A High-Throughput Byzantine Fault-Tolerant Protocol

THÈSE N^{O} 5242 (2012)

PRÉSENTÉE LE 17 FÉVRIER 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE PROGRAMMATION DISTRIBUÉE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nikola KNEZEVIC

acceptée sur proposition du jury:

Prof. B. Moret, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. A. Ailamaki, rapporteur
Dr C. Cachin, rapporteur
Prof. V. Quéma, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

. . . a programmer types in some code, compiles it, runs it, and waits for it to crash. Programs that don't crash are presumed to be running correctly.

— The UNIX HATERS Handbook

To my parents, my brother, and my two amazing nephews. . .

Abstract

State-machine replication (SMR) is a software technique for tolerating failures and for providing high availability in large-scale systems, through the use of commodity hardware. A replicated state-machine comprises a number of replicas, each of which runs an agreement protocol, with the goal of ensuring a consistent state across all of the replicas. In hostile environments, such as the Internet, Byzantine fault tolerant state-machine replication (BFT) is an important technique for providing robust services. During the past decade, we have seen an emergence of various BFT protocols. In order to be adopted, besides providing correctness, a BFT must provide good performance as well. Consequently, all of the new protocols focus on improving performance under various conditions.

However, a closer look at the performance of state-of-the-art BFT protocols reveals that even in best-case execution scenarios, they still remain far behind their theoretical maximum. Based on exhaustive evaluation and monitoring of existing BFT protocols, we highlight a few impediments to their scalability. These obstructions include the use of IP multicast, the presence of bottlenecks due to asymmetric replica processing, and an unbalanced network bandwidth utilization.

The goal of this thesis is to evaluate the actual impact of these scalability impediments, and to offer a solution for a high-throughput BFT protocol in the case in which the network itself is the bottleneck. To that end, we have developed Ring, a new BFT protocol which circumvents the aforementioned impediments. As its name suggests, Ring uses the ring communication topology, in the fault-free case. In the ring topology, each replica only performs point-to-point communications with two other replicas, namely its neighbors on the ring. Moreover, all of the replicas equally accept requests from clients and perform symmetric processing. Our performance evaluation shows that, with the network as the bottleneck, Ring outperforms all other, state-of-the-art BFT protocols. Ring achieves 118 Mbps on the Fast Ethernet – a 24 % improvement in throughput over previous protocols.

Finally, we conducted an extensive practical and analytic evaluation of Ring. In order to analyse benefits (and drawbacks) of Ring (and other protocols) under different settings, without resorting to costly experimentation, we developed an analytical performance model. Our performance model is based on queueing theory, and relies only on a handful of protocol-agnostic measurements of the environment.

Keywords: algorithms, Byzantine fault tolerance, asynchronous systems, scalability, high throughput, correctness proofs, performance, ring topology, analytic modelling, queueing theory, replication and security.

Résumé

La technique dite de *State-Machine Replication*, traduit en français par “réplication de machine d’état”, est une technique logicielle permettant de tolérer les fautes et d’atteindre une haute disponibilité dans les systèmes à grande échelle bâti avec du matériel non spécialisé. Une machine d’état répliquée est constituée de plusieurs répliques, chacune d’elles exécutant un protocole de consensus, dont la tâche est de maintenir un état consistent entre elles.

Dans un environnement hostile tel qu’Internet, la réplication de machine d’état tolérant les fautes dites “Byzantines” est une technique importante pour offrir des services robustes. La dernière décennie à vu proliférer les protocoles de réplication de machine d’état tolérant les fautes Byzantines (dits protocoles BFT). Cependant, en pratique, en plus de fournir la garantie d’une exécution correcte, les protocoles BFT doivent êtres performants pour être adoptés. En conséquence les nouveaux protocoles sont spécialisé pour des types de systèmes particuliers.

Malgré cela les meilleurs protocoles BFT ont des performances très inférieures au maximum théoriquement atteignable. Grâce à une évaluation exhaustive des protocoles BFT existants nous mettons en lumière certaines cause de ce problème. On y retrouve l’utilisation du *multicast IP*, la présence de goulots d’étranglements tels que le traitement non symétrique des requètes, ou la mauvaise répartition de la bande passante réseau.

Cette thèse précise l’impact réel des différentes causes identifiées et propose un protocole BFT haut-débit pour les cas dans lesquels le réseau constitue un goulot d’étranglement. Ce protocole se nome *Ring*, traduit par “Cercle” en français. Comme son nom l’indique, les canaux de communication entre répliques forment un cercle dans ce protocole. En d’autres termes, les répliques sont organisées en cercle et une réplique ne communique que avec ses deux plus proches voisins. De plus, chaque réplique peut recevoir les requètes des clients du système et le traitement de celles-ci est fait de manière symétrique. Nos expériences montrent que, lorsque le réseaux est le point faible du système, le protocole *Ring* dépasse le meilleur débit parmi les protocoles BFT existants d’un facteur de 24 pourcent, pour atteindre un débit de 118 Mbps dans un réseau de type *Fast Ethernet*.

En conclusion, nous avons évalué le protocole *Ring* de manière expérimentale et analytique. Afin d’identifier les bénéfices et coûts des différents protocoles nous avons développé un modèle analytique des performances des protocoles BFT. Ceci nous a permis d’éviter des test difficiles à mettre en place en pratique. Notre modèle est basé sur la théorie des files d’attentes

(*Queuing Theory* en anglais) et ne repose que sur un faible nombre de mesures faites dans l'environnement du système.

Mots-clés : Algorithmes, fautes Byzantines, systèmes asynchrones, évolutivité, haut débit, preuves de protocoles, performance, topologie en anneau, modèles analytiques, théorie des files d'attente, réplication, sécurité.

Zusammenfassung

Die Replikation von Zustandsautomaten ist eine Softwaretechnik, die es erlaubt herkömmliche Hardware einzusetzen und dennoch Ausfälle zu tolerieren und die Verfügbarkeit von grossen verteilten Systeme zu gewährleisten. Ein replizierter Zustandsautomat besteht aus einem Satz von Instanzen die jeweils ein Vereinbarungsprotokoll ausführen um einen konsistenten Zustand bei allen Instanzen zu erreichen. In feindseligen Umgebungen, wie z.B. im Internet, ist die gegen byzantinische Fehler tolerante Replikation von Zustandsautomaten (BFT) ein wichtiges Verfahren um robuste Dienste zur Verfügung zu stellen. Im laufe des letzten Jahrzehnts wurden einige BFT-Protokolle entwickelt. Damit ein BFT-Protokoll Einsatz finden kann, muss es nicht nur Korrektheit, sondern auch eine hohe Geschwindigkeit aufweisen. Aus diesem Grund konzentrieren sich alle neuen Protokolle darauf, ihre Geschwindigkeit unter verschiedenen Verhältnissen zu verbessern.

Eine genauere Untersuchung von aktuellen BFT-Protokollen zeigt jedoch, dass diese sogar unter besten Ausführungsbedingungen weit unter ihren theoretischen Maximalgeschwindigkeiten liegen. Aufgrund von vollständigen Auswertungen und Beobachtungen von existierenden BFT-Protokollen stellen wir einige Hindernisse zu deren Skalierbarkeit heraus. Diese Hindernisse umfassen die Verwendung von IP-Multicast, Engpässe aufgrund von asymmetrischer Verarbeitung durch verschiedene Instanzen und unausgeglichener Verwendung der verfügbaren Datenrate des Netzes.

Diese Arbeit bewertet den tatsächlichen Einfluss dieser Hindernisse und erarbeitet eine Lösung für ein BFT-Protokoll mit hohem Druchsatz, im Falle dass das Netz selbst den Engpass darstellt. Hierzu entwickelten wir *Ring*, ein neues BFT-Protokoll, welches die zuvor genannten Hindernisse umgeht. Wie aus dem Namen hervorgeht, verwendet *Ring* eine ringförmige Kommunikationstopologie solange kein Ausfall vorliegt. Bei der Ring-Topologie kommuniziert jede Instanz nur mit zwei weiteren Instanzen, ihren Nachbarn im *Ring*. Weiterhin nehmen alle Instanzen Anfragen von Clients im gleichen Maße entgegen und bearbeiten diese Anfragen in gleicher Weise. Die Auswertung der Geschwindigkeit zeigt, sofern das Netz die Engstelle darstellt, dass *Ring* alle anderen aktuellen BFT-Protokolle übertrifft — die erreichten 118 Mbps bei Fast Ethernet stellen eine Verbesserung von 24 % dar.

Abschliessend führten wir eine umfangreiche praktische und analytische Untersuchung von *Ring* durch. Um die Vorteile (und Nachteile) von *Ring* (und anderen Protokollen) unter verschiedenen Verhältnissen zu untersuchen, ohne auf aufwendige Experimente zurückgreifen zu müssen, entwickelten wir ein analytisches Geschwindigkeitsmodell. Dieses Geschwindigkeitsmodell basiert auf der Warteschlangentheorie und benötigt lediglich einge wenige protokollunabhängige Messpunkte des Einsatzumfelds.

Stichworte: Algorithmen, Byzantinische Fehlertoleranz, Asynchrone Systeme, Skalierbarkeit, Hoher Durchsatz, Korrektheisbeweise, Geschwindigkeit, Ringtopologie, Analytisches Modellieren, Warteschlangentheorie, Replikation und Sicherheit.

Acknowledgements

As I approach the end of this inspiring, yet arduous journey, which is the process of obtaining a Ph.D., I would like to use this opportunity to thank some of the people without whom getting this far would not have been possible at all.

My deepest gratitude goes to my thesis advisor, Prof. Rachid Guerraoui, for giving me the opportunity to pursue this research in his lab, and under his guidance. I am particularly thankful to him for having had confidence in me, having given me motivation and support, and for sharing with me an invaluable experience. Thanks to Rachid for generously guiding me through all of the phases of my work, and, more importantly, for the full sense of freedom I have felt while working in his laboratory, which has made my research joyful and pleasant.

My special thanks go to my previous advisor, Prof. Dejan Kostić, for inviting me to his lab in the first place. I am thankful for his many useful advice, assistance and dedicated involvement throughout all of our joint projects.

I thank Prof. Bernard Moret for being president of the jury. I also thank Prof. Anastasia Ailamaki Dr. Christian Cachin and Prof. Vivien Quéma for serving on my thesis committee and for providing me with great, insightful comments which make my thesis look more complete.

I am grateful to Vivien Quéma for sharing his experience and knowledge with me on our projects. I owe him for teaching me many tricks, for various, yet always clear explanations of many things, and showing me how to effectively solve problems.

I would like to thank all of the former and current lab members: Dan, Maysam, Marko, Maxime, Radu, Mihai, Giuliano, Vasileios, Fabien, Vincent and Florian, for the numerous discussions that we have had, and for the many activities that we have done together. I would like to especially thank Kristine for always having time to listen to and answer my questions, for her kindness, and for making our lab a lively, warm place. I am thankful to Dan, my friend and office-mate, for so many interesting conversations, both on- and off-work.

My gratitude also goes to Petar Maksimović who, like all great copy-editors, spotted many errors in writing, and corrected them all in his elegant way. Without him, this thesis would be much harder to read.

Special thanks go to my friends Aca, Deki, Nedja, Miloš, Mića, Tamara, Tanja, Voja&Flo, Simon and many others, for spending many unforgettable moments together in Lausanne, and for making my stay so pleasurable. I would also like to thank my friends, scattered around the globe, Grabi&Maja, Sima&Nataša, Branka, Daca, Šale, Pak, Ruža and Teo, for being there for me through all these years.

I thank my dear and loving Marica, for her patience and support, even though the former goes against her nature. With her, everything becomes not only possible, but also easy and enjoyable! I am looking forward to many mutual moments and experiences to come.

Last, but not least, I am profoundly grateful to my parents Boja and Đuro, as well as to my brother Jovan with his family — my two amazing nephews, Stefan and Nikola, for their love, encouragement, and unconditional support throughout my whole life. Thank you for easing my life immensely during all these years.

Preface

This PhD thesis describes the research done at the Distributed Programming Laboratory, School of Computer and Communication Sciences, EPFL, under the supervision of Prof. Rachid Guerraoui, from 2009 to 2011. The work presented in this thesis is centered on high-throughput algorithms, the accompanying challenges, and implementation techniques.
In addition to the presented material, I have also worked on Mirage [Crameri et al., 2007], an integrated software upgrade and deployment system. Afterwards, I have briefly worked on detecting and preventing inconsistencies in a deployed distributed system [Yabandeh et al., 2009]. Next, I have worked on the design and implementation of a cost-effective testbed [Knežević et al., 2010]. The goal of this project was to implement a fully-functional networking testbed, which would harness the power of multicore architectures in order to run many virtualized routers on a single chip, in a time sharing manner, while retaining good performance. Some of the evaluations presented in that paper are used in this thesis. Finally, as an introduction to the work presented in this thesis, I worked on implementing many different BFT protocols in the ABSTRACT framework [Guerraoui et al., 2010b].
The materials presented in this thesis are published as an LDP Technical Report [Guerraoui et al., 2010a], or are under submission [Guerraoui et al., 2011].

List of publications

[Crameri et al., 2007] Olivier Crameri, Nikola Knežević, Dejan Kostić, Ricardo Bianchini, and Willy Zwaenepoel. "Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System". In: *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 2007.

[Guerraoui et al., 2010a] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. *Stretching BFT*. Tech. rep. EPFL-REPORT-149105. EPFL, 2010. URL: http://infoscience.epfl.ch/record/149105.

[Guerraoui et al., 2010b] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. "The Next 700 BFT Protocols". In: *Proceedings of the European conference on Computer systems (EuroSys)*. 2010.

[Guerraoui et al., 2011] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Ali Shoker. “Optimizing BFT Protocols: much ado about nothing?” under submission. 2011.

[Knežević et al., 2010] Nikola Knežević, Simon Schubert, and Dejan Kostić. “Towards a cost-effective networking testbed”. In: *ACM SIGOPS Operating Systems Review* 43 [2010].

[Yabandeh et al., 2009] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. “CrystalBall: predicting and preventing inconsistencies in deployed distributed systems”. In: *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

Contents

List of Figures

List of Tables

1 Introduction

In the today's society, there exists a great dependency on services provided by large-scale computer systems, imposing a need for these services to be highly available: they should work correctly and they should provide uninterrupted service. However, at the large-scale level, computer failures are still considered to be the norm, rather than the exception.

Replication is an intuitive, software-based (and, thus, cost-effective) way of providing high availability: a critical IT service is replicated over a number of machines. In a replication scenario, a specific distributed protocol is put in charge of ensuring consistency across all of the replicas, regardless of any malicious behaviour on part of clients or any subset of the replicas [Bracha and Toueg, 1985; Dwork et al., 1988]. Replication consistency is usually provided by having the protocol order the requests of the clients on all replicas of the service (that is, make the replicas *agree* on what the correct order is). In addition, the replicated service must be implemented as a deterministic state machine [Lamport, 1978; Schneider, 1990]. The most robust replication protocols today are called BFT state machine replication protocols or, simply, BFT protocols. BFT protocols are capable of tolerating arbitrary client failures, as well as failures of up to one third of the replicas [Cachin, 2010; Castro and Liskov, 1999; Lamport, 2004].

For a long time, the interest in BFT protocols has remained solely theoretical, as their performance was considered to be inadequate in a practical setting. However, more than a decade ago, the first practical BFT protocol (called PBFT [Castro and Liskov, 1999]) appeared, opening a new era of research in the systems community. Many protocols [Abd-El-Malek et al., 2005; Clement et al., 2009a; Cowling et al., 2006; Guerraoui et al., 2010a; Kotla et al., 2007] have since been proposed, with the goal of enhancing the performance of PBFT, while targeting, in particular, its throughput and latency. These protocols do indeed provide a boost in the best case performance of PBFT, and the obtained results have implied significant gains once applied to the replication scheme. Moreover, in best-case executions (that is, synchronous executions with no failures) they get close to the performance of non-replicated systems. Arguably, these best-case execution scenarios are achieved frequently in practice.

However, a closer look at the performance of state-of-the-art BFT protocols reveals that even in these best-case execution scenarios, they still remain far behind the theoretical maximum. For instance, our experiments show that, when deployed on a Fast Ethernet network, the most efficient BFT protocols reach a maximum throughput of 93 Mbps (for 4 KiB messages), whereas the theoretical maximum [Guerraoui et al., 2010b] is 124 Mbps[1].

The reason behind such a discrepancy of observed throughputs lies in the fact that the state-of-the-art protocols were implicitly optimized only for CPU-bottleneck workloads, without giving consideration to performance in cases in which the bottleneck is, in fact, the network. Additionally, the throughput performance issue becomes even more relevant with recent advances in deterministic execution on multicore machines [Aviram et al., 2010; Bergan et al., 2010], which make the leveraging of multicore architectures and the achievement of a high CPU execution performance possible. Basically, in the near future, the presence of a bottleneck will no longer be related to the execution speed of the replicated service, but, rather, to the throughput of the agreement phase of the underlying replication protocol. Hence, there exists a pressing need for the development of BFT protocols with high performance in network-bottleneck conditions.

This thesis addresses precisely the problem of attaining high performance in cases in which the network is the bottleneck. Furthermore, this thesis describes a novel algorithm and implementation techniques for building highly-available, high-throughput systems which tolerate Byzantine faults. A more detailed description of the contributions of this thesis is provided in the following section.

1.1 Contributions

In essence, the insight of this thesis is that in a symmetric system, all components become bottlenecks at the same time. In a non-symmetric system, some component will become the bottleneck with a much lower utilization (compared to a symmetric system), thus limiting the total utilization of the system. Thus, the power of symmetry lies in postponing the bottleneck condition to a much higher utilization, that in turn enables a higher total utilization.

This dissertation explores this insight in BFT protocols, and contains the following contributions for the design, implementation, and deployment of high-throughput BFT protocols:

1. We give a thorough analysis of the bottlenecks present in current, state-of-the-art BFT protocols, and the effects of these bottlenecks on the performance of the protocols.

[1]The theoretical maximum for a replicated service is $\frac{n}{n-1}\mathscr{B}$, where n is the number of replicas, and $\mathscr{B}$ is the maximal throughput of a single network link (93 Mbps on the Fast Ethernet network we are using, as reported by the *netperf* tool).

2. We present Ring, a new BFT protocol for high-throughput workloads. Unlike other protocols, Ring approaches the theoretical maximum for throughput. We use formal methods to specify the algorithm and prove its safety.
3. We perform an extensive practical evaluation of Ring, so as to assess its performance under different settings. In addition, we develop an analytic performance model based on queueing theory, in order to be able to reason about the performance without resorting to costly experimentation.

In the next few paragraphs, we consider each of the contributions in more detail.

Throughput bottleneck analysis. The development and the design of Ring was influenced by an extensive study that we have previously conducted, the goal of which was the identification of bottlenecks of the currently most-efficient BFT protocols. This study has helped us to understand the feasibility of throughput-efficient BFT protocols protocols. Our study (detailed in Chapter 3) has revealed the following limiting factors:

- *Asymmetric replica processing:* existing protocols do not equally balance the processing load across different replicas (some replicas perform up to 20 % higher CPU processing when compared with other replicas);
- *Unbalanced network utilization:* existing protocols do not equally use the available networking resources (some replicas do not send or receive any data);
- *IP multicast packet drops:* most BFT protocols rely on IP multicast, which is often inefficient in highly-loaded environments, as it may result in high ratios of packet drops (30 % on the tested hardware [White et al., 2002]).

In the same chapter, we also expose *message handling issues of multicores* as a possible limiting factor in reaching high throughput in the future.

Ring. The *main contribution* of this dissertation is our new BFT protocol, called **Ring**, which circumvents all the impediments that we have discovered in our analysis. Ring attains high-throughput performance, for several reasons:

- *Ring avoids CPU bottlenecks:* all of the replicas in Ring are CPU symmetric and perform (almost) identical processing — any replica can receive a request from any client;
- *Ring avoids wasting of resources:* there are no underutilized network links — the load is fully balanced across all of the available network links.
- *Ring avoids IP multicast:* instead, it uses a point-to-point ring topology for request dissemination and ordering.

The idea of using a ring-based topology as a means for improvement of the throughput of broadcasting protocols is not new in itself: it was adopted, for instance, in LCR [Guerraoui et al., 2010b] and Ring Paxos [Jalili Marandi et al., 2010]. However, both LCR and Ring Paxos focus on crash failures, while Ring is capable of handling their superset — Byzantine failures. Tolerating Byzantine faults while maintaining a ring-based communication pattern with good performance is the main technical difficulty in the design of Ring, and is challenging in various aspects. For instance, a good algorithm has to ensure that a faulty replica must not trick correct replicas into avoiding the execution of correct requests. Also, such an algorithm has to be able to deal with the converse issue of preventing faulty replicas from forcing correct replicas into the execution of non-existent requests, or from bypassing replicas in the ring.

Similarly to PBFT, Ring is implemented as a generic programming library that can be used to provide Byzantine fault tolerance to various services. Additionally, Ring is implemented inside the ABSTRACT framework [Guerraoui et al., 2010a], a framework for designing modular BFT protocols. As such, Ring can be combined with any other protocol developed in the ABSTRACT framework, in order to provide more versatility once the working conditions change.

Extensive practical and analytic evaluation. Even when one uses the ABSTRACT framework, which eases the development of BFT protocols, the design and the deployment of actual protocols is still a costly task. As noted by Singh et al. [2008] and Clement et al. [2009a], all of the protocols behave differently under different conditions, and there does not exist a "one solution fits all" scenario. To further complicate the deployment, protocols may exhibit behaviour unlike the one designers anticipated and reported, once deployed on different platforms.

Toward better understanding of the benefits of Ring, we have conducted both a practical and analytic evaluation:

- We have performed an extensive practical evaluation of Ring on the Emulab testbed [White et al., 2002]. Our benchmarks show that Ring outperforms other protocols in cases in which the network is the bottleneck. More precisely, with clients issuing 4 KiB requests on a Fast Ethernet network, Ring achieves 118 Mbps, whereas other protocols remain at 93 Mbps.
- We have developed a theoretical performance model based on queueing theory, in order to be able to reason about the performance and behaviour of BFT protocols on various platforms. By using representations of different protocols in our performance model and some *protocol-agnostic* measurements[2], we have analysed the performance of Ring in various settings. A thorough analysis shows that our analytic model matches our practical evaluation.

[2]Such as the speed of different cryptographic operations, kernel interactions (the cost of issuing syscalls and/or servicing times for managing multiple connections), and estimates on the data structures handling times, which the protocol designer can either leave out, measure, or provide an estimate based on past experience.

1.2 Roadmap

The rest of the thesis is organized into seven chapters.

In Chapter 2 of the thesis, we give a gentle introduction to Byzantine Fault Tolerant protocols, followed by an overview of the previous, state-of-the-art protocols and related work. Finally, there we also describe the system model which was used to design Ring.

Next, in Chapter 3, we present our findings regarding the impediments to protocol scaling, along with an experimental evaluation.

Chapter 4 is the central point of the thesis, where we present the design and implementation details of Ring. Ring uses two operational modes to handle client requests. In this chapter, we describe both of these modes, the implementation of each of the modes, and provide proofs of their correctness.

The optimizations used in Ring are introduced in Chapter 5. There we also give an outline of authentication challenges on a ring topology in the presence of such optimizations. Finally, in this chapter we propose an algorithm for the detection of the presence of slow replicas in a system, so as to avoid certain kinds of performance attacks [Amir et al., 2008].

In Chapter 6, we describe our analytic framework for modelling the performance of BFT protocols. Consequently, we present our analytic tool for determining the best working conditions of a set of protocols, for any given platform.

In Chapter 7, we present the results of our extensive evaluation of Ring, followed by a comparative assessment of the accuracy of our performance model.

Finally, with Chapter 8, we conclude the thesis.

2 Concepts and Related Work

This introductory chapter lays out the foundation for the remaining chapters to come. First, we briefly introduce the notion of Byzantine Fault Tolerance (BFT). Next, we introduce all of the available, state-of-the-art protocols used later in the analysis of scaling impediments and the evaluation. Additionally, in this chapter we catalog the related work. Finally, we describe the system model under which Ring operates.

2.1 Byzantine Fault Tolerant Protocols

As mentioned in Chapter 1, BFT protocols allow for highly-available services, robust against arbitrary faults, through the use of replication [Lamport et al., 1982]. In this section, we give a short overview of Byzantine Fault Tolerance and the mechanism employed in BFT protocols for guaranteeing consistency. The short overview is followed by a description of the current, state-of-the-art protocols.

2.1.1 Byzantine Fault Tolerance

Byzantine fault tolerant systems [Lamport et al., 1982] are resilient to arbitrary (Byzantine) faults which may occur during the execution of an algorithm in a distributed system. Byzantine faults encompass both *omission* failures (for instance, crash failures, of failures to receive a message) and *commission* failures (for instance, incorrect processing of a request, sending inconsistent responses, or maliciously corrupting a state). Systems that are not resilient to Byzantine faults may respond to such failures in any unpredictable manner. Correctly functioning components of a BFT system will be able to correctly provide the services of the system, assuming there are not too many[1] Byzantine faulty components. The Byzantine failure

[1] Byzantine fault tolerant systems can tolerate up to one third of faulty components, as shown by Bracha and Toueg [1985]; Lamport [2004]; Lamport et al. [1982]; Pease et al. [1980].

assumption models real-world environments in which computers and networks may behave in unexpected ways due to hardware failures, network problems, or even malicious attacks.

Members of a BFT distributed system are called *processes*. Processes are a logical abstraction, represent the execution path of the distributed algorithm, and take steps in the execution. A *faulty* process is one which at some point exhibits any type of failure. A process which is not faulty is *correct*. A BFT algorithm must cope with any failure and satisfy the properties of the problem it is designed to solve. The number of faulty processes a BFT algorithm could tolerate is called the *resilience* of the algorithm, and is usually denoted f. Naturally, there is an upper bound on the number of faults that any BFT algorithm can tolerate [Pease et al., 1980] — a system of n participants tolerates up to one third of faulty components: $n \geq 3f + 1$.

2.1.2 BFT State Machine Replication

BFT state machine replication consists of replicating a service over a group of servers (called *replicas*). Each of the replicas maintains a set of state variables, which are handled by a set of operations which read or modify the variables in atomic fashion. Moreover, these operations need to be deterministic, and each of the operations needs to be executed in the same manner on all of the replicas. In other words, assuming the same initial state on all of the replicas, executing the same operation on any replica should generate the same final state. The properties that a state machine replication algorithm must satisfy are:

- **safety:** all correct replica processes execute the same requests in the same order;
- **liveness:** all correct requests from the clients processes are eventually executed.

These properties could be achieved either through agreement-based, or through quorum-based communication schemes [Lynch, 1996].

Agreement-based protocols were initially proposed for Byzantine-fault-tolerant state machine replication [Cachin, 2000; Cachin and Poritz, 2002; Castro and Liskov, 1999; Schneider, 1990; Zhou et al., 2002]. This class of protocols relies on one primary replica to define a sequential order of client operations, and an agreement protocol which runs amongst the replicas and ensures an agreement on this ordering. This agreement may sometimes, in the presence of faults, have multiple phases of all-to-all communication [Castro and Liskov, 1999; Kotla et al., 2007]. A minimum of $3f + 1$ replicas are required for the system to be able to tolerate f Byzantine faults [Bracha and Toueg, 1985]. Hence, an increase in the number of tolerated faults leads to an increase in message load, which scales quadratically due to all-to-all communication, and may, in turn, pose a scalability problem.

An alternative approach utilizes responses from a quorum of correct replicas, rather than relying on agreement among replicas [Malkhi and Reiter, 1997]. These protocols are inherently optimistic, as each client contacts a quorum of replicas that *independently* order each of the operations. A quorum represents a set of replicas, such that any two quorums intersect on at least one correct replica. In quorum-based communication, operations complete only if the assigned ordering by each of the replicas is consistent. Otherwise, quorum-based schemes require expensive reconciliation phases (and, thus, have poor performance). This occurs most often during write contention, where multiple clients attempt to perform a write operation at the same time. Since high-throughput workloads mostly exhibit high-contention, these protocols are not suitable for the workloads of interest in this thesis.

