seen as an logical extension of the pipelining optimization discussed in this chapter, where instead of overlapping instances in the network links of a single replica (the leader), they overlap instances across the links of all replicas. The two approaches are orthogonal and, in principle, can be combined; individual replicas in a multi-coordinated protocol system could achieve higher throughput by pipelining the instances they coordinate.

8.3 Background

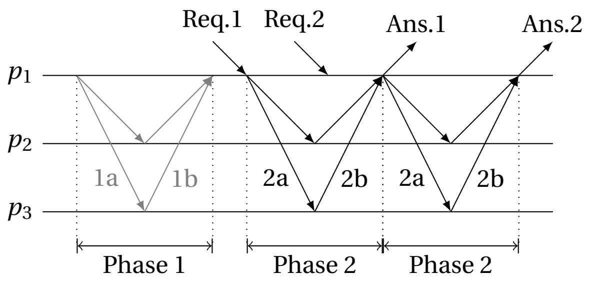
Recall from Section 7.1.2 that Paxos consists of two phases. Since Phase 1 is executed only when a leader is elected, it has a minimal impact on performance when faults are rare. Therefore we ignore Phase 1 in our analysis, and use the term *instance* as an abbreviation for *one instance of Phase 2*.

As seen in Chapter 7, in baseline Paxos the leader proposes one request per instance and executes one instance at a time (Figure 8.1a).

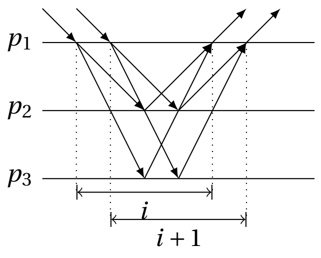
Pipelining As shown in [Lam98], the Paxos protocol can be extended to allow the leader to execute several instances in parallel. This is possible because in MultiPaxos the leader executes Phase 1 for an infinite number of instances, allowing it to start as many instances as it wants within this range of values. In particular, it can start instance $i+1$ even if it is waiting for the decision of instance i, as shown in Figure 8.1b.

Pipelining *improves the utilization of resources*, because the resources that would otherwise be idle can be used to process additional instances. This optimization is especially effective in high-latency networks, as the leader might have to wait a long time to receive the Phase 2b messages.

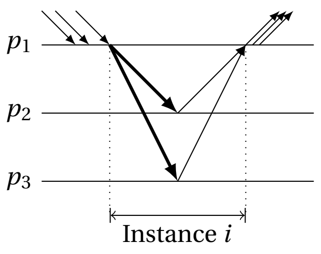
However, there are some potential issues with pipelining that limit the number of instances that is practical to execute in parallel. The first is that although instances may be decided



(a) Basic MultiPaxos



(b) Pipelining



(c) Batching.

Figure 8.1: Paxos: basic message pattern (a) and optimizations (b and c).

out-of-order, the requests must be executed in order, which requires the replicas to explicitly re-order the instances. But if a particular instance takes an exceptionally long time to be decided (maybe due to message loss), it will block the system until it is finally decided. Any higher-numbered instance decided will have to be kept on a re-order buffer, In this case, ordering too many additional instances will not bring any gain, and may even harm the system as it increases the memory requirements and might delay recovery of the late instance.

A second issue with pipelining is that each instance requires additional resources from the system. If too many instances are started in parallel, they may overload the system, either by maxing out the leader's CPU or by causing network congestion, resulting in a more or less severe performance degradation.

For these reasons, the number of parallel instances that the leader is allowed to start is usually bounded. Choosing a good bound requires careful analysis; if set too low, the network will be underutilized, and if set too high the system might become overloaded resulting in a severe performance degradation, as shown by the experiments in Section 8.5. The best value depends on many factors, including the network latency, the size of the requests, the speed of the replicas, and the expected workload.

Batching Batching is a common optimization in communication systems, which generally provides large gains in performance [FR95]. It can also be applied to Paxos, as illustrated by Figure 8.1c. Instead of proposing one request per instance, the leader packs several requests in a single instance. Once the order of a batch is established, the order of the individual requests is decided by a deterministic rule applied to the request identifiers.

The benefits of batching result from spreading the fixed costs of an instance over several requests, thereby decreasing the average per-request overhead. For each instance, the system performs several tasks that take a constant time regardless of the size of the proposal, or whose time increases only marginally as the size of the proposal increases. These include interrupt handling and context switching as a result of reading and writing data to the network card, allocating buffers, updating the replicated log and the internal data structures, and executing the protocol logic. In [CGH$^+$04], the authors show that the fixed costs of sending a packet over a Ethernet network are dominant for small packet sizes, and that for larger packets the total processing time grows significantly slower than the packet size. In the case of Paxos, the fixed costs of an instance are an even larger fraction of the total cost because, in addition to processing individual messages, processes also have to execute the ordering algorithm. Additionally, when stable storage is used, batching decreases dramatically its overhead, because a single stable storage access is enough to log the state of all requests in a batch.

Batching is fairly simple to implement in Paxos: the leader waits until having "enough" client requests and proposes them as a single value. The difficulty is deciding what is "enough". In general, the larger the batches, the bigger the gains in throughput. But in practice, there are several reasons to limit the size of a batch. First, the system may have physical limits on the maximum packet size; for instance, UDP packets can be at most 64KB, and in most cases the underlying network infrastructure imposes an even smaller limit. Second, larger batches take longer to build because the leader has to wait for more requests, possibly delaying the ones that are already waiting and increasing the response time experienced by the client. This is especially problematic with low load, as it may take a long time to form a large batch. Finally, a large batch takes longer to transfer and process, further increasing the latency. Therefore, a batching policy must strike a balance between creating large batches (to improve throughput) and deciding when to stop waiting for additional requests and send the batch (to keep latency within acceptable bounds).

8.4 Analytical model of Paxos performance

8.4.1 Assumptions

We consider the Paxos variant described in Section 8.3 with point-to-point communication. There are other variants of Paxos that use different communication schemes, like IP multicast and chained transmission in a ring [MPSP10]. We chose the basic variant for generality and simplicity, but our analysis can be adapted to other variants. We further assume full duplex links and that no other application is competing for bandwidth or CPU time[1]. Also for

[1] The presence of other applications can be modeled by adjusting the model parameters to reflect the competition for the resources.

Symbol	Description
n	Number of replicas
B	Bandwidth
L	One way delay (latency)
S_{req}	Size of request
k	Number of requests in a batch
w	Number of parallel instances
S_{2a}	Size of a Phase 2a message (batch).
S_{2b}	Size of phase 2b message
S_{ans}	Size of answer sent to client
ϕ_{exec}	CPU-time used to execute a request
WND	Bound on maximum number of parallel instances (Configuration parameter)
BSZ	Bound on batch size (Configuration parameter)

Table 8.1: Notation.

simplicity, we focus on the best case, that is, we do not consider message loss or failures. We also ignore mechanisms internal to a full implementation of Paxos, like failure detection. On a finely tuned system, these mechanisms should have a minimal impact on throughput. Finally, we assume that execution within each process is sequential. The model can be extended to account for multi-core or SMP machines, but this is a non-trivial extension that is out of the scope of this work.

8.4.2 Quantitative analysis of Phase 2 of Paxos

Table 8.1 shows the parameters and the notation used in the rest of the chapter. We focus on the two resources that are typically the bottleneck in Paxos, *i.e.*, the leader's CPU and outgoing network channel.

Our model takes as input the system parameters (n, B, L, and four constants defined below that model the speed of the nodes), the workload parameters (S_{req}, S_{ans} and ϕ_{exec}), and the batching level (k). From these parameters, the model characterizes how an instance utilizes the two critical resources, by determining the duration of an instance (wall-clock time) and the time during which each of the resources is effectively used (the resource busy time). With these values, we can determine the percentage of idle time of each resource, and predict how many additional parallel instances are needed to reach its maximum utilization. The resource reaching saturation with the lowest number of parallel instances is the bottleneck of the system, and therefore determines the maximum number of parallel instances that can be executed.

The model also provides estimations of the throughput and latency for a given configuration, which we use to study how different batch sizes affect the performance and the optimal number of parallel instances for each batch size.

For simplicity, we assume that all requests are of similar size, although our techniques could be extended to consider the case of different requests sizes. Note that with this assumption, we have $S_{2a} = kS_{req} + c$, where c is the size of the protocol headers. For readability, we use simply S_{2a} in the formulas below.

Network busy time

The outgoing channel of the leader is busy for the time necessary to send all the data related to an instance, which consists of $n-1$ Phase 2a messages, one to every other replica, and k answers to the clients. Because of differences in topology, we consider the cases of a LAN and a WAN separately.

In a LAN scenario, the replicas are typically in the same network, usually connected by full-duplex channels, so the effective bandwidth available between them is the bandwidth of the network. Therefore, the leader has a total bandwidth of B available for sending the Phase 2a messages, which leads to the following formula for the per-instance network busy time:

$$\phi_{inst}^{\text{LAN}} = ((n-1)S_{2a} + kS_{ans})/B$$

In a WAN scenario, however, the replicas are in different data centers, so the connection between them is composed of a fast segment inside each replica data center (bandwidth B_L), and of another comparatively slow segment between the different data centers (bandwidth B_W). Since usually $B_W \ll B_L$, in the following analysis we consider B_W to be the effective bandwidth between the replicas, ignoring B_L, *i.e.*, we take $B = B_W$. Moreover, while in LAN a replica has a total bandwidth of B to share among all other replicas, on a typical WAN topology each replica has a total of B_W bandwidth to every other replica. The reason is that the inter-data center section of the connection between the replicas will likely be different for each pair of replicas, so that after leaving the data center, the messages from a replica will follow independent paths to each other replica. Thus, contrary to the case of a LAN, every message sent by the leader uses a separate logical channel of bandwidth B. By the same reasoning, the messages from the leader to the clients also use separate channels. Since sending the answers to the client does not delay executing additional instances, the network bottleneck are the channels between the leader and the other replicas. Therefore, we get

$$\phi_{inst}^{\text{WAN}} = S_{2a}/B$$

From the above formulas, we can compute the maximum network throughput both in terms of instances and requests. The *throughput in instances* is given by $1/\phi_{inst}^{\text{NET}}$, where NET stands for either LAN or WAN. For the throughput in requests, we first derive the network time used for each request, which is $\phi_{req}^{\text{NET}} = \phi_{inst}^{\text{NET}}/k$. With this new formula, the *throughput in requests* is $1/\phi_{req}^{\text{NET}}$.

CPU time

During each instance, the leader uses the CPU to perform the following tasks: read the requests from the clients, prepare a batch containing k requests, serialize and send $n-1$ Phase 2a messages, receive $n-1$ Phase 2b messages, execute the requests, and send the answers to the clients (in addition to executing the protocol logic whenever it receives a message).

These tasks can be divided in two categories: interaction with clients and interaction with other replicas. The CPU time required to interact with clients depends mainly on the size of the requests (S_{req}) and the number of requests that must be read to fill a batch (k), while the interaction with replicas depends on the number of replicas (n) and the size of the batch (S_{2a}). Since these two interactions have distinct parameters and behaviors, we model them by two functions: $\phi_{cli}(x)$ and $\phi_{rep}(y)$. The function $\phi_{cli}(x)$ represents the CPU time used by the leader to receive a request from a client and send back the corresponding answer, with x being the sum of the sizes of the request and the answer. Similarly, $\phi_{req}(y)$ is the CPU time used by the leader to interact with another replica, where y is the sum of the sizes of the Phase 2a and 2b messages. Both functions are linear, which models the well-known [CGH$^+$04] behavior where the time to process a message consists of a constant plus a variable part, the later increasing linearly with the size of message[2]. The values of the parameters of these two functions must be determined experimentally for each system, as they depend both on the hardware used to run the replicas and on the implementation of Paxos. We show how to do so in Section 8.5.2.

Based on the previous discussion, we get the following expression for the CPU time of an instance:

$$\phi_{inst}^{\text{CPU}} = k\phi_{cli}(S_{req}+S_{ans}) + (n-1)\phi_{rep}(S_{2a}+S_{2b}) + k\phi_{exec} \tag{8.1}$$

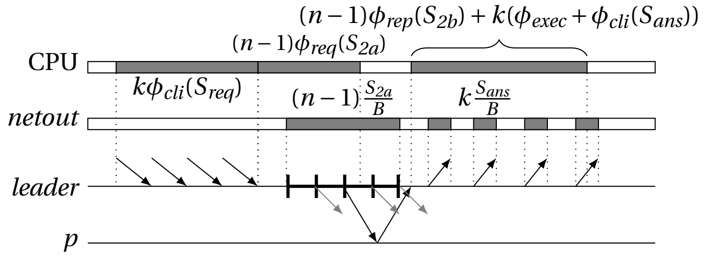
The first term models the cost of receiving k requests from the clients and sending back the corresponding answers, the second term represents the cost of processing $n-1$ Phase 2a and 2b messages and, finally, the last term is the cost of executing the k requests.

From the previous formula, we can compute the time per request as $\phi_{req}^{\text{CPU}} = \phi_{inst}^{\text{CPU}}/k$, and the throughput in instances and requests as $1/\phi_{inst}^{\text{CPU}}$ and $1/\phi_{req}^{\text{CPU}}$, respectively.

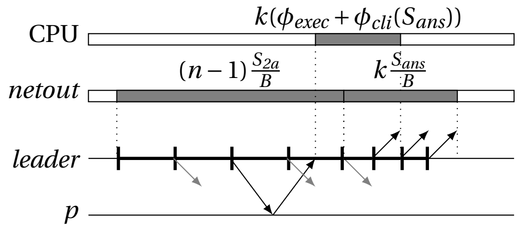
Wall-clock time

Estimating the wall-clock duration of an instance is more challenging than estimating the network and CPU utilization, because some operations that must complete for the instance to terminate are done in parallel. As an example, once the leader finishes sending $\lfloor n/2 \rfloor$ messages to the other replicas, the execution splits into two separate sequence of events. In one of them, the leader sends the remaining phase 2a messages. On the other, it waits for enough phase

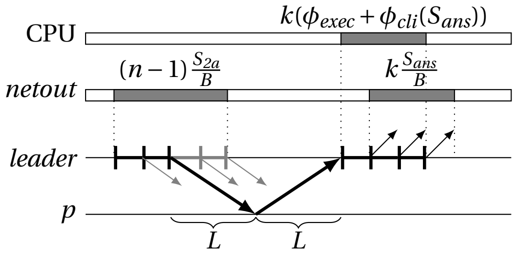
[2] We chose to use a single function to represent sending and receiving a pair of related messages (Phase 2a and 2b), instead of one function for each individual message type. Since the model is linear, this reduces the number of parameters that have to be estimated to half without losing any expressiveness.


$(n-1)\phi_{rep}(S_{2b}) + k(\phi_{exec} + \phi_{cli}(S_{ans}))$
$(n-1)\phi_{req}(S_{2a})$
CPU
$k\phi_{cli}(S_{req})$
$(n-1)\frac{S_{2a}}{B}$
$k\frac{S_{ans}}{B}$
netout
leader
p

(a) CPU is the bottleneck



(b) Bandwidth is the bottleneck



(c) Latency is the bottleneck

Figure 8.2: Utilization of the CPU and outgoing link of the leader during an instance.

2b messages to decide and start executing the requests. Additionally, if after executing the first request in the batch, the leader did not finish sending all the Phase 2a messages, it may have to wait for the outgoing link to be free before sending the answers to the clients. Thus, the exact sequence of events on the critical path that leads to completion depends on the workload and the characteristics of the system. In a fast LAN the wall-clock duration is likely to be limited by the CPU speed, while in a high-latency WAN the latency is likely the dominant factor. Similarly, if the workload consists of large requests and answers, the bandwidth is more likely to be the bottleneck than the CPU or the latency.

Therefore we model the wall-clock time by considering three different cases, each corresponding to a different bottleneck: CPU, bandwidth and latency. For each case, we compute the duration of an instance, resulting in three formulas: T_{inst}^{CPU}, T_{inst}^{band} and T_{inst}^{lat}[3]. The instance time is the maximum of the three:

$$T_{inst} = \max(T_{inst}^{\text{CPU}}, T_{inst}^{band}, T_{inst}^{lat}) \tag{8.2}$$

Once again, due to the differences in topology, we model the LAN and the WAN cases differently.

[3] Note that although both ϕ and T represent real-time, the former represents the time a given resource is used during an action while the second represents the wall-clock time duration of a given action.

For the LAN case, we have:

$$T_{inst}^{\text{CPU}} = k\phi_{cli}(S_{req} + S_{ans}) + (n-1)\phi_{rep}(S_{2a} + S_{2b}) + k\phi_{exec} + \lfloor n/2 \rfloor S_{2a}/2B \tag{8.3}$$

$$= \phi_{inst}^{\text{CPU}} + \lfloor n/2 \rfloor S_{2a}/2B \tag{8.4}$$

$$T_{inst}^{band} = ((n-1)S_{2a} + kS_{ans})/B \tag{8.5}$$

$$T_{inst}^{lat} = \lfloor n/2 \rfloor S_{2a}/B + 2L + k\phi_{exec} + kS_{ans}/B \tag{8.6}$$

Figure 8.2 illustrates these three cases in a LAN. Each sub-figure represents one instance. The two lines at the bottom represent the leader and the replica whose Phase 2b message triggers the decision at the leader. The two bars at the top represent the busy/idle periods of the CPU and of the outgoing link of the leader. The arrows above the leader line represent messages exchanged with the clients (their time-lines are not shown) and the arrows below are messages exchanged with the other replicas.

If the CPU is the bottleneck (Equation (8.3) and Figure 8.2a), the wall-clock time of an instance is dominated by its CPU time (ϕ_{inst}^{CPU}, Formula (8.1)). However, we must also account for the time during which the leader is sending the Phase 2a messages to other replicas, because its CPU will be partially idle during this time, while waiting for the Phase 2b messages from the replicas. This difference between CPU and wall-clock time increases with the size of the batch (confirmed experimentally in Section 8.5, see Figure 8.10). This idle time is represented by $\lfloor n/2 \rfloor S_{2a}/2B$.

If the bandwidth is the bottleneck (Equation (8.5) and Figure 8.2b), the wall-clock time of an instance is the total time needed by the leader to send all the messages of that instance through the outgoing channel, *i.e.*, $n-1$ Phase 2a messages and k answers.

Finally, if the latency is the bottleneck (Equation (8.6) and Figure 8.2c), the wall-clock time of an instance is the time needed to send the first $\lfloor n/2 \rfloor$ Phase 2a messages to the replicas, plus the round-trip time required to receive enough Phase 2b messages from the replicas, followed by the execution time of the requests and the time to send the answers back to the clients.

For the WAN case, the formulas are as follow:

$$T_{inst}^{\text{CPU}} = \phi_{inst}^{\text{CPU}} + S_{2a}/B \tag{8.7}$$

$$T_{inst}^{band} = S_{2a}/B \tag{8.8}$$

$$T_{inst}^{lat} = S_{2a}/B + 2L + k\phi_{exec} \tag{8.9}$$

The difference is that messages can be sent in parallel, because of the assumption that each pair of processes has exclusive bandwidth. Therefore, the time to send a message to the other replicas does not depend on n, and sending the answers to the clients does not affect the duration of an instance (separate client-leader and leader-replica channels).

8.4.3 Maximizing resource utilization with parallel instances

If the leader's CPU and outgoing channel are not completely busy during an instance, then the leader can execute additional instances in parallel. The idle time of a resource R (CPU or outgoing link) is given by $T_{inst} - \phi^R_{inst}$ and the number of instances that a resource can sustain, w^R, is T_{inst}/ϕ^R_{inst}. From these, we can compute the maximum number of parallel instances that the system can sustain as:

$$w = \lceil \min(w^{\mathrm{CPU}}, w^{\mathrm{NET}}) \rceil \tag{8.10}$$

This value can be used as a guideline to configure batching and pipelining. In theory, setting the window size to any value equal to or higher than this lower bound results in optimal throughput, but as shown by the experiments in Section 8.5.2, increasing the window size too much may result in congestion of the network or saturation of the CPU, and reduce performance. Therefore, setting the window size to w should provide the best results.

8.5 Experimental Study

In this section we study the batching and pipelining optimizations from an experimental perspective, and validate the analytical model. We have performed experiments both in a cluster environment (Section 8.5.1) and in a WAN environment emulated using Emulab [Wea02] (Section 8.5.2).

For each scenario, we start by presenting the experimental results, then we determine the parameters of the model that characterize the process speed (parameters of $\phi_{cli}(x)$ and $\phi_{rep}(x)$), and finally compare the predictions for the throughput and optimal window size of the model with the values obtained experimentally. We performed the experiments using JPaxos.

Implementing batching and pipelining in Paxos is fairly straightforward: batching has a trivial implementation and pipelining was described in the original Paxos paper [Lam98]. To control these optimizations, *i.e.*, decide when to create a new batch and initiate a new instance, we use a simple algorithm with three parameters, *WND*, *BSZ* and Δ_B. The parameter *WND* is the maximum number of instances that can be executed in parallel, *BSZ* is the maximum batch size (in bytes), and Δ_B is the batch timeout. The timeout Δ_B is reset whenever the leader opens a new batch, which happens when it receives the first request that is assigned to the batch. The leader then waits until either it has enough requests to fill the batch or the timeout Δ_B expires. It then proposes the batch by starting a new instance as soon as the number of active instances is under *WND*. In the experiments we vary *BSZ* and *WND* while keeping Δ_B set to 50ms. This timeout has no impact on the results, because as explained below, all experiments were performed with the system under high load, so that in the common case the leader is able to fill a batch before the timeout expires.

We consider a system with three replicas. In order to stress the batching and pipelining mechanisms, all the experiments were performed with the system under high load. More precisely, we used a total of 1200 clients spread over three nodes, each running in a separate thread and sending requests in a closed loop (*i.e.*, waiting for the previous reply before sending the next request). During the experiments, the nodes running the clients were far from being saturated, which implies that the bottleneck of the system was on the replicas.

The replicated service keeps no state. It receives requests containing an array of S_{req} bytes and returns an 8 byte array. We chose a simple service as this puts the most stress on the replication mechanisms. JPaxos adds a header of 16 bytes per request and 4 bytes per batch of requests. The analytical results reported below take the protocol overhead in consideration.

All communication is done over TCP. We did not use IP multicast because it is not generally available in WAN-like topologies. Initially we considered UDP, but rejected it because in our tests it did not provide any performance advantage over TCP. TCP has the advantage of providing flow control and congestion control, and of having no limits on message size. The replicas open the connections at startup and keep them open until the end of the run. Each data point in the plots corresponds to a 3 minutes run, excluding the first 10%. For clarity, we omit the error bars with the confidence intervals, as they are very small.

We report our results using the following metrics. The *throughput of instances* is the number of Phase 2 instances executed per second, and the *throughput of requests* is the number of requests ordered per second. These two metrics correspond to the throughput formulas of the analytical model given in Section 8.4.2 (end of paragraphs *Network busy time* and *CPU time*). The *latency per instance* is the time elapsed at the leader from proposal to decision of an instance, *i.e.*, from sending the Phase 2a message to receiving a majority of Phase 2b messages. It corresponds to T_{inst} (Formula 8.2) in the analytical model. The *client latency* is the time the client waits for the reply to a request, which includes the transmission time from the client to the leader, the queuing time of the request at the leader, the time to order the request, and the time to send the answer back to the client.

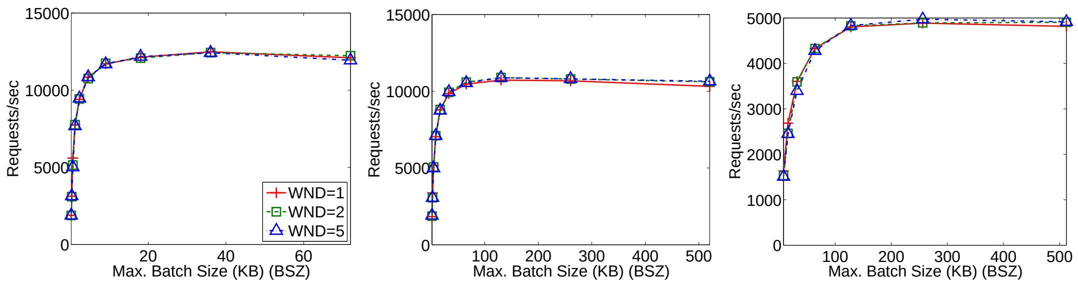
8.5.1 Cluster

The following experiments were run on a cluster of Pentium 4 at 3GHz with 1GB memory connected by a Gigabit Ethernet. The effective bandwidth of a TCP stream between two nodes as measured by `netperf` Linux tool is 940 Mbit/s.

Experimental results

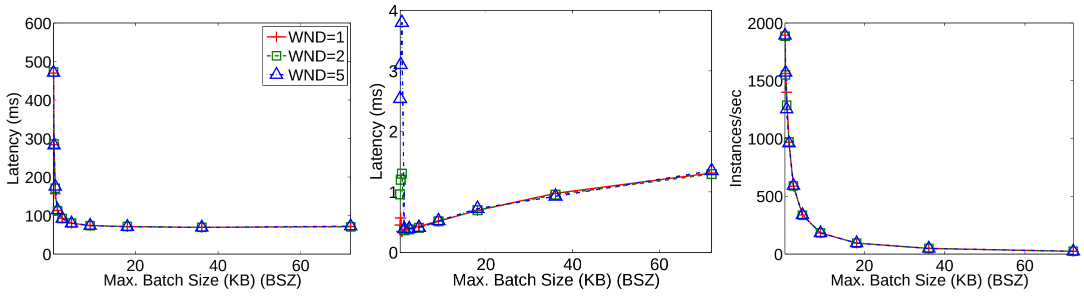
Figure 8.3 shows the request throughput as a function of batch size, for request sizes of 128 bytes, 1KB and 8KB, and for maximum window sizes of 1, 2 and 5.

Batching provides a major improvement in performance in all cases, ranging from an almost ten times improvement with 128 bytes requests to a little over four times with 8KB requests. The



(a) $S_{req} = 128$ (b) $S_{req} = 1\text{KB}$ (c) $S_{req} = 8\text{KB}$

Figure 8.3: Cluster: request throughput as a function of batch size.



(a) Client latency (b) Instance latency (c) Instances/sec

Figure 8.4: Cluster: additional experimental results with requests size of 128 bytes.

batch size where the system reaches optimal throughput varies depending on the request size: around 10KB, 64KB and 128KB for request sizes of 128 bytes, 1KB and 8KB, respectively. On the other hand, increasing *WND* does not improve performance. In fact, except for the smallest batch sizes, the average number of consensus instances open at any time was always one (not shown), which shows that the leader is not able to start additional consensus instances. The cause is the relatively slow CPU by comparison to the network. During each run the average CPU utilization of the leader's CPU is above 90%, suggesting that the leader is CPU-bound and, therefore, is not able to execute additional instances in the time it waits for the answers for the previous instances.

Note that the performance does not drop if *BSZ* or *WND* are increased past their optimal values. This is a desirable behavior, because the system will perform optimally with a wide range of configuration parameters, making it easier to tune. As Section 8.5.2 shows, this is not always the case.

Figure 8.4 shows the effects of batching in several other metrics with request size of 128 bytes. The results show that even small levels of batching have a dramatic effect on performance. The client latency (Figure 8.4a) goes from almost 500ms without batching to under 100ms just by increasing *BSZ* to 2KB. Further increases in batch size provide only small additional improvements in response time, as the bottleneck shifts from the CPU required to process

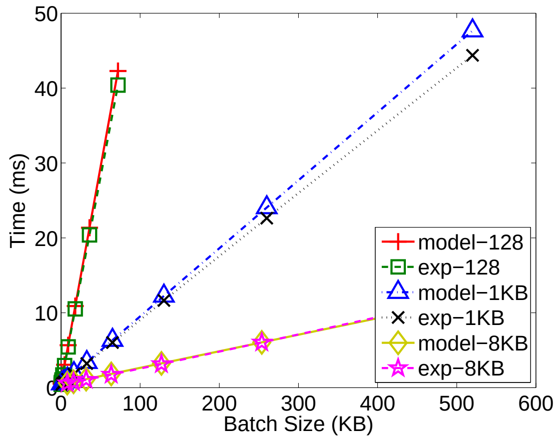


Figure 8.5: Cluster: experimental versus model results for the CPU time of an instance. Fit values: $\phi_{cli}(x) = 0.005x + 0.08, \phi_{rep}(x) = 0.0035x + 0.22$.

each instance of the ordering protocol to the cost of handling each individual client request.

The instance latency (Figure 8.4b) increases linearly with the size of the batch, which shows clearly that its time is composed of a fixed constant time ($\approx$ 0.35ms in this case) plus a variable time that depends on the amount of data that has to be transmitted. The only exception is for the smallest batch sizes with window sizes of 2 and 5, where the instance time is up to eight times higher than what would be expected from the rule above. As the batch sizes are too small to handle the incoming client load, the leader attempts to compensate by executing parallel instances. But with a slow CPU (recall that the nodes are single-core, single-CPU Pentium 4 at 3GHz), the additional instances cause instability resulting in a spike in the consensus execution time.

Finally, Figure 8.4c shows that, as expected, by using larger batches the leader orders a higher number of requests while executing fewer instances. Note that without batching, the leader executes 2'000 instances per second, corresponding to 2'000 requests (Figure 8.3a). As the batch size increases, the number of instances drops to under 200, but the throughput in requests increases to 12'000, because the number of requests per batch increases faster than the reduction in the number of instances.