Interfacing the Service. Most BFT protocols interact with the state machine (which they replicate) as a library. Figure 2.1 outlines the use of such a library. The library is placed as a thin shim beneath the replicated application, on both the replicas and the clients. Clients send messages through the library, which encapsulates the request, adding its own (necessary) headers and authenticators. Replicas receive these messages in an event-driven loop, and exchange them further. Once the replicas have reached the execution point, they unwrap the payload, and make an *up-call* to the application to execute the request. Lastly, replicas wrap the response, send it to the client, that unwrap the reply only if the message can be authenticated, and perform an up-call to the application client code.

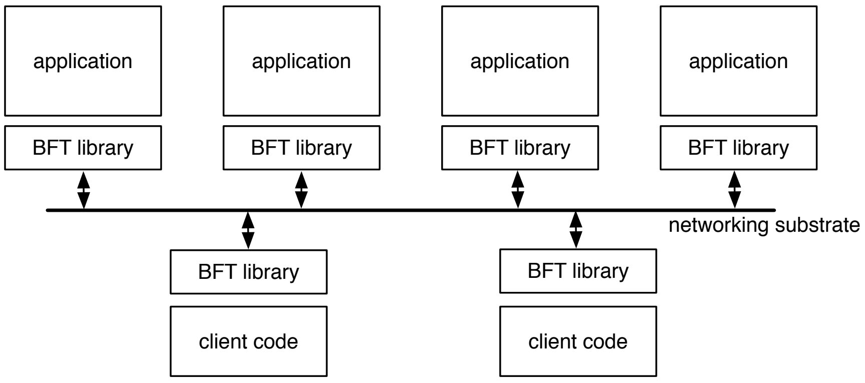

Figure 2.1: BFT library interaction with the replicated application.

2.1.3 Overview of State-of-the-Art Protocols

In this section, we give an overview of the available, state-of-the-art BFT protocols. We focus on protocols known to provide high throughput: PBFT [Castro and Liskov, 1999], Zyzzyva [Kotla et al., 2007], and Chain [Guerraoui et al., 2010a]. These three protocols rely on a dedicated replica which receives requests, called the *primary* (or the *head* in Chain). The primary (respectively, the head) assigns sequence numbers to requests and forwards them to other replicas. It is important to note that all of these protocols require at least $3f+1$ replicas to tolerate f faults (which is optimal, by Bracha and Toueg [1985]; Lamport [2004]). Here, we do not describe quorum-based protocols [Abd-El-Malek et al., 2005; Cowling et al., 2006][2], which are known to perform poorly under contention [Singh et al., 2008].

PBFT. PBFT is the first practical BFT protocol presented in the systems research. The communication pattern of PBFT [Castro and Liskov, 1999] is depicted in Figure 2.2. PBFT relies on a dedicated replica, called the *primary*, to order requests. Clients send requests to the *primary*. There are three rounds of communication between the primary and the backup replicas, and they are *PRE-PREPARE*, *PREPARE*, and *COMMIT*.

After the *primary* receives a request from a client, it appends a sequence number to this new request and broadcasts a *PRE-PREPARE* message to all of the replicas containing the ordered request. When a backup replica receives the *PRE-PREPARE* message, it acknowledges the message by broadcasting a new *PREPARE* message to all of the other replicas. As soon as any replica receives a quorum of $2f+1$ *PREPARE* messages, it promises to commit the request in its local history (at the sequence number appended to the request by the *primary*) by broadcasting a *COMMIT* message. Similarly, when a replica receives a quorum of $2f+1$ *COMMIT* messages, it executes the request and replies to the client. The client commits the request if it receives $f+1$ matching replies. Otherwise, the client retransmits the request.

If the request does not commit after a certain time, the protocol executes a leader election protocol to change the *primary*. This part of the protocol is not executed in the common case (synchronous network, no faults), and, thus, we do not describe it in this thesis.

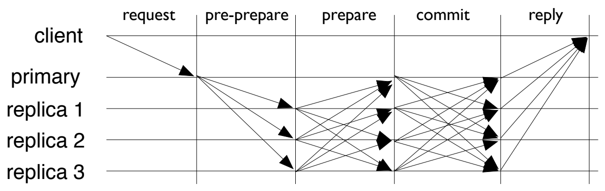

Figure 2.2: Communication pattern of PBFT.

Zyzzyva. Zyzzyva is a successful, high-throughput BFT protocol, that improved upon PBFT. Zyzzyva relies on speculation in order to achieve high performance. The speculation relates to the fact that in the common case there is no faults, and all (correct) replicas will execute requests in the correct order. Thus, there is no need to agree among replicas on the correct order of requests, and it is only necessary to trust one replica to impose the correct ordering of requests.

The communication pattern of Zyzzyva [Kotla et al., 2007] is depicted in Figure 2.3. Similarly to PBFT, Zyzzyva relies on a *primary* to order requests, and clients to issue requests to the *primary*. The *primary* assigns a sequence number to a request and multicasts it to other

[2]These protocols do not rely on a dedicated replica to order requests.

replicas[3]. All of the replicas (including the *primary*) speculatively execute the request and reply to the client. Replicas include the digest of their history in their reply. If the client receives $3f+1$ matching replies, it commits the request.

In the case where client does not receive $3f+1$ matching replies, the protocol executes a slower path, in order to reconcile local histories of the replicas. This part of the protocol is not executed in the common case (synchronous network, no faults), and we do not describe the reconciliation protocol in this thesis.

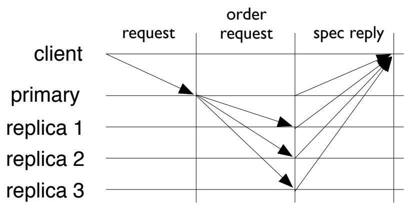

Figure 2.3: Communication pattern of Zyzzyva.

Chain. Chain is another high-performance protocol, that improves upon PBFT. Chain implements efficient pipeline topology in order to achieve high-throughput in fault-free executions. When faults occur, Chain relies on a PBFT-like, backup protocol to handle the request dissemination.

The communication pattern of Chain [Guerraoui et al., 2010a] is depicted in Figure 2.4. Chain relies on two additional, distinct replicas: the *head* and the *tail*. All replicas are arranged in a chain (hence the name of the protocol). A client sends a request to the *head*, which assigns a sequence number to the request. The *head* then forwards the request to the next replica in the chain. Each replica executes the request, appends it to its local history, and forwards the request until it reaches the *tail*. Finally, the *tail* replies to the client. The last $f+1$ replicas include the digest of their history in the forwarded request, which the tail includes in the reply to the client. If these digests match, the client commits the request. Otherwise, the client resorts to a backup protocol to commit the request. This backup protocol is shared by all of the protocols implemented in the ABSTRACT framework, and Vukolic [2008] gives its detailed description.

2.2 Related Work

In the previous section, we described PBFT, Chain, and Zyzzyva. In this section, we contrast these protocols, as well as other relevant work, to Ring, our novel, high-throughput protocol.

[3]Both Zyzzyva and PBFT implement a specific optimization for large requests, which consists in having clients multicast their requests to all of the replicas. Nevertheless, this optimization drastically decreases performance (due to IP multicast packet drops, as explained in Section 3.2).

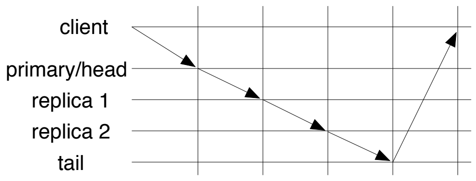

Figure 2.4: Communication pattern of Chain.

PBFT [Castro and Liskov, 1999], Zyzzyva [Kotla et al., 2007] and Chain [Guerraoui et al., 2010a] were known to be the most efficient BFT protocols in terms of throughput under high load. We show that, unlike Ring, none of these protocols features both symmetric CPU processing across replicas and balanced network utilization across different links. Moreover, our evaluation shows that Ring achieves up to 27 % higher throughput than all these protocols.

Scrooge [Serafini et al., 2010] is a primary-based protocol similar to Zyzzyva and PBFT. It reduces the number of replicas needed to achieve low-latency despite faults. In the best case, Scrooge exhibits the same performance as Zyzzyva.

Quorum-based protocols like HQ [Cowling et al., 2006], Q/U [Abd-El-Malek et al., 2005], and Quorum [Guerraoui et al., 2010a] exhibit low latency under very low load, when requests are spontaneously ordered by the LAN switch. When the load increases, these protocols fail to achieve high performance: the spontaneous order observed by the different replicas is often different, which requires replicas to be frequently reconciled, thus resulting in performance degradation.

A set of so-called robust BFT protocols have been recently designed: Aardvark [Clement et al., 2009a], Prime [Amir et al., 2008], Spinning [Veronese et al., 2009] and Zyzzyvark [Clement et al., 2009b]. These protocols aim at offering good throughput when faults occur. Unlike Ring, these protocols do not optimize performance for the non-faulty case. However, their performance under faults is better than that of Ring, and an interesting research challenge would be to design a robust version of the Ring protocol.

A very recent position paper addresses the problem of building scalable BFT protocols [Kapritsos and Junqueira, 2010]. The idea is to improve the throughput of replicated state machine protocols by executing the same protocol multiple times on different (intersecting) sets of machines. This idea is complementary to the one presented in this thesis, and indeed, to get the most benefit out of this multiple-execution mechanism, it is necessary to have a very efficient base protocol.

As we have pointed out, in some of the previous works (Ring Paxos [Jalili Marandi et al., 2010] and LCR [Guerraoui et al., 2010b]), the use of a ring topology in the context of total-order broadcast protocols has been proposed. Ring is not a simple extension of these protocols, as it tolerates Byzantine faults of both replicas and client, while Ring Paxos and LCR can only tolerate crash faults, which makes their design significantly easier. Another difference between

Ring Paxos and Ring is that the former relies on IP multicast to disseminate sequence numbers, whereas the latter does not. Finally, despite running in a crash-fault environment, Ring Paxos achieves lower throughput than Ring.

2.3 System Model

Our model and assumptions are similar to those made by all of the current BFT protocols [Castro and Liskov, 1999; Clement et al., 2009a; Cowling et al., 2006; Kotla et al., 2007].

First, we assume a message-passing distributed system using a fully connected network among participants: clients and servers. The links between processes are asynchronous and unreliable: messages may be delayed or dropped (link failures). However, we assume fair-loss links: a message sent an infinite number of times between two correct processes will be eventually received. We further assume existence of *synchronous* periods: any message sent between two correct processes is delivered within a bounded delay, known to the sender and the receiver, if the sender retransmits the message until it is delivered.

Second, we assume a Byzantine failure model in which (faulty) replicas or clients may behave arbitrarily. Replicas are assumed to fail independently, and we assume an upper bound f on the number of faulty replicas in a given window of vulnerability. There is no upper bound on the number of faulty clients.

Next, we assume the existence of a strong adversary, who may coordinate the actions of faulty nodes in an arbitrary manner. However, the strong adversary cannot see the state of correct replicas. This adversary cannot subvert standard cryptographic assumptions about collision-resistant hashes, encryption and digital signatures. Furthermore, we assume that the replicated state-machine is deterministic.

Finally, our design ensures safety in an asynchronous network which can drop, delay, corrupt, or reorder messages. Liveness is guaranteed only under eventual synchrony [Dwork et al., 1988].

3 Analysis of High-Throughput Working Conditions

Our goal is to design an effective protocol for conditions in which *the network is the bottleneck*. In order to expose and better understand the impediments of throughput scaling in the existing protocols under such conditions, we have analysed the available implementations of PBFT [Castro and Liskov, 1999], Zyzzyva [Kotla et al., 2007][1], and Chain [Guerraoui et al., 2010a]. Although we do not claim that our list is exhaustive, here we highlight the main impediments toward achieving high throughput: asymmetric replica processing, unbalanced network utilization, extraneous processing, and IP multicast packet drops.

Unsurprisingly, the main reasons for poor scalability are due to an imbalance in resource utilization — some replicas become bottlenecks before other replicas, cutting off possibilities for further performance gains. In general, the prerequisite for achieving high performance is efficient execution, in which no time is wasted on unnecessary actions. In the case of unbalanced network utilization, protocols either do not utilize all of the physical network links, or send a lot of excess data per request. Similarly, the use of IP multicast can be seen as introducing unnecessary work, due to higher chances of retransmissions, as this protocol incurs excessive packet drops under load.

Protocols performing superfluous actions may also cause a CPU to become the main bottleneck. For example, processing a large number of messages in bursts causes the kernel to waste a lot of time delivering messages to the application, if the mechanism of delivery is not efficient. Also, in certain circumstances, the memory sub-system cannot optimally transport the messages from the network card, through the kernel, and finally into the application [Menon et al., 2006]. As Dobrescu et al. [2009] and Egi et al. [2008] point out, no processor can handle the Gigabit Ethernet throughput with small messages, precisely due to the aforementioned reasons. Thus, in order to get the highest possible performance, one

[1] We could actually not conduct experiments with the original Zyzzyva code base, as (1) the implementation is incomplete, and (2) there are bugs which prevent from running experiments with a high input load. Therefore, we have used a different implementation of Zyzzyva, called ZLight [Guerraoui et al., 2010a].

needs to resort to using a multicore processor, where processing is carefully placed over multiple cores, with a clear separation of roles present [Egi et al., 2008; Kohler et al., 2000]. This thesis does not consider Gigabit Ethernet since, under that setting, the CPU becomes the first bottleneck in BFT protocols[2]. However, it is still important to mention message handling on multicores as a source of possible obstructions toward high performance, as there is new research in networking [Dobrescu et al., 2009] and deterministic execution [Birman et al., 2009] which suggests that handling Gigabit Ethernet messages may push the execution into being network-bound.

To explore and to assess the effect of each of these limitations in achieving high throughput, we have run different experiments on the Emulab [White et al., 2002] networking testbed. In each of the experiments, we have used *pc3000* machines – Dell PowerEdge 2850s systems, with a single 3 GHz Xeon processor, 2 GiB of RAM, and 4 available network interfaces. Each of the machines runs Ubuntu 8.04, with the default kernel (2.6.24-28). Every replica runs on a separate machine, while the clients are deployed over a total of 15 machines. In all of our experiments, we have used a topology in which replicas belong to one Fast Ethernet LAN, and clients communicate with replicas over a second Fast Ethernet LAN. The reason for choosing this topology is that it yields significantly better performance, especially for Zyzzyva and PBFT, which can be explained by the fact that such a topology reduces the number of IP multicast packet drops.

In our experimentation, we have used a closed-loop benchmark popularized by all of the state-of-the-art BFT protocols [Castro and Liskov, 1999; Guerraoui et al., 2010a; Kotla et al., 2007]. In this benchmark, clients are deployed on a (possibly different) set of machines, from which they issue requests in a closed-loop manner: each client issues a new request only after it has received a reply to its current request. The benchmark makes it possible to modify the size of the requests which are issued by clients and the size of the replies which are generated by the replicas.

3.1 Asymmetry in Resource Utilization

Whenever there is an asymmetry in resource utilization, one of the resources becomes the bottleneck before the others do [Bolch et al., 2005]. Such behaviour is undesirable, because this one resource effectively limits the others, and the system has underutilized, idle resources. In this section, we measure the asymmetry present in the current BFT protocols.

3.1.1 CPU Asymmetry

As we have seen in Section 2.1.3, Chain, Zyzzyva and PBFT all rely on a dedicated replica to handle incoming requests from clients. We monitor the CPU load at each of the replicas, in

[2]This is mostly due to cryptographic operations.

order to detect whether these replicas have a higher CPU load than other replicas and are, thus, bottlenecks.

In order to monitor the CPU load, we use the previously described benchmark. The clients issue 8-byte requests, and we vary the number of clients so as to inject different levels of load. Each of the clients sends 10'000 requests, and we measure the CPU load of different replicas with the `sar` utility [Godard, 2010]. The results are shown in Figure 3.1 for 40, 120 and 200 clients, respectively. For each of the protocols, the first replica (replica 0) is the one handling incoming requests (*primary* in PBFT and Zyzzyva and *head* in Chain).

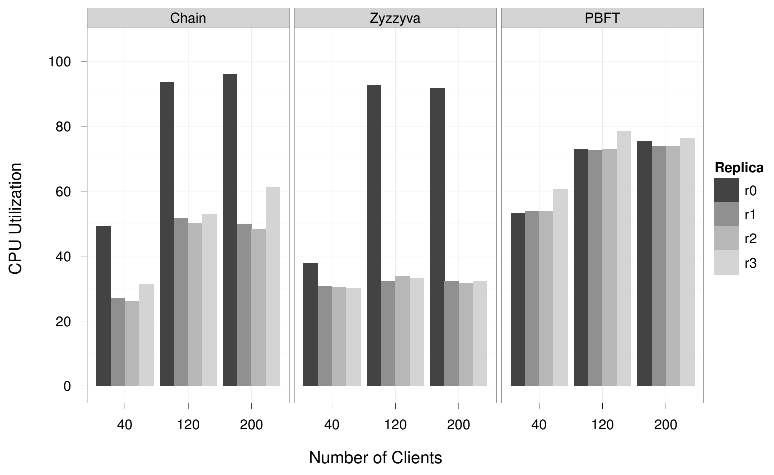

Figure 3.1: CPU utilization on different replicas, for different numbers of clients.

We observe that for each protocol, the replica receiving client requests has a higher CPU load. The difference is quite important for Zyzzyva and Chain (about 20 % higher CPU load). This can be explained by the fact that the primary (respectively, head) receives all of the client requests (while it also manages all of the client connections), thus performing more work (such as connection handling and cryptographic operations) than other replicas. Regarding Chain, we can observe that the tail also performs more work than other replicas, which is explained by the fact that it sends replies to clients. Interestingly, we remark that PBFT has lower CPU consumption than other protocols, and that the CPU usage increase observed at the primary is negligible. We explain this behaviour with the fact that, for every received message, the nodes in PBFT have 4 communication rounds involving an IP multicast. Thus, the replicas in PBFT spend more time sending requests, than actually processing them.

3.1.2 Network Asymmetry

Throughput inefficiency can also be caused by an unbalanced utilization of the available network bandwidth. More precisely, if the network links are not used equally, some may become bottlenecks and limit performance, while others can remain underutilized. Here, we

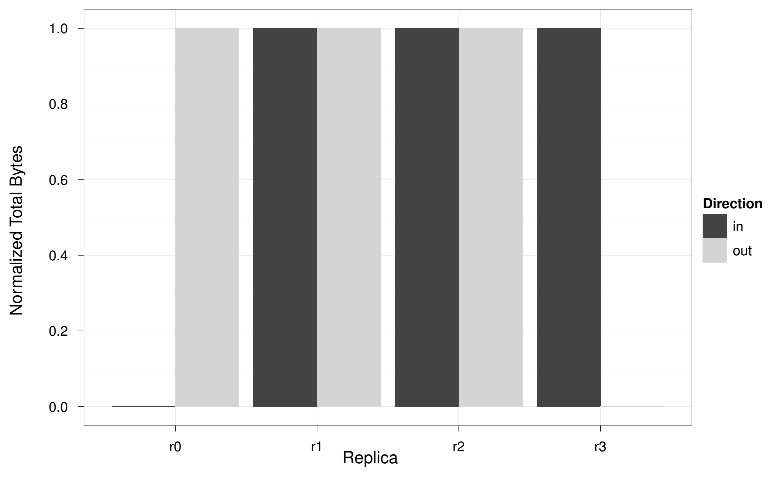

Figure 3.2: Network link utilization in Chain.

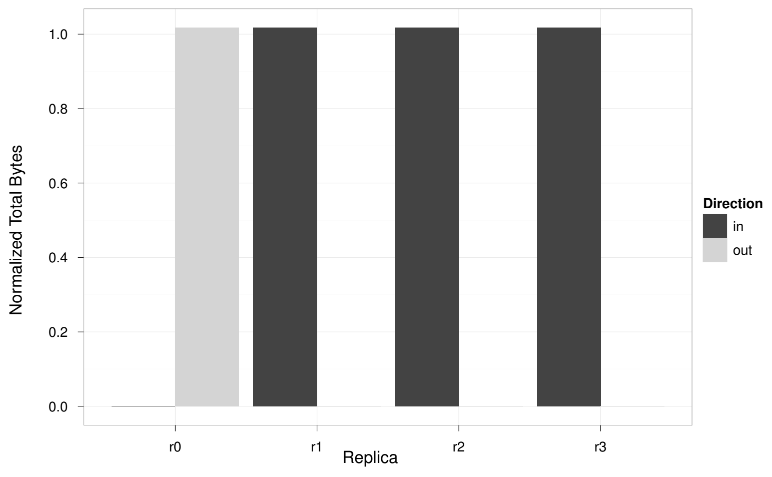

Figure 3.3: Network link utilization in Zyzzyva.

study a setup with 4 replicas. To achieve good performance, each of the replicas is equipped with two network interfaces: one for client-to-replica communications, and one for replica-to-replica communications. We monitor the number of bytes which are sent/received by the replicas for replica-to-replica communications, while the clients issue 4 KiB requests. Figures 3.2, 3.3, and 3.4 illustrate the normalized amount of sent and received bytes over each of the links. In other words, these figures shows how many bytes are sent (or received) for each byte received from a client. The bars *in* (*out*) denote the normalized amount of data on the incoming (respectively, outgoing) links to (respectively, from) the replica, that is, how many bytes were sent (respectively, received) for each byte received from a client.

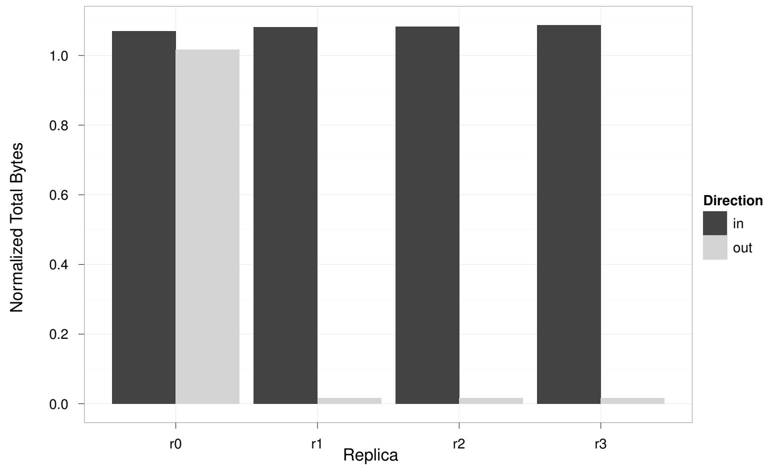

Figure 3.4: Network link utilization in PBFT.

We observe that every protocol exhibits an unbalanced network utilization as well. In Chain, the incoming link of the *head* is not used[3]. Indeed, no replica sends messages to the *head*. For similar reasons, the outgoing link of the *tail* is not used. In Zyzzyva, the *primary* only uses its outgoing link (it does not receive any messages from other replicas), whereas all of the other replicas only make use of their incoming link (they do not send messages to other replicas). Finally, PBFT uses all of the links, but the incoming link of the *primary* and the outgoing links of all other replicas are still underutilized: the slight difference with Zyzzyva stems from different amounts and sizes of *PREPARE* and *COMMIT* messages.

3.2 Protocol Inefficiencies

Another source of throughput inefficiency that we have identified is the usage of IP multicast. Both Zyzzyva and PBFT use IP multicast to send a message to a group of replicas. This optimization might, however, be hazardous to performance, due to packet drops. To quantify the potential impact of IP multicast, we run a simple experiment, in which there is a set of machines which are simultaneously multicasting messages. We vary the number of machines — 3, 6 and 9. Each of the machines multicasts 4 KiB packets to a dedicated machine, which only listens for incoming traffic (a sink). We also vary the sending rate to achieve a total aggregate throughput in the range of 70 Mbps to 110 Mbps. We choose values higher than the maximum throughput on the Fast Ethernet network (100 Mbps) to model the fact that senders cannot be coordinated in Byzantine environments. Figure 3.5 shows the changes in loss rate as the sending rate of each sender increases, for different number of senders. Some points on Figure 3.5 are annotated with the total aggregate throughput at the sink.

[3]Because all replicas have NICs, and the other NIC receives clients requests

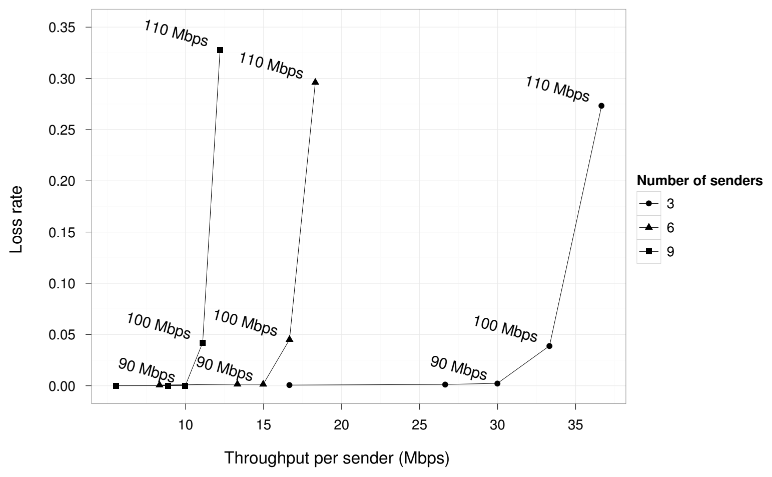

Figure 3.5: Percentage of IP multicast packet drops.

We observe that the loss rate increases non-linearly when the aggregate throughput goes above the link speed. Moreover, the loss rate increases with the number of servers in a group, although the rate stays constant. For example, with 3 servers sending at 36.6 Mbps, almost every 4th packet is dropped. In contrast, with 9 senders serving a total aggregate rate of 110 Mbps (each server sends only 12.2 Mbps), every 3rd packet is dropped (these results are consistent with similar experiments for Gigabit Ethernet networks presented by Jalili Marandi et al. [2010]).

The packet drops can be explained by the fact that IP multicast is an unreliable protocol: under high contention, either machines or the connecting switches drop excess packets [Jalili Marandi et al., 2010]. This leads to retransmissions, which, in turn, congest the network even more. Moreover, the ratio of new versus retransmitted messages drops, which lowers the throughput. These effects are known as multicast storms, and are well known to disrupt entire data centres [Birman et al., 2009; Vigfusson et al., 2010][4].

Note that, with the network topology that we use (a different Fast Ethernet LAN for clients-to-replicas communications and for replicas-to-replicas communications), multicast problems mostly affect PBFT. Zyzzyva is not affected, as there is only a single sender in the multicast group. In contrast, we have observed that in a configuration with only one Fast Ethernet LAN, Zyzzyva is affected by the clients-to-replicas traffic, which creates contention and leads to IP multicast packet drops. Notice, finally, that these experiments explain the poor performance of enabling the client-multicast optimization implemented in PBFT and Zyzzyva. Indeed, with

[4]IP multicast losses can be reduced by carefully configuring buffer sizes, and/or synchronizing distributed senders (as in the Spread communication toolkit [Amir et al., 2004]). However, this is a difficult, if not impossible task in a Byzantine environment, as malicious replicas can simply send traffic at high rate, disrupting complete communication in the group.

this optimization enabled, all of the clients can potentially multicast requests concurrently, resulting in numerous packet drops and a drastic decrease in performance.

3.3 Implementation Inefficiencies

In the previous sections, we have discussed various hindrances toward high-throughput performance, all of which stem from protocol design. In this section, we identify some of the impediments which could further limit the maximum throughput of a protocol, even if it were designed without the aforementioned inefficiencies. These inefficiencies come from the implementation and deployment. We highlight two important issues, namely connection handling and message pipelining on multicore architectures.

3.3.1 Connection Handling

High-throughput environments can sometimes be characterized by a large number of clients accessing servers. When using connection-less protocols, handling messages from clients does not require any additional considerations for the main processing loop. However, connection-less protocols may incur more message losses, implying some sort of trade-off between simplicity and performance. On the other hand, connection-oriented protocols (such as TCP) have implicit message retransmissions, but require additional considerations when handling many clients.

The main implementation problem when using a connection-oriented protocol is how to handle a large number of concurrent connections simultaneously at the server, without a degradation in performance. Even checking for availability of the data on a connection can take precious time, and with a large number of connections present, this unnecessary work becomes rather cumbersome. Therefore, when a large number of connections is present, the server needs to focus only on those connections where events (such as new data arriving) occur, and disregard any inactive connections[5]. For that reason, an efficient event-dispatching mechanism is needed to inform the server when (and where) the data is ready. The community has already tackled this problem, named C10K [Kegel, 2006], which describes this limitation observed on most of the web-servers — more precisely, the lack of ability to handle thousands of simultaneous connections. These studies have sought to improve mechanisms and interfaces for obtaining information about the state of connections from the operating system [Banga and Mogul, 1998; Libenzi, 2002; Provos et al., 2000]. As suggested by Gammo et al. [2004], the often used `select` system call does not scale well, as the kernel needs to be notified about all of the connections of interest, and return the required information. Also, `select` has a limitation in the number of connections it can monitor in one call (most commonly, 1024 connections). The same study has found that the more efficient `epoll` offers

[5]Otherwise, the main processing logic might get blocked waiting for the data to arrive.

much higher scalability, due to a separation of mechanisms for obtaining events from the ones used to declare and control interest in events.

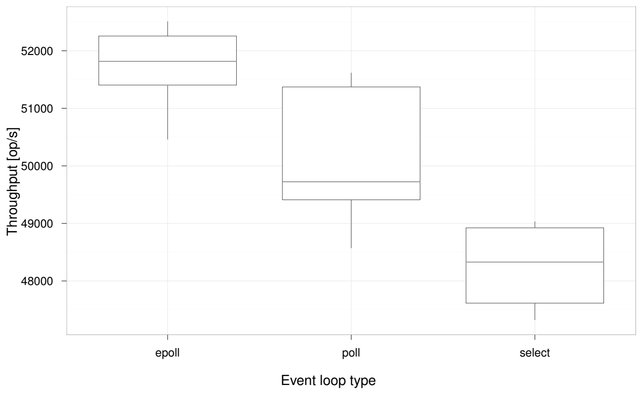

Figure 3.6: Effect of different event loops on the throughput. Boxes represent the standard deviation, while the mid-line represents the mean of the dataset. Vertical lines for each event represent the range of observed throughputs.

To measure the effect of connection handling, we have added support for different event processing loops in Chain — namely, a `select`, `poll`, and an `epoll` event loop. Based on a configuration parameter, Chain uses one of these three event processing loops. The rest of the implementation has remained unchanged. In the experiment, we have used 240 clients to connect to our set of replicas, and we have repeated each of the experiments 10 times, in order to achieve statistically significant accuracy. In order to emphasize the event-processing loop effects on CPU processing, all of the clients have sent 8 B requests (effectively saturating the CPU) in a closed-loop, and have received 8 B replies. Figure 3.6 reports the observed average throughput for these 10 runs, with standard deviation. The obtained results suggest that replacing `select` with the `epoll` event loop increases performance by 6 %, which confirms the premise about the importance of proper connection handling in order to achieve high throughputs.

3.3.2 Message Handling On Multicore Architectures

In this section, we present an important implementation detail for high-throughput processing, related to multicore architectures. Since multicore architectures are becoming prevalent, a reasonable assumption would be that all of the future BFT protocols will utilize them, in one way or another. The presented findings were obtained in a context of a different project [Knežević et al., 2010], but are also applicable to BFT protocols, as high-throughput environments usually encompass a massive processing of messages.

One of the major issues with current BFT protocol implementations is their inherent single-thread execution model, stemming from the transformation from pseudo-code to the realization. The whole pipeline of the execution occurs in a single thread: message verification, message handling, execution within the state machine, and message authentication. This represents a serious bottleneck, and may be changed in the future [Birman et al., 2009]. However, even in such configurations, once the application issues a `send` call to dispatch the request, the kernel takes over the message, and may perform the actual sending on a different core. Even if the execution pipeline is partitioned over multiple threads, these different processing contexts may occur on different cores. Either way, such a behaviour creates stress on the memory subsystem, possibly reducing overall performance, as it increases the chance of the CPU becoming the bottleneck. Thus, a well-thought, efficient implementation and a careful placement of processing contexts are necessary.

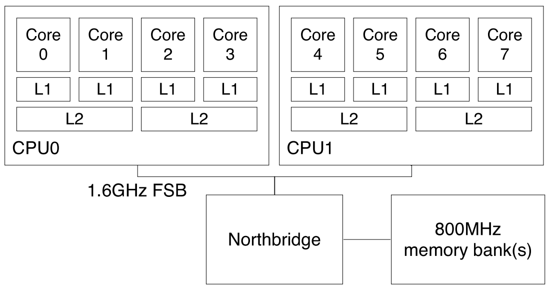

Figure 3.7: CPU and memory architecture of the Intel SR1560 server.

In order to evaluate the impact of a processing unit placement inside a single server, we have conducted the following experiment. The hardware platform is an Intel SR1560 Series rack 1U server with 2 Intel Quad-Core Xeon X5472 processors running at 3 GHz (Figure 3.7). Each CPU is equipped with 12 MiB of L2 cache (2 × 6 MiB). The machine has 8 GiB of 800 MHz RAM that is accessible over the 1600 MHz Front Side Bus. Figure3.7 also shows the memory architecture of the Intel SR1560 server. Each core has its own small L1 cache, and L2 caches are shared by pairs of cores. Each CPU is connected to the memory banks through the northbridge, via one Front-side bus. The major obstacle in achieving high throughputs in such an environment is the hierarchical structure of the memory, and the limited throughput of the memory bus.

For this experiment, we have used FreeBSD-RELEASE-p3 7.1, and all of the measurements were done for processing inside the kernel. On each of the cores, we have run a Click 1.6.0 instance [Kohler et al., 2000], with our patches so that it could work under FreeBSD. Each of the cores did some simple processing of a UDP packet (checksumming, destination checking).

We have used a pipelined, source-sink topology in this experiment, where router 0 generates messages (the source), while router 3 discards them (the sink). Each of the routers on this virtual topology takes the message from the previous router in the pipeline. Several configurations that we have used in this experiment are shown in Figure 3.8. In this topology, the message is copied from one L2 cache to the L2 cache of the processing router. If the next core in line to process the message does not share its L2 cache with the previous core, the modified portion of the message is transferred back to the main memory and, subsequently, the whole packet will be copied to the L2 cache of the following core. As these actions repeat for each of the hops in the topology, significant memory bandwidth is lost on these transfers. Moreover, the whole system is slowed down, because the new messages can not be transferred, as the memory subsystem is working at its peak throughput. Thus, it is important to keep the packets in L2 cache as much as possible, or, in other words, process them locally as much as possible.

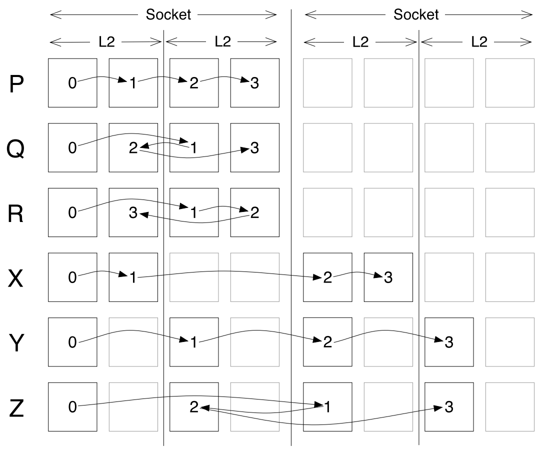

Figure 3.8: Description of different pinnings used in the 4-stage router processing pipeline experiment (Figure 3.9).

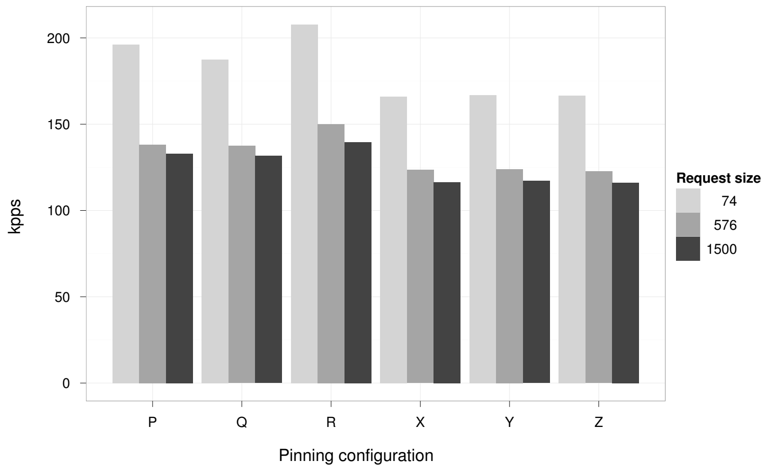

Figure 3.9: Throughput of a 4-stage router processing pipeline, using different router-to-core pinnings.

Figure 3.9 shows how the throughput changes as we vary the placement of the routers (depicted in Figure 3.8). For each of the configurations, the throughput is shown for 74, 576 and 1500 B Ethernet packets[6]. As expected, larger messages lead to an increase in overall throughput that

[6]The minimum, the average, and the maximum size of Ethernet packet.

the system can achieve, although the number of messages drops. The reason for this is the fact that the routers spend more time processing larger packets, thus reducing the stress on the memory subsystem.

In the extreme case, the configurations on the right-hand side of the bar chart exercise more packet transitions from one CPU socket to another, putting stress on the memory subsystem. As seen in Configurations X, Y, and Z, as soon as we start shipping packets between different CPU sockets, the forwarding performance starts to decrease. In Configurations P, Q, and R we can see a smaller drop in performance, although their number of transitions is the same as in Configurations X, Y, and Z, due to Intel's Cache snooping filter [Intel Corporation, 2007], which saves memory bandwidth if the data is in a nearby cache.

As shown by this experiment, thread (or processing) placement plays an important role in handling high-throughput workloads. The reason for this is that the memory subsystem may become the bottleneck, due to increased data traffic. An increase in the number of L2 caches touched by a single message results in lower performance than in the case involving just a single processing core.

3.4 Summary

In this chapter, we have shown that all of the protocols suffer from having underutilized replicas, and use the network in an unbalanced way. Every protocol uses only at most 50 % of the available links, or (in case of PBFT) has moderate overheads per byte of the request of the client. Additionally, all of the protocols except PBFT suffer from asymmetric replica processing. Moreover, PBFT is subject to IP multicast losses, and there is evidence [Guerraoui et al., 2010a] that without IP multicast, the performance of PBFT suffers significantly. Finally, we have shown that, even if a protocol avoids the asymmetry in processing, special considerations must be taken during implementation, to avoid extraneous processing or the creation of bottlenecks.

4 Ring Design and Implementation

In this chapter, we present Ring, a novel high-throughput BFT protocol. The design of Ring is influenced by the observations reported in the previous chapter, and Ring constitutes the main contribution of the thesis. As its name indicates, Ring uses a ring topology for message dissemination between replicas. The task of handling Byzantine faults on a ring topology is complex, as the protocol must ensure that: (1) no replica in the ring can be bypassed, (2) Byzantine clients sending malformed requests cannot corrupt the total order on correct requests, and (3) the reply sent by the last replica[1] in the Ring is not forged.

At the conceptual level, Ring consists of two operational modes: a *fast* mode, which is executed when there are no replica faults, and a *resilient* mode, which is executed only when one or more replicas in the Ring are faulty. Both of the modes can work in the presence of faulty clients. The description of the switching mechanism and the switching conditions are also described in this chapter.

We begin the chapter by giving an overview of our protocol, followed by an in-depth description of the *fast* and the *resilient* mode. Finally, we provide correctness proofs for both of the operational modes.

4.1 Protocol Overview

Ring is named after the ring topology it uses for communications between replicas. Unlike most of the BFT protocols, Ring does not use IP multicast: it only relies on a unicast message exchange. Each replica in Ring has exactly one predecessor, and exactly one successor. Communication flows in only one direction over the ring, with each of the replicas forwarding requests only to its successor. Ring clients send requests to any replica on the ring, and receive

[1] This is the replica which replies to the client.

responses from that replica's predecessor. We assume that all of the nodes[2] know the identities of all of the participating replicas, and that all of the replicas know the identities of all of other nodes. One replica in the Ring, called the *sequencer*, is in charge of assigning a sequence number to each of the new requests that it receives.

As stated previously, Ring has two operational modes: a fast mode and a resilient mode. The fast mode is very efficient during executions in which there are no faulty replicas[3]. Note that, in the fast mode, unlike some of the other protocols, Ring continues committing requests even if there are faulty clients. The resilient mode, on the other hand, ensures progress in the presence of faulty replicas, and Ring uses the ABSTRACT framework [Guerraoui et al., 2010a; Vukolic, 2008] to switch between the two modes when faults are detected. An overview of the ABSTRACT framework is given in Section 4.2.

In the context of this framework, a BFT protocol is an infinite sequence of different, predetermined instances of the ABSTRACT protocol. Each of the instances runs until its own correctness and progress conditions have been satisfied. Otherwise, the instance stops, and the system switches to another instance. The order of the instances is deterministic, known to all of the nodes — in this way, the nodes do not need to run an agreement on the next instance to switch to.

Although Ring contains two operational modes, one ABSTRACT instance corresponds only to a single operational mode. Therefore, performing a switch in Ring could be viewed as the following process:

1. A quorum of replicas in the current instance agrees on the switch.

2. The system instantiates a new instance of the *opposite operational mode*, possibly with a different set of nodes.

3. All nodes of the current instance abort the incoming requests from the clients, effectively stopping the instance. The aborted requests carry the state necessary to initialize the new instance.

Thus, one can observe the execution of Ring as an infinite sequence of alternating instances of fast and resilient operating modes. Whenever we instantiate a new instance of an operating mode in Ring, we also increase the *invocation number* of that mode. The invocation number denotes the number of times a switch to the particular operational mode has been performed, during the lifetime of the system.

[2]Term "nodes" encompasses both replicas and clients.
[3]We use the term "faulty replica" (respectively, "faulty client") to refer to the origin of the fault in the system.

4.1.1 Switching Between Instances

Ring alternates between the fast and the resilient mode in the following way: it first runs in fast mode, with high performance, until a fault occurs. Once a fault has occurred on one (or more) of the replicas, the system switches to an instance in resilient mode. The reason for the switch is that an instance of fast mode does not tolerate faults on the replicas, although it can detect them. Since the resilient mode does not ensure the highest performance possible, Ring tries to stay in this mode for as little as possible. In our design, Ring stays in the resilient mode until it has processed 2^k requests, or until at least $2f+1$ replicas have voted for changing the mode. The parameter k represents the invocation number of the resilient mode. It is reset after reaching a certain threshold, in order to prevent the system to spend long time in the resilient mode.

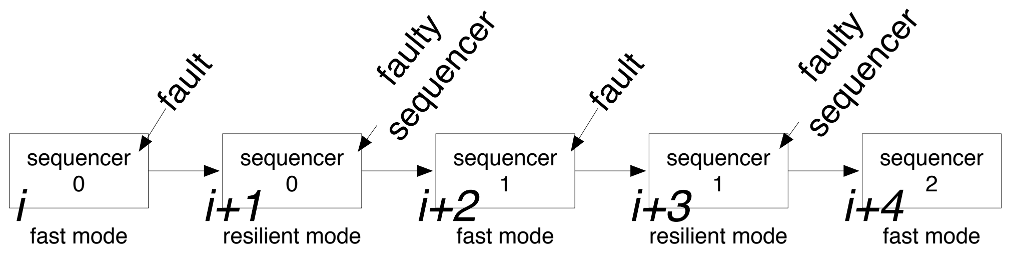

Figure 4.1: Alternating modes in Ring. Each of the boxes represents one instance, and the number in the lower left corner denotes the instance number. Ring runs in fast mode, until a fault[4] occurs, when Ring switches to resilient mode, using the same set of replicas, with the same identities. If a fault occurs in this mode, and the fault happens to be at the sequencer, Ring switches to a new instance of fast mode, with a different configuration.

In Figure 4.1, we illustrate the switching order in Ring. In fast mode, Ring tolerates faulty clients, although with slightly reduced performance than in situations in which there are no faults at all. When a fault occurs, Ring switches to resilient mode, keeping the same configuration (the same set of replicas, and the same identities between nodes). In the resilient mode, Ring tolerates faulty replicas. However, if the sequencer is faulty or the network is slow, the given instance will not progress (this is the *progress condition* of the resilient mode). In that case, the replicas in Ring will be able to detect this faulty situation, and switch to another instance of fast mode but, this time, with a different configuration. In the new configuration, nodes assume new identities, and a new (physical) node is appointed as the sequencer. Most importantly, the nodes have to establish new peering connections. All of these constrains are met by having a predetermined sequence of instances[5]. That way, the cost of switching is minimized (as there are no additional steps to agree on new sequencer), and overheads are reduced. The benefit of using the strategy to change the topology only when switching from the resilient mode

[4]A fault that the fast mode can not tolerate, such are faults on replicas.

[5]Each instance has its own associated configuration: the set of nodes, their peering connections (topology), and running parameters.

instance to the fast mode instance is twofold: (1) it allows Ring to end up in an instance of resilient mode in which the sequencer is correct, and (2) it avoids frequent, lengthy, topology changes[6].

4.2 The ABSTRACT framework

Since the implementation of Ring uses ABSTRACT, or Abortable Byzantine faulT-toleRant stAte maChine replicaTion, we will first present it. ABSTRACT is a new generic abstraction which significantly reduces the development and maintenance cost of BFT algorithms, and makes it considerably easier to develop efficient ones. ABSTRACT resembles state machine replication, and it can be used in order to make any shared service Byzantine fault-tolerant, with one exception: it may sometimes abort a client request. The (non-triviality) condition under which Abstract cannot abort is a generic parameter.

ABSTRACT allows for the composability of protocols: when a particular instance aborts a client request, ABSTRACT returns an unforgeable (digitally signed) request history, which can then be used by the client to perform a "recovery", using another instance of ABSTRACT. Any composition of ABSTRACT instances is possible, and thus, one is free to build modular, composable BFT protocols, which have a small number of lines of code. For example, Guerraoui et al. [2010a] reports that they needed less than 5000 lines of code to implement a protocol which mimics the behaviour of Zyzzyva [Kotla et al., 2007].

As stated previously, an important property of ABSTRACT is that it can abort a request made by a client. In that case, ABSTRACT returns the *abort history*. The client, in turn, uses this abort history to *switch* to another ABSTRACT instance. In this section, we will describe the switching mechanism and necessary formalism. For the thorough treatment of ABSTRACT, we refer to dissertation by Vukolic [2008].

4.2.1 Overview

Every ABSTRACT instance is uniquely identified through its instance number i, and we will, in the continuation of this text, consider these two to be synonyms. When i commits a request, i returns a state-machine reply to the invoking client. Like any state machine replication scheme, i maintains a local history h of committed requests. This local history represents a total order of all of the committed requests, and the reply to the client is put together using this order. In the case in which i aborts a request, it returns to the client a digest of the history of requests that were committed by i (possibly along with some uncommitted requests); this digest is called an *abort history*. In addition to the *abort history*, i returns to the client the identifier of the next instance ($next(i)$) which should be invoked by the client: the next function has to be the same across all abort indications of instance i, and this we denote by we saying that instance i *switches* to instance $next(i)$. ABSTRACT considers only static

[6]Given the assumption that the probability of having a faulty sequencer is less that one third.

switching: the function *next* is always a pre-determined function (e.g., known to the servers which implement a given ABSTRACT instance).

Once the client has obtained an abort indication, it uses the abort history h of i, provided by i, to invoke $next(i)$. However, $next(i)$ considers h to be an *init history*, and $next(i)$ uses h to establish its initial local history, before committing or aborting any requests. Through this initialization step, $next(i)$ obtains the information about the requests committed within instance i, and in this way, the total order of all of the committed requests is preserved across all of the instances.

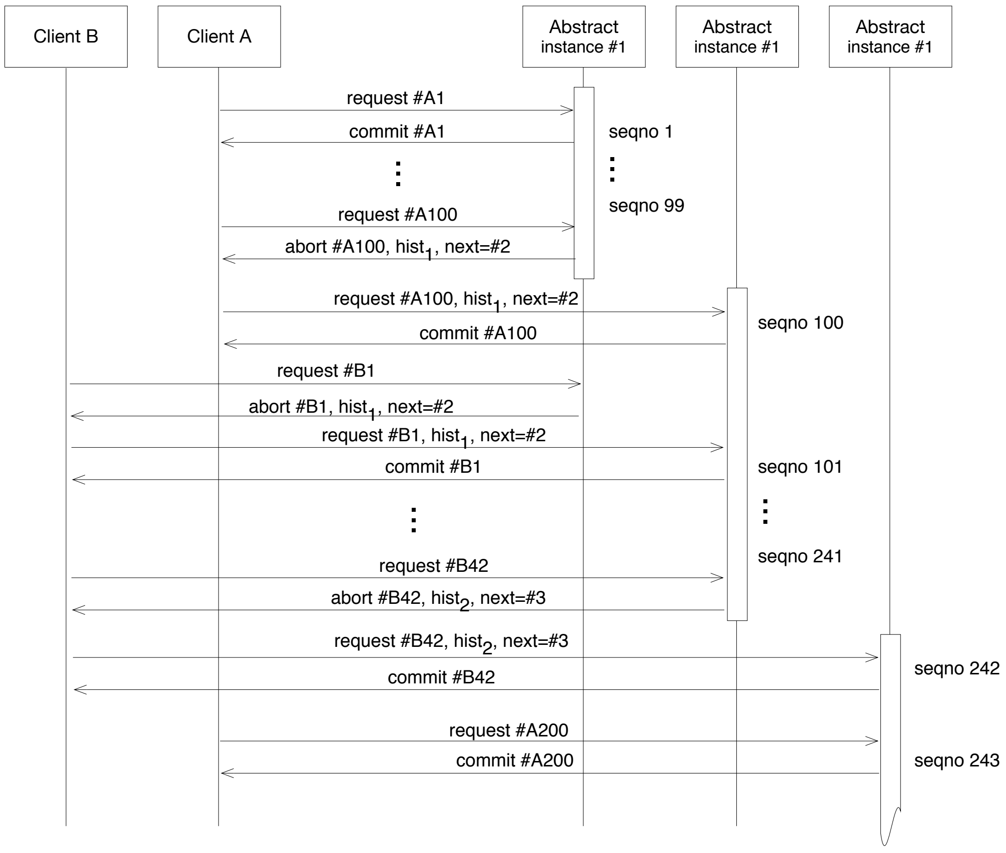

Figure 4.2: Time line diagram of operations in ABSTRACT.

Figure 4.2 depicts a possible run of a BFT system built using ABSTRACT. To preserve consistency, ABSTRACT properties ensure that, at any point in time, only one ABSTRACT instance, called *active instance*, may commit requests. In Figure 4.2, Client A starts sending requests to the first ABSTRACT instance #1. This first instance commits the requests #A1 through #A99, but aborts the request #A100, effectively becoming an inactive instance. In the abort indication to Client A, ABSTRACT instance #1 includes an unforgeable history $hist_1$, as well as the information about the next instance to be used ($next$ = #2). Now, Client A sends this information along with its aborted request to ABSTRACT instance #2. This instance then initializes its own local history, executing request #A100 at position 100. Roughly at the same time, Client B tries to submit its request #B1 to an inactive instance (the first ABSTRACT instance), which returns an abort indication, forcing the client to resend the request (again, along with the abort history) to ABSTRACT instance #2. This instance executes the request of Client B at position 101. This active instance then continues to commits further requests, until the request #B42 from Client B, which it aborts. Analogously to the previous switch, instance #2 returns a new abort history ($hist_2$), along with the next instance to be used ($next$ = #3) to Client B. Similarly to Client A earlier, Client B uses this information to invoke the next ABSTRACT instance #3, which executes and commits the request #B42 at location 242 of its local history. Finally, Client A sends the request #A200 to the third instance, which successfully commits this request at position 243. Note that Client A directly accesses the third instance, unlike Client B, which initially first contacted ABSTRACT instance #1. This is possible if Client A knows which instance is active, or if all three of the ABSTRACT instances are implemented over the same set of replicas: replicas can then, for example, "tunnel" the request to the active instance.

4.2.2 Formal Specification of ABSTRACT

ABSTRACT (without initialization) is defined as follows:

Definition *(Abstract)* ABSTRACT is a State Machine Replication algorithm that exports one operation:

- *Invoke(m)* — we say the client invokes the request m.

Every ABSTRACT instance returns two indications for the *Invoke(m)* operation:

- *Commit(m, h)*, and
- *Abort(m, h)*.

We say the client *commits* (*aborts*) the request m with history h, where h is a sequence of requests that the client can use to compute a reply (respectively, to recover). If the client *commits* (respectively, *aborts*) m with history h, we refer to h as the *commit history* (respectively, *abort history*).

ABSTRACT ensures the following properties:

1. *(Termination)* If a correct client invokes a request m, it eventually commits or aborts m with h, and h contains m.
2. *(Commit Ordering)* Let h and h' be any two commit histories: either h is a prefix of h' or vice versa.

3. *(Abort Ordering)* Every commit history is a prefix of every abort history.

4. *(Validity)* In every commit/abort history h, no request appears twice and every request was invoked by some client.

5. *(Non-Triviality)* If a correct client invokes a request m and some predicate NT is satisfied, the client commits m.

The *Non-Triviality* property is generic; the undefined predicate NT may vary depending on the design goals and the environment in which a particular ABSTRACT implementation is to be deployed.

The property that defined the behavior of ABSTRACT in the case of *abort* is *Abort Ordering*. Intuitively, ABSTRACT returns histories that represent the ordering of the clients requests. In case of a *commit*, this ordering is definitive and the reply of the implemented object is uniquely determined by the order of the requests in the history. This is not the case with the abort history. As specified by *Abort Ordering*, every abort history contains every commit history as its (non-strict) prefix; that is, *Abort Ordering* prevents any new request, invoked after some request is aborted, from being committed.

4.2.3 ABSTRACT **Initialization and Composition**

An instance of ABSTRACT could be used on its own. However, a particular ABSTRACT becomes practically useless after aborting even a single request, since it must also abort every subsequent request. Therefore, an ABSTRACT instance is much more interesting when composed with other ABSTRACT instances. Hereafter, we consider multiple ABSTRACT instances, combined to work for a "common good", with the ultimate goal of producing a flexible, full-fledged BFT state machine replication algorithm that can benefit from good performance under different scenarios.

We assume a fixed, predetermined, ordering among ABSTRACT instances, known by all processes in the system. This ordering is used by clients to know which ABSTRACT instance i' to invoke after aborting from a specific ABSTRACT instance i; we talk about a client *switching* from ABSTRACT instance i to i'. We call i the *preceding* ABSTRACT for i' (respectively, i' is the *succeeding* ABSTRACT for i).

Clients use abort histories received from the preceding ABSTRACT to invoke a special *INIT* request, used to initialize a specific instance of ABSTRACT (Fig. 6.1). *INIT* request invocations have the form *Invoke* (m, h_i), where m is the request aborted by the preceding ABSTRACT i, and where h_i is the corresponding abort history; in the context of the *INIT* request invocation, h_i is called *init history*.

We enhance ABSTRACT properties of Definition 4.2.2 to account for *INIT* requests by: (a) slightly modifying the notion of an invoked request in the *Validity* property to include any request contained in an *init history* h_i of any *INIT* request invocation *Invoke* (m, h_i), and (b) by adding the following property:

6. *(Init Ordering)* Any common prefix of init histories of all invoked *INIT* requests is a prefix of any commit or abort history.

Moreover, client invocations must be *well-formed*, meaning that before invoking a "non-*INIT*" request, correct client c must invoke an *INIT* request. Notice that an *INIT* request may be committed or aborted (according to the specification of a particular ABSTRACT), just like any other request.