Setting model parameters

To estimate the parameters of ϕ_{cli} and ϕ_{rep} we used the Java Management interfaces (more precisely, the `ThreadMXBean` class) to measure the total CPU time used by the leader process during a run. Dividing this value by the total number of instances executed during the run gives the average per-instance CPU time. To prevent the JVM warm-up period from skewing the results, we ignore the first 30 seconds of a run (for a total duration of 3 minutes). We repeat the measurements for several request and batch sizes, and then adjust the parameters of the model manually until the model's estimation for the CPU time (ϕ_{inst}^{CPU}) fits the training

S_{2a}	ϕ^{CPU}_{req}	ϕ^{CPU}_{inst}	ϕ^{NET}_{inst}	T_{inst}	w^{CPU}	w^{NET}
128	0.52	0.52	0.00	0.52	1.00	211.73
256	0.30	0.60	0.00	0.60	1.00	124.18
512	0.19	0.77	0.01	0.77	1.00	79.52
1KB	0.14	1.09	0.02	1.10	1.00	56.97
2KB	0.11	1.75	0.04	1.76	1.01	45.63
4KB	0.10	3.06	0.08	3.07	1.01	39.95
8KB	0.09	5.67	0.15	5.71	1.01	37.11
16KB	0.09	10.90	0.31	10.98	1.01	35.68
32KB	0.08	21.36	0.62	21.51	1.01	34.97
64KB	0.08	42.28	1.23	42.58	1.01	34.62

Table 8.2: Analytical results for the cluster scenario, request size of 128 bytes ($S_{req} = 128$). Results for different batch sizes (S_{2a}). Times in milliseconds.

data. Figure 8.5 shows the training data together with the results of the model, for the final fit of $\phi_{cli}(x) = 0.005x + 0.08$ and $\phi_{rep}(x) = 0.0035x + 0.22$. The figure shows that the CPU time measured experimentally increases roughly linearly with the size of the batch, which validates our choice of a linear model.

Comparison of analytical and experimental results

All the analysis below is done with $\phi_{exec} = 0$, since the request execution time of the service used in the experiments is negligible (recall that the service simply answers with a 8 byte array). Table 8.2 shows detailed results for the case $S_{req} = 128$, while Table 8.3 shows a summary of the analytical results for all request sizes and compares them with the experimental results.

With $S_{req} = 128$ (Table 8.2), the CPU time used by an instance (column ϕ^{CPU}_{inst}) is an order of magnitude larger than the network busy time (column ϕ^{NET}_{inst}). As a result, the wall-clock time of an instance (column T_{inst}) is dominated by the CPU time. Although the network could sustain a large number of parallel instances (column w^{NET}), the CPU cannot sustain more than one (column w^{CPU}) and therefore the system as a whole has no capacity to execute additional instances. The situation is similar for larger requests sizes (columns w^{NET} and w^{CPU} in Tables 8.3b and 8.3c), although the CPU becomes less of a bottleneck as the size of the requests increases. A similar pattern occurs as the batch size increases, with the load shifting from the CPU to the network. But even with the largest messages tested, *i.e.*, $S_{req} = 8$KB and $S_{2a} = 512$KB, the CPU is still the bottleneck, being able to sustain only 1.17 parallel instances as compared to 1.63 of the network. Such a situation is typical of systems where the network is comparatively faster than the process[4], which is typical in a cluster environment.

The model also captures the effect of batching on performance. As the size of the batches increases, the total instance time increases reflecting the larger size, but the average time per request (ϕ^{CPU}_{req}) decreases. This is very noticeable for 128 bytes requests (Table 8.2), with the

[4]The speed of the process depends not only on the CPU speed but also on the efficiency of the Paxos implementation.

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
128	**1**	211.73	1916	1	≈ 1895
256	**1**	124.18	3313	1	≈ 3126
1KB	**1**	56.97	7313	1	≈ 7745
32KB	**1**	34.97	11983	1	≈ 12488
64KB	**1**	34.62	12108	1	≈ 12100

(a) $S_{req} = 128$

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
1KB	**1.01**	31.54	1878	1	≈ 1850
2KB	**1.01**	18.64	3202	1	≈ 3380
8KB	**1.03**	8.93	6791	1	≈ 7050
256KB	**1.04**	5.79	10644	1	≈ 10680
512KB	**1.05**	5.74	10742	1	≈ 10400

(b) $S_{req} = 1\text{KB}$

	Model			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
8KB	**1.05**	4.92	1625	1	≈ 1634
16KB	**1.08**	3.25	2530	1	≈ 2687
64KB	**1.14**	2	4344	1	≈ 4328
256KB	**1.17**	1.68	5293	1-2	≈ 4900
512KB	**1.17**	1.63	5493	1-2	≈ 4900

(c) $S_{req} = 8\text{KB}$

Table 8.3: Cluster: analytical versus experimental results for different batch sizes. Prediction of optimal w [=min($w^{\text{CPU}}, w^{\text{NET}}$)] is in bold. The column w shows the range that was determined experimentally to contain the smallest value of *WND* that produces the maximum throughput.

average time per request dropping from 0.52ms down to 0.08ms for the largest batches.

The results also show that the predicted optimal window size matches closely the value obtained in the experiments. Furthermore, the predicted throughput is close to the experimental results, with less than 5% error for 128 bytes and 1KB requests, and at most 20% for 8KB requests 512KB batch size.

8.5.2 Emulab

Figure 8.6 shows the topology used for the Emulab experiments, which represents a typical WAN environment with the geographically distributed nodes. The replicas are connected point-to-point by a 10Mbits link with 50ms latency. Since the goal is to keep the system under high load, the clients are connected directly to each replica and communicate at the speed of the physical network. The physical cluster used to run the experiments consists of nodes of Pentium III at 850MHz with 512MB of memory, connected by a 100Mbps Ethernet.

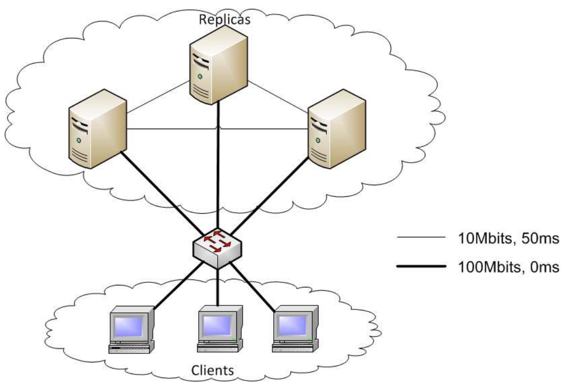
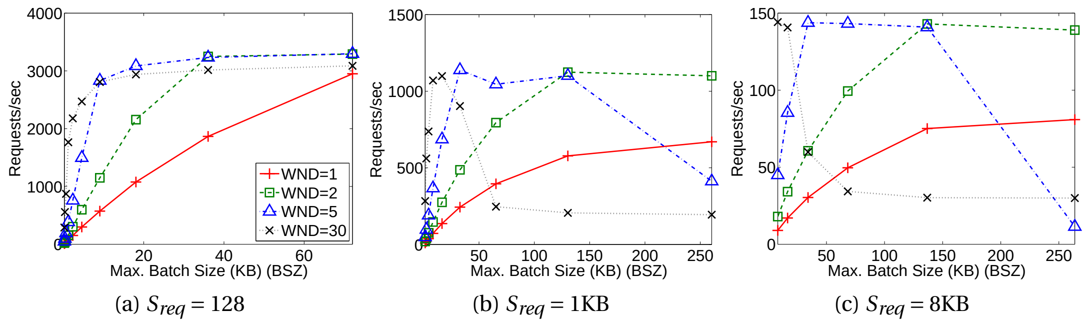


Figure 8.6: Topology used for Emulab experiments



(a) $S_{req} = 128$ (b) $S_{req} = 1\text{KB}$ (c) $S_{req} = 8\text{KB}$

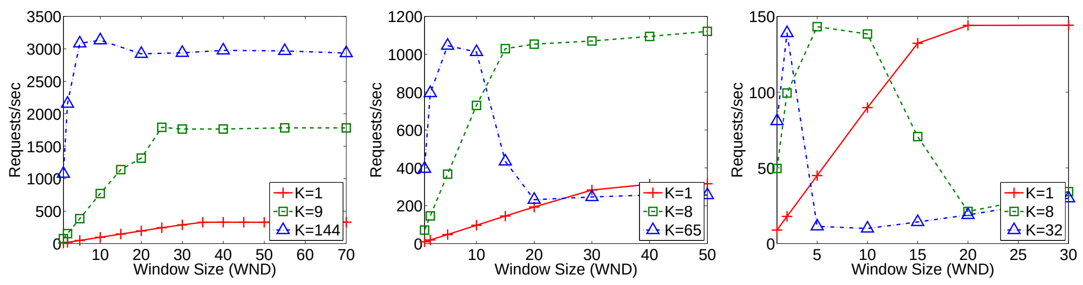
Figure 8.7: Emulab: throughput with increasing batch size.

Experimental results

Figure 8.7 shows the throughput in requests per second for increasing values of *BSZ*, while Figure 8.8 shows similar data but plotted against the values of *WND*. Both figures include results for requests sizes of 128, 1KB and 8KB. Figure 8.9 gives more detail on the tests with request size of 128 bytes, showing the client latency, instance latency and instance throughput.

The results show that, contrary to the cluster experiments, batching alone does not suffice to achieve maximum throughput. Although using only batching (Figure 8.7, $WND = 1$) improves performance significantly, it falls short of the maximum that is achieved with larger window sizes. The difference is greater with large request sizes (1KB and 8KB), where batching alone achieves only half of the maximum throughput, than for small sizes (128 bytes), where it reaches almost the maximum. The reason is that with small request sizes the leader is CPU-bound, so it cannot execute more than one parallel instance, while with larger requests the bottleneck is the network latency. Increasing the window size to 2 is enough for the system to reach maximum throughput in all scenarios if the batch size is large enough (40KB with $S_{req} = 128$ and around 140KB with $S_{req} = 1\text{KB}$ and $S_{req} = 8\text{KB}$). If the window size is further increased, the maximum throughput is achieved with smaller batch sizes.

Using pipelining alone (Figure 8.8, $K = 1$) is not enough to achieve the best performance with



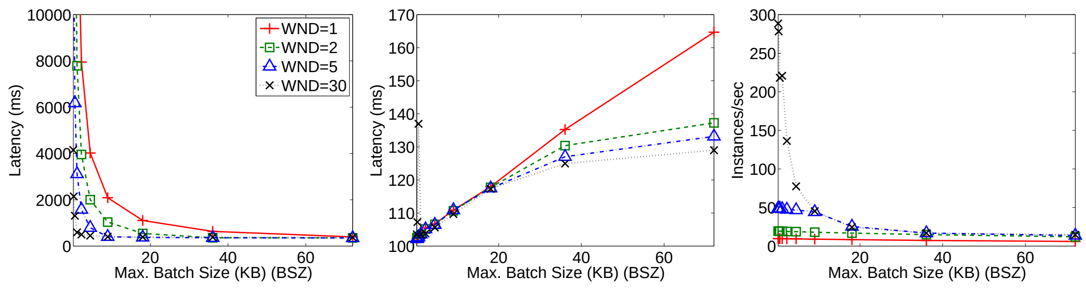
(a) $S_{req} = 128$ (b) $S_{req} = 1\text{KB}$ (c) $S_{req} = 8\text{KB}$

Figure 8.8: Emulab: throughput with increasing window size. K represents the number of requests that fit on a batch, *i.e.*, $K = \lfloor BSZ / reqSize \rfloor$.

small and medium request sizes (128 bytes and 1KB, Figures 8.8a and 8.8b), but is enough for large request sizes (8KB, Figure 8.8c). With small request sizes the performance gains are quite modest, as it does not even reach 500 requests per second, far from the maximum of 3000. With large request sizes, however, with $WND \geq 15$ the system reaches the maximum performance. The difference is once again due to different bottlenecks; with small request sizes the bottleneck is mainly the CPU, so batching provides the most gains by decreasing the average per request CPU utilization. On the other hand, with large requests the bottleneck is the network, either the latency or the bandwidth, and in this case batching does not help. When the limitation is the network latency, pipelining improves throughput significantly, as seen in Figure 8.8c until $WND = 15$. After this point, the system is limited by the network bandwidth, so neither batching nor pipelining can improve the results further.

The experiments also show that increasing the window size too much results in a performance collapse, with the system throughput dropping to around 10% of the maximum. This happens when the leader tries to send more data than the capacity of the network, resulting in packet loss and retransmissions. The point where it happens depends on the combination of S_{req}, WND and BSZ, which indirectly control how much data is sent by the leader; larger values increase the chance of performance collapse. With $S_{req} = 128$ there is no performance degradation, because the CPU is the bottleneck limiting the throughput. With larger request sizes, the network becomes the bottleneck and there are several cases of performance collapse. With $S_{req} = 1\text{KB}$, the performance collapse is noticeable already with $WND = 5$: there is a sharp drop from $BSZ = 128\text{KB}$ to $BSZ = 256\text{KB}$ (Figure 8.7b). And for even larger WND, the performance collapse happens with smaller values of BSZ, *i.e.*, with $WND = 30$ it collapses from $BSZ = 32\text{KB}$ to $BSZ = 64\text{KB}$. Similarly, as the batch size increases performance collapse occurs at smaller and smaller window sizes.

These results show that CPU and network may react to saturation very differently. In this particular system, the CPU deals gracefully with saturation, showing almost no degradation, while the network saturation results in a performance collapse. The behavior may differ significantly in other implementations, because the behavior of the CPU or network when



(a) Client latency (b) Instance latency (c) Instances/sec

Figure 8.9: Emulab: experimental results with request size of 128 bytes.

under load (graceful degradation or performance collapse) depends on the implementation of the different layers of the system, mainly application and replication framework (threading model, flow-control) but also operating system and network stack.

To conclude the analysis of the experimental results, we look in more detail at the experiments with request size of 128 bytes (Figure 8.9). As the batch size increases the client latency goes from a maximum of over 10 seconds down to less than 0.5 seconds (Figure 8.9a). This is a indirect consequence of the batch size. Recall that larger batch sizes result in higher throughput (Figure 8.7a). Since the experiments were performed with a fixed request load, then higher throughput reduces the time that client requests spend waiting in queues, which dramatically decreases the client latency.

The instance latency (Figure 8.9b) increases with *BSZ*, with the minimum being the round-trip time (100ms). For values of *WND* other than 1, the increase is not linear. The reason is that the instance latency depends on the effective batch size, which may be smaller than the maximum batch size (*BSZ*). This is the case with values of *WND* greater than one, because as the leader is allowed to execute parallel instances, it does not always fill the batches completely before the batch timeout expires.

The throughput in instances (Figure 8.9c) is the highest when *WND* is large and *BSZ* small, as the leader is able to fill up batches and start new instances before the previous ones finish. However, since each batch is small, the throughput in requests is low. As *BSZ* increases, the leader will wait more to start a batch, therefore resulting in a smaller throughput in instances. When *WND* is small the throughput in instances is small because the leader is restricted in how many instances it can execute in parallel, so even if it has batches ready, it must wait for the previous instances to finish.

Setting model parameters

Following the same procedure as in the case of the cluster, we have determined the following parameters for the Emulab model: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(y) = 0.002y + 1.5$. Figure 8.10

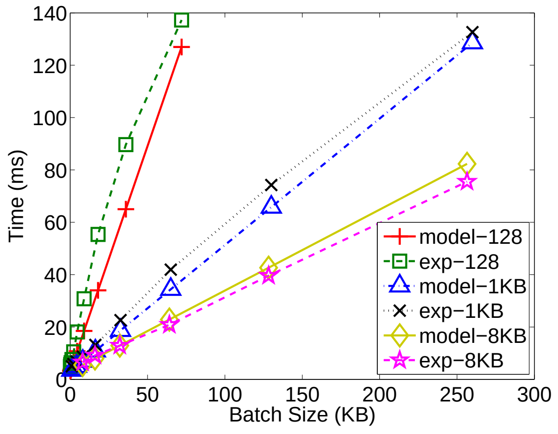


Figure 8.10: Emulab: experimental versus model results for the CPU time of an instance. Fit values: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(y) = 0.002y + 1.5$.

shows the training data and the corresponding model results when parametrized with the values above.

Comparison of analytical and experimental results

Table 8.4 shows the results of the model for the optimal window size of the CPU and network for several batch sizes, and compares them with the experimental results.

The analytical results show that the bottleneck with 128 bytes requests is the CPU (w^{CPU} is smaller than w^{NET}) while for 8KB requests it is the network. With 1KB requests, the behavior is mixed, with the CPU being the bottleneck with small batch sizes and the network with larger batch sizes. These results quantify the common sense knowledge that smaller requests and batches put a greater load on the CPU in comparison to the network. Moreover, as the request size or batch size increase, the optimal window size decreases, because if each instance contains more data, the network will be idle for less time.

The experimental results in Table 8.4 are obtained by determining for each batch size the maximum throughput and the smallest w where this maximum is first achieved.

In all cases the prediction for w is inside the range where the experiments first achieve maximum throughput, showing that the model provides a good approximation. Concerning the throughput, the model is accurate with $S_{req} = 8$KB across all batch sizes. With $S_{req} = 128$, it is accurate for the smallest batches but overestimates the throughput for the larger batches. The reason is that the network can be modeled more accurately than the CPU, as it tends to behave in a more deterministic way[5]. The CPU exhibits a more non-linear behavior, especially when under high load. This in turn requires carefully tuning of the maximum number of parallel

[5]This is true only until reaching a level of saturation where packets are dropped, after which modeling the network becomes difficult.

	Model (predictions)			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
128	**30.88**	833.48	308	30-35	≈ 330
256	**28.77**	422.94	574	25-30	≈ 550
1KB	**20.45**	107.58	1620	20-25	≈ 1800
16KB	**3.38**	7.68	3765	2-5	≈ 3100
32KB	**1.47**	3.12	4032	1-2	≈ 3300

(a) $S_{req} = 128$

	Model (predictions)			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
1KB	**28.89**	119.01	286	30-40	≈ 310
2KB	**25.54**	60.12	502	30-40	≈ 600
8KB	**15.42**	15.8	1155	15-20	≈ 1030
128KB	3.16	**1.93**	1184	1-2	≈ 1120
256KB	2.68	**1.6**	1184	1-2	≈ 1100

(b) $S_{req} = 1KB$

	Model (predictions)			Experiments	
S_{2a}	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
8KB	19.47	**16**	150	15-20	≈ 144
16KB	14.24	**8.5**	150	5-10	≈ 144
64KB	6.72	**2.88**	150	2-5	≈ 144
128KB	4.84	**1.94**	150	1-2	≈ 144
256KB	3.8	**1.47**	150	1-2	≈ 144

(c) $S_{req} = 8KB$

Table 8.4: Emulab: analytical versus and experimental results for different batch sizes. Prediction of optimal w [=min(w^{CPU}, w^{NET})] is in bold. The column w shows the range that was determined experimentally to contain the smallest value of *WND* that produces the maximum throughput.

instances, as higher values increase CPU utilization.

8.6 Conclusion

In this chapter we have studied two important optimizations to Paxos, batching and pipelining.

The analytical model presented in the chapter is effective at predicting the combinations of batch size and number of parallel instances that result in optimal throughput in a given system. The model can be used in the following way to tune batching and pipelining: (i) choose the largest batch size that for a given workload satisfies the response time requirements, then (ii) use the model to determine the corresponding number of parallel instances that maximize throughput. The rationale for this heuristic is the following. As batching provides larger gains than pipelining, the batch size should be the first parameter to be maximized. However, there is a limit on how much it can be increased, because large batches take longer to fill up with requests leading to an higher response time. Given the expected request rate and the desired response time, we can easily compute the largest batch size that satisfies the response time. The model then provides the corresponding window size that maximizes throughput. As an example, consider the Emulab environment. If the average request size is 1KB and we have determined that the batch size should be 8KB, then the window size should be set to 16 (Table 8.4b).

The experiments show clearly that batching by itself provides the largest gains both in high and low latency networks. Since it is fairly simple to implement, it should be one of the first optimizations considered in Paxos and, more generally, in any implementation of a replicated state machine. Pipelining is useful only in some systems, as its potential for throughput gains depends on the ratio between the speed of the nodes and the network latency: the more time the leader spends idle waiting for messages from other replicas, the greater the potential for gains of executing instances in parallel. Thus, in general, it will provide minimal performance gains over batching alone in low latency networks, but provides substantial gains when latency is high. While batching decreases the CPU overhead of the replication stack, executing parallel instances has the opposite effect because of the overhead associated with switching between many small tasks. This reduces the CPU time available for the service running on top of the replication task and, in the worst case, can lead to a performance collapse if too many instances are started simultaneously (see Emulab experiments).

The results presented in this chapter are also relevant to the discussion of monolithic (*e.g.*, Paxos) versus modular implementations of replicated state machines. A typical modular implementation uses atomic broadcast as a black box to order the client requests. This leads to better modularity, as it abstracts the implementation of atomic broadcast, but it may make it harder to implement pipelining. As an example, a common way of implementing atomic broadcast is by reduction to consensus [CT96], where consensus is used as a black box. Batching can be easily implemented in such a scenario, but the same is not true for pipelining. Pipelining is simple when there is a leader responsible for choosing an order, like in Paxos, but rather hard if we make no additional assumptions on the internal working of the consensus box. Therefore, we can use the results obtained with batching without parallel instances as an example of what to expect from atomic broadcast by reduction to consensus. This suggests that this alternative implementation of replication will perform well in networks with low latency, where batching is enough to achieve optimal throughput, but will under-perform when the latency is increased.

This chapter focused on throughput rather than latency because as long as latency is kept within an acceptable range, optimizing throughput provides greater gains in overall performance. A system tuned for high-throughput will have higher capacity, therefore being able to serve a higher number of clients with an acceptable latency, whereas a system tuned for latency will usually reach congestion with fewer clients, at which point its performance risks collapsing to values well below the optimal.

9 Multi-core scalable State Machine Replication

The traditional architecture used by implementations of Replicated State Machines (RSM) does not fully exploit modern multi-core CPUs. This is increasingly the limiting factor in their performance, because network speeds are increasing much faster than the single-thread performance of CPUs. Thus, when deployed on Gigabit-class networks and exposed to a workload of small to medium size client requests, RSMs are often CPU-bound, as they are only able to leverage a few cores, even though many more may be available. In this chapter, we revisit the traditional architecture of a RSM implementation, showing how it can be parallelized so that its performance scales with the number of cores in the nodes. We do so by applying several good practices of concurrent programming to the specific case of state machine replication, including staged execution, workload partitioning, actors, and non-blocking data structures. Furthermore, we implement and evaluate the architecture proposed using JPaxos. With a workload consisting of small requests, we achieve a 6 times improvement in throughput using 8 cores. More generally, in all our experiments we have consistently reached the limits of the network subsystem by using up to 12 cores, and do not observe any degradation when using up to 24 cores. Furthermore, the profiling results of our implementation show that even at peak throughput contention between threads is minimal, suggesting that the throughput would continue scaling given a faster network.

9.1 Introduction

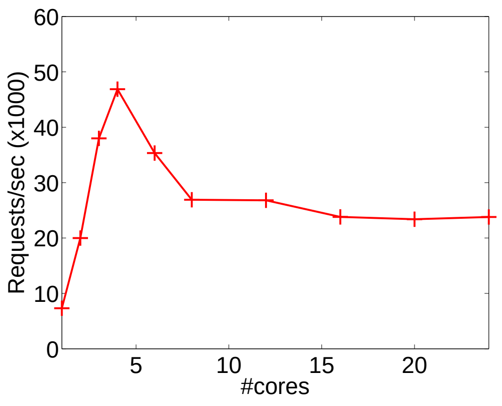
State machine replication is frequently used by online services as a low-cost solution to achieve reliability and availability. However, deploying state machine replication in such a context creates new challenges in the design and implementation of such services. In particular, online services are exposed to a very large number of potential users, and therefore must be engineered for high-throughput.

This raises the question of whether state machine replication can support the required levels of throughput. If the service being replicated is itself expensive, then this is a moot point, as the system will be limited by the performance of the service. But often the services are lightweight, like key-value stores [RST11b], lock servers [Bur06] and coordination services [HKJR10]. These types of services can sustain a very high-throughput, provided that the underlying state machine replication layer can keep up. Additionally, when multiple services within the same data center must be fault-tolerant, it is often easier to delegate ordering to a single, high-performance ordering service instead of having every service implement its own ordering layer [KJ10]. This scenario requires, once again, a state machine replication service capable of reaching very high-throughputs.

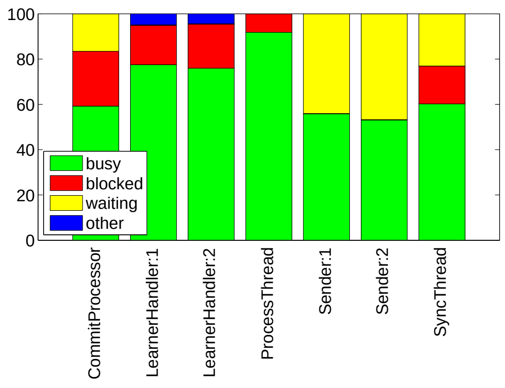
Recently there has been a renewed interest in improving the throughput of the ordering phase. For instance, [MPSP10] and [Lev08] show how to achieve high-throughput with algorithmic improvements to the replication protocol. These protocols use the network efficiently, relying on techniques like ring topologies and IP multicast to achieve an efficiency of over 90% on Gigabit Ethernet, as measured by the amount of data ordered over the bandwidth of the network. However, this high efficiency is achieved only if requests are large enough, usually 8KB or greater. For smaller request sizes, the implementations of these protocols becomes CPU-bound and their efficiency drops significantly [MPSP10]. A similar CPU bottleneck with small request sizes has been identified in many other systems, including in production-quality implementations. For instance, in [HKJR10] the authors report that the atomic broadcast protocol at the core of ZooKeeper becomes CPU-bound at the leader process when running with a workload 1KB write requests. As small request sizes are common in practice (*e.g.*, coordination services, lock servers), the CPU bottleneck is a significant obstacle in achieving high throughput.

It is worth looking more carefully at the causes of this CPU bottleneck. In recent years the single-thread performance of CPUs improved only marginally, while the number of cores increased greatly. A modern server-class CPU contains anywhere between 4 and 16 cores, with even higher numbers on the horizon. Although each core may have a relatively modest single-thread performance, their aggregate performance is considerable. In the case of state machine replication, current implementations are mostly unable to take full advantage of multi-core CPUs, thereby being limited by the single-thread performance.

As an example, consider Figure 9.1a which shows the throughput of ZooKeeper with increasing number of cores, and Figure 9.1b which shows the per-thread profiling results with 24 cores. For each thread, we show the time spent executing (*busy*), blocked trying to acquire a lock (*blocked*), waiting to receive work (*waiting*), and in other states (*other*, see Section 9.6.3 for a detailed explanation). Although ZooKeeper scales well up to four cores reaching a peak of 50K requests per second, its performance degrades to less than 30K as the number of cores increases to eight, after which it remains stable. We can make two main observations from the results. First, ZooKeeper can only use a limited number of cores (around eight), after which adding more cores has no effect. In fact, Figure 9.1b shows that there are only seven threads with a non-trivial CPU utilization (threads with insignificant CPU utilization are not shown). This is a limitation of its design, which does not allow any change to the number of threads. Additionally, this makes ZooKeeper susceptible to single-thread bottlenecks because



(a) Throughput



(b) Per-thread CPU utilization at the leader, when using 24 cores.

Figure 9.1: Performance of ZooKeeper with increasing number of cores. Ensemble of 3 replicas, 128 bytes write-request (`setData()` on a ephemeral node). ZooKeeper 3.3.3 with the default configuration except that the leader does not serve clients and logs and snaphsot directories were mapped to a RAM drive (`/dev/shm`).

once a thread reaches its limit, it will hold back the full system (Figure 9.1b). Second, using more cores may in fact reduce performance, even when there are threads capable of making use of those cores, as seen by the sharp drop in performance from 4 to 8 cores. This is likely caused by a combination of increased contention on locks and cache trashing, because at any point in time there are more active threads contending for access to shared data. The profiling results show that contention is in fact a problem: With 24 cores available, threads spend a large fraction of time blocked (the profiling results are similar for all experiments from 8 to 24 cores).

We believe this is typical of many replicated state machine (RSM) implementations. The traditional threading architecture used by RSMs is based on an event-driven model, with a event loop (thread) doing most of the work (with maybe the exception of IO operations). There are good reasons why RSMs implementations use this design. First, it matches closely the way replication protocols are typically expressed, which is as a set of event handlers. It also simplifies thread coordination, trivially preventing race conditions by not sharing data among threads. This is especially important because RSMs have a complex internal state which is shared by many different internal tasks (*e.g.*, ordering, retransmission, failure detection, snapshotting, and state transfer). Finally, before the multi-core era, a single-thread event-driven design was a good choice, as it avoided the cost of context switches and concurrency control. But this traditional architecture is reaching its limits and must be revisited in order to reach the full potential of modern Gigabit networks, multi-core CPUs and of the new generation of highly efficient replication protocols.

More recent implementations depart from the traditional architecture and use multiple threads. However if threads are introduced without making an effort to carefully partition the internal state of the RSM, they will end up sharing critical parts of the state. This requires the introduction of locks to ensure thread-safety, which in turn leads to high levels of contention. As a result, the improvement in parallelism is limited, as illustrated above with ZooKeeper.