ABSTRACT **Switching**

As stated previously, every ABSTRACT instance has a unique identifier (called instance number) i. When an instance i commits a request, i returns a state-machine reply to the invoking client. Like with all state machine replication schemes, i establishes a total order on all committed requests according to which the reply is computed for the client. If, however, i aborts a request, it returns to the client a digest of the history of requests h that were committed by i (possibly along with some uncommitted requests); this is called an abort history. In addition, i returns to the client the identifier of the next instance ($\mathit{next}(i)$) which should be invoked by the client: next is the same function across all abort indications of instance i, and we say instance i switches to instance $\mathit{next}(i)$. We describe deterministic (static) switching: *next* is a pre-determined function (that is, known to servers implementing a given ABSTRACT instance).

The client uses abort history h of i to invoke $\mathit{next}(i)$; in the context of $\mathit{next}(i)$, h is called an *INIT* history.

Once i aborts some request and switches to $\mathit{next}(i)$, i cannot commit any subsequently invoked request. We impose switching monotonicity: for all i, $\mathit{next}(i) > i$. Consequently, ABSTRACT instance i that fails to commit a request is abandoned and all clients go from there on to the next instance, never re-invoking i. Thus, we add the following property to ABSTRACT:

7. *(Switching Monotonicity)* For every ABSTRACT instance i, $i < next(i)$.

This concludes the formal specification of ABSTRACT. The next sections cover the implementation of Ring in the context of ABSTRACT.

4.3 Fast Mode

In this section, we give an overview of the fast mode, followed by a description of Ring Authenticators, and a detailed description of the implementation.

As previously stated, the role of the fast mode is to provide highest performance in the periods during which there are no faults, and when the computation is not the bottleneck. Hence, in this mode, Ring uses lightweight authenticators, called Ring Authenticators, to reduce both the handling and the computational complexity of message processing. Specifically, Ring Authenticators are a collection of Message Authentication Codes (MACs). As such, Ring Authenticators allow for low-overhead checking of the correct handling of the request over the ring.

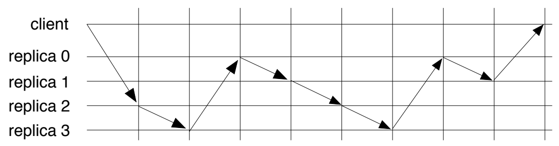

Figure 4.3: Ring communication pattern in fast mode.

The message pattern used in the fast mode is shown in Figure 4.3. A client submits a request to *any* of the replicas, which is called the *entry* replica for that particular request (for instance, replica 2 is the entry replica for the request in the given figure). Each of the submitted requests is then forwarded around the ring, until it reaches the predecessor of the entry replica (replica 1 in the example is the predecessor of the entry replica). At the end of this first round, each of the replicas stores a copy of the request. This request flow is represented by a thick black line. The thickness of the line represents the fact that in the first round replicas may exchange large messages.

Consequently, in this request flow, the request has passed through the sequencer (replica 0 in the example on Figure 4.3, and was assigned a sequence number. This sequence number is added to the header of the message. In order for each of the replicas to know of this sequence number, the predecessor of the entry replica, called the *exit* replica (for that particular request) generates an acknowledgement (*ACK*) for the request, that is forwarded around the ring. The *ACK* message only contains the header of the message, and its flow is denoted, in Figure 4.3, by a thin line, used to represent the fact that the size of these acknowledgements is minuscule. The acknowledgement message is forwarded until it reaches the exit replica (replica 1 in the example). This replica then replies to the client. Note that each of the replicas executes the request only after it has received the acknowledgement message.

The protocol ensures that none of the replicas is bypassed, and that the messages are not corrupted by replicas before being forwarded around the ring. This is achieved using Ring Authenticators (RA), which share some similarities with Chain Authenticators, presented in [Guerraoui et al., 2010a], but with significant differences, due to the presence of ACK messages.

In the case in which a client does not receive a correct reply (see the last step in Figure 4.4), or in the case in which the client does not receive a reply at all, the client sends a *panic* message to all of the replicas, after a given timeout. A *panic* message contains the uncommitted request which has timed out without being committed. The goal of this *panic* message is notify the replicas about a possible fault, and possibly to switch from fast mode to resilient mode. Byzantine clients might deliberately generate fake *panic* messages in order to force the system to switch to resilient mode. To prevent this attack, before switching to resilient mode, Ring uses the following, novel mechanism: upon receiving a *panic* message from a client, a replica handles the request *on behalf* of the client. It forwards the on-behalf request (*OBR*) to the sequencer, waits until the request gets processed along the ring, (possibly) receives the response and replies to the client. If the replica also does not receive a suitable response, a conclusion can be drawn that it is indeed necessary to switch to resilient mode. The replica then broadcasts a message to the other replicas, requesting of them to switch to resilient mode. As soon as $2f + 1$ replicas send such messages, Ring switches to resilient mode.

4.3.1 Ring Authenticators

Ring Authenticators are implemented using message authentication codes (MACs). Roughly speaking, to tolerate f faults, each of the replica generates (verifies) $f+1$ MACs for (respectively, from) its $f+1$ successors (respectively, predecessors).

Message Authentication Codes (MACs)

BFT protocols, in general, rely on cryptography in order to establish the identities of the nodes in the communication, as well as to provide the integrity of messages. The two methods used in practice are digital signatures, and Message Authentication Codes. Ring uses both, depending on the situation.

Digital signatures are computed using public-key cryptography. In communication schemes involving digital signatures, the sender of a message computes a signature, which is a function of the message and the sender's private key. Before sending, the sender appends the signature to the message. Then, upon receiving the message, the receiver can verify authenticity and integrity of the message, by verifying the signature using the public key of the sender.

Since only the sender knows the signing key, and the verification key is public, the receiver can also convince a third party that the message is authentic. It can prove the message was sent by the original sender by simply forwarding the signed message to that third party. However, the drawback of using public-key cryptography lies in its performance — all too often, digital signatures prove to be too slow for practical use.

On the other hand, MAC-based communication schemes use symmetric cryptography to authenticate communication between two parties which share a secret key. The sender of a message computes a MAC, which is a function of the message and the key it shares with the receiver. Prior to sending, the sender appends this (usually small in size) MAC to the message. Upon receiving the message, the receiver can check the authenticity of the message by computing the MAC in the same way as the sender, and comparing it to the one appended to the message. MACs prove to be quite fast, sometimes more than two orders of magnitude faster than digital signatures.

However, MACs are not as powerful as signatures: the receiver is not able to convince a third party that the message is authentic. This is a fundamental limitation, inherent to the symmetry of a MAC computation. The third party is unable to verify the MAC because it does not know the key which was used to generate it. The use of MACs to authenticate client requests also raises additional problems. It may be possible for some of the replicas to be able to authenticate a request, while the others could not. This can lead both to safety violations and liveness problems, and we explain how we deal with these problems in Section 5.2.

Formally, we assume that there exists a pair of secret keys sk for each pair of nodes i and j, such that $sk_{i,j}$ is used to compute the MACs for messages sent from i to j, while $sk_{j,i}$ is used for the opposite direction. Digital signatures require that each node has its own private and public key. Formally, each node i has pki_sk_i as a private key, and pki_pk_i as a public key. We further assume that these keys (except the public keys, that are public) are disseminated between all of the nodes in a secure manner, and further distribution and synchronization is treated as orthogonal to the topic of this thesis.

Point-to-point messages contain only a single MAC. If messages are broadcast to multiple recipients, or if the recipients forward a message, then the message contains *authenticators* — a vector of MACs, one for each of the recipients. The receiver verifies the authenticity of the message by checking the corresponding MAC in the authenticator. Similarly to other protocols, Ring uses authenticators, as replicas forward the message around the ring.

Although MACs can not simply replace [Aiyer et al., 2008; Castro, 2001; Rompel, 1990] digital signatures, and even though MACs are less powerful, they are still commonplace in all of the modern BFT protocols, because of their performance. Consequently, Ring utilizes MACs in fast mode, as it aims for high performance in the situations in which the CPU is not the bottleneck.

Ring Authenticators in Action

Figure 4.4 illustrates the flow of a request in Ring, with the involved MAC operations, when the resilience is 1 ($f = 1$). The red *underlined* labels represent generated MACs, while the green *strikedthrough* labels represent verified MACs in that operation. In step 1 of Figure 4.4, a client chooses replica 2 as its entry replica, and sends the request. The client generates two MACs, one for replica 2, and one for replica 3 ($f + 1$ MACs in total), in order to tolerate against f faults on subsequent replicas). These two MACs represent the Ring Authenticator (RA) generated by the client. Replica 2 receives the message and verifies a single MAC added by the client. In step 2, replica 2 generates its RA – containing two MACs, one for replica 3, and one for replica 0 – and forwards the request to replica 3. Replica 3 receives the request, verifies the MAC from the client, and one MAC from its predecessor – replica 2. Steps 3 and 4 are similar to Step 2.

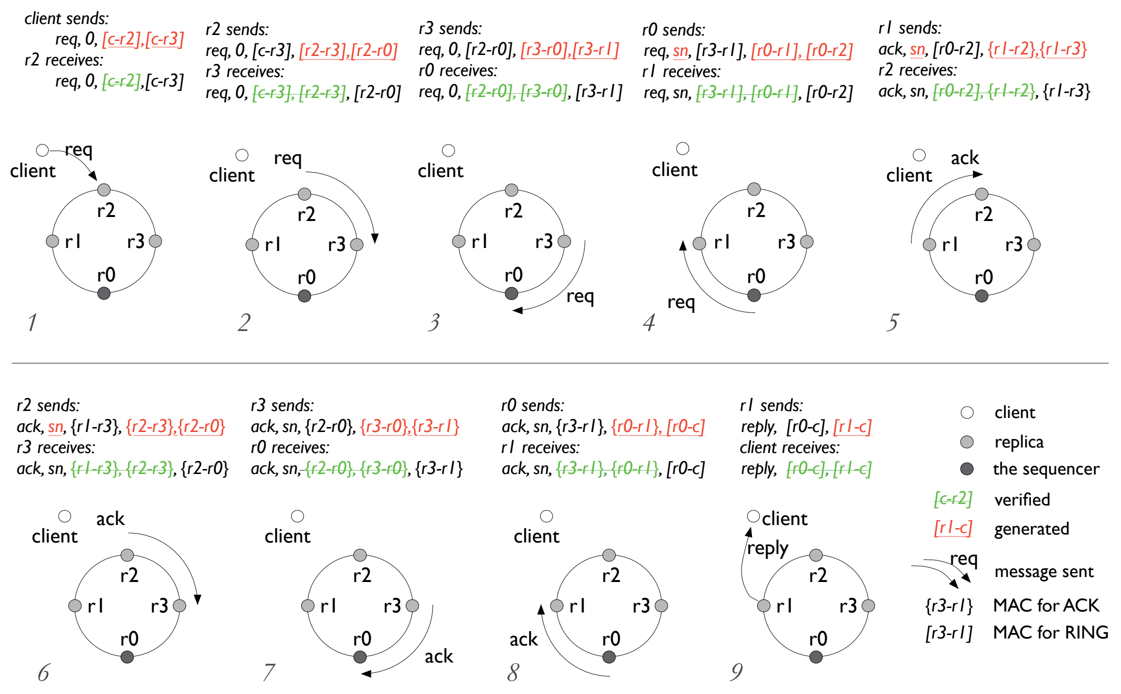
client sends:
req, 0, [c-r2],[c-r3]
r2 receives:
req, 0, [c-r2],[c-r3]
r2 sends:
req, 0, [c-r3], [r2-r3],[r2-r0]
r3 receives:
req, 0, [c-r3], [r2-r3], [r2-r0]
r3 sends:
req, 0, [r2-r0], [r3-r0],[r3-r1]
r0 receives:
req, 0, [r2-r0], [r3-r0], [r3-r1]
r0 sends:
req, sn, [r3-r1], [r0-r1], [r0-r2]
r1 receives:
req, sn, [r3-r1], [r0-r1], [r0-r2]
r1 sends:
ack, sn, [r0-r2], {r1-r2},{r1-r3}
r2 receives:
ack, sn, [r0-r2], {r1-r2}, {r1-r3}
client
req
ack
r0
r1
r2
r3
1
2
3
4
5
r2 sends:
ack, sn, {r1-r3}, {r2-r3},{r2-r0}
r3 receives:
ack, sn, {r1-r3}, {r2-r3}, {r2-r0}
r3 sends:
ack, sn, {r2-r0}, {r3-r0},{r3-r1}
r0 receives:
ack, sn, {r2-r0}, {r3-r0}, {r3-r1}
r0 sends:
ack, sn, {r3-r1}, {r0-r1}, [r0-c]
r1 receives:
ack, sn, {r3-r1}, {r0-r1}, [r0-c]
r1 sends:
reply, [r0-c], [r1-c]
client receives:
reply, [r0-c], [r1-c]
reply
6
7
8
9
client
replica
the sequencer
[c-r2] verified
[r1-c] generated
message sent
{r3-r1} MAC for ACK
[r3-r1] MAC for RING

Figure 4.4: Illustration of Ring authenticators ($f = 1$).

In step 5 of Figure 4.4, replica 1 (the exit replica, or the predecessor of the entry replica) generates an ACK for the given request, and forwards the acknowledgement to its successor – replica 2. Before sending the ACK, replica 1 generates MACs for replica 2 and replica 3. Replica 2 receives the ACK, and verifies the MAC from replica 0 (generated for the request the replica already received), and a MAC from replica 1. In step 6, replica 2 forwards the ACK, after generating MACs for replica 3 and replica 0. Steps 7 and 8 are similar. Finally, in the last step, replica 1 verifies MACs for the ACK from replica 3 and replica 0. Replica 1 then generates one MAC for the client, and sends the reply to the client. The client receives the reply and verifies two MACs – one from replica 0, and one from replica 1. If these MACs are correct, the client commits the reply.

4.3.2 General Notation

In this section, we describe the notation used in the specification and the pseudo-code implementation of Ring.

Nodes in Ring use Message Authentication Codes (MACs) to authenticate data between two entities. MACs are based on the symmetric key cryptography — only the sender and the receiver share a key that is known only to these two entities. In order to used MACs, nodes in Ring rely on two operations: *authenticate* and *verifyauth*.

A node p may authenticate a part (as a bit-string) x of a message m (or full message) for the node q by using *authenticate(p, q, x)* operation. This operation returns an authenticator a. The *verifyauth(q, p, x, a)* operation takes the identity of the receiver node q, the sender node p, a bit-string x, and a putative authenticator a as parameters. Then, *verifyauth* returns boolean $\top$ (true) if and only if *authenticate(p,q,x)* $= a$. Since MACs are based on a shared secret, successful call to *verifyauth(q,p,x,·)* by q confirms that there was a previous call to *authenticate(p,q,x)* by p, that authenticated the same data x. Since MACs are based on fast symmetric cryptographic primitives in practice, they can be computed and verified very fast.

A digital signature scheme provides data authentication in systems where nodes do not need to exchange secrets beforehand. Digital signature scheme relies on associating a public and a private key to each node in the system. As the name implies, the node must keep the private key as a secret, while the public key can be shared with anyone. A node can use its private key to generate a signature for any content. Anyone with access to the public key of the node can verify that signature and confirm the identity of the signer. Ring uses digital signatures for the cases where multiple recipients need to receive the same message. This avoids a possibility for a malicious node to create such message that can be correctly verified on a subset of recipients, and fail on other, as it is possible with MAC scheme. In order to use digital signatures, nodes in Ring rely on two operations: *sign* and *verifysig*.

A node p may sign a part (as a bit-string) x of a message m (or full message) by issuing *sign(p,x)* operation. This operation returns a signature s. The *verifysig(p,x,s)* operation takes the identity of the sender, a bit-string x, and a putative signature s as parameters. Then, *verifysig* returns boolean $\top$ (true) if and only if *sign(p,x)* $= s$. In other words, *verifysig* will return $\top$ if and only if there was a preceding call to *sign(p,x)* by p, that generated signature s for the bit-string x. We assume that all of the nodes have the public keys of all of the other nodes in the system, in order to verify the signatures.

In addition, we denote, by $D(m)$, the digest of message m. Also, we assume that during synchronous periods there exists a time interval Δ, which represents the maximal propagation delay between any two correct nodes in the system. Σ represents the set of all $3f+1$ replicas.

A short-hand notation $\langle m\rangle_{\sigma_p}$ denotes the fact that p signed the message m with its private key. The same notation at the recipient side denotes the fact that any node can successfully verify the signature in such message, sent by node p. For MACs, a short-hand notation $\langle m\rangle_{\mu_{p,q}}$ denotes the fact that p added its MAC for message m, destined for q. Similarly, the same notation on the recipient side denotes the fact that node q can successfully verify this MAC.

Every operational mode of *Ring*[7] has one replica designated as the sequencer, and a fixed ordering of replica IDs (called the *ring order*), which is known to all processes. The sequencer precedes all of the replicas in the ring order, and the last replica in the ring order is the physical predecessor of the sequencer on the ring. Without loss of generality, we assume that the

[7]And, by extension, every ABSTRACT instance corresponding to that mode.

sequencer in the current instance is, in fact, the replica r_0. To simplify the notation, as there is a finite number of replicas, we treat the ring order as a sequence of numbers in the finite group of modulo order $3f + 1$[8]. Thus, the successor of node r_i is $r_{i \oplus 1}$, where $\oplus$ is addition modulo $3f + 1$. When a replica receives a request from a client, the replica becomes the *entry replica* for the request. The replica which replies back to the client is the *exit replica* for a given request. The exit replica is the predecessor of the entry replica ($r_{\text{exit}} = r_{\text{entry}} \ominus 1$, where r_{exit} and r_{entry} hold the IDs of the exit and the entry replicas).

We indicate the *predecessor* set (respectively, the *successor* set) of replica r_j as $\overleftarrow{r_j}$ (respectively, $\overrightarrow{r_j}$). Also, we denote the *sequenced predecessor set* of replica r_j by $\widehat{r_j}$. In the sequenced predecessor set of replica r_j are all of the replicas which may have received a request with a sequence number from the sequencer. We, also, reference by Σ_{last} the set of the last $f + 1$ replicas in the ring order: $\Sigma_{\text{last}} = \{r_j \in \Sigma : j \geq 2f\}$. Further, we denote by $\Sigma_{\text{last}}^{req}$ the set of the last $f + 1$ replicas in the ring order, with respect to request *req*:

$$\Sigma_{\text{last}}^{req} = \left\{ r_j \in \Sigma : \quad req.\text{entry} \ominus (f+1) \leq j \leq req.\text{entry} \ominus 1 \right\}$$

where *req*.entry denotes the entry replica of request *req* (Table 4.2).

4.3.3 Implementation

In this section, we describe the implementation of the fast mode in Ring. First, we present the list of variables used in the implementation, along with the description of their use. Next, we describe all of the steps that the algorithm takes in order to commit a request, starting from a client, through replicas, and then back to the client. The description encompasses all of the possible executions, with the fault-handling cases described separately.

In Table 4.1, we list all of the variables used in the pseudo-code given in this section. As nodes communicate by passing messages, we list the field names of the requests in Table 4.2.

Pseudo code for the fast mode algorithm. For clear presentation, we place the pseudo code close to its first use. In the following text, we give an overview of the pseudo code used in this section.

For brevity, none of the pseudo code presented in this thesis does not contain code for processing *INIT* requests. In ABSTRACT, these messages initialize the replicas of the next instance with a valid abort history. However, the explanation of steps will contain descriptions of how to handle *INIT* requests.

Algorithm 4.3 on page 45 contains the client pseudo code. This algorithm relies on a method called *extract_history*, which is part of the ABSTRACT framework. We explain this method at

[8]In the case the system utilizes more than this optimum, the modulo operation must be changed accordingly.

Variable	Purpose
RASET	The set of Ring Authenticators, used by both the clients and the replicas
MACSET	The set of *MAC* s, authenticating the reply, generated by the replicas
LH	The local history of a replica
self	The variable holding the ID of a replica
sn	The sequence number associated with a request
lastreq	An array indexed by a client ID, holding the last request sent by the client
lastsn	An array indexed by a client ID, holding the last sequence number given to a request from the client
lasthist	An array indexed by a client ID, holding the last history sent to the client
active	A boolean variable, representing the running state of a particular ABSTRACT instance
sequencer_id	A variable containing the ID of the sequencer in the current ABSTRACT instance
pending	A list of pending requests at the replicas
OBRpending	A list of pending on-behalf requests (*OBR*) at the replicas

Table 4.1: An overview of used variables

Field name	Purpose
o	Replicated state machine command
t_c	The time-stamp for the request of the client
cid	The ID of the client
entry	The ID of the *entry* replica to which the client sent the request

Table 4.2: Field names for a request

the end of Section 4.4.1. Algorithm 4.1 on page 42 contains some miscellaneous methods, which are shared between the fast and the resilient mode.

Algorithms 4.2, 4.4, 4.5, and 4.6 represent the pseudo code for the replica in the fast mode of Ring. The full algorithm was split into these four parts for better readability, and also because important functionalities are grouped together in this manner. Algorithm 4.2 on page 44 covers only the initialization of replicas in the fast mode of Ring. Algorithm 4.4 on page 47 contains the code for handling the *RING* (the name of the request message in the implementation of Ring) and the *ACK* messages. Finally, Algorithm 4.5 on page 52 and Algorithm 4.6 on page 53 contain the code for handling faults and *on-behalf* requests (split in two, for easier following).

Algorithm 4.1 Miscellaneous functions used by pseudo code for replicas

```
function distance(src, dst) returns int is
 1:  return dst $\ominus$ src

function updateMACs(MACSET, req, c, LH) returns SET is
 2:  myMACSET $\leftarrow$ MACSET
 3:  if distance(req.entry, self) $> 2f$ then
 4:      myMACSET $\leftarrow$ myMACSET $\cup$ authenticate(self, c, $\langle$req, $D(LH)\rangle$)
 5:  return myMACSET

function authenticateRA(RASET, req, sn, is.req) returns RASET is
 6:  myRASET $\leftarrow$ RASET
 7:  myRA $\leftarrow \emptyset$
 8:  for all RA $\in$ RASET do
 9:      RA' $\leftarrow$ RA
10:      if RA' = $\langle$self$, *, *, *\rangle$ then
11:          myRASET $\leftarrow$ myRASET \ RA'
12:  if distance(req.entry, self) $\leq 2f \vee$ is.req $= \top$ then
13:      end $\leftarrow$ self $\oplus(f+1)$                                   {we write $f+1$ MACs}
14:  else
15:      end $\leftarrow$ req.entry$\ominus 1$      {request is leaving the system in less than $f+1$ communication steps}
16:  for j = self$\oplus 1$ to end do
17:      {iterate clockwise on the circle}
18:      if is.req = $\top$ then
19:          myRA $\leftarrow$ myRA $\cup \big\langle j,$ self$, sn,$ authenticate(self$, r_j, \langle Type^{\mathrm{REQ}}, req, sn\rangle)\big\rangle$
20:      else
21:          myRA $\leftarrow$ myRA $\cup \big\langle j,$ self$, sn,$ authenticate(self$, r_j, \langle Type^{\mathrm{ACK}}, req, sn\rangle)\big\rangle$
22:  return myRASET $\cup$ myRA

function verifyRA(RASET, req) returns boolean is
23:  {checks the well-formedness of Ring Authenticators}
24:  RASET' $\leftarrow$ sort RASET using $<_{(3f+1, req.entry)}$
25:  if $\exists$ R,S $\in$ RASET' :                       {request cannot have two different sequence numbers}
         R=$\langle *, *, sn_1, *\rangle$, S=$\langle *, *, sn_2, *\rangle$,
         $sn_1 \neq$ nil $\wedge\ sn_2 \neq$ nil $\Rightarrow sn_1 \neq sn_2$ then
26:      return false
27:  if $\exists$ R,S $\in$ RASET' :                       {once the sequence number is set, it must persist}
         R=$\langle j_1, i_1, sn_1, *\rangle$, S=$\langle j_2, i_2, sn_2, *\rangle$,
         $(j_1, i_1) <_{(3f+1,i)} (j_2, i_2) \Rightarrow sn_1 \neq$ nil $\wedge\ sn_2 =$ nil then
28:      return false
29:  if $\bigwedge$ distance(req.entry,self) $\leq f+1 \Rightarrow$                      {MAC from the client}
           $\exists RA \in RASET\ :\ RA = \langle i, *, sn', MAC'\rangle,$verifyauth(self$, req.$cid$, \langle req, sn'\rangle) = \top$
         $\bigwedge \forall r_j \in \overleftarrow{\text{self}}, \exists RA \in$ RASET :            {MAC from a predecessor}
           $RA = \langle i, j, sn', MAC'\rangle,$verifyauth$(j,$self$, \langle req, sn'\rangle) = \top$ then
30:      return true
31:  return false
```

Miscellaneous Functions Used in the Pseudo Code

Algorithm 4.1 on page 42 contains various functions used by the rest of the pseudo code, that is shared between the fast and the resilient mode. Function *distance* returns the distance between two nodes on the ring. Function *updateMACs* takes a set of MACs MACSET from a message, associated req, the identity of the client c who issued the request req, and the local history *LH* as parameters. This function adds the replica's authenticator for the request and current local history to MACSET. *updateMACs* returns updated MACSET.

Functions *authenticateRA* and *verifyRA* deal with Ring Authenticators. Ring Authenticators are represented as a *RASET* field in a message. Function *authenticateRA* adds MACs to this set, for any replica r_i. There are two steps. First, replica r_i updates *RASET* by removing all of the MACs destined for itself (line 11 of Algorithm 4.1 on page 42). Then, the replica adds an *RA* authenticating the tuple $\{Type^{\text{REQ}}, req, sn'\}$ (line 19 of Algorithm 4.1) for every replica in its successor set, $\overrightarrow{r_i}$. The first element of the tuple can take the value of either $Type^{\text{REQ}}$ or $Type^{\text{ACK}}$, and serves as a protection against copy attacks. With this in place, none of the replicas can forge *ACK* messages, by using the *RA* of the original *RING* message. This function authenticates the content of the request, and associates it with its sequence number *sn'*. This way, replicas ensure that the information on the right sequence number will propagate, if the sequencer is correct. *authenticateRA* returns an updated *RASET*.

Function *verifyRA* checks if the *RASET* of the message contains correct MACs, and proper authenticators. This function returns a boolean $\top$ indicating that *RASET* is correct and properly authenticates the message. Otherwise, the function returns $\perp$. Function *verifyRA* first check well-formedness of the Ring Authenticator:(1) MACs do not authenticate request for different sequence numbers; and (2) MACs do not authenticate request with no sequence number, although request already had one.Then, the function checks if the MACs in the Ring Authenticator from the client (on first $f + 1$ replica) and the predecessors match the content.

Replica Initialization and Helper Methods in the Pseudo Code

Algorithm 4.2 describes the initialization of replicas in the fast mode, the code for assigning the sequence numbers to requests, and the code for executing the request.

Initialization (described in lines 1–8 of Algorithm 4.2 on page 44) sets all values to $\emptyset$ or `nil`. It also sets the value of variable *active* to `true` to enable the instance.

Procedure *sequence_request* takes a placeholder variable and a request. If the current replica is the sequencer, the procedure increments the replica's global sequence number counter. Then, the procedure assigns this value to the placeholder variable, and returns.

Procedure *execute* executes the request. This procedure takes a sequence number (*sn'*) and a request (*req*). The procedure:

1. executes the request and stores the reply (lines 12–23),
2. appends *req* to its local history LH_i (line 16), and
3. updates the data which reflects the execution of the last request by the client $req.c$ by storing *req* and sn_i into corresponding data structures: $lastsn_i[req.c]$, and $lasthist_i[req.c]$ (line 14).

Algorithm 4.2 Fast mode: pseudo code for initialization with helper methods, of replica r_i.

```
Initialization:
1:  pending ← ∅
2:  sn← 0
3:  active ← true
4:  $T_{OBR}$ ← $(3f+1)\Delta$
5:  for all c ∈ Clients do
6:     lastreq[c] ← nil
7:     lastsn[c] ← 0
8:     lasthist[c] ← nil

Implementation:
procedure sequence_request(sn', req) is
9:     if i = sequencer_id then
10:       sn ← sn + 1
11:       sn' ← sn
procedure execute(sn', req) is
12:    if lastreq[$req$.c].$t_c \geq req.t_c$ then
13:       return
14:    sn ← sn'
15:    sn$_j$ ← sn'
16:    LH ← LH ∘$\langle req \rangle$
17:    lasthist[$req$.c] ← LH
18:    lastsn[$req$.c] ← sn
19:    lastreq[$req$.c] ← req
20:    for req' ∈ OBRPending ∧ req'.c = req.c do
21:       if req'.$t_c$ < lastreq[$req$.c].$t_c$ then
22:          OBRPending ← OBRPending \{$req'$}
23:          stop(T$_{OBR_{req'}}$)
```

Processing Requests in the Best Case, Without Faults

In Figure 4.5, we show the steps taken during the processing of a request, in the case in which there are no faults. The process starts with the client issuing the request to any of the replicas, and continues with the replicas exchanging the request, until the client commits the response. In order to follow the terminology from the ABSTRACT framework, algorithm steps are denoted as *Step **RN***, referring to the N th step, where R stands for fast mode of Ring (to match notation of other protocols in ABSTRACT). In order to avoid confusion between communication steps and algorithm steps, communication steps will be referred to as communication phases. For processing a request in a fault free case, Ring takes 4 algorithm *steps*, that span over 10 communication *phases*. The following text will now explain these steps.

Algorithm 4.3 Ring: pseudo-code of the client.

```
Initialization:
1:  $t_c \leftarrow 0$
2:  entry ← 0
3:  $T_{Ring} \leftarrow (2(3f+1)+2)\Delta$

Implementation:
procedure invoke(o) is
4:      $t_c \leftarrow t_c + 1$
5:      entry ← any number in 0..$3f$
6:      req ← ⟨o, $t_c$, self as cid, entry⟩
7:      clientsig ← sign(req)          {clients may generate MACs here, for f=1, consult Section 5.2}
8:      start_timer($T_{Ring}$)
9:      send ⟨RING, req, nil, clientsig, ∅⟩ to $r_{entry}$

upon event ⟨REPLY, req, MACSET, LH⟩ from $r_{entry \ominus 1}$ do          {predecessor on the ring answers}
10: if $\forall r_i \in \overleftarrow{r}_{entry} : \exists a \in MACSET \Rightarrow$ verifyauth$\big(r_i, \text{self}, \langle req, D(LH)\rangle, a\big)$
    then
11:     trigger ⟨COMMIT(req, LH)⟩
12:     cancel($T_{Ring}$)

upon $T_{Ring}$ expires do
13: panicsig ← sign(self, req)
14: send ⟨PANIC, req, panicsig⟩ to all replicas

upon event ⟨GET-A-GRIP, h, req⟩$_{\sigma_{r_i}}$ from $f+1$ different replicas with the same $h$
15: trigger ⟨COMMIT(req, h)⟩

upon event ⟨ABORT, $LH_{i\sigma r_i}$, req, $r_i$⟩$_{\mu_{r_i,c}}$ from $2f+1$ different replicas, matching $req$
16: $LH' \leftarrow \emptyset$
17: j ← 1
18: hist ← $\bigcup_i LH_i$
19: while $\exists req' : \exists LH^j \subset hist : (|LH^j| \geq f+1) \wedge (\forall h, h' \in LH^j : h[j] = h'[j] = req')$ do
20:     {$LH^j$ is a subset of histories in hist j}
21:     $LH' \leftarrow LH' \circ \langle req' \rangle$
22:     j ← j + 1
23: $abortH \leftarrow$ choose $LH'' : isPrefix(LH'', LH') \wedge (\forall req_1, req_2 \in LH'' : req_1 \neq req_2)$
24: if $req \notin abortH$ then
25:     $abortH \leftarrow abortH \circ \langle req \rangle$
26: trigger ⟨ABORT(req, abortH)⟩
```

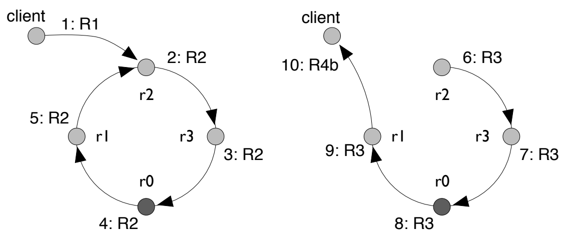

Figure 4.5: A client invokes a request, by sending a message to replica r_2, and receives a reply from replica r_1 afterwards. The circles represent the nodes (clients and replicas), the arrows denote a flow of the communication. The labels of arrows denote the communication phase, and the algorithm step.

Step R1. *A client can send a request to any replica (lines 4–9 of Algorithm 4.3 on page 45)*

When invoking an operation o, the client c first chooses entry, the index of the *entry* replica (line 5 of Algorithm 4.3). Then, the client c forms a $req = \langle o, t_c, c, i\rangle$, which contains the operation in question, the identifier for the request, the id of the client, and the id of the entry replica. Next, the client sets *clientsig* to the signature of *req*. The client creates a request (a *RING* message) for replica r_{entry} (line 9). The client does not fill in the sequence number and *RASET* fields of the message (line 9), as these fields are set by the replicas. Finally, the client sends the message to replica r_{entry}.

Upon sending a *RING* message to the entry replica, the client starts the timer T_{ring}, which is set to expire after a period $(2(3f+1)+2)\Delta$ (line 3). The expiration time is set to match the maximum response delay when the system is synchronous. If the timer expires before the client receives a response, the client will panic (line 13) and notify the replicas. We assume that the clients wait for the response from a replica before issuing new requests (in other words, there is at most one outstanding request from any one of the clients in the system).

(Note) Handling *INIT* Requests The first message sent by the client must contain also the INIT tag and the valid init history *IH* (which is an unforgeable abort history from the preceding ABSTRACT).

Step R2. *Upon receiving a request (a RING message), the replica r_i updates the message fields and forwards the message to its successor (lines 1–14 of Algorithm 4.4 on page 47)*

Upon receiving (line 1 of Algorithm 4.4) a $\langle$*RING*, *req*, *sn'*, *RASET*, *MACSET*$\rangle$ message from the predecessor (or from the client, in the case in which the replica r_i is the *entry replica* for the request), the replica first tries to authenticate and accept the message. This check consists of several conditions (lines 1–5):

Algorithm 4.4 Fast mode: pseudo code for request processing, of replica r_i.

```
upon event ⟨RING, req, sn', clientsig, RASET⟩$_{\sigma_c}$ from $r_{i\ominus 1}$or client $c$
1:  when ⋀ active                                              {only if the instance is active}
2:          ⋀ distance($r_{entry}, r_i$) $\leq f \Rightarrow$ verifysig(c, self, req, clientsig)
3:          ⋀ verifyRA($RASET, req$)
4:          ⋀ req.$t_c$ > $lastreq[req.c].t_c$
5:          ⋀ (sn' $\neq$ nil $\Rightarrow$sn'=sn+1)
    do
6:    pending ← pending ∘{$req$}
7:    sequence_request(sn', req)
8:    if sn' $\neq$ nil then
9:      execute(sn', req)
10:   RASET ← authenticateRA(RASET, req, sn', ⊤)
11:   if i = predecessor(req.entry) then
12:     send ⟨ACK, sn', D(req), req.c, RASET, ∅⟩ to $r_{i\oplus 1}$
13:   else
14:     send ⟨RING, req, sn', RASET, clientsig⟩ to $r_{i\oplus 1}$

upon event ⟨ACK, sn', D', c, RASET, MACSET⟩ from $r_{i\ominus 1}$
15: when ⋀ active
16:         ⋀∃req ∈ pending : req.c = c ∧$D(req) = D$
17:         ⋀ verifyRA(RASET, req)
18:         ⋀ (sn' = sn+1)
    do
19:   execute(sn', req)
20:   pending ← pending \ req
21:   myMACSET ← updateMACs(MACSET, req, req.c, LH)
22:   RASET ← authenticateRA(RASET, req, sn, ⊥)
23:   if predecessor(req.entry) = self then
24:     send ⟨REPLY, req, myMACSET, LH⟩ to req.c
25:   else
26:     send ⟨ACK, sn, D(req), RASET, myMACSET⟩ to $r_{i\oplus 1}$
```

1. The first $f+1$ replicas check whether some Ring Authenticator (*RA*) in the *RASET* contains a valid authenticator (a signature) for the request *req*, generated by the client;

2. every replica r_i checks whether *RASET* contains an *RA* with a valid MAC for every replica r_j in the predecessor set, $\overleftarrow{r_j}$, authenticating *req* and *sn'*. (Note: by definition, the predecessor set of the entry replica is empty $\overleftarrow{r}_{req.\text{entry}} = \emptyset$);

3. every replica accepts a *RING* message only if the timestamp of the client of the request *req* ($req.t_c$) is higher than the last seen (and executed) request timestamp from that same client ($lastreq_i[req.c].t_c$);

4. finally, if the sequence number *sn'* of the message is either equal to nil or $sn_i + 1$, the replica accepts the *RING* message.

Note that the handler is guarded with *active* boolean (line 1). If this boolean is true, that means that the instance is active. Otherwise, the instance is not active, and it will not take any new requests.

If the aforementioned checks succeed, then replica r_i proceeds to the processing of the request. First, every replica stores *req* in *pending*$[req.c]$. The sequencer then increments the local sequence number sn_0, and sets *sn'* = sn_0 (line 7 of Algorithm 4.2). Now, if *sn'* is not nil, then the replica executes the request (lines 12–23 of Algorithm 4.2 on page 44), through several steps. Replica:

- stores *sn'* into the local variable sn_i (line 14 of Algorithm 4.2);
- appends request *req* to its local history *LH* (line 16 of Algorithm 4.2);
- updates the data which reflects the execution of the last request by the client *req.c* by storing *req* and sn_i into corresponding data structures: $lastsn_i[req.c]$, and $lasthist_i[req.c]$ (lines 17–18 of Algorithm 4.2);
- updates the information about the last known request from the client, by storing the request into *lastreq*$[req.c]$ (line 19 of Algorithm 4.2); and
- cancels all alarms related to *OBR* messages, and clears stale information in *OBRPending* list (lines 20–23 of Algorithm 4.2).

After processing *req*, the replica r_i forwards the request, unless r_i is the exit replica. Replica r_i sends the *RING* message containing *req* and *sn'*, as well as the updated set *RASET* calculated in line 10 of Algorithm 4.4, by calling function *authenticateRA*.

Finally, the replica r_i sends the *RING* message, containing *req*, *sn'*, *RASET*, and $\emptyset$ in place of *MACSET* (line 14).

If the replica r_i is the exit replica, instead of forwarding the request, the replica generates an acknowledgement – an *ACK* message (line 12). The replica first updates the *RASET*, this time authenticating the tuple $\{Type^{\mathrm{ACK}}, req, sn'\}$ (line 21 of Algorithm 4.1 on page 42). Finally, the replica r_i sends the *ACK* message to the successor. The *ACK* message initially contains the following fields: $D(req)$, *req.c*, *sn'*, *RASET*, and the empty set (as the *MACSET* field).

(Note) RING message verification failure If a verification of a received *RING* message fails, a correct replica r_i can safely discard the received message.

(Note) Handling *INIT* Requests The first message that the sequencer can assign a sequence number to must contain the INIT tag and a valid init history *IH*. Moreover, if the local history LH_i of replica i is empty, replica i may only execute the *INIT* request with a valid init history

IH. More precisely, replica i executes the entire *IH*, by appending the entire *IH* (instead only *req*) to its (empty) history LH_i. All (non-*INIT*) requests received before are discarded. If LH_i is not empty, replica i neglects *IH* and processes only *req*, as described above.

Step R3. *Upon receiving an acknowledgement (an ACK message), the replica r_i updates the message fields and forwards the message to its successor. (lines 15–26 of Algorithm 4.4 on page 47)*

The replica r_i receives $\langle ACK, D, sn', c', RASET, MACSET\rangle$ from the predecessor, and processes the message in similar fashion to the *RING* message. First, the replica checks whether it can successfully authenticate and accept the message (lines 15–18). The conditions differ from the *RING* message case, namely in that:

1. if there is no stored request *req* corresponding to the *ACK* message in the *pending* list (line 16), the message is discarded; otherwise, *req* is taken from the *pending* list,

2. if *RASET* contains an *RA* with a valid MAC for every replica r_j in the predecessor set $\overleftarrow{r_i}$, the replica authenticates *req* and *sn'*; otherwise, the replica discards the message,

3. finally, every replica accepts the *ACK* message only if the sequence number of the message (*sn'*) equals $sn_i + 1$ (line 18).

If the request was not executed previously by replica r_i, the replica does the same steps as when handling the *RING* message: it calls the *execute* procedure (defined in lines 12–23 of Algorithm 4.2 on page 44). This procedure first check whether it was executed before for the same request, to ensure idempotent execution.

After executing the request, the replica r_i forwards the *ACK* message. Beforehand, the replica updates the *RASET*, and the *MACSET* fields of the message. The *MACSET* is effectively updated only by the $f + 1$ predecessors of the entry replica (line 21). These replicas authenticate the pair $\{req, D(LH_i)\}$, where $D(LH_i)$ denotes the digest of the local history of the replica. If the replica r_i is not the exit replica, the replica forwards the *ACK* message, containing $D(req)$, *sn'*, *req.c*, *RASET*, and *MACSET* (as seen in line 26). Otherwise, the replica r_i sends a *REPLY* message (a reply) to the client named in *req.c*, containing the entire local history of the replica (LH_i) (line 24).

(Note) ACK message verification failure If any of the check conditions does not hold, the replica may safely discard the request. At this point, there is no MAC from the client in the *RASET*, so the replica may assume that some of the predecessors are Byzantine, and simply discard the request.

(Note) Handling *INIT* Requests The first message that the sequencer can assign a sequence number to must contain the INIT tag and a valid init history *IH*. Moreover, if the local history LH_i of replica i is empty, replica i may only execute the *INIT* request with a valid init history *IH*. More precisely, replica i executes the entire *IH*, by appending the entire *IH* (instead only *req*) to its (empty) history LH_i. All (non-*INIT*) requests received before are discarded. If LH_i is not empty, replica i neglects *IH* and executes only *req*, as described above.

Step R4a. *Upon receiving the REPLY message from the exit replica before the expiration of the timer, if the client successfully verifies the reply, the client commits the request. (lines 10–12 of Algorithm 4.3 on page 45)*

If the client c receives the $\langle REPLY, req, *, *, *, MACSET, LH\rangle$ message (line 10) from the exit replica ($r_{\text{entry}\ominus 1}$), that can be successfully verified, then the client commits the request *req* with *Ring commit history LH* (line 12). A successful verification (described in line 10) occurs when the set *MACSET* contains valid MACs from the last $f+1$ replicas in the ring order (predecessors of the exit replica), destined for the client c, which authenticate the pair $\langle req, d\rangle$, where d is the digest of the history ($d = D(LH)$).

Processing Request in the Presence of Benign Faults

Figure 4.6 depicts request processing when there are benign (non-Byzantine) faults, in the system. In such situations, the client does not receive the response, and starts *panicking*. However, the majority of the replicas is able to successfully respond to the client. The client receives the response, and continues with its operation. Ring handles these situations gracefully, with lower performance than in the case without faults.

In this case, the processing involves all of the replicas trying to execute the request *on behalf* of the client. This concept is novel in Ring, and allows the replicas to detect whether the client is misreporting, or is there indeed a faulty replica in the system.

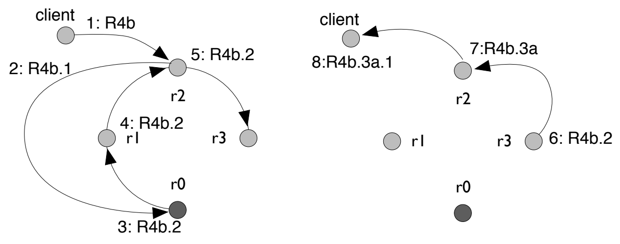

Figure 4.6: A client invokes a request, but does not receive a reply. The client panics, sending a *PANIC* message to all of the replicas. The majority of replicas successfully answer. For clarity, we present only the actions of replica r_2.

Step R4b. *The client **does not** receive the REPLY message from the exit replica, and/or the client can not verify the message before the expiration of the timer. (lines 13–14 of Algorithm 4.3 on page 45)*

If the client does not receive the message before timer T_{Ring} expires, or the message cannot be verified, the client *panics* (line 14). The client c sends a $\langle PANIC, req, panicsig \rangle$ message to all of the replicas. The *PANIC* message is carries the digital sign of the request, by the client. Moreover, the client periodically resends the *PANIC* message to the replicas, until the client commits or aborts the request.

Step R4b.1. *A replica receives a PANIC message from the client, and the replica retries to execute the request on behalf of the client. (lines 1–7 of Algorithm 4.5 on page 52)*

Replica r_i, upon receiving a $\langle PANIC, req, panicsig \rangle$ message (line 1 of Algorithm 4.5), if the message contains a valid signature, tries to commit the request by invoking Steps R1 through R4a on behalf of the client. Toward that end, replica r_i acts as the client and sends the $\langle OBR, r_i, req, panicsig, sn_{OBR} = \mathsf{nil}, RASET_{OBR} = \emptyset, MACSET_{OBR} = \emptyset \rangle_{\mu_{r_i,r_0}}$ message to the sequencer (line 6). Subsequently, the replica starts the timer $T_{OBR_{req}}$. If the timer expires before the replica receives a response for the *OBR* request, the replica will abort the protocol (lines 4–7 of Algorithm 4.5).

The *OBR* message is similar to the *RING* (and the *ACK*) messages, albeit some differences exist:

- the *OBR* message contains an additional field which the replica r_i (the originator of the *OBR* request) populates with its own ID.
- the *RASET* field is initially empty, as the signature of the client is included in the message.

Finally, it is important to note that a correct replica r_i sends an *OBR* request to the sequencer *if and only if* the request is not old (the $req.t_c$ field of the *PANIC* message is greater or equal than $lastreq_i[req.c].t_c$, as seen in the line 2). Moreover, the replica r_i abandons waiting for the *RING* message from replica $r_{sequencer\_id \ominus 1}$ (the predecessor of the sequencer), and cancels its timer if $tr_i[c]$ becomes greater than $req.t_c$ (suggesting there is a new request from client c). These steps are explained on lines 20–23 of Algorithm 4.2 on page 44.

Algorithm 4.5 Fast and resilient mode: pseudo-code for switching, of replica r_i (part 1).

```
upon event ⟨PANIC, req, panicsig⟩ from client c
 1: when ⋀ active
 2:      ⋀ req.$t_c$ ≥ lastreq[req.c].$t_c$
 3:      ⋀ verifysig(c, req, panicsig)
    do
 4:   OBRPending ← OBRPending ∪{ req }
 5:   SIGSET ← sign(self, req)
 6:   send ⟨OBR, self, req, panicsig, 0, SIGSET, ∅⟩ to $r_{sequencer\_id}$
 7:   trigger ⟨ $T_{OBR_{req}}$ ⟩

upon event ⟨PANIC, req, panicsig⟩ from client c                {only when not active}
 8: when ⋀¬ active
 9:      ⋀ verifysig(c, req, panicsig)
    do
10:   send ⟨ABORT, $LH_{\sigma_{self}}$, req, self⟩$_{\mu_{self,req.c}}$ to client denoted in req.c

upon event ⟨OBR, $r_j$, req, panicsig, sn', SIGSET, MACSET⟩ from $r_k$
11: when ⋀ active
12:      ⋀ verifysig(req.c, req, panicsig)
13:      ⋀ $\forall r_j \in \overleftarrow{\text{self}}: \exists sig \in$ SIGSET: verifysig($r_j$, req, $sig$)
14:      ⋀ req.$t_c$ ≥ lastreq[req.c].$t_c$
15:      ⋀self > 0 ⇒ sn' = sn+1
    do
16:   if req = lastreq[req.c] ∧ lastsn[req.c] ≠ nil then
17:     $sn_{OBR}$ ← lastsn[req.c]
18:     $LH_{OBR}$ ← lasthist[req.c]
19:   else
20:     sequence_request(sn', req)
21:     execute(sn', req)
22:     $sn_{OBR}$← sn
23:     $LH_{OBR}$ ← LH
24:   MACSET' ← updateMACs(MACSET,req,$r_j$,$LH_{OBR}$)
25:   if self = sequencer_id ⊖1 then
26:     send ⟨⟨OBR, $r_j$, req, panicsig, $sn_{OBR}$, ∅, MACSET'⟩,$LH_{OBR}$⟩ to $r_j$
27:   else
28:     SIGSET ← SIGSET ∪ sign(self, req)
29:     send ⟨OBR, $r_j$, req, panicsig, $sn_{OBR}$, SIGSET, MACSET'⟩ to $r_{i\oplus 1}$
```

Algorithm 4.6 Fast and resilient mode: pseudo-code for switching, of replica r_i (part 2).

```
upon event ⟨⟨OBR,self,req,panicsig,*,*,MACSET⟩,h⟩ from $r_{\text{sequencer\_id}\ominus 1}$
1:  when active
    do
2:    if $\forall r_j \in \overleftarrow{r}_{\text{sequencer\_id}}, \exists a \in MACSET \Rightarrow$ verifyauth($s_j$,self,D(h)) then
3:      send ⟨GET-A-GRIP, h, req⟩$_{\sigma_{self}}$ to client denoted in req.c
4:      stop($T_{OBR_{req}}$)

upon $T_{OBR_{req}}$ expires do
5:  active ← false
6:  send ⟨STOP, LH, req, self⟩$_{\sigma_{self}}$ to every replica $r_k$
7:  send ⟨ABORT, LH$_{\sigma_{self}}$, req, self⟩$_{\mu_{self,req.c}}$ to client denoted in req.c

upon event ⟨STOP, LH', req, $r_j$⟩$_{\sigma_{r_j}}$ from $2f+1$ different replicas with matching $req$
8:  active ← false
9:  send ⟨STOP, LH', req, $r_j$⟩ to every replica $r_k$
10: send ⟨STOP, LH, req, self⟩$_{\sigma_{self}}$ to every replica $r_k$
11: send ⟨ABORT, LH$_{\sigma_{self}}$, req, self⟩$_{\mu_{self,req.c}}$ to client denoted in req.c
```

Step R4b.2. *A replica receives an OBR message and processes the message in a similar way to both the RING (Step R2) and ACK messages (Step R3.). (lines 11–29 of Algorithm 4.5 on page 52)*

Replica r_i first checks whether it can successfully authenticate and accept a message, upon receiving the $\langle OBR, r_k, req, panicsig, sn, SIGSET, MACSET\rangle$ message from the predecessor (or from replica r_k, in case r_i is the sequencer). This check consists of several conditions:

1. the replica r_i checks whether the signature of the client matches the request (line 13);
2. the replica r_i checks (at line 13) whether the *SIGSET* contains an *RA* with a valid MAC for every replica r_j in the predecessor set, $\overleftarrow{r_j}$, authenticating *req* and *sn'*. (Note: the predecessor set for the sequencer in this case is empty $\overleftarrow{r_0} = \emptyset$);
3. the replica accepts the *OBR* message only if the timestamp of the client which has initiated the request *req* ($req.t_c$) is greater or equal than the timestamp of the last seen (and executed) request from the client ($lastreq_i[req.c].t_c$), as shown in line 14;
4. finally, every replica except the sequencer accepts the *OBR* message if the sequence number *sn'* of the message is equal to $sn_i + 1$.

If these checks succeed, then replica r_i proceeds to the execution part of processing (line 21). The replica only executes the request if the request is new (that is, $req.t_c$ is higher than the last stored request from the client, line 13 of Algorithm 4.4 on page 47). Otherwise, the replica takes the stored response and the sequence number, and skips the next step (line 18 of Algorithm 4.5).

Like in the case of the *RING* and the *ACK* messages, the sequencer updates the sequence number and executes the request (lines 20–21), by calling procedures *sequence_request* and *execute*. the replica stores *sn'* into the local variable sn_i of the replica. Moreover, every replica r_i updates its sn_{OBR} and LH_{OBR} with the corresponding values (lines 17–18 and lines 22–23).

Upon executing *req*, the replica forwards (line 29) the *OBR* message, unless the replica is the predecessor of the sequencer ($r_i \neq r_{sequencer\_id \ominus 1}$). In that case (line 26), the replica $r_{sequencer\_id \ominus 1}$ sends the reply back to the replica r_k (indicated as one of the fields of the *OBR* message). Beforehand, the replica updates the *RASET*.

Step R4b.3a. *The replica commits the request on behalf of the client and forwards the commit history to the client. (lines 4.6–4 of Algorithm 4.5 on page 52)*

If r_i receives an *OBR* message from the predecessor of the sequencer (line 4.6), containing MACs for the pair $\langle req, D(h)\rangle$ for the last $f+1$ replicas in the *MACSET*, as well as the entire history h (line 2), then the replica r_i simply sends the $\langle GET\_A\_GRIP, h, req\rangle_{\mu_{r_i,req.c}}$ message to the client named in $req.c$ (line 3). We say that r_i *commits* the *OBR* for *req* with the history h.

To counter for possible message losses, if a replica receives repeated *PANIC* messages for *req* after committing the *OBR* for *req*, the replica replies to these messages by re-sending the *GET_A_GRIP* message to the client.

Step R4b.3a.1. *The client receives $f+1$ GET_A_GRIP messages containing the same history and commits the request. (Line 15 of Algorithm 4.3 on page 45)*

If the client has received $f+1$ $\langle GET\_A\_GRIP, h, req\rangle$ messages from different replicas, with the same history, the client commits the request by returning $Commit(req, h)$.

Processing Request in the Presence of Faulty Replicas

Figure 4.7 shows the behaviour of the algorithm in the presence of detectable, faulty replicas. In such a case, when processing requests *on behalf* of the client, a replica will not receive an answer. This failure to receive the answer signals that replica that there are faulty replicas in the system, and the replicas will stop processing requests. Further, the replica in question will proceed to abort the current instance.

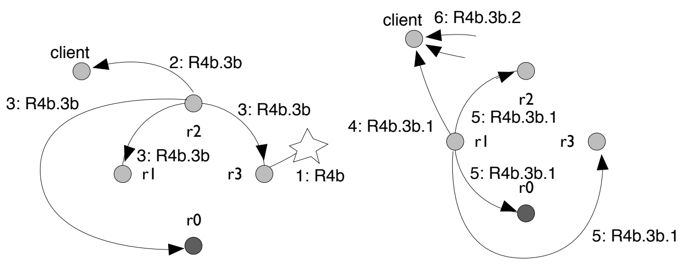

Figure 4.7: An alternative execution to the one shown in Figure 4.6: The replica r_2 cannot commit the request. It then aborts, and stops the current instance.

Step R4b.3b. *The replica **does not** commit the request on behalf of the client, stops processing the new request, and sends a signed history to the client. (lines 4–7 of Algorithm 4.5 on page 52)*

If the replica r_i does not receive the *OBR* request before the expiration of the timer, the replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests (line 5); (b) sends a signed local history to client $req.c$ using an $\langle ABORT, LH_{i\sigma_{r_i}}, req.t_c, r_i\rangle_{\mu_{r_i,req.c}}$ message (line 6); and (c) stops all of the OBR timers. In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica (line 7). Again, to counter possible message losses, we assume that r_i periodically retransmits this *STOP* message.

(Note) Handling *INIT* Requests If the history LH_i of replica i is empty, the replica acts in this step only on a *PANIC* message for an *INIT* request (that contains correctly authenticated init history *IH* from preceding ABSTRACT). In this case, upon receiving the first such *PANIC* message, replica i, before sending an *ABORT* message sets LH_i to *IH*. The following *PANIC* messages for *INIT* requests are treated as described above, neglecting the init histories.