Contribution In this chapter we address the problem of parallelizing RSMs such that their performance scales with the number of cores. For this, we assume that the workload and the system are such that the bottleneck is the single-thread performance of the CPU, which is often the case with small request sizes and fast networks (Gigabit or more). In order to scale the performance with the number of cores, we must ensure that 1) the tasks performed by a replica are evenly balanced among threads, so that no replica becomes limited by the single-thread performance of its CPU, and 2) threads make progress mostly independently of others, with minimal time wasted in coordination (*e.g.*, contention). For the second point, it is essential to minimize shared state among threads.

Some of the tasks performed by a replica, *e.g.*, I/O, are fairly easy to parallelize. But other tasks, like the execution of the ordering protocol and the various housekeeping operations, pose a much greater challenge, either because they are inherently sequential or because their state is complex and shared between several tasks. As mentioned previously, a naive separation into threads is likely to make extensive use of locks, increasing contention, limiting concurrency, and being prone to race conditions and deadlocks. We propose an architecture that avoids these problems by grouping tasks into threading modules according to carefully chosen boundaries that minimize shared state and contention.

We use an hybrid design, with a mixture of event-driven and thread-based modules. This design draws inspiration from SEDA [WCB01] and from the concept of Actors in languages like Erlang and Scala [HO08]. Each module consists of private state, one or more threads, and a well-defined interface for communicating with threads from other modules (usually through message queues). With few exceptions, the private state is accessible only by the threads managed by the module. This organization keeps complex state isolated inside a module, with well-defined access points, simplifying reasoning about thread-safety and parallelism.

We have implemented this architecture in JPaxos and performed an experimental study using a CPU-intensive workload on a cluster of 24-core machines connected by a Gigabit Ethernet.

The results show that the throughput increases with the number of cores, either linearly or very close to linearly, until reaching the limit of the network subsystem, which happens when using between 8 to 12 cores. Furthermore, the profiling results of our implementation show that, even at peak throughput, contention between threads is minimal, suggesting that performance would continue scaling in the absence of non-threading related bottlenecks. This shows that even with a demanding workload, the throughput of state machine replication can be further improved by leveraging multi-core CPUs.

To summarize, our main contributions are:

- An analysis showing that further improvements in throughput of state machine replication require implementations capable of exploiting the potential of multi-core CPUs.
- A multi-core scalable design for RSM.
- An experimental study showing the scalability of this architecture in multi-core CPUs.

Roadmap The remainder of this chapter is organized as follows. Section 9.2 discusses the related work, Section 9.3 provides the background by describing how state machine replication works and how it is typically implemented, Section 9.4 discusses the challenges in parallelizing RSM implementations and describes our general approach, Section 9.5 describes in detail the scalable threading architecture we propose, Section 9.6 presents the experimental results, and Section 9.7 concludes the chapter.

9.2 Related Work

The interest in state machine replication has been increasing in the last years, both in academia [MJM08, Lev08, MPSP10] and in the industry. In the latter case, some of the noteworthy examples are the Chubby lock-server [Bur06] and the ZooKeeper coordination service [HKJR10].

Recently, several works have focused on the performance of state machine replication. Mencius [MJM08] proposes a rotating leader protocol designed for Wide-Area Networks, which improves throughput by distributing over multiple replicas the load that is usually concentrated on the leader. In [KJ10], the authors propose a protocol based on a similar idea but adapted to the LAN scenario. LCR [Lev08] and Ring-Paxos [MPSP10] focus instead on achieving high network efficiency on fast Gigabit LANs. Zab [HKJR10], the atomic broadcast protocol at the core of ZooKeeper, is a modified version of Paxos with focus on high-performance. However, all these works are concerned only with improving the replication protocol, and do not address the potential CPU bottleneck. And in fact, some of the results presented in [HKJR10] and [MPSP10] show that the implementations of these protocols reach the single-thread performance bottleneck of the CPU with workloads consisting of small requests. Our work complements the algorithmic improvements proposed in these papers, by showing how to leverage multi-core CPUs to address the CPU bottleneck. To the best of our knowledge, there is no previously published work addressing this particular problem.

Outside the field of state machine replication, there is a rich body of work on tools and techniques to exploit the parallelism available in multi-CPU/multi-core systems. Our work combines ideas from SEDA [WCB01] and from the notion of Actors from languages like Erlang and Scala [HO08]. SEDA is an architecture for highly-scalable servers consisting of interconnected event-driven stages, each implemented by one or more threads, and using message queues to communicate asynchronously. Actors can be regarded as an extension of the concept of data encapsulation to threading: An Actor encapsulates both data and threading,

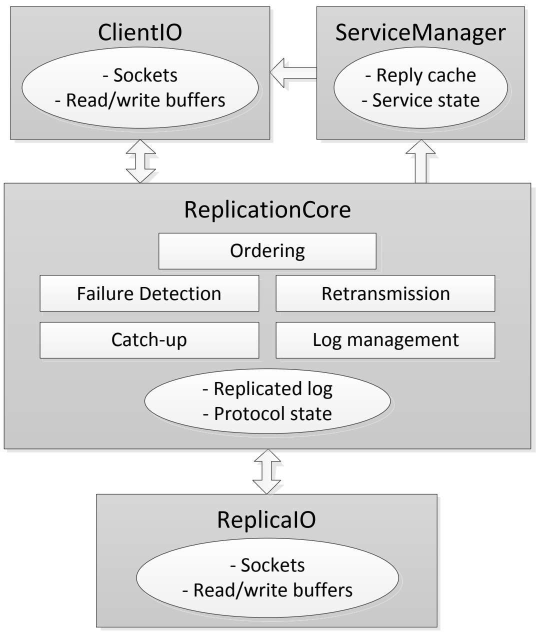


Figure 9.2: Main modules of a Replicated State Machine

forbidding explicit sharing of state and communicating with other Actors only by means of message passing. Our architecture is partly organized as a set of stages, like in SEDA, although we deviated from a pure staged design by also using thread-based modules. Like with Actors, we encapsulate threading within a module using message passing, but deviated from this rule when it would significantly harm performance.

9.3 Generic design of an RSM

Although implementations of replicated state machines differ in many aspects, they are usually organized around the same set of modules, whose functionality and state are roughly equivalent across implementations. In Chapter 7 we have presented JPaxos, which follows a fairly typical structure for RSMs. However, to keep our discussion as general as possible, we will use instead an abstract high-level design, that includes only the major components that are part of any RSM implementation without specifying details like how failure detection or state transfer are done. Our goal is that the threading architecture we propose and the underlying guiding principles can be easily adapted to other designs.

An implementation of a RSM consists of four modules: ClientIO, ReplicaIO, ReplicationCore, and ServiceManager (Figure 9.2).

The *ClientIO* module manages the communication with the clients, which is usually done using TCP connections. Its main tasks are accepting new connections, receiving requests and sending replies, and its state consists of the connection information (sockets) and I/O buffers with partly read/written packets. To achieve high-throughput even with small requests, this module must be designed to handle thousands of connections and up to hundreds of thousands of small messages per second.

The *ReplicaIO* module is similar to the ClientIO module, managing the communication with the other replicas. However, it must be designed for a very different workload, *i.e.*, for a small number of connections (one for every other replica), each transmitting a high amount of data. And contrary to the ClientIO module, the size of the messages exchanged with other replicas is partly under the control of the batching policy of the RSM, which can choose a size that provides good bulk throughput.

The *ReplicationCore* module executes the ordering protocol and all the auxiliary services, like failure detection, log management, message retransmission and catch-up/state transfer. Its state consists of the replicated log containing the information on every known instance of the ordering protocol, and a few other control variables.

Finally, the *ServiceManager* module receives the ordered sequence of requests, executes them on the service, and sends the reply to the clients. Apart from the state of the service, this module may manage a reply cache (used to ensure at-most-once execution of requests) and some additional information to manage snapshots.

9.4 Challenges and Design principles

Our goal is to design a threading architecture whose performance scales with the number of cores. This goal requires first *finding and exploiting parallelism* among the tasks performed by a RSM. The difficulty of this depends greatly on the nature of the particular module of the RSM, which ranges from embarrassingly parallel to inherently serial. Furthermore, scaling the performance requires good *load balancing among threads*, to avoid single-thread performance bottlenecks and ensure that all threads are able to make progress concurrently. Once again, this is easily done inside the modules with homogeneous workload, like I/O, but difficult to do in modules with heterogeneous workloads. Finally, to *ensure correctness* threads must coordinate when accessing shared state. This is a hard problem, susceptible to several classes of errors that can lead to safety violations (race conditions), to liveness violations (deadlock and livelock), or to poor performance (contention).

Our design draws inspiration from the architecture proposed by SEDA, and from the concept of Actors in languages like Scala and Erlang. It consists of a set of modules, with some of them forming a pipeline used to process and order requests, and the others providing auxiliary services.

Like Actors, each module encapsulates both state and threading, and the primary means of communication between threads in different modules are message queues. However, a strict enforcement of this rule would at some places either harm performance and scalability, or result in an unnecessarily complex design. Therefore, we have allowed some carefully designed exceptions where threads access state directly in other modules.

Modules have one or more threads and may be either event-driven or thread-based. Note that single-threaded modules with only private state are naturally thread-safe. Multi-threaded modules, however, must use either locks or state partitioning to protect the internal state. But this is an easier problem than in a monolithic design because the module provides boundaries to the shared state and threading.

This organization has several advantages with respect to RSMs implementations. In addition to the well-known advantages of state and thread encapsulation, it also allows each module to have its own design. This is key to scale the performance of RSMs with the number of cores while keeping complexity under control: The many tasks performed by the implementation of a RSM differ substantially in their structure, so that a single homogeneous design would not provide the best results. The critical factors that guided our design of the modules are the potential parallelism, the complexity of the state and of the operations performed, the frequency and complexity of interactions with other tasks, and the nature of the tasks (sequential or event-handlers). The choice of the appropriate design involves a series of trade-offs between these factors.

As the CPU-intensive tasks have the greatest potential for parallelism, they should be implemented as multi-threaded modules, ideally with a configurable number of threads. This is easy in some cases, like with the I/O tasks which are embarrassingly parallel in nature. However, the ReplicationCore and ServiceManager modules pose a greater challenge: Although they are potential single-thread bottlenecks, they are hard to divide into independent tasks executing concurrently because of their complex state and inter-dependencies. In these cases, we have used the natural single-threaded, event-driven implementation for the core tasks of the module, while offloading as much work as possible from the main event-dispatch thread to auxiliary threads. For the modules that have little or no potential for parallelism (*e.g.*, retransmission and failure detection), we chose the design with the simplest implementation, both in terms of code complexity and thread safety.

9.5 Threading Architecture

To simplify the presentation, we do not distinguish the cases where a process is acting as leader or as non-leader replica, even though the tasks performed are not the same. Instead, we discuss a single case, where the replica does both the leader and non-leader related tasks. Note that this is often the case in reality, where leadership is just an additional responsibility for the replica.

Figure 9.3 shows the threading architecture. Our design uses several types of message queues. The *RequestQueue* connects the ClientIO threads to the Batcher thread, while the *ProposalQueue* does the same between the Batcher and Protocol threads. The *DispatcherQueue* is

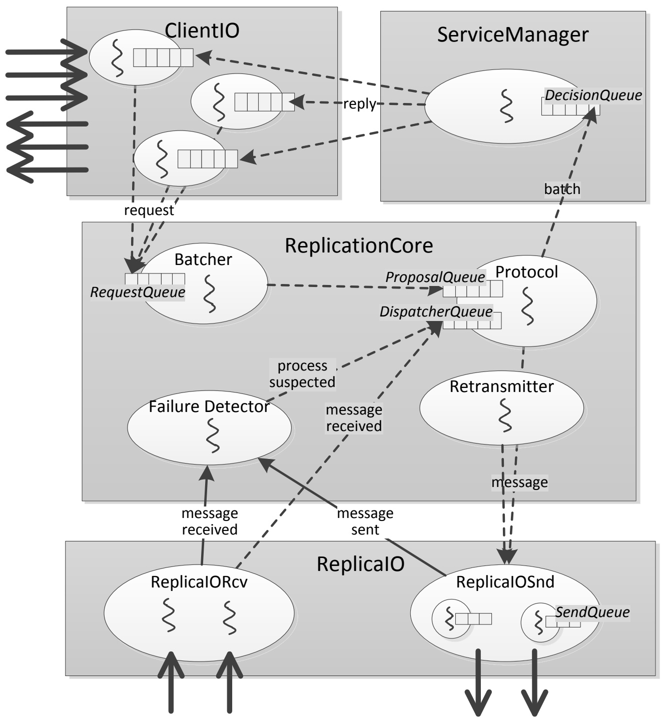


Figure 9.3: Scalable threading architecture. Dashed arrows represent asynchronous calls (putting a message on the message queue of the module), thin solid arrows represent synchronous calls, and thick solid arrows represent network communication.

the queue from where the Protocol thread takes events to process. Each ReplicaIOSnd thread has a queue with the messages waiting to be sent. The *DecisionQueue* is used by the Protocol thread to pass the ordered batches to the ServiceManager. Finally, each ClientIO thread has a queue for the replies to be sent to the respective client.

9.5.1 ClientIO module

Since clients connect to a replica using TCP and remain connected for potentially a long time, the ClientIO module has to handle thousands of simultaneous connections. In this scenario blocking I/O with a thread-per-connection model is inefficient, so the ClientIO module uses instead non-blocking I/O (Java NIO) and an event-driven architecture. For parallelism and load balancing, it keeps a static pool of I/O threads and assigns new connections to a thread in this pool using a round-robin strategy[1]. For each connection, a ClientIO thread is responsible for reading and deserializing requests, checking the reply cache, and then either sending the cached reply back to the client or putting the request in the RequestQueue. After the request is executed, the ServiceManager thread places the answer in the message queue of the ClientIO thread that is handling the connection to the corresponding client. The ClientIO thread will

[1] Our design can easily accommodate more sophisticated load-balancing strategies.

later serialize and send the reply.

Our profiling tests (Section 9.6.3) show that reading and writing requests represent a significant fraction of the CPU utilization in state machine replication. We use a configurable number of ClientIO threads, which allows this module to easily take advantage of the cores available in the system.

The optimal number of ClientIO threads depends on the number of cores and of clients. Based on our experiments, the most important rule to follow in choosing the number of ClientIO threads is to not exceed the number of cores left unused by the other threads in the RSM (or choose only one thread if all cores are used), so that ClientIO threads are not forced to do context switches. Another useful rule, although with a smaller impact, is to set a number proportional to the expected number of clients, aiming to keep each thread busy between 60% and 80% of the time. In our tests the optimal value was typically between 3 and 6.

9.5.2 ReplicaIO module

This module uses blocking I/O and a thread-based design, with two threads per socket, one for reading and another for writing. The reader thread for replica p reads and deserializes the messages received from p, then passes the messages to the Protocol thread using the *DispatcherQueue*. Any thread wanting to send a message to replica p places the message on the queue of the respective sender thread. This thread will later take the message, serialize and send it to p.

Although the reader thread is necessary in a blocking I/O design, the sender thread is not strictly required because other threads can write directly to the socket. However, having a dedicated thread to send messages has several benefits. First, it improves parallelism by offloading to a dedicated thread the work of serialization and of writing to a socket. Second, it prevents the thread running the main event-loop (*e.g.*, the Protocol thread) from blocking on a socket write, which can happen if other replicas are slow or have crashed without closing the TCP connection. Blocking in this situation would at best slow down the main event-loop and, at worst, lead to a distributed deadlock if the Protocol threads of multiple replicas block trying to send messages to each other. By having a dedicated send thread, this situation is detected by other threads without blocking when the *SendQueue* becomes full.

We chose blocking I/O for this module, because the number of connections between replicas is relatively small, usually comparable to the number of replicas, so it did not justify the additional complexity of non-blocking I/O. As our experiments show, a single thread can easily handle the load of reading or writing to one other replica. Additionally, this design scales well with the number of replicas, since the number of ReplicaIO threads is proportional to the number of replicas. Given enough available cores, we can expect that the performance of reading/writing to other replicas will not degrade as the number of replicas increases.

9.5.3 ReplicationCore Module

This module contains four threads: Batcher, Protocol, FailureDetector and Retransmitter. The Protocol thread has the central role, because it executes the replication protocol. As such, it is the critical path for the performance of both the local replica and of the system as a whole. Therefore, we have reduced to a minimum the work done by this thread, delegating as much as possible to other threads. This is challenging, because the tasks done by this module are closely related, sharing and manipulating the same underlying state. Using locks to protect the shared state would lead to a complex design, being prone to contention and race conditions. Instead, this module does not use any lock explicitly: coordination between threads is done either by message passing using queues, or by shared state if concurrent access is not harmful. In spite of this strict rule, we have identified several tasks that are mostly self-contained, having only a few isolated interactions with other threads and sharing only a few variables. We only allow shared variables if accessing them can be done without locks. Several conditions must be met for this to be true, mainly the variable cannot be used in a condition variable (which usually requires locking) and it must not be part of an invariant that includes other variables. If these conditions are true, then we can rely on atomic operations or on the memory model guarantees of the language/runtime to access the variables. An example of the second case are volatile variables in Java, which ensure a global order on the reads and writes to the variable. In the following description, we will use volatile as defined in Java, although other languages have equivalent mechanisms to ensure the same properties.

Batcher thread This thread takes requests from the Request queue, forms batches according to the batching policy, and puts them in the *ProposalQueue*. It accesses directly the state owned by the Protocol thread to read the number of instances that are currently in execution (volatile variable).

The Batcher thread removes from the critical path the task of building a batch, doing it concurrently with the execution of the ordering protocol by the Protocol thread. This design reduces latency of request ordering because when the Protocol thread needs a batch to start a new instance, it can simply take one from the *ProposalQueue*, which is faster than generating a new batch from a list of requests. It also improves parallelism because, as shown by the experiments in Section 9.6.3, Figure 9.11, the total execution time of the Batcher thread can exceed 50% of a CPU, which justifies having a separate thread to offload this work from the Protocol thread.

Protocol thread This thread executes the core replication protocol, implementing an event-loop that takes events from the *DispatcherQueue*. These events include messages from other replicas, suspicions raised by the failure detector, batches ready to be proposed, and other housekeeping events related to log management. This thread has exclusive write access to the bulk of the ReplicationCore module state, including the replicated log and the variables describing the current state of the protocol. In addition to the *DispatcherQueue*, the Protocol thread uses a second queue (*ProposalQueue*) to receive batches from the Batcher thread. This second queue is needed to enforce flow control (explained below) and to allow the Batcher thread to produce a (limited) number of batches in advance.

This design matches closely the logical structure of a replication protocol, usually expressed as a collection of handlers. Equally important, it is a simple design in terms of ensuring thread safety.

FailureDetector thread Depending on the role of the replica, this thread either sends hearbeats to the other replicas (leader role) or waits for heartbeats from the leader. When the leader is suspected, it enqueues a suspect event in the *DispatcherQueue*. It also receives notifications from the Protocol thread whenever the view changes. For every other replica, the FailureDetector thread keeps timestamps of the last message received or sent. These timestamps are updated directly by the ReplicaIO threads when they read or write the corresponding messages. In order to avoid context switches, the ReplicaIO threads do not notify the FailureDetector thread when timestamps are updated. This is safe, because as timestamps never decrease, updating a timestamp always results in delaying the corresponding event (send heartbeat or suspect process), so the failure detector thread can safely wait for the original delay and then decide what to do based on the current values of the timestamps.

Using a dedicated thread for failure detection provides significantly better timing guarantees than using an event-loop, and as such significantly improves the chances that the failure detector will work correctly, even under high-load.

Retransmitter thread This thread ensures that messages essential to the progress of the protocol are eventually delivered. This service is also needed when using TCP, because messages may be lost when a connection fails and later is reestablished. Internally, the Retransmitter thread uses a priority blocking queue containing the messages to be retransmitted sorted by time of retransmission. When the Protocol thread sends a message for the first time, it also enqueues it in the Retransmitter queue. As instances are decided, the Protocol thread cancels the retransmission of the messages. This operation must be very efficient, because under normal conditions it will be done for all messages sent. We do it without acquiring locks and without waking up the Retransmitter thread. The Protocol thread simply sets a (volatile) flag on the control structure associated with the message. Later, when the retransmission timeout of the message expires, the Retransmitter thread wakes up, sees that it was canceled and drops the message.

9.5.4 ServiceManager module

This module contains a single thread, which receives the batches that the Protocol thread puts in the *DecisionQueue* after establishing their order. For each batch, it extracts the requests, passes them to the service in the final order, updates the reply cache with the results of the request execution, and finally hands over the reply to the ClientIO thread responsible for the connection to the respective client.

The reply cache is a potential source of contention: It is queried by each ClientIO thread when a client request is received, and updated by the ServiceManager thread when a request is executed. Under high load, it can be accessed several thousands of times per second from multiple threads. A conventional hash table based on coarse-grained locking performs poorly in this situation, as confirmed by our initial tests. Instead, this table should be implemented using fine-grained locking. In our implementation we have used the class `ConcurrentHashTable` from `java.util.concurrent`, which eliminated any signs of contention on the reply cache.

9.5.5 Queues and flow control

Flow control based on backpressure can easily be implemented in the architecture described above. This is achieved by setting appropriate limits to each queue, so that when a stage is not able to keep up with the incoming workload, the queue fills up, which allows the stages before it to detect the overload and take corrective action.

For instance, under high load the Protocol thread is usually unable to order batches as quickly as the Batcher thread generates them. The *ProposalQueue* therefore fills up, which in turn stops the Batcher thread from taking requests from the *RequestQueue*. This is detected by the ClientIO threads, which in turn temporarily stops reading new requests from the clients. This activates the flow control mechanisms of TCP, resulting in the send buffers at the client side filling up, and the client being blocked from sending more data.

This mechanism proved to be effective in our tests. Even under high load, the resource usage at the replicas remains bounded, without any noticeable performance degradation.

9.6 Performance Evaluation

To evaluate the multi-core scalability of the architecture described above we have implemented it in JPaxos (see Chapter 7).

The experiments were run on two different clusters of the Grid5000[2] testbed, one with 8 core machines and the other with 24 core machines. For the 8 core setup, we used the *edel* cluster at the Grenoble site, consisting of nodes with two quad-core CPUs (Intel Xeon E5520 CPU) running at 2.27GHz. The experiments with 24 core machines were run on the *parapluie* cluster

[2]https://www.grid5000.fr

of the Rennes Grid5000 site, consisting of nodes equipped with two 12-core CPUs (AMD Opteron 6164 HE) running at 1.7Ghz. Both clusters used a 1 Gigabit Ethernet network with an effective inter-node bandwidth of 114MB/s. Nodes were running Linux, kernel version 2.6.26-2, and the JVM used was Oracle's JRE version 1.6.0_25.

We restricted the number of cores used by the JVM by setting the process affinity with the GNU command `taskset`. In choosing the cores from the two CPUs, we tried to co-locate cores in a single CPU as much as possible. This strategy performs in general better than if using cores from different CPUs, since all communication between cores is done inside a single CPU by means of the L3 cache, thereby avoiding costly main memory accesses.

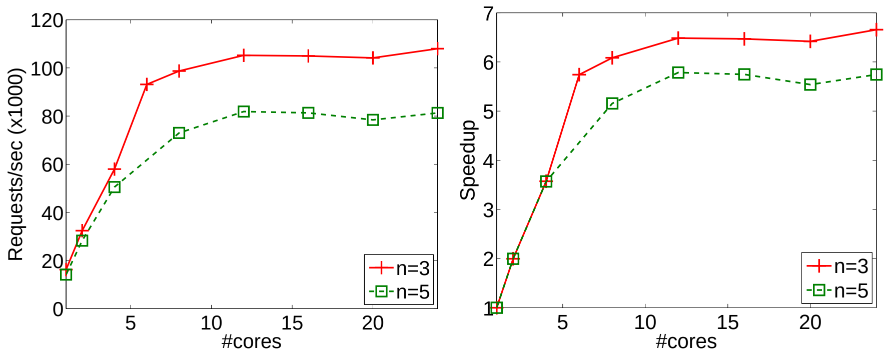
The workload was generated by nodes located in the same cluster as the replicas, each running several client threads in the same Java process. The clients send the requests directly to the leader, using persistent TCP connections. After establishing the connection, they send requests in a loop, waiting for the answer to the previous request before sending the next one. Each experiment was run for 3 minutes, with the first 10% ignored in the calculation of the results. The request size was 128 bytes and the answer size was 8 bytes. To focus our evaluation on the ordering protocol, we used a null service, which discards the payload of the request and sends back a byte array of the size specified by the test. We have not used stable storage, as it would introduce an additional bottleneck making it harder to test the multi-core scalability.

We used a total of 1800 clients distributed over six machines. For the pipelining optimization we set the maximum number of parallel instances to 10, and for batching we used a maximum batch size to 1300 bytes. With these settings the system is CPU-bound, thereby allowing us to better observe the gains from parallel execution.

We use as metrics the throughput, the speedup, the CPU utilization and the total thread blocking time. The *throughput* is measured in requests per second. The *speedup* is defined as the ratio between the throughput with k cores and with one core [3]. The *CPU utilization* of a replica is measured using the GNU `time` command and is shown as a percentage of one core, *i.e.*, 100% is equivalent to one core being fully utilized. These values are the average over the full run, including the warm up period. The plots labeled *Total blocked time* show the sum across all threads of the time spent blocked trying to acquire a lock. The values displayed are normalized to the run time, *i.e.*, 100% corresponds to 3 minutes. This metric gives an indication of the level of contention inside the JVM, and therefore, the efficiency of the threading architecture. These values are obtained using the Java Management interface (`ThreadMXBean`) by a dedicated background thread. This thread takes samples every second, starting 10 seconds after JVM startup and continuing until shutdown. The values reported below are the cumulative times between the first and last samples.

Recall from Section 9.5 that the number of ClientIO threads is configurable. This parameter

[3]Although our definition differs from the traditional definition of speedup, which measures the reduction in time needed to complete a certain task, it is equivalent, because if the throughput increases by s, then the time needed to process a fixed number of requests is divided by s as well.



(a) Throughput (b) Speedup

Figure 9.4: JPaxos performance with increasing number of cores, parapluie cluster.

has a significant impact on performance in multi-core machines: too low and the cores are underutilized, too high and performance drops due to contention. Therefore, for each data point (number of CPUs) in the plots in Section 9.6.1 we have repeated the experiments with various number of ClientIO and we show the best results.

9.6.1 Multi-core Scalability

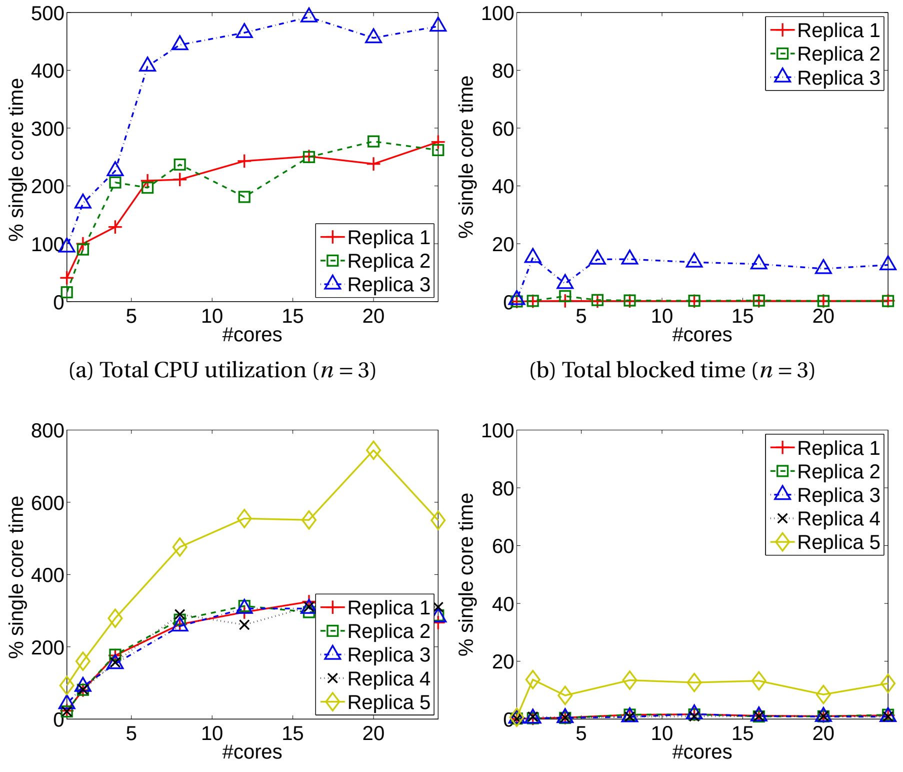
We performed the experiments using configurations with three and five replicas. In each set of experiments, we varied the number of cores from one to the maximum in the nodes. Figures 9.4 and 9.5 show the results for the parapluie cluster.

For $n = 3$, the speedup (Figure 9.4b) is linear up to six cores, then sublinear up to twelve where it reaches the maximum speedup of over 6.5, with a throughput of around 100K requests/sec (Figure 9.4a). The throughput then remains stable up to the maximum of 24 cores.

A common pattern in all our results is that the replica in the role of leader (replica 3 and 5 in the tests with three and five replicas, respectively) has significantly higher CPU utilization and contention than the other replicas (Figure 9.5), which is to be expected from leader-based protocols. Therefore, in the following we will focus our analysis on the leader.