Step R4b.3b.1. *The replica receives a STOP message from some other replica, stops processing new requests, and sends its signed history to the client. (lines 7–11 of Algorithm 4.5 on page 52)*

The replica r_i now aborts all of the requests from clients, similarly as in the Step R4b.3b. The replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests; (b) sends a signed local history to all of the clients referenced in the active OBR timers (there is a timer for every outstanding *OBR* request *req'*), using an $\langle ABORT, LH_{i\sigma_{r_i}}, req'.t_c, r_i\rangle_{\mu_{r_i,req'.c}}$ message; and (c) stops all of the OBR timers. In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica. Again, to counter possible message losses, we assume that r_i periodically retransmits this *STOP* message.

Step R4b.3b.2. *A client receives $2f+1$ matching ABORT messages, extracts the abort history, and aborts the request. (lines 16–26 of Algorithm 4.3 on page 45)*

A matching *ABORT* message for a $\langle PANIC, req \rangle$ message is any *ABORT* message with a matching request identifier $req.t_c$. When a client receives a matching *ABORT* message from $2f+1$ different replicas, the client extracts the abort history *abortH* in the following way:

- the client generates the history *LH'*, such that $LH'[j]$ equals the value that appears at position $j \geq 1$ of $f+1$ different histories LH_i received in the *ABORT* messages. If such a value does not exist for position j, then *LH'* does not contain a value at positions j and higher.
- the longest prefix *LH”* of *LH'* is selected such that no request appears in *LH”* twice.
- if $req = \langle o, t_c, c \rangle$ does not exist in *LH”*, the request is appended to *LH”*. The resulting sequence is an abort history *abortH*.

Then, the client c aborts *req* by returning $Abort(req, abortH)$. To prove the validity of *abortH*, the abort history is accompanied by the set of $2f+1$ *ABORT* messages.

4.4 Resilient Mode

The role of the resilient mode is to handle situations in which there are severe faults in the system, such as the presence of Byzantine replicas which may block the flow of requests. In this mode, Ring handles *RING*, *ACK*, and *PANIC* messages in the same way as the fast mode does. The main difference, however, stems from the approach in handling of *OBR* messages.

In the resilient mode, Ring tries to prevent malicious replicas from blocking the flow of requests, by utilizing more than one successor connections. Here, replicas use connections to their $f+1$ successors[9]. Under the assumption that only f replicas may be faulty, this assures that at least one correct replica will receive an *OBR* message, and forward it further. Also, in this mode, Ring breaks off the linear (circular) structure, and replies only when the request has been processed by at least $2f+1$ replicas. Ring uses digital signatures as a method of discovering the identities of the replicas which have processed the request. The propagation of an *OBR* message is shown in Figure 4.8, following a *panic* message received by Replica 2 (all of other replicas also send the message to the sequencer, but we omit those for sake of clarity).

Similarly to the case of the fast mode, the replicas initially send the *OBR* message to the sequencer, and await the arrival of the response. If the sequencer is Byzantine, a replica may not receive this response, and in that case, the replica will panic and stop the processing, triggering a switch to a new instance of Ring, with a different configuration, and a new sequencer.

[9]Here, when we refer to $f+1$ successors of a replica, we, in fact, refer to the successor of the replica, the successor of the successor of the replica, etc., rather than implying that a replica actually has $f+1$ successors.

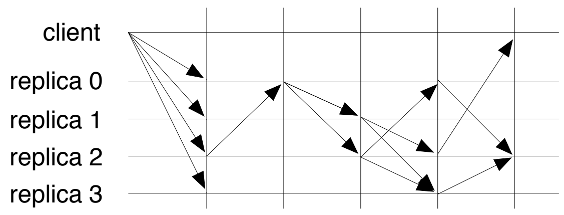

Figure 4.8: The Ring communication pattern in resilient mode, for the *on-behalf* request.

For brevity and clarity, we present only the difference between the fast and the resilient mode in the following text. The client code, as well as the handling of *RING*, *ACK*, and *PANIC* messages, all remain the same.

4.4.1 Implementation

In this section, we describe the implementation of the resilient mode in Ring. The following text uses the same notation as in Section 4.3.3. Similarly as in Section 4.3.3, we describe all of the steps that the algorithm takes in order to commit a request, starting from a client, through replicas, and then back to the client. In this section, however, we focus only on the execution after a fault has occurred at a *panicking* client. In the exposition, we consider the cases in which a replica does receive a response to its *on behalf* request, and those in which it does not receive one.

Pseudo code for the resilient mode. Algorithm 4.7 describes the difference between a fast mode and a resilient mode instance. The difference stems mainly from the differences in handling the *on behalf* requests. As in Section 4.3.3 we place the algorithm close to the description in the text of the algorithm steps.

Processing Requests When No Faults Occur

Similarly to Section 4.3, Figure 4.9 illustrates a flow of messages. Steps are denoted *Step* $\boldsymbol{R^+}$ `N`, referring to `N`th step of the resilient mode ($\mathbf{R^+}$ stands for the resilient mode).

Figure 4.9 illustrates the steps in a situation after a client has started to panic. Replica r_2 receives the *PANIC* message, and sends an *OBR* request to the sequencer (r_0). The sequencer, after assigning the sequence number to the request, signs the request, and forwards it to its $f+1$ successors. Replicas r_1 and r_2 receive this message, execute it in correct order, and sign it, forwarding it to their $f+1$ successors. In the next step, replicas r_2 and r_3 receive the message carrying $2f+1$ different signatures, execute the request, unless they have done it so far, and send a response to the client. We elaborate on these 5 algorithm steps in the text to follow.

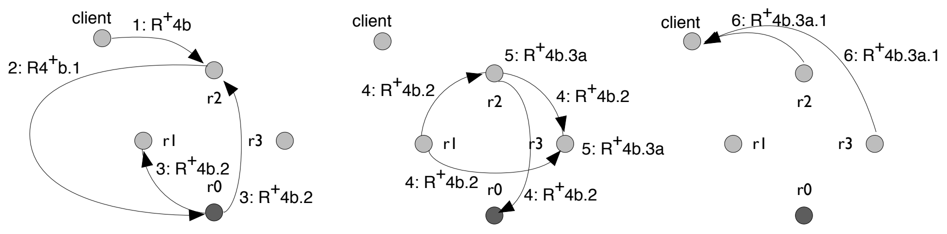

Figure 4.9: A client panics, sending a *PANIC* message to all replicas. For clarity, here we proceed as if only replica r_2 has received the message. The replicas utilize the links to their $f+1$ successors, and after receiving a response processed by at least $2f+1$ other replica, a replica replies back to the client (in this execution, replicas r_2 and r_3 reply after 5 communication steps.

Step R$^+$4b. *The client* ***does not*** *receive the RING message from the exit replica, and/or the client can not verify the message, before the expiration of the timer. (Line 14 of Algorithm 4.3 on page 45)*

If the client does not receive the message before the timer T_{Ring} expires, or the message cannot be verified, the client *panics*. The client c sends a $\langle PANIC, req, panicsig \rangle$ message to all of the replicas. The *PANIC* message is carries the digital sign of the request, by the client. Moreover, the client periodically resends the *PANIC* message to the replicas, until the client commits or aborts the request.

Step R$^+$4b.1. *A replica receives a PANIC message from the client, and the replica retries Ring on behalf of the client. (lines 1–7 of Algorithm 4.5 on page 52)*

Replica r_i, on receiving a $\langle PANIC, req, panicsig \rangle$ message, if that message contained a valid signature, attempts to commit the request by invoking Steps R1-R4a on behalf of the client. Toward that end, replica r_i acts as a client and sends the $\langle OBR, r_i, req, panicsig, sn_{OBR} = \text{nil}, RASET_{OBR} = \emptyset, MACSET_{OBR} = \emptyset \rangle_{\sigma_{r_i}}$ message to the sequencer. Subsequently, the replica starts the timer $T_{OBR_{req}}$. If the timer expires before the replica has received a response for the *OBR* request, the replica will abort the protocol.

The handling of the *OBR* message is similar to the handling of the *RING* (and the *ACK*) message, albeit some differences exist:

- the *OBR* contains an additional field, which the replica r_i (the originator of the *OBR* request) populates with its own ID.
- the *RASET* field is initially empty, as the signature of the client is included in the message. Note that the replica authenticates the message with the signature.

Algorithm 4.7 Resilient mode: pseudo-code of replica r_i (differences with respect to fast-mode implementation)

```
Initialization:
1: pending ← ∅
2: sn← 0
3: active ← true
4: stored_sigs ← ∅
5: $T_{OBR}$ ← $(3f+1)\Delta$

Implementation:
upon event ⟨OBR, $r_j$, $req_{\sigma_{req.c}}$, sn', SIGSET, MACSET⟩ from $r_k$
6:  when $\bigwedge req.t_c \geq lastreq[req.c]_t.c$
7:          $\bigwedge req_{\sigma_{req.c}}$ is valid
8:          $\bigwedge i > 0 \Rightarrow (sn' = sn+1)$
    do
9:    SIGS ← valid signatures in SIGSET from different servers
10:   stored ← stored_sigs[req]
11:   if SIGS ⊂ stored then
12:     break upon
13:   stored_sigs[req] ← stored ∪ SIGS
14:   if stored $\geq 2f+1$ then
15:     if req = lastreq[req.c] ∧ lastsn[req.c] ≠ nil then
16:       $sn_{OBR}$ ← lastsn[req.c]
17:       $LH_{OBR}$ ← lasthist[req.c]
18:     else
19:       sequence(sn', req)
20:       execute(sn', req)
21:       $sn_{OBR}$← sn
22:       $LH_{OBR}$ ← LH
23:     SIGSET ← SIGSET ∪ $\sigma_{self}$(self, req.c, D(req)) ∪ stored
24:     if j=i then
25:       send ⟨GET-A-GRIP, h, req⟩$_{\sigma_{self}}$ to req.c
26:     send ⟨OBR, $r_j$, $req_{\sigma_{req.c}}$, $sn_{OBR}$, SIGSET, MACSET'⟩ to $\overrightarrow{r_i}$
27:   else
28:     SIGSET ← SIGSET ∪ $\sigma_{self}$(self, req.c, D(req)) ∪ stored
29:     send ⟨OBR, $r_j$, $req_{\sigma_{req.c}}$, sn', SIGSET, MACSET'⟩ to $\overrightarrow{r_i}$
```

Finally, it is important to note that a correct replica r_i sends an *OBR* request to the sequencer *if and only if* the request is not old (the $req.t_c$ field of the *PANIC* message is greater or equal than $lastreq_i[req.c].t_c$). Moreover, replica r_i abandons waiting for the *RING* message from replica r_{3f} (predecessor of the sequencer), and cancels its timer, if $tr_i[c]$ becomes greater than $req.t_c$ (suggesting there is a new request from client c).

Step R$^+$4b.2. *A replica receives an OBR message and processes the message, possibly executing the request. The replica then forwards the message to its $f+1$ successors. (lines 6–29 of Algorithm 4.7 on page 59)*

Replica r_i first checks whether it can successfully authenticate and accept a message, upon receiving the $\langle OBR, r_k, req, panicsig, sn', RASET, MACSET\rangle$ message from one of its predecessors (or from replica r_k, in case that r_i is the sequencer). This check consists of several conditions:

1. the replica r_i checks whether the signature of the client matches the request (line 7);
2. the replica accepts the *OBR* message only if the client's timestamp of the request ($req.t_c$) is greater or equal than the timestamp of the last seen (and executed) request from the client ($lastreq_i[req.c].t_c$, shown at line 6);
3. finally, every replica except the sequencer accepts the *OBR* message if the sequence number *sn'* of the message is equal to $sn_i + 1$ (line 8).

If these checks succeed, then replica r_i collects all of the signatures in the *OBR* message, and verifies each in turn (line 9). If at least one of the signatures has not been seen previously, then the replica continues processing the request. Otherwise, the replica drops the request (line 12).

If there are more than $2f + 1$ valid signatures, the replica proceeds to the execution part of processing (line 14). The replica only executes the request if the request is new (that is, $req.t_c$ is higher than the last stored request from the client). In this case, the sequencer updates the local sequence number sn_0, and sets $sn' = sn_0$, while other replicas just store *sn'* into their local variable sn_i (line 19). Otherwise, if the request is old, the replica takes the stored response and the sequence number, and skips the execution step.

Each replica executes the request, if the request is new. Every replica r_i: (1) appends *req* to its local history LH_i, and (2) updates the data which reflects the last request by the client $req.c$ by storing *req*, sn_i and LH_i into $lastreq_i[req.c]$, $lastsn_i[req.c]$, and $lasthist_i[req.c]$, respectively. Finally, every replica stores *req* in $pending[req.c]$.

Upon executing *req*, replica adds its own signature to the message (line 23). Then, the replica forwards the request to its $f + 1$ successors (line 26).

If there were less than $2f + 1$ valid signatures in the request, the replica adds its own signature, and forwards the updated request to its $f + 1$ successors (lines 28–29).

Step R$^+$4b.3a. *The replica commits the request on behalf of the client and forwards the commit history to the client. (lines 23–26 of Algorithm 4.7 on page 59)*

If r_i receives an *OBR* message with at least $2f+1$ valid signatures from other replicas, then the replica simply sends the $\langle GET\_A\_GRIP, h, req\rangle_{\mu_{r_i,req.c}}$ message to client named in $req.c$ (line 25). We say that r_i *commits* the *OBR* for *req* with the history h.

To counter possible message losses, if a replica has received repeated *PANIC* messages for *req* after committing the *OBR* for *req*, the replica replies to these messages by re-sending the *GET_A_GRIP* message to the client.

Step R$^+$4b.3a.1. *The client receives $f+1$ GET_A_GRIP messages containing the same history and commits the request. (Line 15 of Algorithm 4.3 on page 45)*

If the client has received $f+1$ $\langle GET\_A\_GRIP, h, req\rangle$ messages from different replicas, with the same history, the client commits the request by returning $Commit(req, h)$.

Processing Requests When the Sequencer Is Faulty

If the sequencer is faulty, it may omit sequencing the message, or might not reply to any of the *OBR* messages. In that case, the replicas will detect a fault, and proceed to switching to another instance, with another configuration. The steps that the replicas take are shown in Figure 4.10, which is similar to Figure 4.7, and the differences between the two are brought on by the situation in which a replica does not receive a response to the *OBR* message, after a predefined timeout.

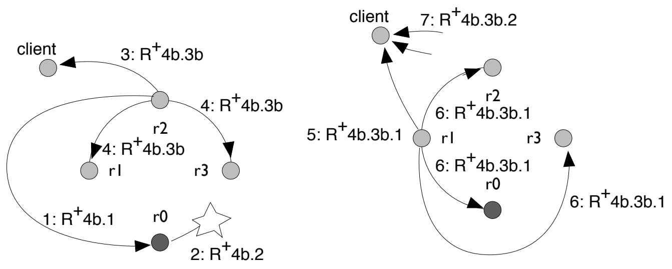

Figure 4.10: Alternative execution to the one shown in Figure 4.9: Replica r_2 cannot commit the request, because the sequencer does not respond. The replica aborts, and stops the current instance.

Step R$^+$4b.3b. *The replica* ***does not*** *commit the request on behalf of the client, stops processing new requests, and sends a signed history to the client. (lines 4–7 of Algorithm 4.4 on page 47)*

If replica r_i does not receive the *OBR* request with at least $2f+1$ valid signatures from other replicas, before the expiration of the timer, the replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces the current instance of the resilient mode to stop accepting requests (line 5); (b) sends its signed local history to client *req.c* using an $\langle ABORT, LH_{i\sigma_{r_i}}, req.t_c, r_i\rangle_{\mu_{r_i,req.c}}$ message (line 6); and (c) stops all of the OBR timers. In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica (line 7). Again, to counter possible message losses, we assume that r_i periodically retransmits this *STOP* message.

Step R$^+$4b.3b.1. *The replica receives a STOP message from some other replica, stops processing new requests, and sends a signed history to the client. (lines 7–9 of Algorithm 4.4 on page 47)*

Replica r_i aborts all of the requests from clients, similarly as in Step R$^+$4b.3b. Replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests; (b) sends its signed local history to all of the clients referenced in the active OBR timers[10], using an $\langle ABORT, LH_{i\sigma_{r_i}}, req'.t_c, r_i\rangle_{\mu_{r_i,req'.c}}$ message; and (c) stops all OBR timers. In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica. Again, to counter possible message losses, we assume that r_i periodically retransmits this *STOP* message.

Step R$^+$4b.3b.2. *A client receives $2f+1$ matching ABORT messages, extracts the abort history, and aborts the request. (lines 16–26 of Algorithm 4.3 on page 45)*

A matching *ABORT* message for a $\langle PANIC, req\rangle$ message is any *ABORT* message with a matching request identifier $req.t_c$. Once a client has received a matching *ABORT* message from $2f+1$ different replicas, it extracts the abort history *abortH* in the following way:

- the client generates the history *LH'*, such that *LH'*$[j]$ equals the value that appears at position $j \geq 1$ of $f+1$ different histories LH_i received in the *ABORT* messages. If such a value does not exist for position j, then *LH'* does not contain a value at positions j and higher.
- the longest prefix *LH"* of *LH'* is selected such that no request appears in *LH"* twice.
- if $req = \langle o, t_c, c\rangle$ does not exist in *LH"*, the request is appended to *LH"*. The resulting sequence is an abort history *abortH*.

[10]There is a timer for every outstanding *OBR* request *req'*.

Then, the client c aborts *req* by returning $Abort(req, abortH)$. To prove the validity of *abortH*, the abort history is accompanied by the set of $2f+1$ *ABORT* messages.

4.5 Correctness

Both operational modes of Ring are, in fact, an implementation of ABSTRACT, each with its own *Non-Triviality* property. The Non-Triviality property in an ABSTRACT model defines the conditions under which a protocol should commit client requests.

Definition *(Fast Mode Non-Triviality)*

If (a) a correct client c invokes a request m, (b) there are no replica failures, (c) the set of replicas (Σ) is synchronous, and (d) messages from c to Σ (and back) are synchronous, then the client c commits m.

Definition *(Resilient Mode Non-Triviality)*

If (a) a correct client c invokes a request m, (b) the sequencer is not faulty, (c) the set of replicas (Σ) is synchronous, and (d) messages from c to Σ (and back) are synchronous, then the client c commits m.

In addition, we say that a correct replica r_j executes *req at position pos* if $sn_j = pos$ when r_j executes *req*.

The list of properties every ABSTRACT instance must satisfy is given in Section 4.2. Before proving these properties, we first prove a set of auxiliary lemmas.

Definition *(Ring order)* The *ring order* defines the total order of replicas on the ring. We say that this ordering *starts* at a particular replica r_j, and define a total order operation such that: $j < j+1 < \cdots < j+3f$.

Figure 4.11 illustrates the circular topology of Ring. For the ring order which starts at replica r_0, we have the following relation: $r_0 < r_1 < r_2 < r_3$. On the other hand, if the order were to start at, for instance, r_2, we would have: $r_2 < r_3 < r_0 < r_1$.

Note that by protocol design, *Switching Monotonicity* holds for both Fast mode and Resilient mode instances, hence we will not prove this property.

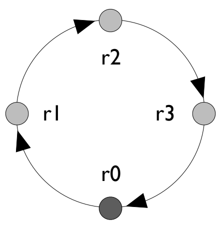

Figure 4.11: The circular topology of Ring

4.5.1 Fast Mode Correctness Proof

In this section, we prove that fast mode implements ABSTRACT with *Fast Mode Non-Triviality*. To do so, we need to show that fast mode satisfies the properties listed in Section 4.5. First, we prove some necessary lemmas.

Lemma 4.5.1. *Let r_j be a correct replica and let LH_j^{req} be the state of LH_j when r_j executes req. Then, LH_j^{req} remains a prefix of LH_j forever.*

Proof. A correct replica r_j modifies its local history LH_j only in *Step R2* (page 46) or *Step R3* (page 49) or *Step R4b.2* (page 53) by sequentially appending requests to LH_j. Hence, LH_j^{req} remains a prefix of LH_j forever. □

Lemma 4.5.2. *If a correct replica r_i accepts a request req (via the RING message) at time t_1, then all of the correct replicas r_j (req.entry $\le j < i$)*[11] *have accepted the request before t_1. Note that we do not discuss execution of the request. If replica* accepts *a request, it means that it has previously verified the request, and stored it in some internal structure.*

Proof. By contradiction. Assume that the lemma does not hold, and fix r_j to be the first correct replica that accepts *req*, such that there is a correct replica r_x ($x < j$) that never accepts *req*. We say that r_j *animates req*. Since *RING* messages are authenticated using RAs, r_j accepts *req* only if r_j receives a *RING* message with MACs authenticating *req* from all of the replicas from $\overleftarrow{r_j}$, that is, only *after* all of the correct replicas from $\overleftarrow{r_j}$ have accepted *req*. If $r_x \in \overleftarrow{r_j}$, r_x must have accepted *req* — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica which animates *req*, since any correct replica (at least one) from $\overleftarrow{r_j}$ animates *req* — a contradiction. □

Lemma 4.5.3. *If a correct replica r_i accepts a request req, then the request was invoked by a client.*

Proof. By contradiction, assume that some correct replica has accepted a request not invoked by any client, and let r_j be the first correct replica to accept such a request *req'* in *Step R2*

[11] If not stated otherwise, we presume to use the ring ordering.

(page 46). In the case in which $j \in \{req'.\text{entry} \dots req'.\text{entry} \oplus (f+1)\}$, r_j accepts the *req'* only if r_j receives a *RING* message with a signature from the client, that is, only if some client invoked *req*, or if *req* is contained in some valid *INIT* history. On the other hand, if j is not in that set, Lemma 4.5.2 yields a contradiction with our assumption that r_j is the first correct replica to accept *req'*. □

Lemma 4.5.4. *If a correct replica receives a non-nil sequence number (sn) for a request req, either through a RING, an ACK, or an OBR message, that sn was generated by the sequencer.*

Proof. By construction. The guard conditions in *Step R2*, and *Step R3* (page 49) prevent such case, along with the check of Ring Authenticators. □

Lemma 4.5.5. *If a correct replica r_i executes a request req, at position sn, at time t_1, then all of the correct replicas r_j ($0 \le j < i$) have executed the request at position sn before t_1. Note that here, we refer to the ring order.*

Proof. By contradiction, assume that the lemma does not hold, and fix r_j to be the first correct replica which executes *req* (at position *sn*), such that there is a correct replica r_x ($x < j$) which never executes *req*. We say that r_j is the first replica for which *req skips*. Since *RING* (and *ACK*) messages are authenticated using RAs, r_j executes *req* at position *sn* only if replica r_j receives a *RING* (or an *ACK*) message with MACs authenticating the pair $\langle req, sn\rangle$[12] from all of the replicas from $\overleftarrow{r_j}$, that is, only *after* all of the correct replicas from $\overleftarrow{r_j}$ have accepted *req*. If $r_x \in \overleftarrow{r_j}$, then r_x must have accepted *req* — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica at which *req* skips, since at any correct replica (at least one) from $\overleftarrow{r_j}$ *req* skips — a contradiction. Similar reasoning applies to the handling of an *OBR* request.

Note that the sequence number *sn* associated by the sequencer is indeed equivalent to the position at which a replica executes *req*, since (1) if the replica is the sequencer, *sn* is incremented by one, and (2) if the replica is not the sequencer, the replica accepts *req* with the associated sequence number, only if $sn' = sn + 1$. These conditions are described in *Step R2* (on page 46), *Step R3* (on page 49), and *Step R4a* (on page 50). □

Lemma 4.5.6. *If a correct replica r_i receives an ACK for the request req, at position sn and time t_1, then all of the correct replicas r_j ($req.\text{entry} \le j < i$) have executed the request req at position sn, before t_1. Note that we use the ring order, which starts at req.*entry.

Proof. If a replica r_i receives a valid *ACK*, that means that all of the correct replicas have received the request (execution condition in *Step R3* from page 49, and Lemma 4.5.2). From *Step R3*, and Lemma 4.5.5, we have that all of the correct replicas r_j ($0 \le j < i$) have executed the request. Let us fix the ring order, so that the sequence starts from 0, and ends at $3f$. We consider two cases:

[12]Where *sn* is not nil.

1. if $0 \leq req.\text{entry} < i$, then the claim follows immediately from Lemma 4.5.5;

2. if $0 \leq i < req.\text{entry}$, from *Step R2* on page 46 we get that *ACK* was generated at $req.\text{entry} \ominus 1$. It holds that $0 \leq i \leq req.\text{entry} \ominus 1$. From *Step R3*, by construction, we have that all of the correct replicas r_x ($x \in req.\text{entry} \ominus 1 \dots i$) have received the *ACK*. From the previous case, we have that the request is executed on all of the correct replicas r_k ($req.\text{entry} \leq k < 0$), and from Lemma 4.5.5 we have that request is executed on all of the correct replicas r_j ($0 \leq j < i$).

□

Lemma 4.5.7. *If a benign (that is, non-Byzantine) client c commits the request req with history h (at time t_1), then all of the correct replicas in $\Sigma_{\text{last}}^{req}$ execute req (before t_1) and the state of their local history upon executing req is h.*

Proof. To prove this lemma, notice that a correct replica $r_j \in \Sigma_{\text{last}}^{req}$ generates a MAC for the client authenticating *req* and $D(h')$ for some history h' (*Step R2*, or *Step R3*): (1) only after r_j has executed *req* and (2) only if the state of LH_j upon execution of *req* equals h'. Moreover, by *Step R2/R3*, no correct replica executes the same request twice. By *Step R4a* on page 50, a benign client (respectively, a replica) cannot commit *req* with h unless it receives a MAC authenticating *req* and $D(h')$ from every correct replica in Σ_{last}. Using Lemma 4.5.5, we get the claim. By *Step R4b.3a.1* (page 54), a benign client (respectively, a replica), cannot commit *req* with h unless it receives a *GET_A_GRIP* message with a MAC authenticating *req* and $D(h')$ from every correct replica in Σ_{last}. Again, using Lemma 4.5.5, we get the claim.

□

Next, we proceed with proving that Ring satisfies every ABSTRACT property.

Well-formed commit indications. By *Step R4a* (on page 50), in order to commit a request, the client needs to receive MACs authenticating $Digest_{LH} = D(h')$ for some history h' and a reply digest from all of the replicas from $\Sigma_{\text{last}}^{req}$, including at least one correct replica. By *Step R3* from page 49, the digest of the reply sent by a correct replica is $D(rep(h'))$. Hence, h' is exactly the commit history h and is uniquely defined, due to our assumption of collision-free digests.

Moreover, since a correct replica executes an invoked request before sending an *ACK* message in *Step R3* (or a *GET_A_GRIP* message in *Step R4b.3a* on page 54), it is straightforward to see that if *req* is committed with a commit history h_{req}, then *req* is in h_{req}.

□

Validity. For any request *req* to appear in an abort (commit) history h, at least $f+1$ replicas must have sent h (respectively, a digest of h) in *Step R3* on page 49 (or in *Step R4b.3a.1* on page 54), such that $req \in h$. Hence, at least one correct replica has executed *req*.

Directly from Lemma 4.5.6, we observe that all of the correct replicas execute only requests invoked by clients.

Moreover, by *Step R2* or *Step R3* or *Step R4b.1*, no replica executes the same request twice (every replica maintains a list of last-seen identifiers — $t_j[c]$). Hence, no request can appear twice in any local history of a correct process, and consequently, no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction.

□

Termination. By assumption of a quorum of $2f+1$ correct replicas and fair-loss links: (1) correct replicas eventually receive a PANIC message sent by a correct client c (in *Step R4b* on page 51) and (2) c eventually receives $2f+1$ *ABORT* messages from correct replicas (sent in *Step R4b.3b* from page 55). Hence, if the correct client c panics, the client eventually aborts the invoked request *req*, in the case in which c did not commit *req* beforehand.

Moreover, to see that a committed request *req* must be in its commit history h_{req}, notice that the client needs to receive a MAC for the same local history digest $D(h_{req})$ from all of the $f+1$ replicas from $\Sigma_{\text{last}}^{req}$, including at least one correct replica r_j. By *Step R2*/*R3*, r_j executes *req*, and appends the request to the replica's local history LH_j before authenticating the digest of LH_j. Therefore, $req \in h_{req}$. By *Step R4b.2* on page 53, the replica r_j executes *req*, and appends the request to its local history LH_j. Furthermore, the replica embeds this history in the *OBR* message. Only after these steps, and prior to sending the *GET_A_GRIP* message to the client, does the replica r_j authenticate the digest of LH_j. Hence, $req \in h_{req}$.

□

Commit Order. Assume, by contradiction, that there are two committed requests *req* (by a benign client c) and $req' \neq req$ (by a benign client c') with different commit histories h_{req} and $h_{req'}$, such that neither is the prefix of the other. By Lemma 4.5.7, all of the correct replicas in $\Sigma_{\text{last}}^{req}$ ($\Sigma_{\text{last}}^{req'}$) have executed the request *req* (respectively, *req'*), with history h_{req} (respectively, $h_{req'}$). Let r^{req} be the first correct replica in $\Sigma_{\text{last}}^{req}$, and let $r^{req'}$ be the first correct replica in $\Sigma_{\text{last}}^{req'}$. There are two distinct cases:

- these replicas are the same ($r^{req} = r^{req'}$). A contradiction with Lemma 4.5.1.
- one of the replicas precedes the other, in the ring order which starts from the sequencer. Without loss of generality, we can assume that $r^{req} < r^{req'}$. By Lemma 4.5.5, r^{req} has executed all of the requests that $r^{req'}$ has executed, at the same positions. A contradiction. □

Abort Order. Let us assume, by contradiction, that there is a committed request req_C (by some benign client) with commit history h_{req_C} and an aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. By Lemma 4.5.7, and the assumption of at most f faulty replicas, all of the correct replicas (at least one) from $\Sigma_{\text{last}}^{req_C}$ execute req_C, and their state upon executing req_C is h_{req_C}. Let $r_j \in \Sigma_{\text{last}}^{req_C}$ be a correct replica with the highest (w.r.t. the ring order which starts at $req_C.entry$) index among all of the replicas in $\Sigma_{\text{last}}^{req_C}$. By Lemma 4.5.6, all of the correct replicas r_k ($req_C.entry \le k < j$) execute all of the requests in h_{req_C} at the same positions that these requests have in h_{req_C}.

In addition, a correct replica executes all of the requests in h_{req_C} before sending any $ABORT$ message (*Step R4b.3b.1*, page 55); indeed, before sending any $ABORT$ message, a correct replica must stop any further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an $ABORT$ message, h_{req_C} is a prefix of LH_j.

Finally, by *Step R4b.3b.2* (page 56), a client which aborts a request waits for $2f+1$ $ABORT$ messages, including at least $f+1$ from correct replicas. By construction of the abort history, every commit history including h_{req_C} is a prefix of every abort history, including h_{req_A}. A contradiction.

□

Init Order. Under the constraint that if a replica's local history is empty, the first request to which the sequencer can assign the sequence number and the first request a replica may execute must be *INIT* requests, we obtain that replicas initialize their local histories before sending any *RING*, *ACK* or *ABORT* request.

Since any common prefix CP of all of the valid init histories is a prefix of any particular init history IH, CP is a prefix of every local history sent by a correct replica in a *RING* or $ABORT$ message. Init Order for commit histories immediately follows. In the case of abort histories, notice that out of at least $2f+1$ $ABORT$ messages received by a client on aborting a request in *Step R4.3b.2* (page 56), at least $f+1$ are sent by correct processes and contain local histories that have CP as a prefix. Hence, by *Step R4b.3b.2*, CP is a prefix of any abort history.

□

Non-Triviality. Non-Triviality relies on the fact that the timer of the client, triggered in *Step R1* is set in such a way that it does not expire in the case when the set of replicas, including the client, is synchronous.

Let us assume, by contradiction, that there is a correct client *c* which panics and denote the first such time by t_{PANIC}. The client *c* has invoked the request *req* at $t = t_{PANIC} - (2(3f+1)+2)\Delta$. Since no client has panicked by t_{PANIC}, all of the replicas will have executed all of the requests they have received by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that: (i) by the time $t+\Delta$, the entry replica receives *req* and takes *Step R2*, and (ii) by the time

$t+3f\Delta < t_{PANIC}$ all of the replicas take *Step R2* for *req*, and (iii) by the time $t+(2(3f+1)-1)\Delta < t_{PANIC}$ all of the replicas take *Step R3.* Since the sequencer is correct, then we have that all of the replicas execute all of the requests received before t_{PANIC} in the same order (established by the sequence numbers assigned by the sequencer). Hence, by $t+(2(3f+1)+2)\Delta = t_{PANIC}$, c receives $f+1$ identical replies (*Step R4a*), commits the request *req*, and never panics. A contradiction.

In addition, a correct replica r_i executes *Step R4b.3b* (on page 55) and stops appending new requests, only if r_i fails to commit an OBR request for a *RING* message signed by some client. Since such an OBR request cannot raise a verification failure, r_i can fail to commit such a request only in the case in which there is asynchrony in the set of replicas, or in the case in which some replica has failed.

□

4.5.2 Resilient Mode Correctness Proof

In this section, we prove that resilient mode implements ABSTRACT with *Resilient Mode Non-Triviality.* First, we prove several auxiliary lemmas.

Lemma 4.5.8. *If a correct replica r_i receives a request req (via the OBR message) at time t_1, then all of the correct replicas r_j ($0 \le j < i$) have received that request before t_1.*

Proof. By contradiction. Let us assume that the lemma does not hold, and let us fix r_j to be the first correct replica which receives *req*, such that there is a correct replica r_x ($x < j$) that never receives *req*. We say that r_j is the first replica for which *req obr-skips.* A correct replica sends a request to its $f+1$ successors. Hence, if $r_x \in \overleftarrow{r_j}$, r_x must have received *req* — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica for which *req obr-skips*, since any correct replica (at least one) from $\overleftarrow{r_j}$ obr-skips *req* — a contradiction. □

Lemma 4.5.9. *When processing OBR requests, after at most* $\min(f+1,4)$ *communication steps from the time the non-malicious sequencer has received an OBR request, all of the replicas will have received the message.*

Proof. By contradiction. Let us assume that it takes more than four steps for all of the replicas to receive the request. Let R_1 be the last replica in the ring order which has received the request in the first step. Similarly, let R_2 (R_3, R_4, R_5) be the last replicas which have received the request in the second (respectively, third, fourth, fifth) step. Let d_0 be the distance between the starting replica r_0 and R_1. Likewise, let d_1 be the distance between R_1 and R_2, d_2 be the distance between R_2 and R_3, etc... We have the following equations:

$$d_0 + d_1 + d_2 + d_3 + d_4 < 3f + 1 \tag{4.1}$$
$$1 \leq d_0, d_1, d_2, d_3, d_4 \leq f + 1 \tag{4.2}$$
$$f + 1 \leq d_0 + d_1 \tag{4.3}$$
$$f + 1 \leq d_1 + d_2 \tag{4.4}$$
$$f + 1 \leq d_2 + d_3 \tag{4.5}$$
$$f + 1 \leq d_3 + d_4 \tag{4.6}$$
$$2f + 1 \leq d_0 + d_1 + d_2 \tag{4.7}$$
$$2f + 1 \leq d_1 + d_2 + d_3 \tag{4.8}$$
$$2f + 1 \leq d_2 + d_3 + d_4 \tag{4.9}$$

Equation 4.1 states that, after five communication steps, we reach all of the correct nodes on the ring (at most $3f + 1$). Equations 4.3-4.6 state that a replica reached in two steps could not have been reached in a single step. Similarly, Equations 4.7-4.9 state that a replica reached in three steps could not have been reached in less than three steps. From Equations 4.7 and 4.6, we get a contradiction with Equation 4.1:

$$(d_0 + d_1 + d_2) + (d_3 + d_4) \geq 3f + 2 \tag{4.10}$$

When $f = 1$ or $f = 2$, we take less equations into consideration. In the case in which $f = 1$, only d_0, d_1, and d_2 exist. Similarly, when $f = 2$, only d_0–d_3 exist. □

Lemma 4.5.10. *When processing OBR requests, after at most* $\min(2f + 2, 8)$ *communication steps from the time the non-malicious sequencer receives an OBR request, all replicas will receive the message with* $2f + 1$ *correct signatures.*

Proof. Due to Lemma 4.5.9, after $\min(f + 1, 4)$ communication steps, all correct replicas will have a copy of the message (assuming that the sequencer is correct). All correct replicas will forward the message further. Thus, if we apply Lemma 4.5.9 once more, treating each replica as the sequencer, then after additional $\min(f + 1, 4)$ steps, all replicas will have messages from all other correct replicas. Since all correct replicas memorize the set of previously seen signatures for the request (Line 13 of Algorithm 4.7), after $\min(2f + 2, 8)$ communication steps all replicas will receive the message with $2f + 1$ correct signatures. □

Well-formed commit indications. The proof is the same as for the fast mode case.

Validity. The proof is similar as the proof for the fast mode case.

Init Order. The proof is the same as for the fast mode case.

Termination. The proof is the same as for the fast mode case.

Commit Order. Let us assume, by contradiction, that there are two committed requests *req* (by a benign client *c*) and *req' ≠ req* (by a benign client c') with different commit histories h_{req} and $h_{req'}$, such that neither of them is the prefix of the other. The clients commit requests either as a response to a *RING*, or to a *PANIC* message. There are three possible cases:

- Both of the committed requests are direct responses to *RING* messages. By Lemma 4.5.7, all of the correct replicas in $\Sigma^{req}_{\text{last}}$ ($\Sigma^{req'}_{\text{last}}$) have executed the request *req* (respectively, *req'*), with history h_{req} (respectively, $h_{req'}$). Let r^{req} be the first correct replica in $\Sigma^{req}_{\text{last}}$, and let $r^{req'}$ be the first correct replica in $\Sigma^{req'}_{\text{last}}$. There are two distinct cases:
  - these replicas are the same ($r^{req} = r^{req'}$). A contradiction with Lemma 4.5.1.
  - one precedes the other, in the ring order which starts from the sequencer. Without loss of generality, we can assume that $r^{req} < r^{req'}$. By Lemma 4.5.5, r^{req} has executed all of the requests that $r^{req'}$ has executed, at the same positions. A contradiction.
- Both of the committed requests are a direct response to *OBR* messages. From *Step R⁺4b.3a.1* (on page 61), a client commits a request, if there are $f+1$ matching *GET_A_GRIP* messages. By *Step R⁺4b.3a* on page 61, a replica executes a request and sends a *GET_A_GRIP* message if there are at least $2f+1$ correct signatures. Thus, each of the clients commits a request after receiving a message executed by at least $f+1$ correct replica. These two sets (carried in *GET_A_GRIP* messages) of correct replicas intersect on one correct replica, which has executed both of the requests. A contradiction with Lemma 4.5.1.
- One committed request is a direct response to a *RING* message, while the other is a direct response to an *OBR* message. Without loss of generality, let us assume that client *c* has committed *req* as a direct response to the *RING* message, while the client c' has committed *req'* as a direct response to the *OBR* message. By Lemma 4.5.7, all of the correct replicas in $\Sigma^{req}_{\text{last}}$ have executed *req*. Let r^{req} be the first correct replica in $\Sigma^{req}_{\text{last}}$. By Lemma 4.5.5, all of the correct replicas in the range $\{r_{req.\text{entry}} \dots r^{req}\}$ have executed the request, and there are at least $f+1$ correct replicas in that range (as r^{req} belongs to the last $f+1$ replica in the ring orders starting from *req*.entry). Similarly to the previous case,

the client c' commits the request after receiving $f+1$ matching *GET_A_GRIP* messages. Every replica which has sent the *GET_A_GRIP* message had executed the request after receiving an *OBR* message with at least $2f+1$ signature. Thus, the set of correct replicas which has executed req, and the set of replicas which has executed *req'* intersect on at least one correct replica. A contradiction with Lemma 4.5.1.

Abort Order. Let us assume, by contradiction, that there is a committed request req_C (by some benign client) with a commit history h_{req_C} and an aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. There are two different cases:

- req_C was committed without the client sending a *PANIC* message. By Lemma 4.5.7, and the assumption of at most f faulty replicas, all of the correct replicas (at least one) from $\Sigma_{\text{last}}^{req_C}$ execute req_C, and their state upon executing req_C is h_{req_C}. Let $r_j \in \Sigma_{\text{last}}^{req_C}$ be a correct replica with the highest (w.r.t. the ring order which starts at req_C.entry) index ind among all of the replicas in $\Sigma_{\text{last}}^{req_C}$. By Lemma 4.5.6, all of the correct (at least $f+1$) replicas r_k ($req_C.\text{entry} \le k < j$) execute all of the requests in h_{req_C} at the same positions that these requests have in h_{req_C}.
- req_C was committed during the handling of the *PANIC* message sent by the client. By Lemma 4.5.10, and *Step R⁺4b.3a* (on page 61), all of the correct replicas (at least $2f+1$ replicas) execute req_C.

In addition, a correct replica executes all of the requests in h_{req_C} before sending any $ABORT$ message (*Step R⁺4b.3b.1*, listed on page 62); indeed, before sending any $ABORT$ message, a correct replica must stop any further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an $ABORT$ message, h_{req_C} is a prefix of LH_j.

Finally, by *Step R⁺4b.3b.2* from page 62, a client that aborts a request waits for $2f+1$ $ABORT$ messages, including at least $f+1$ from correct replicas. By construction of the abort history, every commit history, including h_{req_C}, is a prefix of every abort history, including h_{req_A}. A contradiction.

□

Non-Triviality. Non-Triviality relies on the fact that the timer of the replica, triggered in *Step R⁺4b.1* (on page 58) is set in such a way that it does not expire in the case when the set of replicas, including the client, is synchronous.

Let us assume, by contradiction that there is a correct replica r which stops, and denote the first such time by t_{STOP}. Replica r has sent the *OBR* message m at $t = t_{STOP} - ((2f+1)+1)\Delta$. Since no client has panicked by t_{PANIC}, all of the replicas will have executed all of the requests they have received by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that: (i) by the time $t + \Delta$, the sequencer receives m and takes *Step R$^+$4b.2* (page 60), and (ii) by the time $t + (f + 1 + 1)\Delta < t_{STOP}$, all of the correct replicas take *Step R$^+$4b.2* for m, and (iii) by the time $t + ((2f + 1) + 1)\Delta < t_{STOP}$, all of the correct replicas take *Step R$^+$4b.3a* (page 61). Since the sequencer is correct, then we have that all of the correct replicas execute of the all requests they have received before t_{STOP} in the same order (established by the sequence numbers assigned by the sequencer). Hence, by $t + ((2f + 1) + 1)\Delta = t_{STOP}$, r receives a message with at least $2f + 1$ signatures (*Step R$^+$4b.3a*), commits the request *req* (associated with m) and does not abort. A contradiction.

In addition, a correct replica r_i executes *Step R$^+$4b.3b* (on page 62), and stops appending new requests, only if r_i fails to commit the OBR request for a *RING* message signed by some client. Since such an OBR request cannot raise a verification failure, r_i can fail to commit such request only in the case there is an asynchrony in the set of replicas, as, per Lemma 4.5.10, if the sequencer is correct, a malicious replica cannot prevent correct replicas from receiving the *OBR* message.

$\square$

5 Optimizations and Extensions

We have designed several techniques which improve the efficiency of BFT implementations, and have implemented some of them in Ring. This chapter addresses these techniques, which range from protocol optimizations to protocol extensions which enable the detection of low-performance operations. In Section 5.1, we have described protocol optimizations, while the methods of handling authentication are described in Section 5.2. The last section elaborates on the methods of detecting and defending against slow replicas.

5.1 Optimizations

We have implemented a set of optimizations to further improve the performance of Ring. These optimizations mostly aim at reducing the number of performed MAC operations per request, and the number of sent messages. These optimization are suitable for the fast mode of operation, although they may also be applicable to the resilient mode.

The tasks of optimizing for the reduction of the number of performed MAC operations, and lowering the overall number of sent messages have been challenging, as Ring Authenticators carry dependencies on the content of the request during the next $f+1$ communication steps, and each of these fields in the message is serially processed by a different replica. Let us consider, for instance, a request entering the system at replica 1. At some later point in time, replica 1 receives the acknowledgement from the sequencer (replica 0), and needs to authenticate the request using a MAC from both replica 3 and replica 0. Replica 3 has created the MAC for the request, without a sequence number being set. Replica 0 created the MAC for the acknowledgement, with the sequence number being set (replica 0 had updated this sequence number). Replica 1 needs to take both of these conditions into account during the verification of MACs. The next section explains the encountered challenges in detail, along with our approach to solving them.

5.1.1 Piggybacking

The goal of this optimization is to reduce the number of messages which are sent over the network. The optimization works as follows: when a replica generates an *ACK* message (the acknowledgement message), it takes one (or more) client request(s), and piggybacks the acknowledgement onto the request. Next, the replica generates Ring Authenticators for the union of the requests and the acknowledgement. Lastly, the replica forwards the piggybacked request. When such a message reaches the last replica for the request, acknowledged by the piggybacked acknowledgement, this last replica needs to take special care in order to generate a proper MAC for the client, and also to generate proper MACs for the request(s) onto which the acknowledgement was piggybacked.

Figure 5.1 illustrates the process of the piggybacking of acknowledgements, when two clients send requests. Replica 1 receives a forwarded request from replica 0, and detects an outstanding request from client *b*. First, replica 1 creates an acknowledgement for the request forwarded by replica 0 (from client *a*), and then attaches that acknowledgement to the request from client *b*. The subsequent replicas process both the acknowledgement and the request. Later, replica 1 receives back the acknowledgement it has created for the request from client *a*, and the acknowledgement of the request from client *b*[1]. Then, replica 1 performs the following steps: (1) splits the message, (2) takes the acknowledgement of the request from client *a*, (3) replies to the client *a*, and (4) forwards the rest of the message to replica 2.

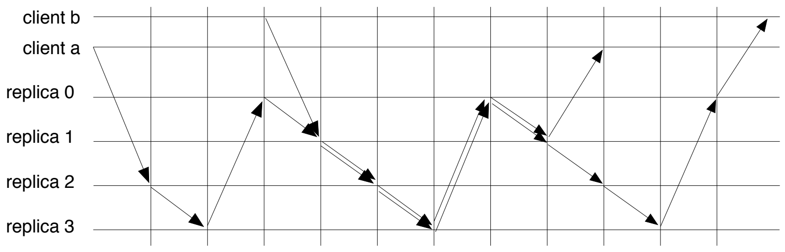

Figure 5.1: An illustration of piggybacking (f=1). Solid lines represent the request from the client *a*, while dashed lines represent the request from the client *b*. Thick lines denote requests, thin lines denote corresponding acknowledgements. Piggybacked requests are represented as a line parallel to the line of the carrying request.

Note that this optimization can be considered fragile, as malicious clients could disrupt the performance of Ring by sending malformed messages, which would be dropped at later replicas. Indeed, when an acknowledgement is piggybacked onto a new request, and the request authentication fails, both the acknowledgement and the request will be dropped. For that purpose, we have decided to disable this optimization when the number of committed requests between two switchings to the resilient mode is below a configurable threshold.

[1]Replica 0 has created the acknowledgement for the request of client *b* in the previous step

5.1.2 Batching

The goal of this optimization is twofold. First, it aims at the decrease of the number of messages sent over the network, and to reduce the number of MAC operations which are performed per request. Upon receiving a request from a client, a replica checks whether there are other pending requests from other clients. If there are such requests, the replica batches them together, and generates RAs for the union of requests. Lastly, the replica forwards the batch. Note that the first $f+1$ replicas need to verify client MACs, for each single request of the batch, and a joint MAC for the whole batch. Moreover, the last $f+1$ replicas need to generate MACs for the whole batch for their successors in the Ring, and a MAC for every client separately. Finally, note that when generating the acknowledgement for the batch, the replica creates a batch of acknowledgements, to allow for message fragmentation. Request batching is shown in Figure 5.2.

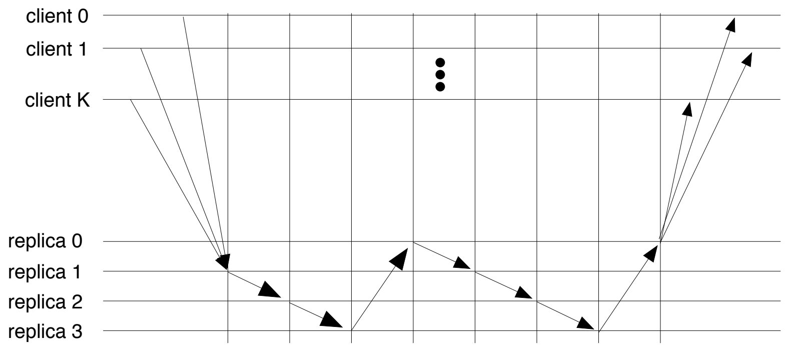

Figure 5.2: An illustration of batching (f=1). Thick line represents a batch of requests, while a thin line between replicas represents a batch of acknowledgements. The difference in thickness of the lines symbolizes the fact that the acknowledgements are smaller in size than the requests.

Similarly to piggybacking, this is a fragile optimization that we choose to disable when the number of committed requests between two switches to the resilient mode is below a configurable threshold.

5.1.3 Read Optimization

The goal of the read optimization is to reduce the latency of read requests. In general case, read requests do not need to be totally ordered, because they do not affect the state of replicas. However, if different replicas have a different state, then read operation will return mismatching replies. In such cases, clients submit read requests so that these read operations are totally ordered by the protocol.

Similar approach is taken in Ring. Having to totally order read requests would get them to circulate twice around the ring, which would unnecessarily add to the latency, and increase the processing time per request on each replica. Thus, we first submit read requests so they exit the ring after being processed by $f+1$ consecutive replicas, as shown in Figure 5.3. Once a client has received the reply to its read request, it compares the $f+1$ MACs contained in the reply. If they match, the client commits the reply. Otherwise, it is a signal to the client that the replicas had been in different states. Thus, the client sends the read request as a write request, in order for it to be totally ordered.

Note that read requests can be batched with write requests. However, that would complicate the authentication and verification of requests (generation of MACs). Therefore, in order to keep the protocol implementation simple, read requests are only batched with other read requests.

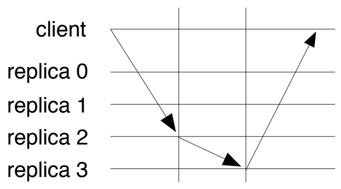

Figure 5.3: Illustration of a read-only operation (f=1).

Note that we have also tested the read optimization used in state-of-the-art BFT protocols [Amir et al., 2008; Castro and Liskov, 1999; Clement et al., 2009a; Kotla et al., 2007], in which clients multicast their read requests to $f+1$ replicas and wait for $f+1$ matching replies. We have observed that this approach does not yield good performance. With such a read optimization, we heve observed a high number of request retransmissions, due to mismatching replies (as different replicas on the ring were in different states). The reason for this behaviour is that the pipelining approach used for request propagation interferes with the parallel approach used to send read requests.

5.1.4 Out-of-Order Caching

The protocol description (Section 4.3.3 and Section 4.4.1) dictates that replicas should process requests in ascending order of their sequence numbers. In practice, this is hard to achieve in Ring, because the requests arrive in different order, due to the fact that request obtain their sequence number at different stages of processing. Thus, to increase performance, Ring caches all messages, processes them out-of-order (of arrival), but according to their appropriate sequence numbers.

5.1.5 Checkpointing

As the execution proceeds, message logs of all of the executed requests may grow without bound. Thus, the replicas must truncate the log containing the messages that have been executed on all of the replicas. The replicas have to be careful not to delete requests which have been seen or executed by less than $2f+1$ replicas, as safety would be violated otherwise. To truncate the execution history, Ring uses the Lightweight Checkpointing Subprotocol (LCS) of the ABSTRACT framework, which we briefly outline in the this section.

LCS truncates the history by a predefined number of messages, a number which is the same for every replica. LCS consists of the following steps, which are executed by every replica, independently of the main request processing:

- Every replica keeps a checkpointing counter, and increments it for every executed request;
- Whenever that counter reaches a predefined threshold value, every replica sends a checkpoint message to all of the other replicas, containing the signed digest of the history and the value of the counter;
- Whenever a replica sends the checkpoint, it triggers a checkpoint timer. If the timer expires, the replica stops and aborts all future requests;
- When a replica receives the same checkpoint from all of the other replicas, it collapses its history. The replica truncates the history up to the checkpointing counter referenced in the checkpoint message.

5.2 Authentication Challenges in Ring

As mentioned in Section 4.3.1, MAC authenticators are an important overall optimization, due to their good performance characteristics over digital-signatures. Another difference between the two is that digital signatures are written only once per message, per sender, while the authenticators are usually a vector of MACs, each for one designated recipient. Thus, handling MAC authenticators is not as straightforward as handling digital signatures, especially with the optimizations described in the previous section. In this section, we outline several challenges of using MACs over complex message interactions. Also, we describe an attack [Clement et al., 2009a], which causes a crash in all state-of-the-art BFT protocols, while reducing performance in Ring. We also describe our counter-measures for this attack.

5.2.1 Optimizations and Authenticators

In this section, we describe the effect of each of the optimizations on MAC handling.

When creating a batch of requests, a replica will create an authenticator for all of the messages in the batch. All successive replicas will verify the whole batch, create a batch of acknowledgements, and process each acknowledgement in the batch. However, there are some issues which need to be taken into consideration during the handling of batched messages. The first $f+1$ replicas will need to read a MAC per client, for a single message, and for each of the replicas. Similarly, the last $f + 1$ replica will need to write, for each of the messages in the batch, a single MAC for the client who created the request. Clearly, the process of handling MACs for batched messages becomes a complex task, as there are many conditions that need to be taken into account.

When handling piggybacked requests, authenticator processing becomes more complicated. The reason is as follows — when a replica receives a piggybacked request, and if the replica is one of the f successors of the entry replica of the request, it needs to do the following tasks: (1) check the MAC written by the client for the carrier request; (2) check the MACs, written by the predecessors of the entry replica, of the carrier request, *for the piggybacked acknowledgement*; (3) check the MACs written by its predecessors, for the joint, piggybacked request (both for the request, and the acknowledgement). Furthermore, the f predecessors of the exit replica *for the piggybacked acknowledgement* need to write the MACs for the carrier request *and* the acknowledgement separately.

Therefore, writing MAC authenticators for a combination of batched and piggybacked requests becomes quite a complex operation, one which requires careful design and implementation. Our evaluation shows that batching and piggybacking do indeed improve performance. However, the trade-off lies between the performance of the code, and its maintainability.

5.2.2 Attacks On MACs

Clement et al. [2009a] described a well-known attack against all state-of-the-art BFT protocols which use MAC authenticators. The crux of the attack is in the independence of MACs in the authenticator — there is one MAC per receiver, stored in the authenticator. For example, Zyzzyva's communication pattern is such that the primary first receives the request, sequences it after authentication, and then multicasts it to other replicas. These other replicas receive the request, authenticate it, and execute it. When a client writes MAC authenticators in such a way that the primary can only correctly verify the MAC for itself, while other replicas fail the authentication, it will create a discrepancy in the state of the replicas. Thus, the system will need to go through an expensive phase of re-consolidation, in order to correctly continue its operation. In theory, this attack should affect performance, while, in practice, it causes of the all replicas to crash, leaving the system in an unusable state. A similar behaviour occurs with Chain, as well.

As Ring's fast mode aims toward high performance in the best case (in which there are no errors), any additional processing may negatively affect the performance. Thus, we are faced with a trade-off between high-performance and increased robustness. In the general case, one solution would be to use digital signatures instead of Ring Authenticators. However, this approach severely reduces performance. Thus, in the general case, we leave the task of fighting off such attacks to either blacklists, or by forcing the system to go through the resilient mode.