Interestingly, the leader's CPU utilization (Figure 9.5a) increases slower than the throughput (Figure 9.4a): from one to six cores it goes from 100% to 400%, while the throughput increases six times. A possible explanation is that with more cores available, threads run for longer between context switches, resulting in less overhead and better caching behavior than with fewer cores.

Note that the CPU utilization (Figure 9.5a) and total blocked time (Figure 9.5b) remain stable up to the maximum number of cores. In particular, the total blocked time remains under 20%, showing that increasing the number of cores up to 24 does not cause additional contention. Recall from the results in Figure 9.1 that this is not always the case, even in production quality



(a) Total CPU utilization ($n = 3$)

(b) Total blocked time ($n = 3$)

(c) Total CPU utilization ($n = 5$)

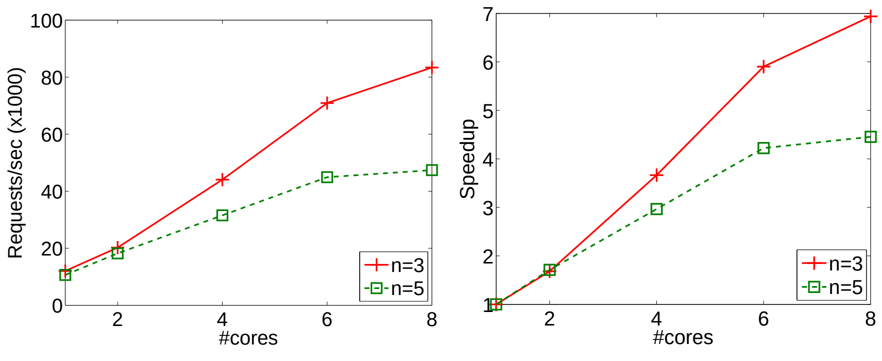
(d) Total blocked time ($n = 5$)

Figure 9.5: JPaxos CPU usage and contention, parapluie cluster.

implementations like ZooKeeper.

For $n = 5$ the results are similar, with the exception of a smaller speedup, which reaches a maximum of 5.5 instead of 6.5. This is likely a consequence of the higher number of messages (approximately the double) that the leader has to handle. Receiving and sending are done by two dedicated threads per replica, so these tasks scale linearly with the number of cores. However, the main event-loop has to handle all those messages and since it is single threaded (Protocol thread) it is limited by the single-thread performance of the CPU, which accounts for the lower speedup. This effect is likely more pronounced with higher number of replicas.

The bottleneck limiting the performance to 100K requests/second is in the network subsystem of the leader, as we explain next. At this point, the leader is exchanging 100K network packets per second with the clients, in addition to the packets exchanged with other replicas, which amounts to 150K packets per second in each direction. In Section 9.6.5 we analyze this bottleneck, performing experiments with different parameters including smaller requests, larger batch sizes, and larger maximum number of parallel instances, obtaining slightly better results in throughput in some cases. But in all cases the throughput reached a peak when the leader was handling around 150K network packets per second. As we discuss in Section 9.6.5 and in Chapter 10, this bottleneck is in the networking subsystem of the Linux kernel. The



(a) Throughput (b) Speedup

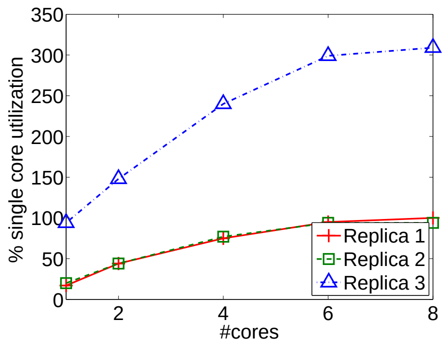
Figure 9.6: JPaxos performance with increasing number of cores, edel cluster.

version used in the experiments (2.6.26) is known to have several scalability bottlenecks when running in multi-core machines, some of which affect the networking subsystem [BWCM+10].

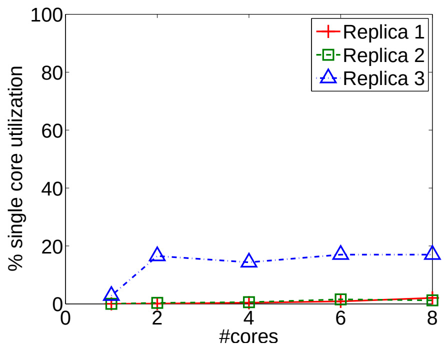
Note that although by using a larger batch size it is possible to pack more requests into a network packet, thereby making more efficient use of the network, this works only among the replicas. The communication pattern between clients and replicas depends on the particular workload of the application, which is generally not under the control of the RSM implementation.

The experiments performed in the edel cluster (Figures 9.6 and 9.7) show similar trends, except that with only eight cores in each node the system does not reach the limit of the network subsystem. With $n = 3$, the speedup increases almost linearly, reaching a maximum of 7 with eight cores (Figure 9.6b), for a throughput of just above 80K requests per second (Figure 9.6a). The shape of the curve suggests that with more cores available, the performance will increase further. Another indication that JPaxos would scale further, comes from the results in the parapluie cluster, which reached a maximum throughput of 100K requests per second. As both clusters use similar network infrastructure and operating system, we can expect that the bottleneck from the network subsystem will be reached at similar levels of throughput, suggesting that the network subsystem in the edel cluster still has room for higher performance than the one exhibited in the tests.

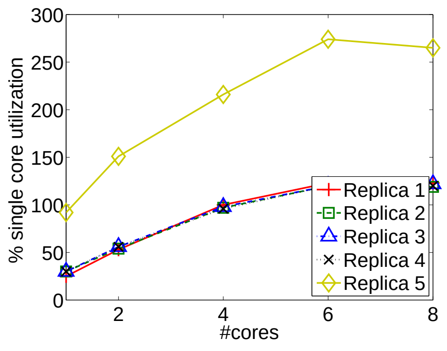
Concerning CPU utilization and blocking time (Figure 9.7), the results are once again similar to the ones in the parapluie cluster, with the CPU utilization increasing slower than the speedup; more precisely, for a 7x speedup there is a 3x increase in CPU utilization. The total blocking time is once again relatively low, with a total aggregate time across threads of under 20% of a single core time. This shows that having more cores allows a well-designed multi-threaded application to run more efficiently, by avoiding the overhead associated with sharing a small number of cores among a larger number of threads.



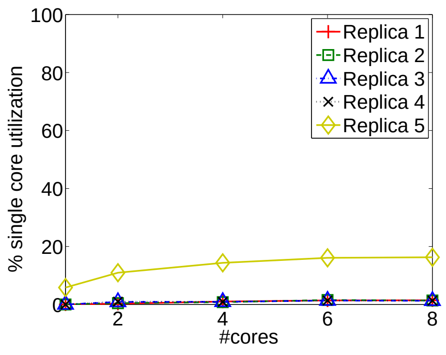
(a) Total CPU utilization ($n = 3$)



(b) Total blocked time ($n = 3$)



(c) Total CPU utilization ($n = 5$)



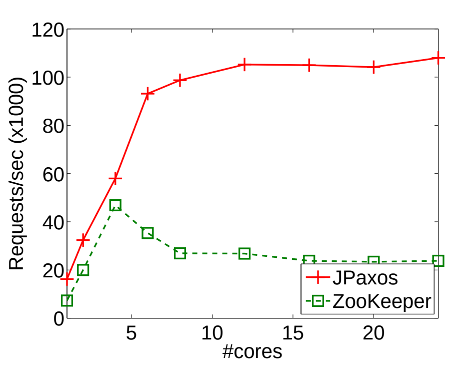
(d) Total blocked time ($n = 5$)

Figure 9.7: JPaxos CPU usage and total blocked time, edel cluster.

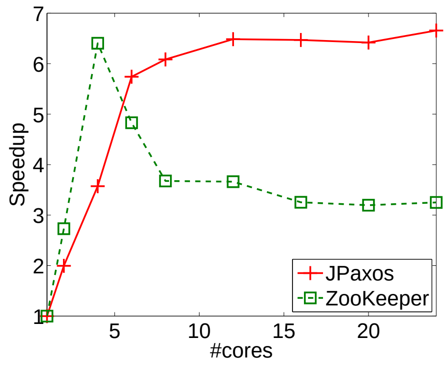
9.6.2 Comparison with ZooKeeper

In this section we compare the results of our threading architecture with ZooKeeper, to show how it improves over a production-quality implementation of RSM. Since ZooKeeper implements additional functionality on top of the core replication protocol, the absolute results are not directly comparable. However, there are some interesting trends in ZooKeeper's scalability with multi-core CPUs that illustrate the typical limitations faced by the traditional architectures, which are worth understanding and comparing with the improved behavior of JPaxos.

The tests in this section were performed with ZooKeeper 3.3.3. We used the default behavior for the connections between the clients and replicas, *e.g.*, the clients connect via TCP to a replica chosen randomly. As recommended in the ZooKeeper's documentation for achieving better performance, we configured the leader not to accept client connections. Each ZooKeeper client creates an ephemeral node at startup and then enters a loop issuing write requests (`setData()`) in that node. We used write requests to force ZooKeeper to execute the full ordering protocol, thereby making the results comparable with JPaxos which does not support local reads. Otherwise, the workload settings were similar to the ones used by JPaxos. As mentioned previously, all tests with JPaxos were done without stable storage. However,

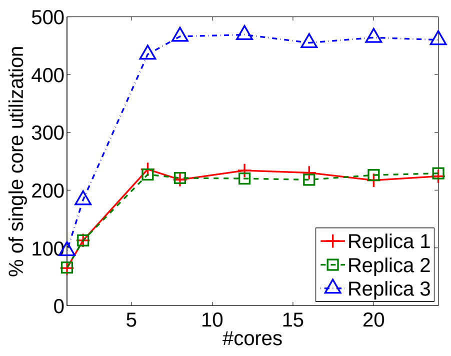


(a) Throughput (parapluie)

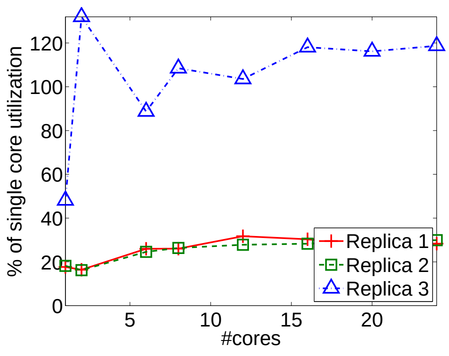


(b) Speedup (parapluie)

Figure 9.8: JPaxos vs ZooKeeper with increasing number of cores: parapluie cluster, $n = 3$



(a) CPU Usage



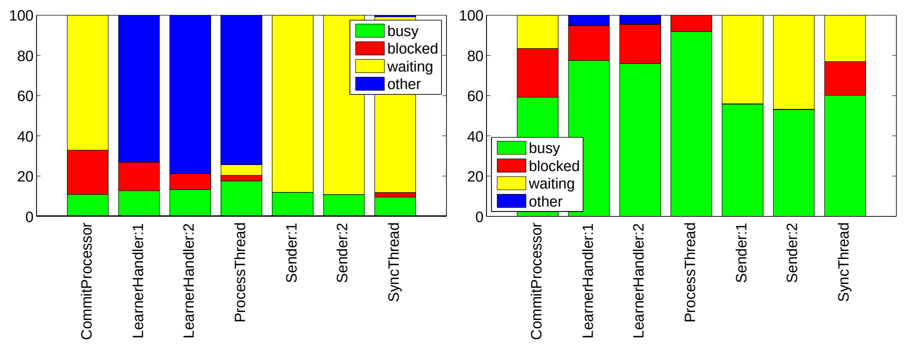
(b) Total blocked time

Figure 9.9: ZooKeeper's CPU usage and total blocked time. Parapluie cluster, $n = 3$

ZooKeeper does not allow disabling stable storage. As a workaround, we have used a ramdisk (`/dev/shm`) for its transaction log and for saving snapshots, which effectively removes most of the cost of disk writes.

ZooKeeper scales super-linearly up to four cores where it reaches a speedup of six (Figure 9.8b). However, for higher number of cores its performance degrades substantially, finishing at only four when all 24 cores are used. This degradation is caused by contention.

Figure 9.9b shows that the leader, Replica 3, suffers from very high levels of contention, with a very high aggregate blocking time. In comparison, the blocking time in JPaxos does not exceed 20% (Figure 9.5b). ZooKeeper's CPU utilization (Figure 9.9a) is another sign of the problems with its architecture. Although the throughput drops when using more than four cores, the CPU utilization continues increasing up to ten cores, when it finally stabilizes; this means that the additional CPU utilization is spent contending for locks. With JPaxos, on the other hand, the CPU utilization (Figure 9.5a) closely follows the throughput, indicating minimal or no CPU wasted because of contention. Looking in more detail at how threads spend their



(a) One core

(b) 24 cores

Figure 9.10: ZooKeeper: per-thread CPU utilization of the leader process.

time (Figure 9.10) further confirms the high level of contention. Even when using a single core, several threads spend between 10 and 30% of their time blocked. The situation gets worse when 24 cores are used, with one thread, the `CommitProcessor`, spending around 40% of its time blocked.

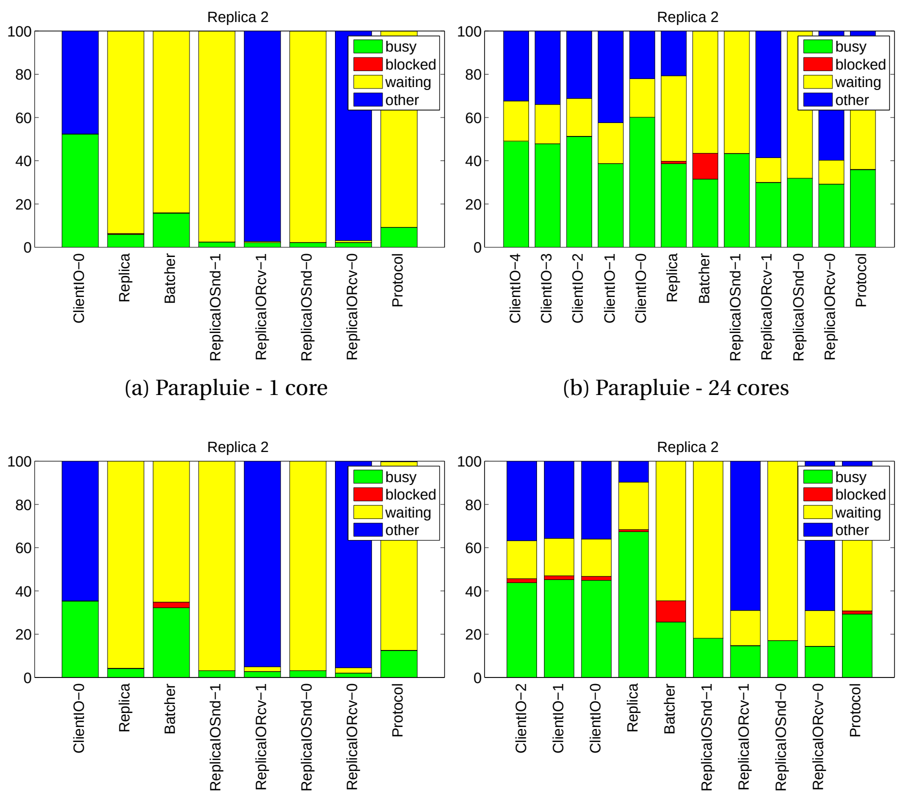
ZooKeeper also suffers from several single-thread bottlenecks, because the workload is poorly balanced among its threads. Figure 9.10b shows that when 24 cores are used, three of its main threads are busy or blocked 100% of their time, which limits the overall performance in spite of having more cores available.

9.6.3 Scalability limits of threading architecture

The results in Section 9.6.1 confirm that the threading architecture scales efficiently with the number of cores up to the limits of the networking subsystem of the nodes. Although this shows that the initial goal of this work was reached, it leaves open the question of what are the scalability limits of the threading architecture itself. We can, however, use the previous results to try to infer what would happen if the networking subsystem were faster. For instance, the aggregate CPU utilization is far from the maximum (500% for a maximum of 2400%), and the contention does not increase with the number of cores, suggesting that the performance could continue scaling.

In this section, we look under the hood of JPaxos by analyzing how the threads spend their time when the system is under load. The goal is both to better understand how the architecture works internally, and to identify any potential architectural bottlenecks. We discuss only the results with three replicas, since the results with five replicas do not differ substantially.

Figure 9.11 shows the CPU usage of the main threads in JPaxos. For each thread, we show the time spent executing (*busy*), blocked trying to acquire a lock (*blocked*), waiting on a condition variable (*waiting*), and in other states (*other*). The *waiting* state is a sign that the thread is idle, either because its input queue is empty or because its output queue is full, and thus must wait



(a) Parapluie - 1 core (b) Parapluie - 24 cores

(c) Edel - 1 cores (d) Edel - 8 cores

Figure 9.11: JPaxos per-thread CPU utilization of the leader process, $n = 3$.

for other threads to make progress. The *blocked* state indicates contention for locks. Finally, the *other* state accounts for the remainder of the time including, among others, the time spent sleeping (*i.e.*, calling `Thread.sleep()`), the time spent blocked on a system call (*e.g.*, waiting for I/O), and the time waiting to be scheduled by the operating system.

Figures 9.11a and 9.11c show the results for the tests with one core in the parapluie and edel clusters, respectively. JPaxos is CPU-bound in this test, as the sum of the busy time of all threads is close to 100%, which is the maximum with one core. In both cases, the ClientIO and the Batcher threads account for most of the CPU utilization, with the sum of their busy time reaching 80%. With all cores enabled in the parapluie cluster (Figure 9.11b) all threads are busy between 30 and 60% of the time. In the edel cluster (Figure 9.11b), the Replica thread stands out somewhat, at more than 60% busy time, while the other threads are under 40%. We will discuss why the Replica thread is likely to be a fundamental single-thread bottleneck in any RSM implementation. Nevertheless, the workload is fairly well balanced between the threads, as even the Replica thread is not more than 20% higher than the others.

The results also confirm that there is little contention among threads, with most threads spending almost no time blocked. The exception is the Batcher thread, which spends around

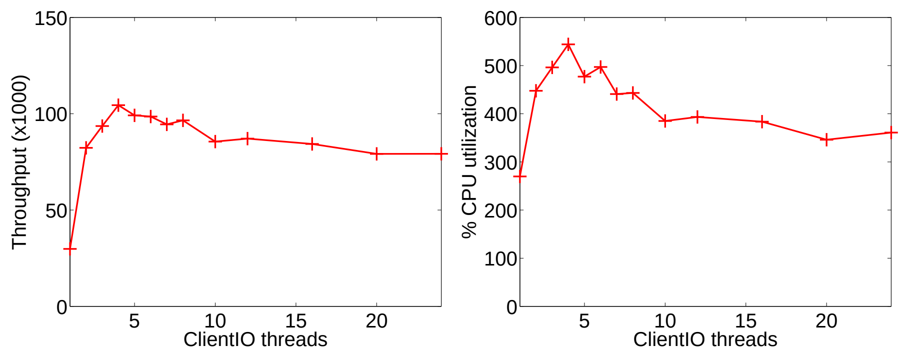
15% of its time blocked. Recall that this thread competes for locks both with the ClientIO threads (Request queue), and with the Protocol thread (Proposal queue), so it has a higher chance of being blocked. Although this is undesirable, it is not affecting the performance because the Batcher thread still spends over 50% of its time in the waiting state, *i.e.*, waiting for work.

From these results, we can extrapolate what is likely to happen in the absence of bottlenecks other than the ones inherent to the threading architecture. Interacting with clients is likely to continue scaling with the number of cores and with the workload. The absence of contention among the ClientIO threads confirms that this is a highly parallelizable task, so adding additional ClientIO threads will improve performance as long as enough cores are available.

Communication with other replicas should also scale. By using a pair of dedicated send/receive threads per replica, if more replicas are deployed there will be also more ReplicaIO threads, which can run independently given enough cores. The only potential bottleneck is at the level of an individual connection, since all the reading from (resp. sending to) another replica is done by a single thread. However, the per-connection workload is mainly a function of the size of the batches (which is under control of the RSM), being mostly independent of the number of the clients and number of replicas. And in our tests, even at peak throughput, the ReplicaIO threads are busy less than 40% of the time, suggesting that there is room for more than doubling the throughput given a faster network.

The busy time of the Replica, Batcher and Protocol threads is between 40 and 50%, suggesting that if no other bottlenecks were present, the nodes used in this experiment could sustain up to double the current throughput before hitting the single-thread performance limits of the CPU. Further improvements require changes to the architecture.

Next are some ideas that could extend even further the scalability of our architecture. The creation of batches can be parallelized by using several Batcher threads, each with its own queue of incoming requests. However, this change is far from being trivial to implement, since it requires addressing load balancing among Batcher threads and can potentially increase latency as batches take longer to be formed. The work done by the Protocol thread cannot be easily parallelized because of the complexity of the state managed by this thread. But since the work of this thread is proportional to the number of batches ordered, it is possible to reduce its workload by increasing the batch size. The Replica thread poses the biggest challenge, because its work is proportional to the number of requests and it cannot be easily parallelized, as at this stage requests are put in their final sequential order. A possible improvement is to separate this thread into two, one that handles all the housekeeping tasks related to replication and another that only executes the service. This would introduce some additional overhead in coordination, but it might improve performance overall. Additionally, it is also possible to optimize the single-thread performance. of the Replica thread.



(a) Throughput (b) Total CPU utilization at leader

Figure 9.12: Varying the number of ClientIO threads. Parapluie cluster, $n = 3$

9.6.4 Number of ClientIO threads

As mentioned previously, the number of ClientIO threads is an important factor in the scalability of JPaxos. In Section 9.6.1 we have reported for each number of cores tested the best results among several tests with different number of ClientIO threads. In this section, we look in more detail at the effect of this parameter on throughput.

Figure 9.12 shows the results of an experiment using all 24 cores of the parapluie nodes while varying the number of ClientIO threads, using the same experimental settings as in Section 9.6.1. The results show clearly that handling client connections offers a major opportunity for parallel execution: The throughput goes from 40K requests per second with one ClientIO thread to over 100K with four threads (Figure 9.12a), a 2.5x improvement with just three additional threads. However, the performance degrades slightly with more than eight threads, dropping to 80K requests per second. Figure 9.12b shows that the CPU utilization mirrors the behavior of the throughput, reaching a maximum of 550% with four ClientIO threads, then decreasing slightly.

Although we were not able to determine precisely the cause of the degradation in performance with more than eight threads, we ruled out contention inside the JVM. In JPaxos, contention could manifest either directly, as an increase in the time spent blocking by threads (for lock-based data structures), or indirectly, as an increase in CPU utilization (for non-blocking data-structures based on test-and-set instructions). But even with 24 ClientIO threads the total blocked time of all threads is under 10% of the run time, even less than with the optimal of four threads (See Section 9.6.3). And the CPU utilization with 24 threads is less than with eight threads, where it reaches the maximum throughput, which also suggests there is no contention in the JVM. However, one possible source of contention is the Linux TCP stack, which as mentioned previously suffers from scalability bottlenecks in the networking subsystem in the kernel version used in the tests [BWCM+10].

These results illustrate the importance of carefully choosing the level of parallelism, as often there is a narrow range in which the performance is optimal.

9.6.5 Determining the system bottleneck

In the experiments in the `parapluie` cluster, the throughput peaked at around 100K requests per second with eight cores, remaining at this level as the number of cores was increased further. The results did not show any clear indication of a bottleneck. From Figure 9.11b we can conclude that threads are not spending a significant amount of time blocked on locks, that there is no single-thread bottleneck (*i.e.*, the busy time of every thread is under 60%), and that the total CPU utilization is well under the maximum. Therefore, it remains to identify the bottleneck limiting the performance. In this section, we show that at this performance level the system is limited by the network subsystem of the leader replica. This confirms that we have not hit the scalability limits of the threading architecture of JPaxos, even in a 24 core cluster with high-end network equipment.

In the rest of this section we use *WND* to denote maximum number of parallel instances the leader is allowed to execute, and *BSZ* to denote the maximum batch size. Note that both are limits defined by configuration parameters. The actual number of parallel instances in execution at any given time and the actual size of batches created by the leader, vary during a run according to the workload.

Internal queues

We start by looking at what happens inside JPaxos during a run, by analyzing the average size of the internal queues used for communication between threads. Among the queues described in Section 9.5, there are three that are particularly important: the *RequestQueue*, the *ProposalQueue* and the *DispatcherQueue*. The first two show if the leader is waiting for client workload (queues are mostly empty) or if it is the clients that are waiting for the leader to process their requests (queues mostly full). The DispatcherQueue gives an indication of how busy the Protocol thread is.

We performed several tests varying the window size, measuring the average size of the three queues mentioned above. Additionally, we also measured the average number of parallel instances, which gives an indication of whether the leader is slow or if it is waiting for the answers from the other replicas. Table 9.1 shows the results.

In all tests, the *RequestQueue* is always more than one-quarter full (maximum 1000) and the *ProposalQueue* more than half full (maximum 20). This shows that the bottleneck is not between the clients and the leader, as the leader has batches waiting to be ordered. On the other hand, the *DispatcherQueue* is in average empty, indicating that the Protocol thread is most of the time idle. The reason for this is clear from the average number of parallel instances, which is always very close to the limit. This indicates that the Protocol thread cannot start new instances because it is waiting for messages from other replicas in order to decide the current

WND	*RequestQueue*	*ProposalQueue*	*DispatcherQueue*	Avg parallel instances
10	629.70 ± 23.94	14.33 ± 0.36	2.14 ± 0.25	9.63 ± 0.12
35	550.29 ± 8.11	14.94 ± 0.29	1.26 ± 0.41	34.67 ± 0.20
40	440.30 ± 7.49	14.97 ± 0.30	1.47 ± 0.31	39.50 ± 0.26
45	406.52 ± 8.36	14.85 ± 0.33	1.54 ± 0.54	43.88 ± 0.47
50	255.91 ± 10.90	12.99 ± 0.46	4.51 ± 1.14	45.86 ± 0.85

Table 9.1: Average size of internal queues and of the number of parallel instances. Parapluie cluster, $n = 3$, $BSZ = 1300$.

instances.

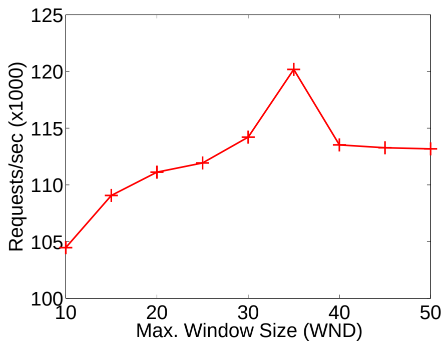
This shows that the bottleneck is located somewhere between the leader sending the messages to other replicas and receiving the corresponding replies. There are three main possibilities for the bottleneck: the bandwidth, the latency or the other replicas themselves. The bandwidth is not the limit because in the experiments above the maximum data rate reached by the leader[4], is 38MB/sec out and 22MB/sec in, which is far from the limit of 114MB/sec. We can also exclude the other replicas as the cause of the delay, as they were very lightly loaded in all experiments and showed no signs of contention. In the next section we investigate the remaining possibility, that is, that the latency is the cause of the delays.

Effect of window size

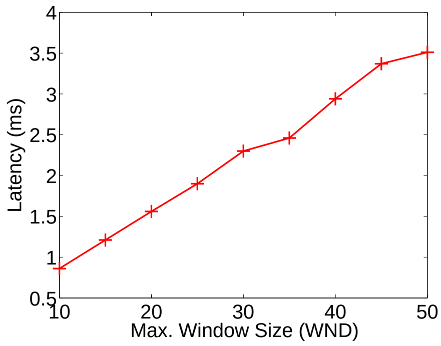
If the communication latency between the leader and the followers is the cause of the poor performance, then by increasing the limit *WND* the leader should be able to use its idle time to start more instances while waiting for the messages of the previous ones, thereby improving throughput. Figure 9.13 shows the throughput, latency, effective batch size and effective window size for the same set of experiments as in Table 9.1, *i.e.*, where *WND* varies from 10 to 50.

Increasing *WND* improves the throughput from 100K to a peak of 120K requests per second with $WND = 35$ (Figure 9.13a). But for higher values of *WND*, the throughput drops to 110K. The other plots explain the reason. Figure 9.13c shows that batches are always full and Figure 9.13d shows that the average number of parallel instances in execution at any given time is always very close to the limit. But in spite of executing more instances in parallel, the throughput does not increase after $WND = 35$. The reason can be inferred from the instance latency (Figure 9.13b), *i.e.*, the time the leader has to wait since it proposes a value for an instance until receiving at least one Phase 2b message from another replica, and thus deciding the instance. The instance latency grows steadily with the maximum number of parallel instances: up to 35, it grows slower than *WND*, resulting in a net gain in throughput, while after 35 it grows faster than *WND*, resulting in a decrease in throughput.

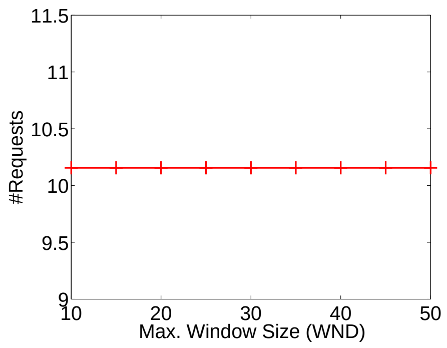
[4]These values were measured with the Ganglia monitoring interfaces of Grid 5000: https://helpdesk.grid5000.fr/ganglia/.



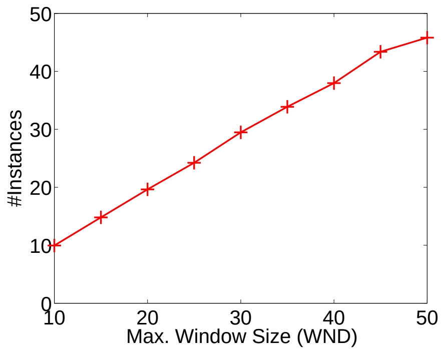
(a) Requests/sec



(b) Instance latency



(c) Avg batch size (*bsz*)



(d) Avg window size (w)

Figure 9.13: Performance as a function of window size. Parapluie, 24 cores, n=3, $BSZ = 1300$.

This increase in latency is not due to delays in processing the messages inside the replicas JVM, because as seen before both the leader and the non-leader replicas are far from being overloaded. To determine the cause, we have instead to look at the network latency between replicas during an experiment. Table 9.2 shows the round-trip-time (RTT) between nodes in the cluster before and during an experiment, measured using the `ping` command. While idle, the RTT is consistently around 0.06ms between any node. During an experiment, it is also 0.06ms between nodes not involved in the experiment. Between nodes involved in the experiment, but excluding the leader, the RTT becomes more irregular, varying between 0.06 and 0.08, a marginal increase. But between the leader and any other node in the cluster it increases to 2.5ms. This value matches the instance latency in this experiment (Figure 9.13b), which is also around 2.5ms. Since the RTT only increases when the leader node is involved, this leaves only the network subsystem of the leader as the cause of the delays. Also note that the `ping` command uses directly the low-level networking system of the kernel, which further confirms that the source of the delay is within the kernel and not in the JVMs or even in the TCP stack. This conclusion is validated by results published in [BWCM$^+$10], which show that the network subsystem of the Linux kernel in versions prior to 2.6.35 (so including version 2.6.26 used in our experiments) suffered from poor scalability in multi-core machines when

		ping (ms)
idle	any ↔ any	0.06
experiment	other ↔ other	≈ 0.06
	follower ↔ other	≈ 0.06
	follower ↔ follower	≈ 0.06-0.08
	leader ↔ any	≈ 2.5

Table 9.2: Ping times between nodes of the parapluie cluster, while idle and during an experiment ($WND = 35$, $BSZ = 1300$, $n = 3$). *other* denote a node in the cluster not involved in the experiment, and *follower* denotes non-leader replicas.

BSZ	Throughput	Packets/s (out/in)	Bandwidth (out/in)
650	83K	150/145 K	38/22 MB/s
1300	114K	150/145 K	44/25 MB/s
2600	119K	150/140 K	44/25 MB/s
5200	120K	150/135 K	44/25 MB/s

Table 9.3: Throughput and network utilization as a function of maximum batch size. Parapluie cluster, $n = 3$, $WND = 35$.

processing a large number of packets.[5]

Effect of batch size

The limiting factor in the experiments above is the number of packets processed by the network subsystem of the leader. We now investigate if increasing the maximum batch size improves the throughput beyond the maximum of 120K requests per second achieved in the previous section. The rationale is that for the same throughput, larger batches may decrease the total number of individual network packets handled by the leader, because the larger messages sent by the leader to the other replicas will be split into packets of an higher average size. Figure 9.14 shows the results.

As *BSZ* increases from 1300 bytes to a little over 10KB, the throughput remains at 120K requests per second (Figure 9.14a). Therefore, higher batches do not improve the throughput. To understand the reason, we show in Table 9.3 the average number of packets per second sent and received by the leader and the total outgoing and incoming bandwidth, as reported by the monitoring interfaces of Grid 5000.

For all batch sizes displayed, the number of packets per second sent by the leader is at 150K, once again the limit of the network subsystem of the leader. The leader uses the same network interface to communicate with the clients and the replicas. Therefore, for a throughput of X

[5] Since the experiments describes were performed, we have determined that the bottleneck is caused by the way the Linux kernel handles interrupts from the network card. By default, it directs all the network interrupts to a single queue that is served by only one core, which creates a single-thread bottleneck as the number of packets increases. However, it is possible to distributed the load among cores using mechanisms like Receive Side Scaling (RSS) and Receive Packet Steering (RPS) [HdB11]. We have repeated some experiments with these settings enabled and in most cases the throughput doubled.

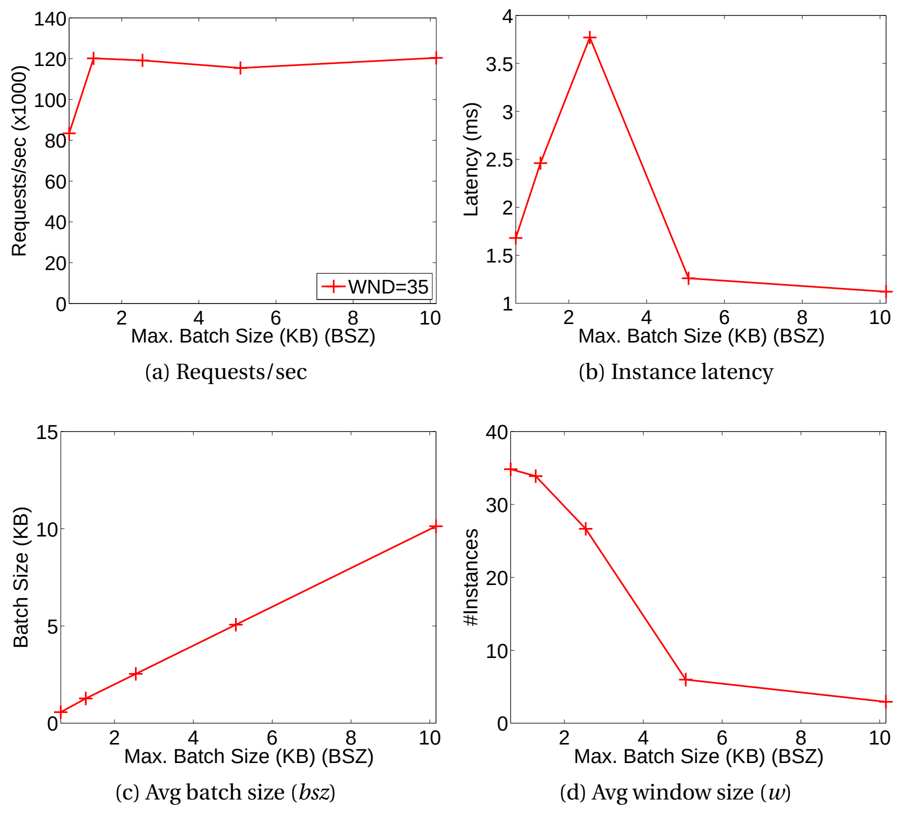


Figure 9.14: Performance as a function of batch size. Parapluie, 24 cores, $n = 3$, $WND = 35$.

requests/sec, the leader has to received/send X packets/sec from/to the clients (each request is sent in a single packet), regardless of *BSZ*. The leader must also use a number Y of packets/sec to order these requests, but this number will depend on *BSZ*: higher *BSZ* decrease the number of packets required to order the same number of requests. With this in mind, we can interpret the results.

With *BSZ* = 650 (Table 9.3), the leader sends a total 150K packets/sec, out of which 83K packets/sec are sent to the clients (one for each reply) and 67K to the replicas. This is an inefficient configuration, because an Ethernet frame can be up to 1500 bytes in a typical deployment, while the leader is sending batches of 650 bytes to the other replicas. With *BSZ* = 1300, the leader packs the double number of requests in a single Ethernet frame, therefore halving the number of frames exchanged with other replicas for the same throughput. The spare budget of network frames allows the leader to process more client requests, resulting in a higher throughput. Further increases in *BSZ* do not translate in a significant increase in the average size of each Ethernet frame sent by the client, because with *BSZ* = 1300 bytes the packets were already close to the limit. Therefore, the increases in throughput are small for *BSZ* > 1300.

9.7 Conclusion

As replication protocols improve and the network infrastructure becomes faster, implementations of replicated state machines are increasingly limited by the single-thread performance of the system. The reason is that most of these implementations, including research projects and production-quality implementations like ZooKeeper, are not designed to take full advantage of multi-core systems. In this chapter we have shown how to parallelize a generic implementation of a replicated state machine, so that its performance scales with the number of cores in the nodes.

The proposed threading architecture divides the internal state and tasks into a set of modules, with well-defined boundaries. At the core, there is a pipeline of event-driven stages that handle requests, with several satellite modules providing auxiliary services. As these modules differ substantially in complexity we used a variety of techniques, choosing the implementation of each module based on its potential for parallelism and complexity. We believe that the architecture proposed is general enough to be applied in a variety of implementations of state machine replication, with only minor adaptations needed.

The experiments show that in a 24 cores system, the architecture scales with the number of cores, until reaching the limits of the network subsystem. The results also suggest that in the absence of other bottlenecks, the performance would continue scaling with additional cores.

10 Scaling State Machine Replication - Eliminating the Leader Bottleneck

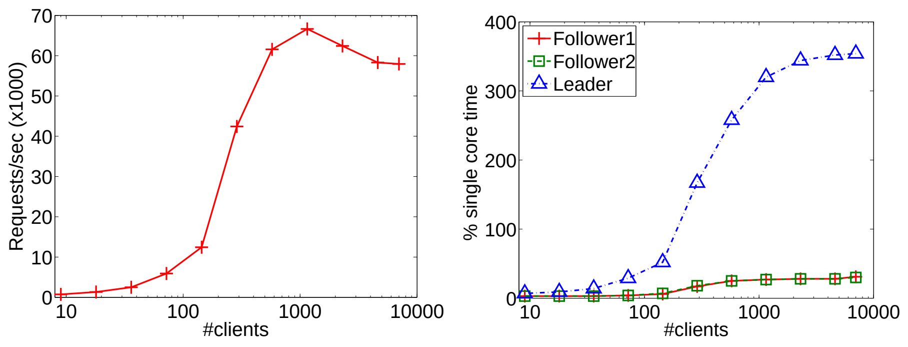
Implementations of state machine replication are prevalently using variants of Paxos or other leader-based protocols. Typically these protocols are also leader-centric, in the sense that the leader performs more work than the non-leader replicas. As a result, as the load of the system increases the leader quickly reaches its maximum capacity and becomes a bottleneck, while the other replicas are lightly loaded. In this chapter we show that much of the work performed by the leader in a leader-centric protocol can in fact be evenly distributed among all the replicas, thereby leaving the leader only with minimal additional workload. This is done (i) by distributing the work of handling client communication among all replicas, (ii) by disseminating client requests among replicas in a distributed fashion, and (iii) by executing the ordering protocol on ids. We derive S-Paxos, a variant of Paxos incorporating these ideas, and show that it achieves an almost perfectly balanced use of resources among replicas. By balancing resource use among all replicas, our protocol not only achieves significantly higher throughput for any given number of replicas, but also increases its throughput with the number of replicas.

Publication: Martin Biely, Zarko Milosevic, Nuno Santos and André Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *31st IEEE International Symposium on Reliable Distributed Systems (SRDS 2012)*, Irvine, California, USA, October 8-11, 2012.

Joint work with Zarko Milosevic.

10.1 Introduction

In Chapter 9, we have shown how to improve the performance of state machine replication by taking advantage of the large number of cores available in modern multi-core CPUs. In more general terms, this is an example of how to improve performance of state machine replication by using system resources that are usually left unused by conventional designs and



(a) Throughput (b) CPU usage

Figure 10.1: Performance of a typical leader-centric Paxos. Request size 20 bytes, four core machines. See Section 10.4 for experimental settings.

implementations, in this case, CPU cores. In this chapter we apply the same idea to a different type of resource: the resources of the follower replicas in leader-based protocols.

Implementations of state machine replication are prevalently using variants of Paxos [Lam98] or other leader-based protocols. Being leader-based means that there is a replica that assumes a distinguished role, in the case of Paxos, the distinguished proposer. This is a fundamental property of this approach to establishing a total order. Most of these protocols are also leader-centric, in the sense that the leader does significantly more work than the followers. As a result, when the load in the system increases, the leader is the first replica to run out of resources (such as CPU and network bandwidth), thereby being the bottleneck of the system. This happens even though other replicas usually are only lightly loaded, with plenty of idle capacity.

Figure 10.1a illustrates our observations, by showing the throughput of JPaxos (cf. Section 10.4 for details of the experimental setup) for different number of clients. The maximum throughput is achieved with just over 1000 clients. At this point, the leader's CPU becomes the bottleneck (cf. Figure 10.1b). When this happens, an additional increase in the number of clients results in the throughput decreasing. The other replicas, however, are only lightly loaded. Since the bottleneck is at the leader, introducing additional replicas will not improve performance; in fact it will lead to a slight decrease in throughput since the leader will need to process additional messages.

Contribution In order to overcome this shortcoming of leader-centric protocols, we propose S-Paxos (S stands for scalable), a novel SMR protocol for clustered networks derived from Paxos. S-Paxos achieves high throughput by balancing the load among replicas to use otherwise idle resources, thereby avoiding the bottleneck at the leader. Furthermore, its throughput keeps increasing with the number of replicas (up to a reasonable number). This way, S-Paxos overcomes the traditional trade-off between fault tolerance and throughput present in most state machine replication protocols: in S-Paxos higher fault tolerance actually leads to higher throughput.

In order to derive S-Paxos, we start by observing that a replicated state machine performs the following tasks:

- *Request dissemination:* receiving requests from clients and distributing them to all replicas
- *Establishing order:* reaching agreement on the order of requests
- *Request execution:* executing requests in the determined order and sending replies to clients.

In leader-centric protocols such as Paxos, most responsibility for these three tasks rests with the leader, while the followers are left only with acknowledging the order proposed by the leader and executing requests. The S-Paxos key design guideline is to *distribute the three tasks evenly across replicas.* In order to do so, we turn the first two tasks into separate layers: dissemination and ordering layer. The *dissemination layer* is then balanced among all replicas by having all replicas accept requests from clients and disseminate them. Since request dissemination is handled by the dissemination layer, the role of the *ordering layer* is only to determine the order in which requests will be executed. We use normal Paxos for this layer with one difference: it orders request ids instead of full requests. Therefore, the additional overhead of being the leader in S-Paxos consists only in ordering ids. After *executing* the request, the replica that received the request from the client sends the corresponding reply.

We have implemented a prototype of S-Paxos and performed a detailed experimental evaluation, which shows how our protocol circumvents the different bottlenecks at the leader that appear in leader-centric protocols like Paxos. As we will demonstrate, our protocol is able to reach unprecedented performance for an application-agnostic ordering service. For example, for typical replicated state machine deployment sizes (3-7 replicas according to [JRS11]), S-Paxos achieves a throughput of 500K requests per second for 3 replicas and 750K requests per second for 7 replicas.

We focus our discussion in LAN environments and do not discuss stable storage. We do so because our goal is to study the performance of the ordering protocol, which would be obscured in the presence of bottlenecks like disk access or WAN latency. However, S-Paxos does not rely on any LAN-specific service, like IP multicast, and therefore can also be deployed in a WAN environment.

Roadmap The remainder of the chapter is organized as follows. In Section 10.2 we identify potential bottlenecks in Paxos and how they limit its performance. We then derive S-Paxos from Paxos in Section 10.3. In Section 10.4 we evaluate the performance gains of S-Paxos over Paxos, and explain how S-Paxos circumvents the limitations of Paxos. Section 10.5 summarizes related work and Section 10.6 concludes the chapter.

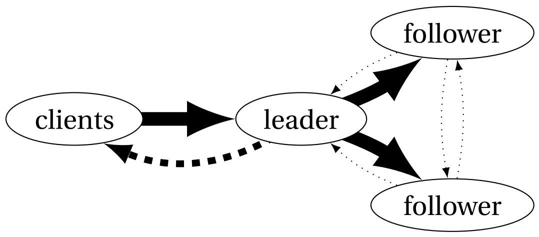


Figure 10.2: Data-flow in Paxos for $n = 3$; solid lines for requests, dashed for replies, and dotted for consensus related messages; thickness depends on number and size of messages.

10.2 Why is Paxos leader-centric?

We now look in more detail at the traditional leader-centric variants of Paxos, with the goal of identifying the reasons why the protocol is unbalanced. Chapter 7 describes in detail the Paxos protocol. Here, we focus only on the resource usage of the different stages of the protocol.

Figure 10.2 shows a rough overview of the data-flow of Paxos. Usually, the leader does all communication with the clients, *i.e.*, it receives all requests and sends all replies. Some implementations, however, allow the clients to contact any replica, which will then forward the request to the (current) leader [MJM08, AK08, JRS11]. In both cases, the leader is responsible for starting instances containing one or more client requests, which involves sending a message containing the requests to all replicas. It then waits for a majority of acknowledgments before deciding. If the leader is doing the communication with the clients, it also has to send the replies to the respective clients after executing the requests.

From the above, it is clear that the traditional design of Paxos is heavily leader-centric. Even if the work of receiving client requests is distributed among all replicas, the leader is still responsible for most of the work of disseminating each client request to all replicas, which increases significantly its resource usage as compared to the followers.

There are three types of resources that can cause a bottleneck. The first resource that we consider is the network bandwidth. The CPU is also a critical resource, as it is used to serialize and deserialize messages, to execute the replication protocol, and to run the service itself. A less obvious resource that can also become the bottleneck is the network subsystem [DB96] (network interface card, network driver and operating system networking stack), which can only handle a certain number of packets per second.

Bandwidth When the average request size is large enough, the outgoing channel of the leader saturates before the CPU or network subsystem limits are reached. The reason for this is that request dissemination requires the leader to inform the other replicas of the requests. Recall that this is done through the Phase 2a message, in which the leader sends requests to all other replicas. Therefore, in this case the maximum throughput of Paxos is limited by $\frac{B}{n-1}$, where B is the bandwidth available on the leader's outgoing link. In this case, we say that the system is *bandwidth bound*.

Note that when a multicast primitive is available (for instance, IP multicast), then the maximum throughput we can reach is roughly the leader's total outgoing bandwidth (*e.g.*, [MPSP10]) minus what is required to send replies to the clients. However, support for IP multicast is not always available, and if available it may be prohibited from use as it can easily lead to congestion collapse when overused. This is the case in many data-centers, as Vigfusson et al. point out [VALB$^+$10]. As we are focusing on clusters (in data-centers), we do not consider IP multicast in the remainder of this chapter.

CPU and network subsystem When the average request size is small (contrasting the above case), then the leader's CPU can become the bottleneck long before the outgoing link becomes saturated [MPSP10, JRS11]. In this case, throughput is determined by the processing power of the leader.

We have identified two possible locations for the bottleneck, which we will discuss separately: the user-level code (*i.e.*, application and state machine implementation) and the network subsystem. In the first case, we say that the system is *CPU bound*, while in the second we say it is *network-system bound*.

Contrary to the case of the bandwidth, where the maximum throughput is a simple function of the available bandwidth and of the request size, in the case of the CPU the situation is more complex. The maximum number of requests that a node can process depends not only on the speed of the CPU, but also on the efficiency of the Paxos implementation, both in terms of single-thread efficiency and scalability with multiple cores. In Chapter 9 we have shown that a Paxos implementation with a carefully designed multi-threading architecture can scale its performance with the number of cores. However, if no other bottleneck is present in the system, eventually the leader will reach the limit of the CPU to execute the user-level code, *i.e.*, the system is CPU bound.

The network system can also become the bottleneck. The reason why this might happen is that as the network card receives packets, it raises interrupts to inform the CPU of the arrivals. The CPU has to execute the interrupt handler, which does some work to read the packet and redirect it for further processing. With very-high packet rates, the work done by the interrupt handlers might become enough to saturate a single core, thereby creating a bottleneck. The number of cores allocated to process the interrupt handlers depends on the network card, network drivers and configuration of the network subsystem, but in many configurations it is often a single core. As we will see in Section 10.4 this can indeed become a bottleneck when a single core is used to handle interrupts.[1]

Although the exact point of saturation of the CPU will differ greatly among implementations of

[1] In Section 10.4.4 we show that when the hardware and OS allow using multiple cores, this bottleneck is greatly alleviated.

Paxos and of the network stack, the general rule is that the smaller the request, the more likely is that the leader will become either CPU bound or network-system bound before hitting the bandwidth limit.

In any case, the system will reach a bottleneck when the leader runs out of one of these resources, even though the followers might have plenty of spare capacity on the same resource. Adding more replicas to the system will make things even worse because the leader will need to do even more work (it has to send and receive messages to and from more replicas [AK08, JRS11]). Therefore, the traditional leader-centric Paxos is clearly an unbalanced protocol, which does not make efficient use of all the resources available on the system, limiting throughput and scalability.

10.3 S-Paxos

In the previous section we have seen that in Paxos the bottleneck at the leader is mainly caused by the leader having to *receive client requests* and then *disseminate them* to all other replicas. The latter is done through the Phase 2a messages. A key insight behind S-Paxos is that the Phase 2a message in leader-centric variants of Paxos serves the dual role of disseminating requests and proposing an order; while having a leader propose an order is the core idea of Paxos, using the leader to disseminate the requests is not. In fact, request dissemination (together with receiving requests from clients) is a task that can be performed by all replicas. For this reason S-Paxos separates these two roles into two different layers: the *dissemination layer* and the *ordering layer*. As explained in detail below, in S-Paxos request dissemination is a fully distributed protocol with a balanced workload: all replicas accept requests from clients and disseminate them directly to all other replicas. The ordering layer uses the conventional Paxos protocol, with the difference that the order is established on request ids instead of on full requests.

Since S-Paxos is a variant of Paxos, we make the same assumptions about the system as Paxos (see Chapter 7). We defer explaining how S-Paxos deals with failures to Section 10.3.2, and first focus our attention on the normal case.

10.3.1 Normal operation

We now describe in detail how our dissemination layer works and how it interacts with the ordering layer (Paxos). In the pseudo-code of Algorithm 10.1 we use propose(v) (line 17) to pass a value to the ordering layer, which (later) signals decision in the i^{th} instance of Paxos by the decide(i, v) event (line 19). Moreover, since only the leader can propose an order on ids, the dissemination layer also needs to be aware of the identity of the leader once it successfully finishes Phase 1.

As for Paxos we focus our exposition of S-Paxos on how a request is processed. Clients can

Algorithm 10.1 Dissemination layer algorithm (code of process p).

```
Parameters:
  $\eta$ /* delay for retransmission of ACK */
  $\Delta$ /* delay before polling for request */
Initialization:
  $requests_p \leftarrow \emptyset$ /* requests/ids known to dissemination layer */
  $proposed_p \leftarrow \emptyset$ /* Tracks requests proposed */
  $decided_p \leftarrow \emptyset$ /* Maps instances to request ids */
  $stableIds_p \leftarrow \emptyset$ /* ids acknowledged by at least $f+1$ replicas */

upon receive request $\langle uid, r\rangle$ from a client
  send $\langle$FORWARD, $uid, r\rangle$ to all

upon receive $\langle$FORWARD, $uid, r\rangle$ from $q$
  $requests_p \leftarrow requests_p \cup \{\langle uid, r\rangle\}$
  send $\langle$ACK, $\{uid\}\rangle$ to all

upon receive $f+1$ distinct acks for $uid$
  $stableIds_p \leftarrow stableIds_p \cup \{uid\}$
  if $p$ is leader and $uid \notin proposed_p$ then
    propose($uid$)
    $proposed_p \leftarrow proposed_p \cup \{uid\}$

upon decide $(i, uid)$
  $decided_p \leftarrow decided_p \cup \{\langle i, uid\rangle\}$

upon exists $uid, r, i$ such that $\langle i, uid\rangle \in decided_p$
    and $\langle uid, r\rangle \in requests_p$ and $r$ not executed
  execute $r$ in order $i$

every $\eta$ time do
  $S \leftarrow \{uid : \langle uid, r\rangle \in requests_p\}$
  send $\langle$ACK, $S\rangle$ to all

upon receive $\langle$ACK, $S\rangle$ from $q$
  for all $uid \in S : \langle uid, r\rangle \notin requests_p$ do
    if within $\Delta$ time request $\langle uid, r\rangle$ was not yet received then
      send $\langle$RESENDFORWARD, $uid\rangle$ to $q$

upon receive $\langle$RESENDFORWARD, $uid\rangle$ from $q$
  send $\langle$FORWARD, $uid, r\rangle$ to $q$

upon view change finished $\langle reproposed\rangle$
  /* executed by new leader only */
  $proposed_p \leftarrow proposed_p \cup reproposed \cup \{uid : \langle uid, r\rangle \in decided_p\}$
  for all $uid \in stableIds_p \setminus proposed_p$ do
    propose($uid$)
  $proposed_p \leftarrow proposed_p \cup stableIds_p$
```

send requests to any replica.[2] When a replica receives a request from a client, it forwards the request, along with its unique id, to all other replicas (lines 9-10). When a replica receives a forwarded request (either from itself or from another replica), it records the id and the request in the *requests* set. It then sends an acknowledgment containing only the id to all (lines 11-13). Since messages can be lost during periods of asynchrony (partial synchronous system), all acknowledgments are periodically retransmitted (lines 23-25).

After receiving $f+1$ acknowledgments for a particular request id a replica records this fact by adding the id to its *stableIds* set (we will discuss the importance of this set in the next subsection). Moreover, the leader replica also passes the request id to the ordering layer, which will then use the Paxos protocol to order it (lines 14-18).

As in Paxos the order in which requests are executed is defined by the sequence of decisions of the consensus instances. The only difference is that our ordering layer only orders ids. As it is possible (especially in periods of asynchrony) that an id is ordered before the corresponding request is received, S-Paxos cannot immediately execute requests as they are decided. Instead the decision is recorded (line 20) and execution is done only once the request is available at the replica (lines 21–22). After executing the request the replica that received the request from the client sends the corresponding reply.

10.3.2 Handling failures

Since the ordering layer uses Paxos without any modifications, failure detection, leader election, view change, and retransmission of messages in the ordering layer are done as in Paxos. However, due to the separation of dissemination from ordering, failures can affect S-Paxos in ways not handled by the ordering layer alone. Therefore we now discuss the fault tolerance of the dissemination layer.

Ensuring stability of client requests One aspect that makes performing ordering on ids instead of requests non trivial is the need to ensure that once an id is decided, the corresponding request is available in the system [ES06]. Requests that will remain available in the system even in the presence of crashes are called *stable*. In Paxos, stability is ensured by waiting for at least $f+1$ Phase 2b messages before deciding, which indicates that at least one correct process received the request as part of the Phase 2a message.

As S-Paxos disseminates requests in a separate layer (*i.e.*, outside Paxos), it cannot rely on the ordering layer to ensure stability of ordered requests. Rather, S-Paxos ensures that a request is stable before proposing the corresponding id. More specifically, the leader proposes an id only once it has received $f+1$ acknowledgments for the corresponding request (line 14). This ensures that decided ids always correspond to client requests that reached at least one

[2] We do not address the issue of load balancing in this work. However, when all clients generate similar loads, a simple strategy is to let clients randomly pick a replica.

correct replica, say p. As p will include the corresponding request id in the ACK message sent periodically to all, other replicas can (upon receiving such ACK message) retrieve the request from p (lines 14–18, 23–25, 26–29). The same mechanism ensures (as $n \geq 2f + 1$) that any request received by at least one correct replica will eventually be ordered.

Conversely, if the request does not reach any correct replica, then it will never be proposed by the leader (as there will never be $f + 1$ acks). In this case, the clients that sent the requests will have to time out and retransmit it to another replica. Here unique request ids do not only prevent duplicate execution, but also allow replicas to send the correct response to previously executed requests.

View change As long as there are no leader changes, the mechanisms above ensure that any request that becomes stable at some replica is proposed once and only once. However, in the presence of leader changes this is no longer the case.

During view change, the new leader will establish a new view based on the information received in the Phase 1B messages. This view includes all the ids that were decided or could have been decided in a previous view. To complete view change, Paxos will propose again the ids that may have been decided and mark locally as decided the ones that were flagged as decided in at least one of the Phase 1A message.

The dissemination layer must take this in consideration in choosing which ids to pass to the ordering layer for ordering. On the one hand, to avoid ordering the same id more than once, the dissemination layer should not pass any id that was already decided or was re-proposed as part of view change. On the other hand, to ensure that the id of every stable request is ordered at least once, the dissemination layer has to pass to the ordering layer all ids that are not part of the new view. These rules apply regardless of whether the request corresponding to the id has already become stable locally or not.

We address these two cases with a call-back (lines 32–37) from the ordering layer that informs the dissemination layer that the new leader has finished Phase 1 of Paxos (Figure 7.1). At this point the dissemination layer (of the leader) will propose all stable ids (line 36), unless it is known that these ids have been decided (the ids are in the *decided* set) or they are already (re)proposed from within Paxos (the *reproposed* set).[3] After this call-back is executed the new leader can again use the set *proposed* to ensure that no id is proposed more than once.

10.3.3 Optimizations

Since the ordering layer uses the standard Paxos protocol, it can use the traditional optimizations of batching and pipelining, as well as any other optimization that applies to Paxos.

[3]If this information is not available an alternative approach is to filter out duplicate requests in the ordering layer or during execution. While the former requires changing the ordering layer which we try to avoid, the latter solution incurs some wasted resources. Therefore using the *reproposed* set can also be seen as an optimization.

Batching can also be used as an optimization at the dissemination layer: Instead of forwarding requests directly, a replica p can first group them into batches, assign them a unique batch id, *e.g.*, of the form $\langle p : sn \rangle$, where sn is a local sequence number, and then broadcast them. The remainder of the protocol will then operate on batch ids and batches of requests rather than request ids and individual requests.

As presented in the algorithm above, the size of the periodic ACK messages will grow forever. This can be easily avoided, by using the id schema for requests proposed above for the batching optimization. This allows a replica to group all consecutive batch ids generated by one replica into intervals, and only transmit the bounds of these intervals. When one uses TCP for communication between the replicas the dissemination and reception of batches will occur in a FIFO manner (as long as connections are not interrupted). Therefore, in practice, the number of transmitted intervals is very low, and so this approach is sufficient to keep the size of acknowledgment messages within reasonable bounds.

Another possible optimization is to piggy-back the acknowledgments on the messages used to forward batches, with explicit acknowledgments sent only as a fall-back mechanism if no other messages are being exchanged. This optimization is especially effective when the system is under high load.

10.3.4 Discussion

The protocol above distributes *request reception*, *request dissemination* and *sending replies* across all replicas. The only remaining leader-based task, ordering, is now very lightweight since it is done only on ids. As the experimental results presented in Section 10.4 confirm, in S-Paxos there is no perceptible difference between the leader and the followers in terms of resource usage, which results in better performance than traditional leader-centric protocols.

Additionally, S-Paxos is also able to scale with the number of replicas, up to reasonable numbers. This contrasts with conventional leader-centric protocols, where higher fault tolerance typically results in lower performance. The reason for S-Paxos to scale is that the most resource intensive tasks of the protocol, *i.e.*, client request reception and dissemination, are now distributed across all replicas so that adding more replicas to the system may increase the throughput of the dissemination layer. As we will show on Section 10.4, the gains depend on what resource is the bottleneck, with the biggest gains being achieved when either the CPU or network subsystem are the bottleneck.

The fact that only ids are proposed, also allows the dissemination of requests to continue throughout leader change. It is only the ordering layer that does not make progress while a new leader is determined. Moreover ordering only ids also leads to another advantage during view change: the state that has to be transferred from the replicas to the new leader during Phase 1b can be significantly smaller for S-Paxos when compared with Paxos. All this makes view changes very lightweight in S-Paxos (cf. Section 10.4.3).

Cluster	CPU	Network	Used in Section
Helios	2×2cores@2.2GHz	1Gbps	10.4.1, 10.4.2
Parapluie	2×12cores@1.7GHz	1Gbps	10.4.1
Paradent	2×4cores@2.5GHz	1Gbps	10.4.1, 10.4.3

Table 10.1: Grid5000 clusters used for experiments.

		S-Paxos		Paxos
Cluster	Req Size	*cbsz*	*bsz*	*bsz*
Helios	20	1450	50	1450
Parapluie	20	1450	50	1450
Paradent	1024	8700	50	8700

Table 10.2: Experimental settings. Sizes in bytes. CBSZ - client batch size (dissemination layer), BSZ - ordering layer batch size.

S-Paxos is designed for high throughput, which does not come for free. Compared to Paxos, it requires a higher number of communication steps to order a client request. Additionally, the two levels of batching (at the dissemination and at the ordering layer) can also harm response time. As our evaluation in Section 10.4.2 shows, we can indeed observe a moderate increase (less than 10ms) in average response time with low and medium client load. However, for high load the situation is reversed: the performance of Paxos eventually reaches its peak, leading to dramatically higher client response time, while for S-Paxos the throughput continues to increase and the response time remains low.

10.4 Performance evaluation

We have implemented S-Paxos by modifying JPaxos, and have compared the two implementations experimentally. The ordering layer of S-Paxos is identical to JPaxos. The dissemination layer implements the optimizations described in Section 10.3.3, *i.e.*, batching of client requests, compact representation of acknowledgments, and piggybacking of acknowledgments. As seen in Chapter 7, JPaxos follows a conventional leader-centric design. And since JPaxos and S-Paxos differ only on the aspects described in this chapter, the results of our experiments reflect the inherent differences between a leader-centric and the balanced approach proposed in this chapter.

The experiments were run in several clusters of the Grid5000[4] testbed (see Table 10.1). The network was always a Gigabit Ethernet with an effective inter-node bandwidth of 930Mbps (measured using `iperf`). Nodes were running Linux, kernel version 2.6.32-5, and the Java Virtual Machine used was Oracle's JRE version 1.6.0_25.

The workload was generated by nodes located in the same cluster as the replicas, each running

[4] `https://www.grid5000.fr`

several client threads in a single Java process. The clients use TCP to communicate with replicas. In JPaxos they communicate with the leader only, while in S-Paxos clients connect to a random replica. Clients send requests in a closed loop, waiting for the answer to the previous request before sending the next one[5]. Each experiment was run for 3 minutes, with the first 10% ignored in the calculation of the results. To focus our evaluation on the ordering protocol, we used a null service that discards the payload of the request, sends back a fixed 8 bytes response and does not use stable storage. The overhead of using a more complex service or stable storage would easily dominate the ordering part.

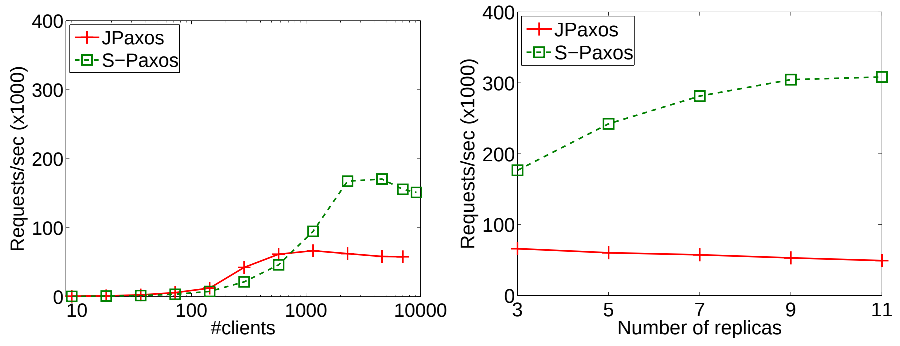
In all experiments the bound on the number of parallel Paxos instances (pipelining) was set to 30. Table 10.2 summarizes other experimental settings (for S-Paxos, *cbsz* refers to the batching of request ids of Section 10.3.3, and *bsz* to the batching of ids in the Paxos ordering layer; for Paxos, *bsz* refers to the batching of requests in Paxos). The values were chosen to match the natural limits of the underlying Ethernet network (1500 bytes maximum payload of a frame and optimal performance usually with messages of around 8KB).

We use as metrics the throughput in requests per second and in data ordered per second (Mbps), the client response time, and the CPU utilization. The *client response time* is the time from when the client sends a request until it receives the corresponding reply, which includes dissemination, ordering and execution. The *CPU utilization* of a replica is measured using the GNU `time` command and is shown as a percentage of one core, *i.e.*, 100% is equivalent to one core being fully utilized.

In Section 10.4.1 we describe the throughput of JPaxos and S-Paxos in different scenarios when the system bottleneck is the CPU, the networking subsystem, or the bandwidth. For each of these scenarios, we choose the workload (number of clients and request/response size), the algorithm parameters (*e.g.*, batch size), and clusters (Table 10.1) such that JPaxos hits the intended bottleneck. In Section 10.4.2 we measure the client response time of JPaxos and S-Paxos under different client loads. In Section 10.4.3 we study the effect of view changes and crashes on performance, and finally in Section 10.4.4 we explore what is the maximum throughput that S-Paxos can achieve with recent (as of 2011) mid to high-end range commodity servers, which are representative of current data-centers.

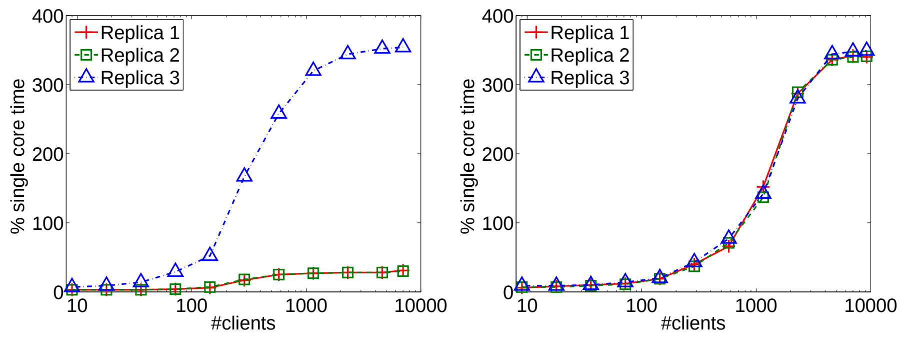
In the tests that show the throughput with increasing number of replicas (n), we were interested in the maximum throughput of each protocol. However, JPaxos and S-Paxos achieve their maximum throughput with very different number of clients (see Figure 10.3a): S-Paxos reaches its maximum with a number of clients that is significantly higher than what JPaxos can support without its performance starting to degrade. Therefore, we repeated each experiment using different number of clients, and report the highest throughput for each value of n.

[5] Since clients can have only one outstanding request, in order to saturate the system we had to use a high number of clients. In case clients can issue several parallel requests, a smaller number of clients would be sufficient.



(a) n=3, different client load (b) Max throughput for different n

Figure 10.3: Throughput of JPaxos and S-Paxos when CPU is the bottleneck. Helios cluster.



(a) CPU usage of JPaxos (b) CPU usage of S-Paxos

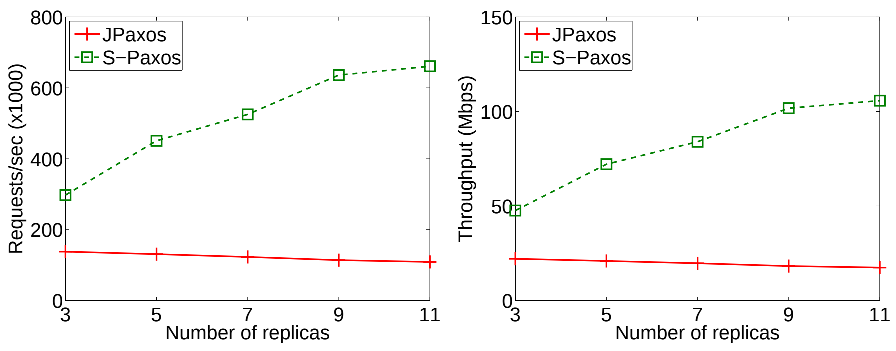
Figure 10.4: CPU usage at different replicas with a) JPaxos and b) S-Paxos with increasing number of clients. Helios cluster.

10.4.1 Throughput

When the CPU is the bottleneck

For testing with the CPU as the bottleneck, we have chosen a request size of 20 bytes. As discussed in Section 10.2, small request sizes put a much greater stress on the CPU than on the bandwidth. Moreover, we used a cluster with a small number of cores, *e.g.*, 2×2cores@2.2GHz. As our results show, in these conditions the CPU is indeed the bottleneck both for JPaxos and S-Paxos (although at very different throughput levels).

When $n = 3$, the throughput of JPaxos increases with the number of clients until it reaches a maximum at around 75K requests per second with just over 1'000 clients, (Figure 10.3a). At this point, we can see in Figure 10.4a that one replica, the leader, is using over 300% of CPU, which is close to the maximum of 400% for this setup (recall that the nodes in the Helios cluster are four-core machines). The other replicas, however, are only lightly loaded. In contrast, with S-Paxos (Figure 10.4b) the CPU usage is equally distributed among all replicas and grows



(a) Throughput(req/sec) (b) Throughput(Mbps)

Figure 10.5: Throughput of JPaxos and S-Paxos when network subsystem is the bottleneck. Parapluie cluster.

slower than on the leader when using JPaxos. Like with JPaxos, the throughput also levels off when the CPU usage exceeds 300%. However, S-Paxos reaches a throughput of almost three times the one of JPaxos, with 200K requests per second while serving 8'000 clients.

When the number of replicas is increased (Figure 10.3b), S-Paxos throughput increases, reaching a maximum of 300K requests per second with eleven replicas. This contrasts sharply with JPaxos, whose throughput decreases with the number of replicas, going down from around 75K with $n = 3$ to 50K with $n = 11$. With $n = 11$, S-Paxos achieves six times the throughput of JPaxos. The reason is that S-Paxos is able to spread the workload of handling client connections and dissemination among all replicas, while in JPaxos the leader must perform all these tasks, thus quickly maxing out its CPU bottleneck.

When the network subsystem is the bottleneck

For these tests, we use the Parapluie cluster. In contrast to the Helios cluster used in the previous experiments, the nodes in the Parapluie cluster have a high number of cores (24 versus 4). And since both JPaxos and S-Paxos are designed to make efficient use of multiple cores, this means that in the Parapluie cluster neither of them is CPU bound anymore. Instead, the bottleneck shifts to the network subsystem which can only use a single core, due to limitations on the network stack of the version of Linux used for the tests.

In Figures 10.5a and 10.5b we show the throughput of JPaxos and S-Paxos in requests per second, and data ordered per second (Mbps). The results confirm that the bottleneck in this case is indeed the network subsystem: The CPU usage during the tests is always below 600% out of a maximum of 2400% (results not shown here), and the peak data rate of 100Mbps is far from the maximum bandwidth of a Gigabit network (Figure 10.5b). This is in agreement with the results presented in Section 9.6.5, where we determined that when JPaxos is run on the Parapluie cluster, the maximum number of packets per second that the network subsystem of

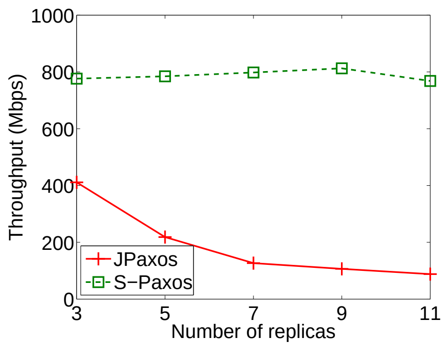


Figure 10.6: Throughput of JPaxos and S-Paxos when bandwidth is the bottleneck. Paradent cluster.

each node is able to handle is around 150K.

Like in the case where the bottleneck is the CPU, as the number of replicas increases the throughput of JPaxos gets worse while the one of S-Paxos improves (Figure 10.5a). Additionally, the throughput of S-Paxos is always substantially higher than the one of JPaxos: For $n = 3$ S-Paxos has double the throughput (300K versus 150K), while for $n = 11$ the advantage increases to six times (600K versus 100K).

The reason is once again that in JPaxos the leader is the sole replica interacting with the clients, while in S-Paxos this work is shared among all replicas. Therefore, for the same number n of replicas, S-Paxos is capable of handling approximately n times more client connections. So, as n increases, S-Paxos can handle more client connections, where JPaxos is still limited to what the leader can handle.

When the network bandwidth is the bottleneck

For the tests focusing on the bandwidth, we use a request size of 1KB. With this request size, in all the clusters we used for the experiments, both JPaxos and S-Paxos are easily able to saturate the bandwidth available before hitting the limits of the CPU or of the network subsystem of the replicas. Even so, we chose the cluster with the fastest CPUs (Paradent, see Table 10.1), which further decreases the likelihood of CPU-related bottlenecks. Recall that the effective inter-node bandwidth in the clusters used for all the experiments is approximately $B = 930$Mbps.

Figure 10.6 shows the throughput in Mbps of JPaxos and S-Paxos with an increasing number of replicas. On the one hand, the maximum throughput of JPaxos is $B/(n-1)$, since the leader has to send requests to $n-1$ followers. This explains the throughput being approximately 400Mbps for $n = 3$, and dropping as additional replicas are added to the system. On the other hand S-Paxos orders around 800Mbps of application data (without TCP/IP headers and protocol headers) irrespective of the number of replicas. As expected, for S-Paxos the limiting factor

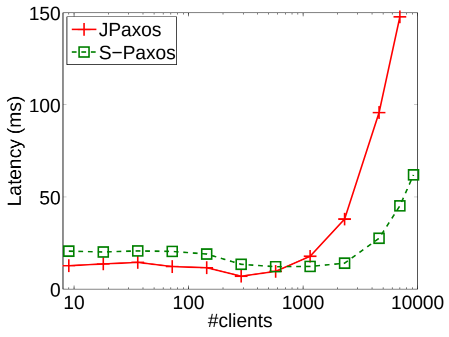


Figure 10.7: Response time with JPaxos and S-Paxos with different number of clients (log scale). Helios cluster.

is the amount of data any replica can receive. This difference between JPaxos and S-Paxos comes from the fact that in both protocols every request has to be received by every replica, but only in JPaxos the leader has to send every request.

10.4.2 Response time

In this section we study the response time of JPaxos and S-Paxos under different client loads. We run the experiments on the Helios cluster, using a request size of 20 bytes.

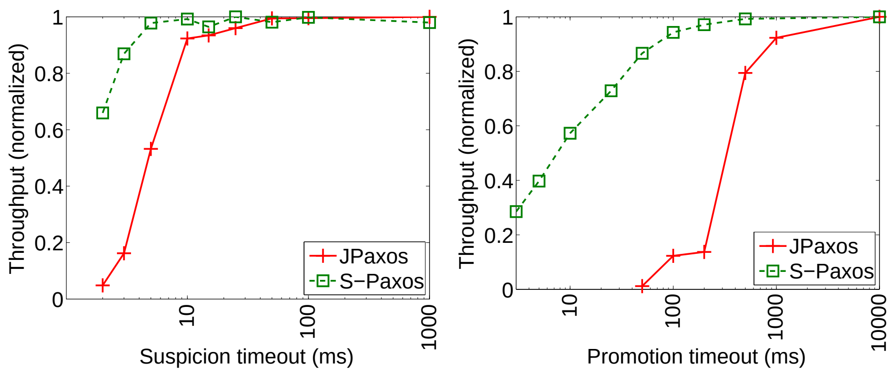
Figure 10.7 shows the results. Under small and medium load (up to 1'000 clients), S-Paxos has a slightly higher response time than JPaxos (approximately 20ms versus 13ms). However, as the client load increases, the situation is reversed, with the response time of JPaxos deteriorating faster than the one of S-Paxos (note the log-scale). These results are a good example of the typical trade-off between latency and throughput. To optimize for throughput, S-Paxos uses a longer request processing pipeline, with more levels of batching and more communication steps than JPaxos. Therefore, under low load it does not perform as well as JPaxos. But as the client load increases, JPaxos is not able to keep up with the demand, forcing the client requests to queue for a long time before being ordered and executed. S-Paxos, however, can cope with higher client loads, therefore avoiding the queuing delays and providing a much better response time.

10.4.3 Failures

For the evaluation of the failure case we return to the Paradent cluster and use a request size of 1KB (See Table 10.2).

Cost of View Change

We start by studying the cost of view change in two scenarios: view changes due to false suspicions and forced view changes. For the first scenario, we perform several runs where



(a) Suspicions

(b) Promotions

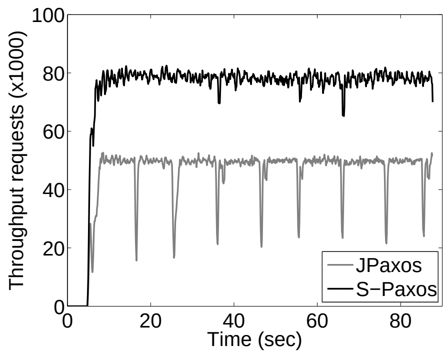
Figure 10.8: Performance with varying failure suspicion timeout and leader promotion timeout. $n = 3$

we lower the suspicion timeout of the failure detector, therefore increasing the frequency of false suspicions. For the second scenario, we rotate the role of leader periodically between replicas, by having the replica with id immediately greater than the one of the leader (modulo n) promote itself some time after the current leader was elected. The results are shown in Figure 10.8.

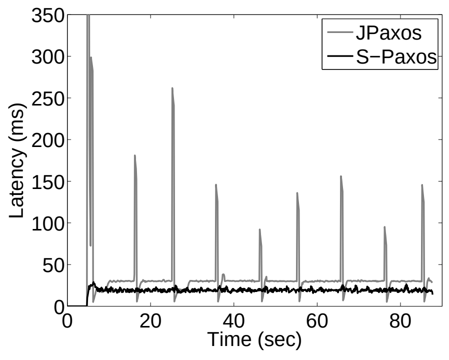
With timeouts as low as 100ms, false suspicions are rare so neither JPaxos nor S-Paxos show any drop in performance (Figure 10.8a). For smaller timeouts, false suspicions become common enough to affect performance. While S-Paxos has a graceful behavior, achieving a still respectable 60% of its peak performance even with suspicion timeouts as low as 3ms, the performance of JPaxos quickly collapses with timeouts smaller than 10ms. Since the failure detector used is the same in both cases, the number of false suspicions is roughly equivalent, which leaves the cost of view change itself as the cause of the difference. The results of the test with leader induced view changes (Figure 10.8b) confirm that this is indeed the case. With JPaxos, the throughput degrades quickly with view changes every 1 second, and collapses with view change intervals of 200ms. S-Paxos, however, tolerates view changes much better, achieving still 70% of its peak throughput with view change intervals of 50ms.

To better understand the effect of view change, we show in Figure 10.9 the time series of the response over a single run. For each x coordinate, the plot shows the average response time of the requests sent at time x. Note that this time includes retransmissions in case of connection or replica failures. Figure 10.9 shows clearly that view changes do not affect the performance of S-Paxos in any noticeable way. However, in JPaxos, each view change causes a temporary drop in performance.

S-Paxos performs better in these cases because of two main reasons. First, the amount of data exchanged by S-Paxos during view change is, on average, much smaller than in JPaxos; this is because Phase 1b messages of S-Paxos contain only ids, while in JPaxos they contain full

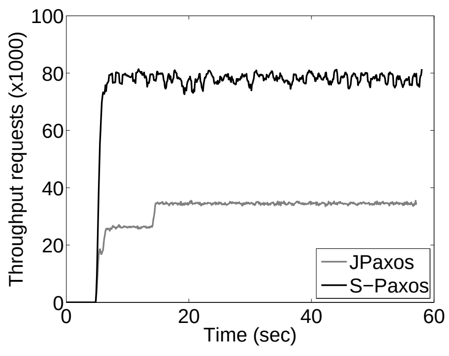


(a) Throughput

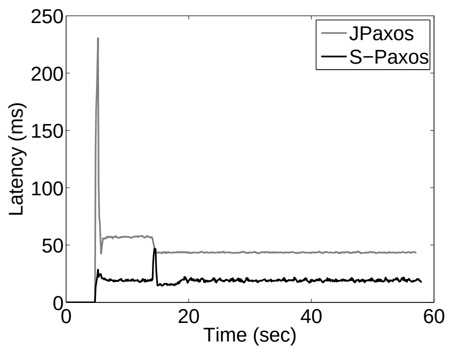


(b) Latency

Figure 10.9: Performance over time with leader promotion every 10 seconds. $n = 3$



(a) Throughput



(b) Response time

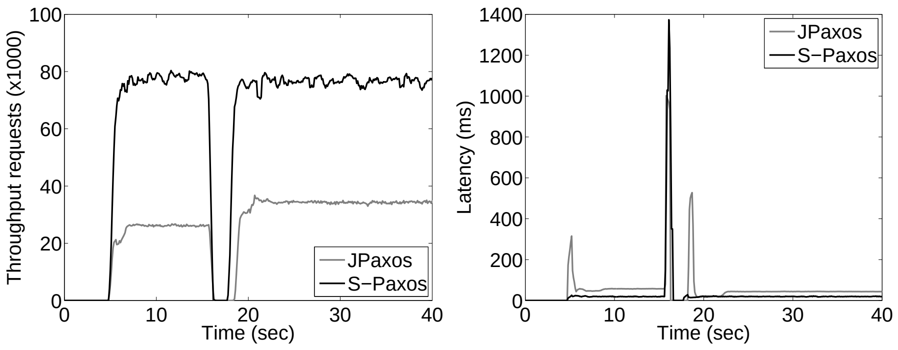
Figure 10.10: Crash of a follower. $n = 5$

batches. Second, in S-Paxos clients do not need to reconnect to a different replica, while in JPaxos all clients must disconnect from the previous leader and reconnect to the new one. This second problem can be minimized by allowing every replica to receive client connections and having replicas forward requests to the leader, like Zookeeper [JRS11] does. But even with this improvement, view changes will still be more expensive than in S-Paxos, because after a view change replicas may have to forward to the new leader some requests that were already forwarded to the previous leader, thus wasting resources. This is not necessary in S-Paxos.

Performance with crashes

We now look at the effect of a crash on the performance of the system. Each experiment lasts for 40 seconds, with a crash being triggered 15 seconds after startup. In these experiments, clients start sending requests to the replicas 5 seconds after startup.

The crash of a follower has little impact on the performance of the system (Figure 10.10). The throughput of S-Paxos is not at all affected, while the one of JPaxos increases because now



(a) Throughput (b) Response time

Figure 10.11: Crash of the leader. $n = 5$

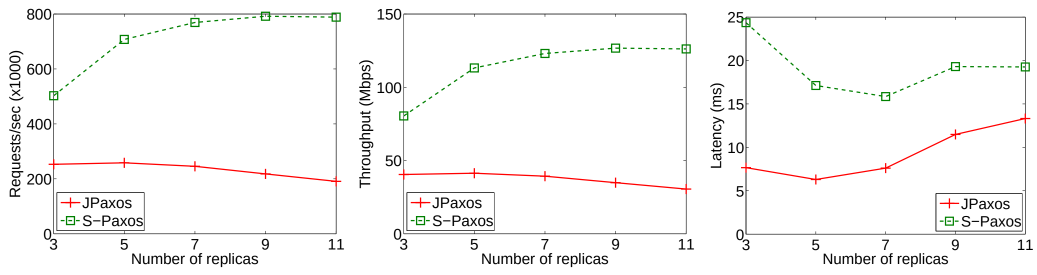
the leader is sending messages to one less follower, since the TCP connection to the crashed follower is closed.[6] For the same reason, the response time of JPaxos improves slightly after the crash. On the other hand, the response time of S-Paxos has a spike during the crash, then quickly returns to the same levels as before the crash. This spike is caused by the clients that were connected to the replica that crashed (approximately, 1/5 of all clients), which have to reconnect to another replica after a small random backoff delay (between 0.1 and 0.5 seconds). This is not the case for JPaxos because clients do not connect to followers.

When the leader crashes (Figure 10.11) both JPaxos and S-Paxos stop ordering requests until a new leader is elected.[7] The gap in the plots is dominated by the suspicion timeout of the failure detector (2 seconds). S-Paxos recovers faster than JPaxos because clients remain connected to the other 4 replicas and, therefore, as soon as a leader is elected these clients and the replicas resume normal operation, with only a small portion of the clients having to reconnect to some other replica. In JPaxos all clients have to reconnect to the new leader. Depending on the timeouts and reconnection strategy, this may happen only some time after a new leader is elected. The response time of both JPaxos and S-Paxos shows a spike just before the crash, corresponding to the requests that were sent before the crash of the leader, but ordered only after the recovery of the system. The second spike of JPaxos is, once again, a consequence of the clients having to connect directly to the leader, which introduces additional delays.

10.4.4 Achieving record throughput with S-Paxos

In the experiments presented above we studied how S-Paxos improves over a leader-centric version of Paxos in the presence of specific bottlenecks. Therefore, we have chosen the experimental settings to highlight the bottleneck in study and focused the analysis on the relative performance between JPaxos and S-Paxos. We now look instead at absolute numbers, as we investigate what is the maximum throughput that S-Paxos can achieve with recent

[6] Recall from Section 10.4.1 that with these experimental settings the bandwidth is the bottleneck.
[7] The response time for periods where no request is ordered is shown as 0.



(b) Throughput(Mbps)

(c) Response time (ms)

Figure 10.12: Throughput of JPaxos and S-Paxos with kernel 3.0.0-2 and hardware queues. Parapluie cluster.

(as of 2011) mid to high-end range commodity servers, which are representative of current data-centers. We focus on throughput for small requests, because there are already many SMR implementations capable of achieving the maximum bandwidth of a Gigabit LAN given large enough requests. To the best of our knowledge, the results achieved by S-Paxos presented in this section represent a new record among the publicly available results for small request sizes.

As S-Paxos is able to use multiple cores effectively, we perform the tests in this section in the Parapluie cluster as it is the one with the highest number of cores (2x12). We have used this cluster for the experiments in Section 10.4.1 (Figure 10.5), where the bottleneck was the network subsystem. Recall that the reason for this bottleneck, was that all interrupts from the network card were being handled by a single core. However, modern high-end network cards have a featured called “Receive Side Scaling” (RSS) that allow interrupt processing to be distributed among multiple cores, improving parallelism. These cards have multiple hardware receive queues, allowing them to map interrupts of these queues to different cores based on (IP and TCP) packet headers. A second advantage is that a good implementation of RSS can steer all the packets from a connection directly to the core that is executing the thread handling the connection, thereby decreasing the number of context switches and the amount of bus traffic. Although RSS depends on a feature that is not available in all network interface cards, a software implementation of the same principle is available in recent versions of the Linux kernel under the name “Receive Packet Steering” (RPS).[8]

The nodes in the Parapluie cluster are equipped with network cards that support RSS, however the version of the Linux kernel used in the experiments in Section 10.4.1 (2.6.32-2) does not support this feature. Therefore, for the experiments in this section, we use instead a more recent Linux kernel (3.0.0-2) which supports RSS. By using RSS, we can push even more the limits of S-Paxos, and show what kind of throughput is currently attainable when using a multi-threaded, decentralized implementation of Paxos coupled with a modern commodity hardware.

Figure 10.12 shows the results. As can be seen in Figure 10.12a, with RSS S-Paxos achieves a

[8]http://lwn.net/Articles/362339/

throughput of 800K requests per second (request size is 20 bytes). In this set of experiments we are serving more than 18K clients with average client latency always below 25 ms (Figure 10.12c). Furthermore, we are able to achieve a very high throughput even with a small number of replicas: 500K requests per second with $n = 3$ and 700K with $n = 5$. Having a good performance with 3 and 5 replicas is important are these are the most common case in practice [CGR07, JRS11].

Note that with such high throughput (around 800K requests per second with $n = 11$) again the CPU becomes bottleneck—more precisely, the sequential part of the protocol. Although the total CPU usage is around 600%, one core is busy at 100% with the thread responsible for executing the ordered requests (which in our case is just returning the same 8 bytes array) and performing some bookkeeping which must be done sequentially for every request. It turns out that at this level of performance on a 1.7Ghz machine one is left with about 3000 cycles per request, which is not much when considering that a cache miss can easily incur a delay of a few hundred cycles. By performing optimizations and using faster CPUs one might increase performance beyond this point, as we are still far from the limit of network bandwidth.

10.5 Related work

According to the classification of [DSU04], Paxos belongs to the group of fixed sequencer protocols because a distinguished process (the leader) is responsible for establishing the order of messages. Another example of a high-throughput fixed sequencer protocol is Ring-Paxos [MPSP10]. Ring-Paxos relies on efficient use of IP multicast to disseminate messages to servers. This, together with the fact that it executes consensus on ids (like S-Paxos), makes it a very efficient protocol in the case where the outgoing channel of the leader is the bottleneck (large requests). Another fixed sequencer protocol of interest is Zab [JRS11], a variant of the Paxos protocol used in primary-backup systems such as Zookeeper. While Zab distributes client communication to all replicas, it follows the conventional leader-centric design by having replicas forward requests to the leader.

All the above fixed sequencer protocols share the same short-coming: They burden the sequencer to the point it becomes the system's bottleneck. In this chapter, we have shown that by offloading the work from the leader, we can derive a fixed sequencer protocol that has good performance and that uniformly utilizes system resources at all replicas.

An alternative approach to prevent the leader from becoming the bottleneck is taken by Mencius [MJM08] and by the position paper [KJ10]. These protocols are based on the observation that rotating the role of the sequencer avoids contention on a single replica.[9]

In Mencius the sequence of consensus protocol instances is partitioned among all replicas with each taking the job of being the (initial) leader of an instance in a round-robin fashion.

[9]According to the classification [DSU04], the protocols that rotate the sequencer role among replicas are called *moving sequencer protocols*.

In order to exclude failed replicas from this schedule some reconfiguration is necessary in case of failures. Contrary to S-Paxos, which is designed for clustered environment, Mencius is designed with the goal of being an efficient SMR protocol for WAN environments. Since Mencius and S-Paxos are based on different high level concepts (moving sequencer vs. fixed sequencer), a detailed analysis would be necessary to understand trade-offs between the two approaches, which is out of the scope of this work. However, we expect the performance in the failure-free case to be comparable (both approaches balance the work among replicas). In the failure case we expect S-Paxos to be more efficient: the crash of any replica in Mencius stops progress, while in S-Paxos this is the case only if the leader replica fails (the crash of a follower does not affect system progress).

Kapritsos and Junqueira [KJ10] assign each consensus instance to different "virtual clusters", instead of to a different leader as in Mencius. Each virtual cluster consists of $2f+1$ virtual replicas which are mapped to overlapping sets of physical replicas. To reach maximal throughput each virtual cluster's leader should be on a different physical replica. This clearly necessitates some inter-cluster coordination in case of leader changes. Otherwise, one physical replica could become the bottleneck as it has the job of being leader in multiple virtual clusters. Although this approach can potentially achieve the same goal as S-Paxos, we cannot at the time of writing compare them in more detail because [KJ10] is the only publicly available description of the protocol and it does not provide many important details (in particular, how the system maintains a balanced distribution of work in the presence of failures).

In both solutions just discussed, multiple streams of decisions have to be merged into a single total order. This is done by taking one request from each process/virtual cluster in a round robin fashion. If one process/cluster does not have any request to order, it has to propose a special *skip* request, to allow other replicas to continue ordering requests. There is no need for such mechanism in S-Paxos.

While all aforementioned work is based on the ideas behind Paxos, there are other approaches that also balance the workload among all replicas that are not based on Paxos. One protocol that belongs to this group is LCR [GLPQ10], which arranges replicas along a logical ring and uses vector clocks for message ordering. LCR is a high-throughput protocol where work is equally divided among servers, thereby utilizing all available system resources. The drawback of the approach taken by LCR is that latency increases linearly with the number of processes in the ring; moreover, maintaining the ring structure adds overhead to the protocol. Although LCR has a slightly better bandwidth efficiency than S-Paxos for large requests (according to [MPSP10] it achieves 95% efficiency in a cluster setting), it requires *perfect failure detection*. Perfect failure detection implies stronger synchrony assumption than required by S-Paxos. While not limited by a leader being the bottleneck, it is unclear whether adding replicas increases the throughput of LCR.

State partitioning [GHOS96] is another technique commonly used to achieve scalability. Recently, it has been considered in the context of state machine replication [MPP11, MPP12].

In [MPP11], Marandi et al. show that state partitioning leads to improvements in both throughput and response time for a replicated B-tree service. However, as they argue, perfect state partition is often not possible. One can gain performance even with imperfect state partitioning, as it lifts the requirement that every request needs to be received by every replica. In [MPP11] the authors use Ring-Paxos as an ordering protocol. However, as they point out in [MPP12], when the number of partitions increases (which is required for good throughput), Ring-Paxos becomes the bottleneck due to its leader-centric nature. In order to avoid Ring-Paxos being the bottleneck, [MPP12] introduces groups, and let a single instance of Ring-Paxos be responsible for ordering messages within a single group. Then each learner subscribes only to the groups it wants to receive messages from. This approach, which is conceptually similar to [MJM08, KJ10], requires a mechanism for merging requests from different groups and a skipping mechanism for inter-group coordination. Note that S-Paxos can be easily used in this context by configuring the dissemination layer to send requests only to interested learners, while the ordering layer would deliver the order to all learners. The benefits would be using a single cluster instead of several, and the fact that there is no need for additional mechanisms (merging requests, skipping consensus instances, reconfiguration).

The benefit of running consensus on ids in the context of Atomic Broadcast is explored in [ES06]. Their approach requires modifying the consensus algorithm, so that it only decides an id when the corresponding message is stable. Our work differs in two main aspects. First, we consider a different problem, that is, balancing the load in State Machine Replication using Paxos. Second, we ensure stability of the request before initiating ordering of the id, and thereby avoid the need to modify the ordering protocol.

10.6 Conclusion

In this chapter we have shown that leader-based protocols do not have to be leader-centric, *i.e.*, they do not need to have the leader responsible for a significantly higher workload than the followers. We presented S-Paxos, a variant of the Paxos protocol that offloads work from the leader by delegating it to the other replicas. This allows S-Paxos to benefit fully from the resources of all replicas and to increase its performance with the number of replicas. We implemented a prototype of S-Paxos and evaluated its performance in different cluster settings. In many common settings, S-Paxos achieves between 2 and 3 times the request throughput of a leader-centric Paxos implementation for $n = 3$. Furthermore, the throughput of S-Paxos improves when additional replicas are added to the system, while the performance of most SMR implementations drops. With eleven replicas, S-Paxos achieves a throughput up to seven times higher than a leader-centric variant of Paxos. Therefore, with S-Paxos there is no need to make a trade-off between fault tolerance and performance, contrary to what must be done in most SMR implementations.

The source code of S-Paxos is available at https://github.com/nfsantos/S-Paxos.

11 Conclusion

11.1 Research assessment

In this thesis we have investigated the problem of state machine replication, focusing on understanding its performance characteristics and proposing techniques to achieve high-throughput. Using the popular Paxos protocol as a basis for our work, we have looked at the problem from different perspectives and at different components of a replicated state machine.

In the first part, we performed an analytical analysis of several consensus algorithms, including a version of the Paxos protocol expressed on a round model. We then looked at ways of improving the performance of round models. Finally, we proposed a leader-election algorithm that takes network topology into consideration to choose a well-connected leader. This can significantly improve the performance of leader-based algorithms, like Paxos, in systems with asymmetric latencies.

In the second part, we proposed several techniques to achieve high-throughput in state machine replication. First we looked at the optimizations of batching and pipelining, studying their interaction and showing how to parametrize them in a way to maximize the throughput in a given system. We then looked at how to parallelize the implementation of state state machine replication, in order to make effective use of modern multi-core CPUs. This is especially important to cope with the workload generated when dealing with a large number of clients and small to medium request sizes. The last chapter proposed a new variant of Paxos, S-Paxos, which overcomes the traditional bottleneck at the leader process, by distributing the workload evenly across all replicas.

We now look in more detail at each of these contributions and discuss future work.

Quantitative Analysis of Consensus Algorithms In this chapter we have shown that the Heard-Of round model provides a suitable framework for comparing the performance of consensus algorithms quantitatively, making this task easier than in the failure detector model or the partially synchronous model. We have shown how to express two classes of algorithms in the HO model, one of them a variant of Paxos, and derived formulas for several important performance metrics. These include the duration of a good period required to solve an instance of consensus when the system switches from a bad to a good period, the time required to solve additional instances while inside a good period, and the message complexity.

The results show that the performance of round-based algorithms largely depends on the implementation of rounds. The results also show that the number of rounds of an algorithm is not always a good metric for the performance of an algorithm. Finally we can observe trade-offs in resilience, minimum duration of a good period for a decision, decision time during long good periods, and message complexity.

Swift Algorithms for Repeated Consensus The traditional perception that round algorithms are slow is a side-effect of the way round models have been traditionally implemented, where for every round some process has to expire a timeout before the system advances to the next round. This contrasts with algorithms for the partially synchronous and the failure detector model, where progress is made whenever enough messages are received.

We have formalized this difference between algorithms with the notion of a *swift algorithm*. Roughly speaking, an algorithm is swift if eventually it progresses at the speed of the system, *i.e.*, by message reception, without expiring timeouts. Using this notion, we have shown that failure detector algorithms are naturally swift, while round algorithms are naturally non-swift.

We have then described a round implementation that is swift, thereby showing that there is no fundamental performance gap between the round model and the models that are commonly used in practice, *i.e.*, the partially synchronous and the failure detector system models. Furthermore, we have validated our round implementation experimentally, by comparing two versions of the OTR consensus algorithm, one expressed in the round model and the other in the failure detector model. The results show that their performance is similar.

Latency-aware Leader Election Most of the algorithms used for state machine replication are leader-based. When the latencies between the processes are not symmetric, the choice of the leader has a large impact on the performance of the system: choosing a well-connected leader, that is, one that can communicate quickly with a majority of processes, can significantly improve the performance of the system. Based on this insight, we have developed a leader election algorithm that elects a process that is not only correct (as captured by the Ω failure detector abstraction), but also optimal with respect to the transmission delays to its peers. We gave the definitions of this problem and a suitable model, thus allowing us to make a detailed analytical analysis of the problem, stating precisely what being "optimal" means. Finally, we present the results of an experimental study comparing our latency-aware leader election algorithm with traditional, non-latency aware algorithms, showing that, in networks with asymmetric latencies, our algorithm provides a remarkable improvement in the performance of an implementation of state machine replication based on Paxos.

Tuning Paxos for High-Throughput with Batching and Pipelining The batching and pipelining optimizations are widely used by Paxos implementations, as they can substantially improve performance. However, to extract the maximum from these two optimizations, they must be correctly configured, taking in consideration many system parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. As we have shown using experiments, this is not an easy task to do in an ad-hoc fashion, as these optimizations have a complex behavior, especially when they are used in combination.

We addressed this problem with an analytical model of the performance of Paxos, which can be used to determine good configuration values for batching and pipelining optimizations, in the sense of achieving high-throughput. We validated the model by presenting the results of experiments where we investigate the interaction of these two optimizations both in LAN and WAN environments: The experimental results are close to the predictions of the model.

Furthermore, both the model and the experiments give useful insights into the relative effectiveness of batching and pipelining. They show that although batching by itself provides the largest gains in all scenarios, in most cases combining it with pipelining provides a very significant additional increase in throughput, not only on high-latency network but also in low-latency networks.

Multi-core scalable State Machine Replication Recently there has been a growing demand for state machine replication capable of achieving very high-throughputs, motivated mainly by the proliferation of online services that require both fault-tolerance and the capability to handle a large number of clients. Traditional implementations of state machine replication have difficulties in coping with these demands, because they are based on a event-driven, single threaded design whose performance is bounded to the single thread performance of the nodes. We have addressed this problem, by revisiting the traditional architecture and enabling it to take advantage of modern multi-core CPUs. The revised architecture is based on several good practices of concurrent programming, including staged execution, workload partitioning, actors, and non-blocking data structures. Our experimental evaluation has shown that with a workload consisting of small requests, we achieve a six times improvement in throughput using eight cores. And more generally, in all our experiments we have consistently reached the limits of the network subsystem by using up to twelve cores, and do not observe any degradation when using up to 24 cores.

Scaling State Machine Replication - Eliminating the leader bottleneck We have argued that most leader-based protocols like Paxos are also leader-centric, in the sense of the leader doing a high fraction of the work, which causes an imbalance where the leader becomes the system bottleneck even when there are plenty of resources available at the followers. However, we have shown that this does not have to be the case, by developing S-Paxos, a variant of the Paxos protocol that offloads the leader by delegating to the other replicas some of the work typically done by the leader. S-Paxos key ideas are to reduce to the bare minimum the work of the leader, by having all replicas doing client interaction and request dissemination, leaving to the leader the relatively minor task of establishing order on request ids.

We implemented a prototype of S-Paxos and evaluated its performance in different cluster settings. S-Paxos achieves between 2 and 3 times the request throughput of a leader-centric Paxos implementation for $n = 3$. Furthermore, for the most common configurations of $n = 3$ to 7, the throughput of S-Paxos improves when additional replicas are added to the system, while the performance of most SMR implementations drops. This shows an interesting side-effect, which is that with S-Paxos there is no need to make a trade-off between fault tolerance and performance, contrary to what must be done in most SMR implementations.

11.2 Open questions and future research directions

Latency-aware Leader Election A latency-aware leader election algorithm will by necessity cause leader changes even when the current leader is correct. As these changes disrupt the protocol that is using the leader oracle, typically causing a significant temporary slowdown, they should be avoided as much as possible. That is, there is a trade-off between leader optimality and leader stability. The parameter ϵ in our model controls this trade-off, but our work left open the question of how to set ϵ. An interesting future work is to understand how to set ϵ to reach certain objectives defined in terms of frequency of leader changes and proximity from optimality, given an underlying network where latencies may change.

Another interesting extension is to consider a model where the link latencies follow a probabilistic distribution instead of eventually stabilizing to a fixed (but unknown) value. A probabilistic distribution is closer to the real behavior exhibited by networks, especially the ones on busy data centers or spanning the open Internet, and thus may provide more accurate results.

Tuning Paxos for High-Throughput with Batching and Pipelining The model we present has some limitations, the most significant being that it assumes single-core CPUs. Although the current version of the model can also be used on multi-core CPUs by adjusting the parameters of the functions modeling the processing time to reflect the higher speed of multi-core CPUs, the results will likely not be as accurate as with a model developed specifically for multi-core CPUs. Thus, it would be interesting to explore how to extend the current model to take in consideration multi-core CPUs.

Batching and pipelining are key elements in controlling the trade-off between response time and throughput: large batches and small window sizes favor throughput, while small batches and large window sizes favor response time. A static configuration will necessarily be optimized for one of these metrics. This suggests a possible follow-up to our work, where these parameters are set dynamically, based on the current state of the system. A possible policy is to aim for low response time when the system is lightly loaded, while aiming for high-throughput at the cost of response time if the system is heavily loaded.

Multi-core scalable State Machine Replication In Section 9.6.3 we have proposed several incremental improvements that can be applied to our architecture.

Another direction for future work is to validate the applicability of the architecture by implementing it on a mature, production-quality implementation of state machine replication, for instance ZooKeeper, and then comparing with the unmodified version. Although JPaxos is fairly feature-rich, it is only a research prototype and may be missing some of the features found on fully-functional products. We believe that the threading architecture could be extended to support any additional functionality, but this should be validated.

Another possible research direction is to explore languages like Scala, which have a more extensive support for concurrency than Java and than most other mainstream languages. Therefore, they may provide a more natural environment to write a high-performance Paxos implementation.

Scaling State Machine Replication - Eliminating the leader bottleneck With S-Paxos, we have considered only the case where stable storage is not used. Although this is a valid use case for many deployments where only a small subset of the replicas are expected to crash at any given time so that a recovery mechanism can restore the resilience, in many other cases it is necessary to tolerate catastrophic failures, *i.e.*, the failure of potentially all replicas. Therefore, it would be interesting to explore the implications of stable storage on the mechanisms proposed. In principle the same principles that were used to offload work from the leader, can also be applied to distribute stable storage accesses across replicas, therefore reducing the burden on the leader.

We have evaluated S-Paxos on a cluster environment, as this is the most common context where State Machine Replication is used. However, S-Paxos will also work on a WAN environment, as it does not rely on any cluster-specific functionality. Many of the techniques that allow S-Paxos to perform so well should work equally well on a WAN, but it would be worth to investigate this question with an experimental study, and eventually revise the architecture to be more suitable for a WAN.

A Proofs for Chapter 4

Lemma 4.1. *In a good period, a time interval of length $\tau_C = \beta\tau_L$, measured by some process p, corresponds to a real time interval of length in $[\tau_L, \tau_U]$, with $\tau_U = \frac{\beta}{\alpha}\tau_L$.*

Proof. Let $[t_1, t_2]$ be a real time interval. Then, $C_p(t_2) - C_p(t_1) = \tau_C$ is the duration of the interval as measured by p, and $t_2 - t_1$ the real time duration. From equation (4.1), we have $t_2 - t_1 \geq \left[C_p(t_2) - C_p(t_1)\right]/\beta = \tau_L$, and $t_2 - t_1 \leq \left[C_p(t_2) - C_p(t_1)\right]/\alpha = \frac{\beta}{\alpha}\tau_L$, which proves the result. □

Lemma 4.3 (Timeout τ_{C1})**.** *Consider Parametrization 4.3 with $\tau_{C1} = [3\Delta + 3n\Phi]\,\beta + \tau_{C4}\left(\frac{\beta}{\alpha} - 1\right)$. Assume every process starts round $4(\phi - 1)$ in a $\left(\frac{n+1}{2}\right)$-good period and phase ϕ has a unique coordinator c. Then (i) c hears from a majority of processes in round $4\phi - 3$, and (ii) all processes in Π_0 hear from c in round $4\phi - 2$.*

The proof is illustrated in Figure A.1, between t_{s4} and t_{e1}.

Proof. We start with (i), and compute first the latest time at which all round $4\phi - 3$ messages are ready to be received.

Assume p_1 is the first process to execute a send step for round $4(\phi - 1)$ at t_{s4}. In the following, we use p_2 to represent the process that takes the longest to send a round $4\phi - 3$ message. By $t_{s4} + (n-2)\Phi$, p_1 finishes sending its round $4(\phi - 1)$ messages. $\Delta + n\Phi$ time later, p_2 receives the message from p_1 and starts the timeout τ_{C4}. This timeout expires at most $\tau_{U4} + \Phi$ later, which means that p_2 starts round $4\phi - 3$ by $t_{s4} + \Delta + (2n-1)\Phi + \tau_{U4}$. Then, at most Φ time later, p_2 has sent its message to the coordinator (p_1 in Figure A.1), and Δ time later this message is ready for reception. Thus, by time $t_0 = t_{s4} + 2\Delta + 2n\Phi + \tau_{U4}$ the messages from all processes in Π_0 are ready to be received by the coordinator. The coordinator (p_1 in Figure A.1) enters round $4\phi - 3$ not before $t_{s4} + \tau_{L4}$, thus its timeout for round $4\phi - 3$ will not expire before $t_{s4} + \tau_{L4} + \tau_{L1}$, which is after t_0. In addition, since we are in a $\left(\frac{n+1}{2}\right)$-good period, there is at least a majority of processes in Π_0 that has sent a round $4\phi - 3$ message; so the coordinator receives a majority of messages. This proves (i).

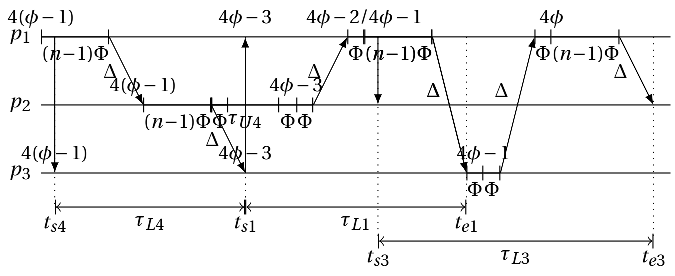


Figure A.1: Synchronization by a Coordinator: Lemmas 4.3 –4.5.

To show (ii), recall that by t_0 all round $4\phi-3$ messages are ready to be received by the coordinator p_1. One receive step later the coordinator advances to round $4\phi-2$, and $n-1$ send steps later it sends all its round $4\phi-2$ messages. The message is ready for reception at all processes (p_3 in Figure A.1) by $t_{e1} = t_0 + n\Phi + \Delta = t_{s4} + 3\Delta + 3n\Phi + \tau_{U4}$. We now show that no timeout for round $4\phi-3$ expires before t_{e1}. Since timeouts are started after sending, by definition, t_{s4} is also the earliest time a process p_3 in Π_0 starts the timeout for round $4(\phi-1)$.[1] By Lemma 4.1, p_3 then waits at least until $t_{s1} = t_{s4} + \tau_{L4}$ before sending its round $4\phi-3$ message to the coordinator and start the timeout τ_{C1}. The timeout for round $4\phi-3$ expires at p_3 the earliest at $t_{s4} + \tau_{L4} + \tau_{L1}$, which is equal to t_{e1}. So all processes in Π_0 hear from the coordinator in round $4\phi-2$, which completes the proof of (ii). □

Lemma 4.4 (Timeout τ_{C3})**.** *Consider Parametrization 4.3 with the timeout* $\tau_{C3} = [3\Delta + 2n\Phi]\beta$. *Assume every process starts round* $4(\phi-1)$ *in a* $\left(\frac{n+1}{2}\right)$*-good period, and* ϕ *has a unique coordinator c. Then (i) c hears from a majority of processes in round* $4\phi-1$, *and (ii) all processes in* Π_0 *hear from c in round* 4ϕ.

Proof. Since round $4(\phi-1)$ started in a $\left(\frac{n+1}{2}\right)$-good period, by Lemma 4.3, the coordinator receives all the round $4\phi-3$ messages, and sends its round $4\phi-2$ messages.

We start with (i), and compute first the earliest time by which all round $4\phi-1$ messages are ready to be received by the coordinator. Let us assume that the coordinator sends its first round $4\phi-2$ message at time t_{s3} to a process p_2. The coordinator finishes sending its round $4\phi-2$ messages at latest by $t_{s3} + (n-2)\Phi$; $\Delta + 2\Phi$ later all processes have received this message, advanced to round $4\phi-1$ and sent a round $4\phi-1$ message back to the coordinator. By time $t_0 = t_{s3} + n\Phi + 2\Delta$ these message are ready for reception by the coordinator. The coordinator starts its timeout for round $4\phi-1$ not before t_{s3}, *i.e.*, the timeout expires not before $t_{s3} + \tau_{L3}$, which is after t_0. Thus the coordinator receives a majority of round $4\phi-1$ messages, which proves (i).

We prove now (ii), and start our considerations at time t_0 (see above) when the coordinator is ready to receive a majority of round $4\phi-1$ messages. One receive step later the coordinator

[1] Note that our reasoning is different here to the one in the proof of Lemma 4.2, where we start our considerations with the start of the timeout, while here we start with the first message that is sent.

advances to round 4ϕ, and $n-1$ steps later it finishes sending its round 4ϕ messages. This message is ready for reception at any process by time $t_{e3} = t_{s3} + 3\Delta + 2n\Phi$. We show now that no timeout for round 4ϕ expires before t_{e3}. The earliest p_2 can receive the round $4\phi - 2$ message from the coordinator p_1 is at time t_{s3}. Upon receiving the coordinator's message, p_2 enters round $4\phi - 2$ (line 1 of *NextRound*), advances to round $4\phi - 1$ (line 4 of *NextRound*), and sends a message to the coordinator. So the earliest time at which p_2 may start round $4\phi - 1$ is t_{s3}, and the earliest time that any process in Π_0 expires the timeout for round $4\phi - 1$ is $t_{s3} + \tau_{L3} = t_{s3} + 3\Delta + 2n\Phi$. This time is not before t_{e3}, which proves (ii). □

Lemma 4.5 (Timeout τ_{C4})**.** *Consider Parametrization 4.3 with the timeout* $\tau_{C4} = [2\Delta + (2n-3)\Phi]\beta$*. Assume that a k-good period, $k \geq 1$, starts at time t_g and that round r_0 is the highest round started by any process in Π_0 by time t_g. Then every round $4\phi > r_0$ started after time t_g is uniform with non-zero cardinality.*

Proof. We show that every process in Π_0 receives a message from all processes in a non-empty subset of Π_0 and not from any other process.

By the same argument as in the proof of Lemma 4.2, item (ii), we can conclude that in round $4\phi > r_0$ no process in Π_0 receives a message from processes in $\Pi \setminus \Pi_0$.

We consider now messages received from processes in Π_0. Consider Figure A.1 while renaming $4(\phi-1)$ to 4ϕ. Let t_{s4} be the earliest time some process $p_1 \in \Pi_0$ sends a round $4\phi > r_0$ message. Since timeouts are started after sending, by definition, t_{s4} is also the earliest that a process can start the timeout for this round. The timeout $\tau_{C4} = [2\Delta + (2n-3)\Phi]\beta$, together with Lemma 4.1, ensures that no timeout of length τ_{C4} started at time t_{s4} expires before $t_{s1} = t_{s4} + 2\Delta + (2n-3)\Phi$.

We will now show that all round 4ϕ messages are received before t_{s1}. Process p_1 finishes his send steps by time $t_{s4} + (n-2)\Phi$. By time $t_{s4} + (n-2)\Phi + \Delta$ the messages sent by p_1 are ready to be received by all other processes. This is the latest time some process, say p_2, may start the send steps for round 4ϕ, since if p_2 receives the round 4ϕ message while on a lower round, it will skip the send steps of this round (line 3 of *Dest*). Assume p_2 started its send steps at $t_{s4} + \Delta + (n-2)\Phi$. Its messages will be ready for reception at all processes $(n-1)\Phi + \Delta$ later, *i.e.*, by $t_{s4} + 2\Delta + (2n-3)\Phi$ which, as seen previously, is before any timeout expires. So when the timeout expires, all messages sent in round 4ϕ are either received or ready to be received; thus in round 4ϕ every process in Π_0 receives all messages sent in round 4ϕ, hence the round is uniform.

Trivially, at least one message is sent in round 4ϕ, which shows the non-empty cardinality. □

Theorem 4.2. *In any good period of length*

$$y\left[\left(2\Delta+(2n-3)\Phi\right)\frac{\beta}{\alpha}+4\Delta+(2n+5)\Phi\right]+$$
$$+\left(2\Delta+(2n-3)\Phi\right)\frac{\beta^2}{\alpha^2}+\left(5\Delta+(5n-3)\Phi\right)\frac{\beta}{\alpha}+\Delta+(3n+1)\Phi$$

the generic algorithm with Parametrization 4.3 ensures y consecutive phases ϕ that fulfill $\mathscr{P}_{lv4}(\phi)$.

Proof. We first compute the time by which all processes have entered the first round of a good phase, which we will call the *initialization* period. The duration a phase ϕ is measured as the time since the last process enters round $4\phi-3$ until the last process ends the round 4ϕ.

Assume a good period starts at time t_g (see Figure 4.6). We start by computing the latest time a process p will enter a new round $4(\phi-1)$ after t_g.

After t_g, there will be a process p_1 that will either start (i) a new round $4(\phi-2)$, or (ii) a new round $4(\phi-1)-2$ before any other process which depends on p_1's round at t_g. If p_1 is in round two or three of a phase, then the case (i) happens first, the latest by $t_{s4a}=t_g+n\Phi+\tau_{L3}\frac{\beta}{\alpha}$, which is the time required to complete rounds two and three of a phase. If p_1 is in round four or one of a phase, then case (ii) happens first. The time by which it happens depends on whether p_1 advances to round $4(\phi-1)-2$ by timeout or by receiving messages.

Process** p_1 **advances by timeout **(line 3 of** *NextRound*) Then p_1 will skip rounds $4(\phi-1)-2$ and $4(\phi-1)-1$. If p_1 is a coordinator, then by line 2 of *SkipRound* it will skip the round without executing send steps. Otherwise, the *Dest* function ensures that p_1 will not send to anyone and line 4 of *NextRound* makes it advances immediately to round $4(\phi-1)-1$. In either case, p_1 will skip round $4(\phi-1)-1$ by line 3 of *SkipRound*, since no round $4(\phi-1)-2$ was sent at this time. Thus, p_1 starts round $4(\phi-1)$ immediately, proposing itself as coordinator because of line 3 of *Dest*. This happens the latest by $t_{s4b}=t_g+(n+1)\Phi+(\tau_{L4}+\tau_{L1})\frac{\beta}{\alpha}$, which is the maximum time required to complete rounds $4(\phi-2)$ and $4(\phi-1)-3$.

Process** p_1 **advances by receiving messages This may happen either by receiving a message from round $4(\phi-1)-2$ (line 1 of *NextRound*) or by receiving a majority of messages in round $4(\phi-1)-3$ (line 3 of *NextRound*). The former could not have happened in this situation, because p_1 is the first process to enter round $4(\phi-1)-2$, so no round $4(\phi-1)-2$ message was sent by this time. In the latter case, p_1 may be the coordinator. If it is not, then it will skip round $4(\phi-1)-2$ and $4(\phi-1)-3$ for the same reasons as if p_1 had advanced by timeout.

Otherwise, if p_1 is the coordinator, let $\mathscr{M}_0$ be the set of processes from which p_1 received round $4(\phi-1)-3$ messages. Processes in $\mathscr{M}_0$ must have started round $4(\phi-2)$ before t_g, since by assumption no process started a new round $4(\phi-1)$ between t_g and p_1. Thus, the latest by $t_g + n\Phi + \tau_{L4}\frac{\beta}{\alpha}$ they are in round $4(\phi-1)-3$, and $\Delta+\Phi$ time later the message was received by the coordinator. Therefore, the latest by $t_g + \Delta + (n+2)\Phi + \tau_{L4}\frac{\beta}{\alpha}$ the coordinator has received a majority of messages and $n\Phi + \Delta$ later, the round $4(\phi-1)-2$ messages of p_1 are ready for reception at all processes. If all alive processes are still in round $4(\phi-1)-2$ or lower when the messages arrive, and if they have as coordinator p_1 (recall that the phase is not necessarily well coordinated), then they will receive the message, advance to round $4(\phi-1)-1$, and reply to p_1. Since we are in a good period, the remaining rounds of the phase will complete successfully and the phase will satisfy the predicate. Since this case does not correspond to the longest it takes to satisfy the predicate, we will not consider it. If there is a process q that doesn't have p_1 as coordinator, or that ended round $4(\phi-1)-3$ before receiving p_1's message, then by lines 2 and 3 of *SkipRound*, q will advance to round $4(\phi-1)$ and send a message to all, by line 2 of *ElectCoord*. This happens the latest at $t_{s4c} = t_g + 2\Delta + (2n+2)\Phi + \tau_{L4}\frac{\beta}{\alpha}$, which is the time by which p_1's messages are received by all processes.

Considering all the cases above, and taking the maximum of t_{s4a}, t_{s4b} and t_{s4c} we can conclude that by time $t_g + (n+2)\Phi + (\tau_{L4} + \tau_{L1})\frac{\beta}{\alpha}$ there is a process p that started a new round $4(\phi-1)$. $(n-1)\Phi + \Delta$ time later, every process has a round $4(\phi-1)$ message ready to be received and n steps later, all processes will have received it and started round $4(\phi-1)$. $\tau_{U4} + \Phi$ later all processes will have entered round $4\phi-3$, which marks the end of the initialization. This happens by $t_e = t_g + \Delta + (3n+1)\Phi + (2\tau_{L4} + \tau_{L1})\frac{\beta}{\alpha}$.

We can use Corollary 4.2 to show that $\mathscr{P}_{lv4}(\phi)$ will be true since round $4(\phi-1)$ starts in a good period. Using the timeouts specified in Corollary 4.2, the algorithm is able to complete y phases before the end of the good period specified in this Theorem, then by Corollary 4.2 all those phases will satisfy $\mathscr{P}_{lv4}(\phi)$, which proves this Theorem.

The duration of the good phase can be computed as follows starting from t_e. Rounds $4\phi-3$ to $4\phi-1$ are message driven, consisting of two participants-coordinator-participants message exchange, each taking $2\Delta + (n+2)\Phi$. Therefore, at time $t_e + 4\Delta + (2n+4)\Phi$ all processes have advanced to round 4ϕ, and $\tau_{L4}\frac{\beta}{\alpha} + \Phi$ later have completed round 4ϕ. Therefore, the first phase ends at $t_e + 4\Delta + (2n+5)\Phi + \tau_{L4}\frac{\beta}{\alpha}$. By applying the same reasoning, we can show that the following phases will take the same time. Therefore, y good phases will complete by $t_e + y[4\Delta + (2n+5)\Phi + \tau_{L4}\frac{\beta}{\alpha}]$. By expanding and simplifying this expression, we show that the duration of the good period specified in this theorem is enough to complete y phases of $\mathscr{P}_{lv4}$. □

B Proofs for Chapter 5

Lemma 5.1. *Consider Algorithm 5.1 with $TO \geq 2\Delta + (2n+5)$ and $n > 3f$. Let R be a Δ-partially synchronous run. Let t_r be the time the first process starts a new round r after GST. Then round r is uniform.*

Proof. (See Figure B.1 for illustration.) Let p be the first process to finish the input and send steps for round r, at time t_s ($t_s \leq t_r + (n+1)$). We show that (i) all round r messages from all alive processes are ready for reception[1] by time $t_s + TO$, and (ii) no process expires its round r timeout before $t_s + TO$. This implies that round r is uniform.

(i) By time $t_s + \Delta$ the round r message from p is ready for reception at all processes. Every process q will make a receive step at most $(n+2)$ time later (if at time $t_s + \Delta$ q was on a output step of a round $r' < r-1$, then it must make one input step and n send steps before the next receive step). After receiving the round r message, every process performs an output step for its current round, advances to round r, performs one input and n send steps. Therefore, by time $t_s + \Delta + (2n+5)$, all processes have finished sending their round r messages, and Δ time later, by time, $t_s + 2\Delta + (2n+5) = t_s + TO$, all round r messages are ready for reception at all alive processes. Note that this time is still in the good period, since $t_s + TO = t_r + TO + (n+1)$.

(ii) Since all processes start the timeout for round r after p, no process will expire its timeout before $t_s + TO$. Additionally, no process advances to a higher round by receiving a higher round message because for a new round to start, the timeout of round r of some process has to expire. □

Lemma 5.2. *Consider Algorithm 5.1. Let R be a Δ-partially synchronous run. Then by time $GST + TO + (n+2)$ at least one process has started a new round r_0.*

Proof. Let p be the process with the highest round number r among all processes. Then the lemma is fulfilled, if p is at least in round $r+1$ by the given time. However, in a good period,

[1] We call a message ready for reception if it must be received with the next receive step of the receiver process.

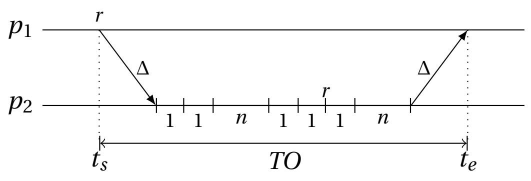


Figure B.1: Non-swift rounds: Lemma 5.1

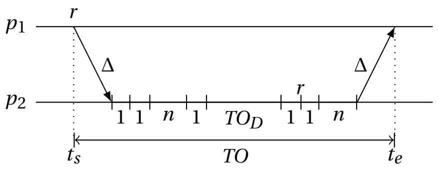


Figure B.2: Swift-rounds: Lemma 5.6

p can be in round r at most for $TO + (n+2)$ time, the timeout and the time for an input, an output, and n send steps. □

Lemma 5.6 (Timeouts TO and TO_D)**.** *Consider Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$. Let R be a Δ-partially synchronous run, and t_r the time the first process starts a new round r after GST, such that all processes have the same Alive set after t_r. Then round r is uniform.*

Proof. Let p_1 be the first process to finish sending its round r messages at time $t_s = t_r + (n+1)$, and starting the timeout for round r (see Figure B.2). These messages are ready for reception at most Δ time later, at $t_s + \Delta$. These messages are received in the next receive step, which occurs the latest after $(n+2)$ steps (an output step followed by an input step, and n send steps). This is because some process (p_2 in Figure B.2) might be just started executing an output step for some round $r' < r$. Therefore, p_1's message is received by all processes the latest at time $t_1 = t_s + \Delta + (n+3)$. Any process that receives this message in round $r-1$ for the first time, might set its timeout to $t_1 + TO_D < TO$ (see lines 21-22). And start round r the latest by time $t_1 + TO_D + 1$, after an output step for round $r-1$. By time $t_2 = t_1 + TO_D + 1 + 1 + n$, any process (including p_2) has performed an input step and n send steps for round r. This message is ready for reception the latest at time $t_e = t_2 + \Delta = t_s + TO_D + 2\Delta + (2n+5)$. The timeout $TO = TO_D + 2\Delta + (2n+5)$ ensures that no timeout started at time t_s expires before t_e (see line 16). So when the timeout expires, all messages for round r are either received or ready to be received. Before, calling the transition function for round r (in line 23), a receive step is performed (in line 11); thus every process in round r receives a message from every process, and round r is uniform.

Note that no process in round r can receive a message from round $> r+1$. We prove this by contradiction. Let p be a process in round r that receives a message from round $r+2$. This means that there is some process q that sent round $r+2$ messages. This requires that either (i)

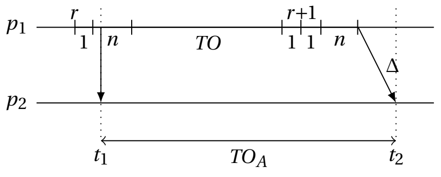


Figure B.3: Swift-rounds: Lemma 5.7

q receives all round $r+1$ messages, including p's message, which is not possible; or (ii) the timeout for round $r+1$ expires, which is not possible inside the given interval.

If a process ends round r at time t before the end of timeout *TO*, because it has received all round r messages from its *Alive* set (line 15), any other process does so the latest by time $t+(n-1)+\Delta$. From lines 21-22, a process in round r that receives a message from round $r+1$ for the first time, waits until $t+TO_D$ time before starting round $r+1$, which is enough to receive all round r messages. By the assumption, since all processes have the same actual *Alive* set in the given interval, round r is also uniform in this case. □

Lemma 5.7 (Timeout TO_A)**.** *Consider Algorithm 5.4 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)$, $TO \geq TO_D + 2\Delta + (2n+5)$, and $TO_A \geq TO + \Delta + (2n+1)$. Let R be a Δ-partially synchronous run. Let t_r be the time the first process starts a new round r after GST. Then by time $t_r + 2 + TO_A$ all processes have the same Alive set.*

Proof. By time $t_1 = t_r + 2$ the first round r message can be received (see Figure B.3). From the code of the algorithm, every process starts a new round the latest every $TO + (n+2)$ steps: one input step followed by n send steps, *TO* receive steps followed by an output step. From the fact that a process sends at most one message in each step, every process p_1 sends messages to any process p_2 every $TO + (n+2) + (n-1)$ steps. Since a message can take at least 0 and at most Δ time to be received, every process receives a message every $x = TO + (2n+1) + \Delta$ time. From the code of the algorithm, process p_2 excludes process p_1 from its *Alive* set, if it does not receive a message within TO_A steps (see line 13). Comparing TO_A with x we have $TO_A = x$, which is sufficient to receive a message from any alive process. □

C Proofs for Chapter 6

Lemma 6.5. *Let M a matrix with $\left|M[i][j] - \overline{rtt}_{ij}\right| \leq 2\varepsilon$ for all $i, j \in \Pi$. Then SelectLeader(M, ℓ) returns a process p such that: (i) $p \in \mathcal{O}$ and (ii) if $p \neq \ell$ then $\overline{majrtt}_p < \overline{majrtt}_\ell$.*

Proof. For (i), lines 2 and 3 of Algorithm 6.2 order each line of M in ascending order. Line 5 computes *minRtt*, which is the minimal majority value among the estimations contained in M. Let q be the process with the lowest id such that $M[q][\lfloor n/2 \rfloor + 1] = minRtt$.

From the assumption on M, we have $M[p_m][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_{p_m} + 2\varepsilon$. And since $M[q][\lfloor n/2 \rfloor + 1] \leq M[p_m][\lfloor n/2 \rfloor + 1]$ (because of the definition of q), we get

$$M[q][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_{p_m} + 2\varepsilon \tag{C.1}$$

If the condition at line 6 is true, then *SelectLeader* returns q and by (C.1) we have that $q \in \mathcal{O}$. Else, if the condition is false, *SelectLeader* returns ℓ, and we have $curRtt \leq minRtt + 4\varepsilon$, which is equivalent to:

$$M[\ell][\lfloor n/2 \rfloor + 1] \leq M[q][\lfloor n/2 \rfloor + 1] + 4\varepsilon. \tag{C.2}$$

By assumption, we have $\overline{majrtt}_\ell - 2\varepsilon \leq M[\ell][\lfloor n/2 \rfloor + 1]$. Using this inequality on the left side of (C.2) and (C.1) on the right side, we obtain $\overline{majrtt}_\ell - 2\varepsilon \leq \overline{majrtt}_{p_m} + 6\varepsilon \Leftrightarrow \overline{majrtt}_\ell \leq \overline{majrtt}_{p_m} + 8\varepsilon$. Therefore, $\ell \in \mathcal{O}$.

To prove (ii), since $p \neq \ell$, the condition on line 6 is true, that is, $minRtt < curRtt - 4\varepsilon$, which is equivalent to:

$$M[p][\lfloor n/2 \rfloor + 1] < M[\ell][\lfloor n/2 \rfloor + 1] - 4\varepsilon \tag{C.3}$$

By assumption, we have $\overline{majrtt}_p - 2\varepsilon \leq M[p][\lfloor n/2 \rfloor + 1]$ and $M[\ell][\lfloor n/2 \rfloor + 1] \leq \overline{majrtt}_\ell + 2\varepsilon$. Using these two inequalities in (C.3), we obtain $\overline{majrtt}_p - 2\varepsilon < \overline{majrtt}_\ell + 2\varepsilon - 4\varepsilon$ and thus

$\overline{majrtt}_p < \overline{majrtt}_\ell$. □

Lemma 6.6. *After time $GST + \eta + 4\Delta$, if SelectLeader(M, ℓ) is called, $|M[i][j] - \overline{rtt}_{ij}| \le 2\varepsilon$ for all $i, j \in \Pi$.*

Proof. Let ℓ be the process that calls *SelectLeader*(M, ℓ) at time $t \ge GST + \eta + 4\Delta$ while in round r_ℓ. By Algorithm 6.1, ℓ is the leader for round r_ℓ and the last time it sent ALIVE messages was at time $t - 2\Delta$. At this time it reset all entries of the matrix M to ∞. The ALIVE message is received by all processes during the interval $[t - 2\Delta, t - \Delta]$. All alive processes answer with a REPORT message containing their local vector. These messages are received by ℓ before time t and are used to update the matrix M. If a process q is crashed when it received the ALIVE message, then line $M[q]$ is not updated and remains equal to a vector of ∞.

Thus, the matrix M contains values that are no older than 2Δ. Since $t - 2\Delta = GST + \eta + 2\Delta$, by Lemma 6.2 we have $|M[i][j] - \overline{rtt}_{ij}| \le 2\varepsilon$ for all $i, j \in \Pi$. □

Lemma 6.7. *If there is a time t after which the maximum round reached by any process does not increase, then eventually a process from $\mathcal{O}$ is elected as leader.*

Proof. This proof is in two parts: (i) show by contradiction that a leader is elected, (ii) show that this leader must be in $\mathcal{O}$.

Let us assume that no leader is ever elected. Let r_h be the highest round of any process after t. Other processes might advance to higher rounds if they are in lower rounds but, by assumption, will never exceed r_h. Since by assumption processes cannot increase rounds forever, there is a time t' after which no process advances to a new round anymore.

Let $\mathcal{H}$ be the set of processes in round r_h after time t'. Let p be the candidate leader for round r_h (*i.e.*, $p \bmod n = r_h$).

If $p \in \mathcal{H}$, then p sends ALIVE messages to all processes every 2Δ. Therefore, at most at $t' + 3\Delta$, all processes will have received an ALIVE message for round r_h. Since by assumption no process is in a round higher than r_h, all alive processes will accept this message and advance to round r_h if not already there. Therefore, p becomes the leader for all alive processes, contradicting the assumption that no leader is elected.

If $p \notin \mathcal{H}$, then processes in $\mathcal{H}$ will timeout waiting for the ALIVE messages from the leader and will advance to a higher round, contradicting the assumption that no process advances to a new round after time t'.

To show (ii), let p be the process elected as leader. By assumption, p remains leader forever and keeps sending ALIVE messages every 2Δ time. Let t^* be the first time that p sends ALIVE messages after $GST + \eta + 2\Delta$. We can apply Lemmas 6.5 and 6.6 to show that by time $t^* + 4\Delta$ a process $\ell \in \mathcal{O}$ is elected leader. If $\ell \ne p$, then p advances to a higher round which contradicts the fact that p remains leader forever. Therefore, $\ell \in \mathcal{O}$. □

Lemma 6.8. *Let ℓ be a process that at time $t > GST$, is in round r_ℓ for which ℓ is the leader. Let t' be the time when ℓ sends the next* ALIVE *message. If all alive processes are in a round not higher than r_ℓ when they receive the* ALIVE *message from ℓ, then (i) no process $q \neq \ell$ advances to a round $r' > r_\ell$ while ℓ remains in round r_ℓ, and (ii) ℓ only advances to a new round if SelectLeader$_\ell(M, \ell)$ returns a process different than ℓ.*

Proof. To prove (i), note that since we are after GST the ALIVE message sent by ℓ at time t' is ready for reception at q the latest by time $t' + \Delta$. By assumption, at this time q is in a round not higher than r_ℓ. If q is in a lower round, it advances to r_ℓ. We now show that if ℓ remains in round r_ℓ, then q does not advance to a round higher than r_ℓ.

We proceed by contradiction. Let q be the first process to advance to a round higher than r_ℓ. Then either (a) q received a higher round message, (b) the timer of q expired, or (c) q called the procedure *SelectLeader*. (a) cannot happen because q is the first process to advance to a round higher than r_ℓ. (b) is not possible either, because the ALIVE messages from ℓ are received always with less than 3Δ of interval. Finally, q cannot call *SelectLeader* because it is not the leader for round r_ℓ.

To prove (ii), note that since ℓ is the leader, it will never timeout on itself. Additionally, by (i) we know that no other process will advance to a new round before ℓ, so ℓ will never receive a higher round message. Therefore, the only other way in which ℓ can advance to new rounds is if *SelectLeader*$_\ell(M, \ell)$ returns a process different than ℓ. □

Lemma 6.9. *Assume there is a process p and a round r_p such that p is the first process starting round r_p at time t_0, $t_0 > GST + 4\Delta$. Then before time $t_0 + 2\Delta$ no process will advance to a round $r' > r_p$.*

Proof. We proceed by contradiction. Let us assume there's a process $q \neq p$ that is the first to start a round $r' > r_p$ at time t', with $t_0 < t' < t_0 + 2\Delta$. This could have happened in one of the following ways: (i) receiving a message from a higher round, (ii) timeout on ALIVE messages from the leader, and (iii) calling *SelectLeader*.

Case (i) is impossible, because q is the first process starting round r'. In case (ii), q advances from round $r' - 1$ to round r' when this timeout occurs. By Algorithm 6.1, q must have been in round $r' - 1$ for at least 2Δ, which means that q started round $r' - 1$ before time t_0. Since $r' - 1 \geq r_p$, we get a contradiction with the assumption that p is the first process starting round r_p.

Finally, for (iii), let's assume that q advanced at time t' because of calling *SelectLeader*. This means that q was in a round $r_q < r'$ for which q is the leader, and after calling *SelectLeader* it advanced to round r'. We'll show that this contradicts the assumption that p was the first process to advance to round r_p at time t_0.

Process q was in round r_q for at least 2Δ, which is the time a process waits after entering a round until calling *SelectLeader* for the first time. Therefore, round r_q was started no later than $t' - 2\Delta$. And since $t_0 < t' < t_0 + 2\Delta$, we have $t' - 2\Delta < t_0 < t'$. From the above, it comes immediately that $r_q < r_p$, because by assumption no process started a round equal or higher than r_p before t_0.

Since q is the leader for round r_q, it sends ALIVE messages every 2Δ while in round r_q. Since $t' - 2\Delta$ is the latest that q entered the round, q sent an ALIVE message between $t' - 4\Delta$ and $t' - 2\Delta$. Since $t' - 4\Delta > GST$, this message is received by all alive processes between $t' - 4\Delta$ and $t' - \Delta$. If there is any process in a round higher than r_q when it receives the ALIVE from q, it answers with a $(START, r)$ message, which is received before t', forcing q to advance to a new round, which contradicts the assumption that q calls *SelectLeader* at time t'. Therefore, all processes are in a round $r \le r_q$ when they receive the ALIVE message from q. Applying Lemma 6.8, we conclude that no process will advance to a new round until q calls *SelectLeader*. But by assumption the next time this happens is at time t', contradicting the assumption that p advances to round r_p at time $t_0 < t'$. □

Bibliography

[ADGFT01] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*, pages 108–122. Springer-Verlag, 2001.

[ADGFT03] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*. ACM Press, 2003.

[ADGFT04] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceeding of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 328–337, St. John's, Newfoundland, Canada, 2004. ACM Press.

[AGGT08] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *Proceedings of the 22th International Conference on Distributed Computing (DISC'08)*, pages 32–46, 2008.

[AK08] Yair Amir and Jonathan Kirsch. Paxos for system builders. Technical Report CNDS-2008-2, Johns Hopkins University, 2008.

[BBH$^+$11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI'11*, pages 11–11, 2011.

[BCP$^+$03] Alberto Bartoli, Cosimo Calabrese, Milan Prica, Etienne Di Muro, and Alberto Montresor. Adaptive message packing for group communication systems. In *OTM 2003 Workshops*, LNCS. Springer, 2003.

[BHSS12] Fatemeh Borran, Martin Hutle, Nuno Santos, and André Schiper. Quantitative Analysis of Consensus Algorithms. *Transactions on Dependable and Secure Computing*, 9(2):236–249, March/April 2012.

[Bur06] 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, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[BW09] Martin Biely and Josef Widder. Optimal message-driven implementations of omega with mute processes. *ACM Trans. Auton. Adapt. Syst.*, 4(1):1–22, 2009.

[BWCM$^+$10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI’10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[CBS06] Bernadette Charron-Bost and André Schiper. Improving Fast Paxos: being optimistic with no overhead. In *Pacific Rim Dependable Computing, Proceedings*, 2006.

[CBS09] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, pages 49–71, 2009.

[CGH$^+$04] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman. High throughput reliable message dissemination. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, NY, USA, 2004.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC ’07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[Chu98] F. Chu. Reducing Ω to $\Diamond\mathcal{W}$. *Information Processing Letters*, 67:289–293, June 1998.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): a network subsystem architecture for server systems. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI ’96, pages 261–275, New York, NY, USA, 1996. ACM.

[DGDF$^+$08] Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With finite memory consensus is easier than reliable broadcast. In *OPODIS*, pages 41–57, 2008.

[DGK07] Partha Dutta, Rachid Guerraoui, and Idit Keidar. The overhead of consensus failure recovery. *Distributed Computing*, 19(5-6):373–386, April 2007.

[DGL05] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 22–27, Los Alamitos, CA, USA, 2005.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Don10] Benjamin Donzé. *Implementing and testing leader election algorithms*. PhD thesis, Ecole polytechnique fédérale de Lausanne, EPFL, 2010.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36, December 2004.

[ES06] Richard Ekwall and Andre Schiper. Solving atomic broadcast with indirect consensus. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 156–165, Washington, DC, USA, 2006. IEEE Computer Society.

[FH06] R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *Symposium on Reliable Distributed Systems, SRDS'06*, October 2006.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FR95] Roy Friedman and Robbert Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical Report TR95-1527, Department of Computer Science, Cornell University, 1995.

[FRT99] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 470 –477, 1999.

[Gaf98] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceeding of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 143–152, Puerto Vallarta, Mexico, 1998. ACM Press.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96*, 1996.

[GLPQ10] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2), 2010.

[HdB11] Tom Herbert and Willem de Bruijn. Scaling in the linux networking stack, November 2011.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX-ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[HMSZ06] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Brief announcement: Chasing the weakest system model for implementing Ω and consensus. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '06)*, 2006.

[HO08] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[HS07] Martin Hutle and André Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Dependable Systems and Networks (DSN 2007)*, pages 92–10. IEEE, June 2007.

[HW05] Martin Hutle and Josef Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Self-Stabilizing Systems*, pages 153–170, 2005. Appeared also as Brief Announcement at PODC'05.

[JRS11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN'11*, pages 245–256, 2011.

[KJ10] Manos Kapritsos and Flavio P. Junqueira. Scalable agreement: toward ordering as a service. In *Proceedings of the Sixth international conference on Hot topics in system dependability*, HotDep'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[KS06] Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 169–178, New York, NY, USA, 2006. ACM Press.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.

[Lam89] Leslie Lamport. The part-time parliament. Technical report, SRC Research, September 1989.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[Lam06] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–102, October 2006.

[Lev08] Ron Levy. *The Complexity of Reliable Distributed Storage*. PhD thesis, EPFL, 2008.

[LMZ10] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41:63–73, 2010.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[MMN$^+$04] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04*, pages 8–8, 2004.

[MOZ05] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Ω meets Paxos: Leader election and stability without eventual timely links. In *Dependable Systems and Networks (DSN 2005)*, 2005.

[MPP11] P.J. Marandi, M. Primi, and F. Pedone. High performance state machine replication. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 454 –465, june 2011.

[MPP12] P.J. Marandi, M. Primi, and F. Pedone. Multi-ring Paxos. In *Dependable Systems Networks (DSN), 2012 IEEE/IFIP 42st International Conference on*, june 2012.

[MPSP10] Parisa Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN'10)*, June 2010.

[Nag84] John Nagle. Congestion control in IP/TCP internetworks. Technical Report RFC 896, IETF, January 1984.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM.

[PLL97] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, volume LNCS 1320/1997, pages 111–125. Springer-Verlag, 1997.

[PM95] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1-2), 1995.

[RST11a] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4:243–254, 2011.

[RST11b] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

[SB05] L. Sampaio and F. Brasileiro. Adaptive indulgent consensus. In *Dependable Systems and Networks (DSN 2005)*, pages 422–431, June-1 July 2005.

[SBCF03] Livia M. R. Sampaio, Francisco V. Brasileiro, Walfredo Cirne, and Jorge C. A. Figueiredo. How bad are wrong suspicions? towards adaptive distributed protocols. In *Dependable Systems and Networks (DSN 2003)*, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[Sch97] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[SKŻ$^+$11] Nuno Santos, Jan Kończak, Tomasz Żurkowski, Paweł Wojciechowski, and André Schiper. JPaxos - State machine replication in Java. Technical Report 167765, EPFL, July 2011.

[SNBJP05] L. Sampaio, R. C. Nunes, F. Brasileiro, and I. Jansch-Pôrto. Efficient and robust adaptive consensus services based on oracles. *Journal of the Brazilian Computer Society (JBCS), Special Issue on Dependable Computing (2005)*, 2005.

[SW89] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proc. 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, February 1989. Springer-Verlag.

[VALB$^+$10] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Gregory Chockler, Haoyuan Li, and Yoav Tock. Dr. multicast: Rx for data center communication scalability. In *EuroSys '10*, 2010.

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP, pages 230–243, New York, NY, USA, 2001. ACM.

[Wea02] Brian White and et al. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

Publications

Chapter 4

Fatemeh Borran, Martin Hutle, Nuno Santos, and André Schiper. Quantitative Analysis of Consensus Algorithms. In *Transactions on Dependable and Secure Computing (TDSC)*, 9(2):236–249, March/April 2012.

Chapter 5

Fatemeh Borran, Martin Hutle, Nuno Santos and André Schiper. Swift Algorithms for Repeated Consensus. In *29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*, New Delhi, India, October 31 – November 3, 2010.

Chapter 6

Nuno Santos, Martin Hutle and André Schiper. Latency-Aware Leader Election. In *2009 ACM symposium on Applied Computing (SAC 2009)*, Honolulu, Hawaii, USA, March 9-12, 2009.

Chapter 8

Nuno Santos and André Schiper. Tuning Paxos for high-throughput with batching and pipelining. In *3rd International Conference on Distributed Computing and Networking (ICDCN 2012)*, Hong Kong, China, January 3-6, 2012.

Chapter 9

Nuno Santos and André Schiper. Achieving high-throughput State Machine Replication in multi-core systems. EPFL Technical Report 170480, 2011.

Chapter 10

Martin Biely, Zarko Milosevic, Nuno Santos, André Schiper S-Paxos: Offloading the leader for high throughput state machine replication. In *31st IEEE International Symposium on Reliable Distributed Systems (SRDS 2012)*, Irvine, California, USA, October 8-11, 2012.

Curriculum Vitae

I was born in Coimbra (Portugal) in 1978. In 2000 I graduated from the University of Coimbra with a B.Sc. in Mathematics, with a specialization in Computer Science, receiving an award for graduating at the top of my class. Three years later, in 2003, I obtained M.Sc. in Computer Engineering also from the University of Coimbra, with a thesis topic of "Security in Mobile Agents Platforms". In 2002 and 2003 I did two summer internships: the first at Ericsson in Aachen, and the second at CERN in Geneva. After completing my M.Sc. I worked for one year as a software engineer at Wit-Software in Coimbra, Portugal. In November 2004 I returned to CERN, where I worked for two years developing fault-tolerant Grid Computing middleware. Since November 2006 I have been working at the Distributed Systems Laboratory as a research and teaching assistant, and doing a PhD on the topic of fault-tolerant distributed computing under the supervision of Professor André Schiper.