Nevertheless, in a configuration for tolerating at most one fault ($f = 1$)[2], Ring's topology allows for a simple protection against such an attack. The attack poses a problem only if a replica, other than the primary (respectively, the head, the sequencer) receives the request to which it assigns a sequence number, but that request still contains a MAC from the client, intended for other replicas. In Ring, we solve this problem by prohibiting the sequencer to sequence any requests for which it is the entry replica. This forces the sequencer to assign the sequence number to the *the acknowledgement* of the request. This approach does not violate the safety, nor the correctness of the algorithm, because all of the other replicas in the system will receive the acknowledgement with the sequence number.

5.3 Low Performance Detection

Ring has a highly symmetrical topology but, on the other hand, it exhibits a linear communication pattern: a replica receives a request from its predecessor, and forwards it to its successor. These two factors make Ring sensitive to performance disruptions. If there is at least one replica being slower then the rest, the performance of the whole system will be affected. To this regard, Ring is similar to quorum-based systems, where any single member is capable of affecting the performance of the whole system. Essentially, Ring is as fast as the slowest replica, entailing that malicious or faulty replicas can significantly hurt its performance.

To remedy this situation, Ring runs its slowness detection algorithm, which detects if there are any slowdowns in the system. If the algorithm returns a positive answer, the replicas can signal the operator about the observed fault, or switch to another instance, possibly using a different set of replicas.

5.3.1 Collected Metrics

Each replica r_i in Ring collects the following statistics, in last 1, and 5 s:

- throughput of requests processed from the predecessors, per entry replica ($b_q^i[j], j \in (0, 3f+1)$)
- throughput of requests sent to the successor (B_q^i)
- throughput of requests processed from clients (b_c^i)
- throughput of requests sent to clients (B_c^i)
- average round trip time (RTT) for requests for which the replica is the entry replica (RTT^i)
- ping time (denoted T_{hop}^i), which represents the time it takes the request to reach replica $i+1$.

[2]Which should be the most prevalent configuration.

5.3.2 Slowness Detection Algorithm

This algorithm relies on the symmetry of Ring. The intuition here is that if all of the replicas were to share the information on their working conditions, then the majority of them will report what the good working conditions are.

Another insight comes from the manner in which the replicas in Ring process the requests. Each of the replicas keeps two queues of incoming requests — one queue accepts requests from clients, while the other queue holds requests from the predecessor. The replica processes requests from both of the queues, one-by-one, so that the ratio between the throughputs from both queues is around $3f + 1$, in the favour of the predecessor queue, because that is the number of flows (from different replicas) that the predecessor queue holds. We use the well-known token bucket algorithm Shenker and Wroclawski, 1997 for keeping the ratio between the flows at $3f + 1$.

In our approach to detecting the slowness in the system, we rely on the fact that if a request were to leave from one replica to another, and immediately return, the whole round trip would capture the propagation time and the queueing time on both replicas. The queueing time accounts for the processing of all of the previous requests, giving an insight into the working conditions.

In brief, the outline of the slowness detection algorithm is:

- each of the replicas measures the average time it takes a request to go once over the ring ($\text{RTT\_}avg$ - Round trip time).
- each of the replicas measures the time it takes the successor to accept and process a message.
- each of the replicas exchanges its previous two values with other replicas.
- based on the gathered data, if the round trip time and the handling time at the successor do not match, the replica can suspect the presence of slow replicas in the system.

The intuition behind Algorithm 5.1 is quite simple. A replica sends a ping (or piggybacks it to another message) to another replica, and measures how long it takes for it to receive an answer. Since both replicas operate in the same conditions, half of that time is the propagation of the message to the processing site at the next replica.

Algorithm 5.2 describes a part of the slowness detection algorithm, in charge of exchanging data received in Algorithm 5.1, and deciding if there exists a slow replica in the system or not. A replica sends a signed message with its own time, and each other replica appends (after signing) its own time to the message. Once the message has returned to its originating replica, it will contain times measured by all of the other replicas. At that moment, the originating replica can decide if there are any slow replicas in the system. Also, it forwards this message

Algorithm 5.1 Ring: calculating propagation time between neighbours

```
procedure send_ping() is
1:   {this procedure is executed periodically}
2:   $T_{ping} \leftarrow$ current_time()
3:   nonce $\leftarrow$ new nonce
4:   send ⟨PING, nonce⟩ to $r_{i+1}$

upon event ⟨PING, nonce⟩ from $r_{i-1}$ do
5: send ⟨PONG, nonce⟩ to $r_{i-1}$

upon event ⟨PONG, nonce'⟩ from $r_{i+1}$ do
6: if nonce' = nonce then
7:   $T_{pong} \leftarrow$ current_time()
8:   $T_{hop} \leftarrow \frac{T_{pong} - T_{ping}}{2}$
```

along the ring once more, so all of other replicas can obtain all of the data values, and make a decision for themselves.

Algorithm 5.2 Exchange of propagation times

```
procedure exchange_ping_times() is
1:   {this procedure is executed periodically}
2:   nonce $\leftarrow$ new nonce
3:   HOPSET $\leftarrow \emptyset \cup \langle T_{hop}, nonce\rangle_\sigma$
4:   send ⟨RT_CHECK, nonce, i, HOPSET⟩ to $r_{i+1}$

upon event ⟨RT_CHECK, nonce, entry_replica, HOPSET⟩ from $r_{i-1}$ do
5: if entry_replica = i
   then
6:   if $\forall$ i $\in$ HOPSET: signature(i) is correct then
7:     send ⟨RT_REPORT, nonce, i, HOPSET⟩ to $r_{i+1}$
8:     decide_slowness(HOPSET)
9: else
10:  HOPSET $\leftarrow$ HOPSET $\cup \langle T_{hop}, nonce\rangle_\sigma$
11:  send ⟨RT_CHECK, nonce, entry_replica, HOPSET⟩ to $r_{i+1}$

upon event ⟨RT_REPORT, nonce, entry_replica, HOPSET⟩ from $r_{i-1}$ do
12: if i $\neq$ entry_replica then
13:   if $\forall$ i $\in$ HOPSET: signature(i) is correct then
14:     decide_slowness(HOPSET)
```

The slowness detection algorithm (Algorithm 5.3) is used to detect the presence of slow replicas. Calculations in this algorithm rely on K_{lat}, a known network-specific constant which accounts for latency variability. After receiving all ping times, a replica sorts these times, picks the $f+1$-th time from the top of the list, and uses this value to estimate the message propagation time around the ring. If the calculated time (with latency variability taken into account) is less than the measured time, we have a signal that there is a slow replica in the system. The replica broadcasts its finding (in the form of a *yes* or *no* decision). Then, the replica waits to receive findings from $2f$ other replicas. If the majority ($f+1$) of decisions is *yes*, the replicas need to either switch to a new instance of ABSTRACT, or try to detect which replicas are slow.

When performing the comparison between the calculated and the measured time, the replicas in Ring take into the account a constant ϵ. ϵ is set by the operator, and describes the allowed extent of variance in the performance.

Algorithm 5.3 Slowness detection algorithm

```
Initialization:
1:  certificate ← ∅

Implementation:
procedure decide_slowness(HOPSET) is
2:    times ← []
3:    for all H ∈ HOPSET do
4:        {get all times in a list}
5:        times ← times || H.$T_{hop}$
6:    sort_ascending(times)
7:    $T_{cand}$ ← times[f+1]
8:    $TAT_{exp} \leftarrow (3f+1) * T_{cand} * K_{lat}$
9:    nonce ← new nonce
10:   if $TAT_{exp} * (1+\epsilon) < TAT_{measured}$ then
11:       send ⟨RT_DECIDE, i, nonce, $\langle YES, nonce\rangle_\sigma$⟩ to all
12:       certificate ← certificate $\langle YES, nonce\rangle_\sigma$
13:   else
14:       send ⟨RT_DECIDE, i, nonce, $\langle NO, nonce\rangle_\sigma$⟩ to all
15:       certificate ← certificate $\langle NO, nonce\rangle_\sigma$

upon event ⟨RT_DECIDE, id, nonce $\langle ANSWER, nonce\rangle_{\sigma(id)}$⟩ from $r_{i-1}$ do
16: if $\langle ANSWER, nonce\rangle_{\sigma(id)}$ is correctly signed then
17:   certificate ← certificate ∪$\langle ANSWER, nonce\rangle_{\sigma(id)}$
18:   if size(certificate) $\geq 2f+1$ then
19:       DECIDE ← majority(certificate)
```

Attacks

The attacker can mount numerous attacks against the aforementioned algorithms. We assume that the attacker cannot drop messages, since that will be a signal that the system is not synchronous, and will cause Ring to switch to another instance of ABSTRACT. A malicious replica can:

- respond to the ping message immediately upon receipt. This way, T_{hop} will be shorter than the actual time.

- wait longer to respond to the ping message.
- report a large time for the *RT_CHECK* message.
- report *NO* during the *RT_DECIDE* phase.

Clearly, for a malicious replica, the goal of which is to slow down the traffic, sending premature responses to ping message is ill-advised. Such an action will cause that its predecessor gets a lower T_{hop}, and will increase the chance that other replicas will pick a small ping time in the *decide_slowness* method, increasing the chance that the malicious activity will be detected.

Correctness of the Algorithm

Proposition 1. *Malicious replicas can not insert wrong times in the* RT_CHECK *message.*

Proof. This is guaranteed by the properties of digital signatures. If a malicious replica alters any signature, this discrepancy will be detected at subsequent replicas, and a switch will be made to the next (more robust) instance of ABSTRACT. □

Definition A correct T_{hop} is the T_{hop} value measured between two correct replicas.

Proposition 2. *A minimum number of correct T_{hop} values in the RT_REPORT message is $f+1$.*

Proof. Every malicious replica could affect T_{hop} value measurements at two replicas. The first one is its predecessor, while the second one is the malicious replica itself. Hence, for the f malicious replicas in the system, there will be at most $2f$ affected values. Since *RT_REPORT* message carries values from $3f+1$ different replicas, it leaves at least $f+1$ correct T_{hop} values. □

Proposition 3. *Malicious replicas can not force correct replicas to choose a T_{hop} which is greater than all of the correct T_{hop} values.*

Proof. The proof comes directly from Proposition 2 and the fact that in the *decide_slowness* method, a replica chooses the $f+1^{\text{st}}$ value. □

Definition There is an attack *if and only if* the system is acting slower than a system comprised of the slowest of all correct replicas.

Proposition 4. *Malicious replicas can not cause a misdetection of a performance attack. The performance attack causes a degradation of throughput.*

Proof. If there is an attack which causes slower traffic, all of the replicas will see a higher *RTT* (higher than in the case without an attack). Based on Proposition 3, all of the correct replicas will obtain a T_{hop} which is less than or equal to the maximal correct T_{hop}. By definition, the chosen value is at most K_{lat} times smaller than the maximal correct T_{hop}. Next, if there is an attack, $(3f+1) * \max(T_{hop})$ would be less than the measured *RTT* (by the definition of an attack). Hence, by transitivity, we have that $(3f+1) * K_{\text{lat}} * T_{\text{chosen}}$ will be higher than the measured *RTT*, as $K_{\text{lat}} * T_{\text{chosen}} \leq \max(T_{\text{hop}})$. □

5.3.3 Preventing Replicas From Discriminating Clients

One attack which malicious replicas may try to perform in Ring is to discriminate clients. A malicious replica, acting as the exit replica for a client's request, may reply to the client slowly, or with a delay. This attack reduces the overall throughput. To combat such a behaviour, the clients in Ring perform similar actions as replicas in Amir et al. [2008]:

- Before sending a request, the client starts a timer set to some value T_c.
- Upon receiving the response, the client stops the timer.
- If the timer expires, the client reissues the same request to another replica.
- Periodically, the client sends a special message with instructions for replicas to send back the statistics. This message traverses the system as a regular request (over the ring).
- When a replica receives such a message, it appends its Δi times to the message.
- When the client receives the reply to its special message, it takes all of the listed Δ times, sorts them in ascending order, and picks a value greater than the $2f+1^{\text{th}}$ value in the list. The client then sets this value as its next T_c.

6 Performance Model

Even extensive experimentation before the deployment of a protocol is often not enough to assess all of the benefits (and, more importantly, drawbacks) of the protocol, nor to predict its performance in the deployment environment. Therefore, we turn to analytic models, which can provide invaluable assistance in explaining experimental results and in predicting performance in situations for which experimental data does not exist. However, an analytic model is only useful if it matches reality.

In this chapter, we present an analytic model for the performance of BFT protocols, developed using queueing theory. To the best of our knowledge, this is the first application of queueing theory to modelling performance of BFT protocols. We start the chapter by explaining the assumptions which we take, followed by the presentation of the analytic model itself. Finally, we show how Ring is represented in our model, along with the representations of other protocols.

6.1 Queueing Theory Overview

The term "queueing theory" refers to the mathematical theory of queues (waiting lines) [Kleinrock, 1975]. More generally, queueing theory is concerned with mathematical modelling and analysis of systems which provide service to random demands, in a stationary regime [1]. Queueing theory is used for performance analysis of different processes in various technical systems, such as telephone and computer networks, production systems, hospitals, etc.

The main element in queueing theory analysis is a *queue*. Here, a queue may hold either a finite or an infinite number of *requests*, which arrive according to some distribution of *arrival times*. The *queueing discipline* of a queue denotes how, when and which requests are taken

[1]A stationary regime is a regime in which the mean and the variance of the observed property do not change over time.

out from the queue and passed to an associated processing element. The most usual queueing discipline is FIFO (First-In-First-Out), while the other disciplines include prioritization, LCFS (Last-Come-First-Served), processor sharing, or pre-emption [Boudec, 2010]. Each queue may have one (or more) associated processing elements, called *servers*[2]. The most important characteristic of a server is its *type of service process*, and this process denotes the distribution of servicing times. A queue with its associated server(s) forms a simple *station*. A station is a general term in queueing theory, denoting a collection of queues (along with their associated servers) which perform the same, high-level, work.

Clearly, a simple station in isolation cannot be used to represent many real-world systems. As realistic models of information and communication systems involve interconnected systems, many queues may represent such a system through a *queueing network* [Boudec, 2010; Walrand, 1988]. In queueing networks, as the name implies, queues (or, more precisely, stations) are interconnected. One such simple network, modelling users submitting jobs to a batch processing system is shown in Figure 6.1.

In Figure 6.1 we observe a station representing 3 users, which submit jobs to a service station consisting of a single CPU and two disks. Users submit only one outstanding request (per user) that goes to the CPU. After a request is serviced on the CPU, it can either return to the user, or go to either of the disks. In general, the path the request takes is probabilistic — there is a certain probability associated with each of the paths. These probabilities influence the mean number of visits to each queue, which, in turn, affects the total response time (we discuss the actual mechanisms later in the chapter). Note that there is no queue in front of the station denoted “users”. The reason for this is that each of the users submits only one request, and so the requests do not wait for service upon returning to users.

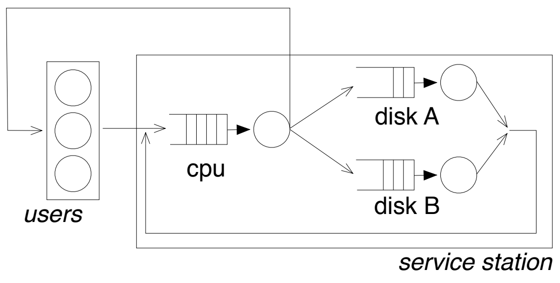

Figure 6.1: Example of a queueing network

As Figure 6.1 illustrates, queueing networks may consist of many different queues, with various interconnections among the queues. As such, queueing networks are not solvable in general. However, there is a broad class of queueing networks, the so-called *multiclass product-form queueing networks* [Baskett et al., 1975; Gordon and Newell, 1967; Jackson, 1963; Kelly, 1979] for which exhaustive results exist. Requests visit stations, where they either queue or receive

[2]“Server” is a well-established term in queueing theory. Consequently, we use the term “replica” to denote the participants in the BFT system.

service according to a particular servicing disciple of that station. Upon service, requests move to another station, or leave the system (if this the case, the network is called an *open* network[3]). Each of the request has an attribute (taken from a finite set of possible values), called the *class*. Requests change their class in transit between stations[4], according to Markov routing [Walrand, 1988]. Markov routing defines routing over different queues in the network, and is, essentially, represented by a matrix (a routing matrix) which defines the probabilities of moving a request from the pair (*queue*, *class*) to another pair (*queue'*, *class'*). Routing probabilities are denoted as $q_{s,c}^{s',c'}$, where s is the current queue, c is the current class, s' is the next queue, and c' is the next class. Thus, the concept of classes helps in determining the next queue for a request to take. The purpose of the routing matrix is to allow for the calculation of visiting rates.

The visiting rate of a station is a per-class quantity which represents the rate with which a request of a particular class visits the station. It could be either calculated (for example, by using the routing matrix), inferred from the system description, or measured from the real-world system. Given the visiting rate and the mean servicing time, one may calculate some important statistics related to a particular system, such as: the average number of requests in a queue, the average waiting time, the average servicing time (or a *response time*), and the average throughput.

The modelling process. When modelling any real system, the modeller first builds a model in which he is evaluating the performance of the system. In our case, the model is built using queues and stations (elements of queueing theory models). The model describes how given elements in the real system are mapped onto elements in the mathematical representation of the system. The entire modelling process consists of several phases:

1. *Building the representation.* In the first phase, the modeller (in our case, the protocol designer/evaluator) builds a queueing network which represents the modelled protocol. The queueing network must contain queueing stations, representing all of the parts of the real system (in our case, the replicas) which are accessed by a request, during its lifetime. Moreover, the routing matrix must be such that the path the request takes in the queueing network (visited resources) matches the resources visited in the actual system.

2. *Visit rate calculation.* Next, using the queueing network from the previous phase, the protocol designer/evaluator calculates the visiting rates — the statistical mean of times a request has visited each of the resources, during its lifetime.

3. *Parametrization of servers.* Unlike the previous two phases, this phase depends on the environment in which the protocol will run (or currently runs). In this phase, the protocol designer/evaluator measures various properties of the system, such as the

[3]In an open networks, customers can join and leave the system. Conversely, in a *closed* networks the total number of customers within the system remains fixed.

[4]Transits are instantaneous.

latencies or processing times for various operations. Using these measurements, the protocol designer/evaluator then calculates the total processing time for each of the requests, for each of the server resources in the queueing network.

4. *Application.* Finally, the protocol designer/evaluator uses the results from the previous three phases as an input to some analytic or numerical method (algorithm) for computing the metrics of interest for the given multiclass product-form queueing network. In our approach, we use the Mean Value Analysis (MVA) algorithm [Reiser and Lavenberg, 1980]. The output of the MVA algorithm is some metric of interest — the throughput, the response time, or the average queue length.

The first two phases usually occur only once during the entire evaluation process. The parametrization phase has to be re-assessed every time the conditions change — a change in hardware, a change in network settings, or a change in system parameters (for example, using requests of different size). On the other hand, the second phase does not depend on the hardware, and only depends on the associated routing probabilities of the queueing network. The modeller goes through the last phase any time he needs to query the model and obtain the metrics of interest.

MVA algorithm. The Mean Value Analysis algorithm [Reiser and Lavenberg, 1980], outlined in Algorithm 6.1, is an efficient method of calculating the metrics of interest in multiclass product-form queueing networks, and it has extensions for a broad class of queueing networks. The MVA could be applied to both the *open* and *closed* types of queueing networks.

Algorithm 6.1 Mean Value Analysis algorithm

```
Input: $\theta_c^s$, visit rates for all stations, such that $\theta^1 = 1$
Input: $\overline{S}^s$, average processing time of each of the stations
Output: $\lambda^s$, throughput for all of the stations
1: K ← population size
2: $\lambda \leftarrow 0$                                        {throughput}
3: $Q^s \leftarrow 0$ for all station $s \in$ FIFO    {total number of customers at station s}
                                                   {$Q^s = \sum_c \overline{N}_c^s$}
4: $\theta^s = \sum_c \theta_c^s$ for every $s \in$ FIFO
5: $h = \sum_{s\in \mathrm{IS}} \sum_c \theta_c^s \overline{S}^s + \sum_{s\in \mathrm{FIFO}} \theta^s \overline{S}^s$     {a constant term}
6: for $k = 1 : K$ do
7:     $\lambda = \frac{k}{h + \sum_{s\in \mathrm{FIFO}} \theta^s Q^s \overline{S}^s}$
8:     $Q^s = \lambda \theta^s \overline{S}^s (1 + Q^s)$ for all $s \in$ FIFO
9: return $\lambda^s = \lambda\theta^s$
```

The MVA algorithm aims at the computation of expected queue lengths (denoted Q in Algorithm 6.1) in the equilibrium state. The essence of this iterative algorithm lies in the *Arrival theorem* [Boudec, 2010]: in a system with k clients, an arriving client observes the rest of the system to be in an equilibrium state for a system with $k-1$ clients. The MVA starts by initializing the waiting times for all of the queues in the system, and then iteratively adds clients one by one, adjusting these waiting times. Using Little's Law, which relates throughput (denoted λ) with the waiting time:

$$\lambda = \frac{k}{\sum w}$$

where k is the total number of requests, and $\sum w$ is the total waiting time on all of the queues, one obtains the overall throughput of the system. The additional output of the MVA are the mean response times, as the algorithm calculates the waiting time at each of the queues [Bolch et al., 2005].

6.2 Model

In this section, we first describe the assumptions that we make in order to build our model. Next, we describe the model itself — how the elements from a real system (such as replicas, clients, network links, requests, ...) are mapped onto the elements of our queueing theory model. Afterwards, we outline how different queues in our model are parametrized. Finally, we describe the effects of various optimizations on the parametrization process.

Assumptions. We assume a *closed* system, i.e., that there is a fixed, finite, total number of clients accessing a finite number of replicas. This is a reasonable assumption for existing BFT systems, which require all of participants to exchange their cryptographic keys, thus limiting the number of clients which can participate in the system.

In a traditional BFT system, the clients submit their requests to the service over links which have a certain communication delay. The replicas enqueue requests from clients if they are currently processing another request. Consequently, every request spends some time in *transit*, and some time being *processed* (and also executed in the replicated service) by the replicas.

The model. To model both the networking and computational effects, we represent each of the replicas as a station containing several substations. The substations represent each of the incoming and outgoing network links, and one substation represents the CPU. The representation of a replica in our model is shown in Figure 6.2. Note that, as we use 2 network interface cards in the actual deployment, we have to represent that in the model by having multiple substations for each of the network interface cards. Next, the incoming and outgoing network links are represented as separate substations, as all of the links are full-duplex. Indeed, all of the modern network cards indeed have different hardware queues. A substation which represents a network link consists of single-server FIFO queues and a delay station[5] (denoted

[5]A delay station does not incur any queueing, and only adds a delay (called *thinking time Z*) to any of the requests that it receives.

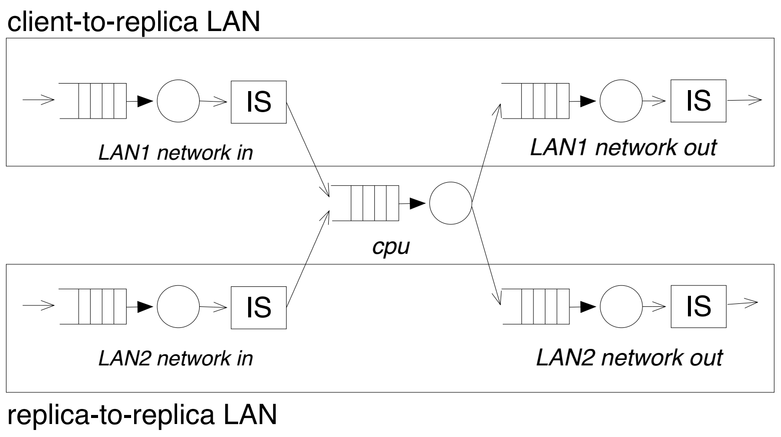

Figure 6.2: Queueing model for a replica: IS denotes a delay station, while all of the other queues adhere to the FIFO discipline.

IS), in a chain. The FIFO queues adhere to the First-In-First-Out processing discipline, matching the behaviour of the network links. In the context of network link related substations, the FIFO queues model the use of bandwidth, while delay stations model the latency. The CPU is modelled as a single-server [6] FIFO queue, with the processing time independent of the processing times of other FIFO queues in the replica station. The output of the incoming link substations is connected to the input of the CPU queue, while the output of the CPU queue is connected to the inputs of the outgoing link substations. This arrangement matches the fact that in all of the implementations, a separate thread fetches incoming requests to the main memory, where requests are processed by the CPU, and then propagated further over the network.

The clients in the system are represented only by their requests, as we assume that clients have only one outstanding request. The delays on the client links, and the client processing, are represented by a single delay station [Boudec, 2010], which is shown later in the chapter.

Although we can analyse different execution scenarios (as long as there exists a stationary working regime[7]), we focus only on the best case execution, for two reasons:

- arguably, the best-case execution is the predominant working mode, and, as such, has a stationary working regime.

- practitioners are usually interested in improving the best performance [Guerraoui et al., 2010a; Kotla et al., 2007; Serafini et al., 2010].

[6]Although any other representation of a CPU would also be valid. Our model aims at simplicity, and so we have chosen to use just a single queue.

[7]This is a shortcoming of queueing theory.

In each of the protocols, the clients issue requests, which then get processed by replicas. The clients commit the requests upon receiving a response or a quorum of responses. Thus, in the best execution scenario, the commit rate at the client is determined by the slowest response it receives. In order to capture this observation, we focus only on the longest, single path (the critical path) taken by a request during its lifetime. We do not try to model multicast, where one request spawns multiple other requests. The longest path is protocol-dependent, and using the longest path is a trade-off between the simplicity of the model and its accuracy. Indeed, relying on the longest path removes some interactions (e.g., waiting for the client to collect a quorum of responses). A special type of queueing stations, called fork-join [Duda and Czachórski, 1987] queues, allow for the representation of previously stated interactions, at the expense of added complexity of the calculation, as well as convoluted modelling. Our experiences and the results we report in Chapter 7 show that considering only the longest path yields very satisfactory results.

Model parameters. In order to have an accurate model, we need to accurately parametrize each of the server in our model, by conducting several measurements. One of our goals is to make our model simple to use and versatile, while retaining accuracy. The main difficulty lies in capturing all of the important behaviour through measurements (as these factors directly affect the accuracy), while keeping the number of measurements as small as possible. We achieve the balance by requiring only a few, *protocol-agnostic* parameters from the environment. These protocol-agnostic parameters are easy to obtain from every system, and in order to ease the process of obtaining the set of system parameters, we have developed a set of simple benchmarks to measure them, requiring no operator involvement.

Our model includes the representation of both CPU and network elements, and, therefore, the parametrization process has to capture the parameters of both of them. In the following text, we list the system parameters that our benchmarks measure:

- *Parametrization of servers representing replicas:* we parametrize the server associated with computational aspects (the central, CPU queue in Figure 6.2) by measuring the speeds of different operations on a given platform: cryptographic operations (authentication and verification of messages that are sent/received by the replicas), the execution time in the application layer, and kernel level operations (such as the `send()` system call). The processing time of the server (the inverse of the processing speed) is represented as a linear combination of the costs of these operations. Note that we do not measure the involved data structures handling speed, as it highly depends on the particularities of the implementation, and is not quantifiable in a generic manner. Nevertheless, the protocol designer can measure (or estimate) the data structure handling time and add it to the model if he wants to make more accurate predictions.
- *Parametrization of servers representing network links:* we parametrize the servers for the networking queues by measuring the latency between any two of the replicas in

the system, for all (or a subset of) possible message sizes. For each message size, the processing speed of the servers representing the network links is equal to the inverse of the measured latency[8]. Also, to parametrize the bandwidth related queues, we measure the maximum bandwidth on a link.

The exhaustive list of parameters which we use to parametrize the servers associated to both the replicas and the network links is given in Table 6.1.

Variable	**Purpose**
T_{exec}	application execution time
T_{digest}	the time needed to compute a digest of a message
T_{macg}	the time needed to generate a MAC for a message
T_{macv}	the time needed to verify a MAC for a message
T_h	the time needed to handle the necessary data structures
T_{send}	the time needed to issue a `send()` system call
$T_{proc(i)}$	the time needed to *process* a message at replica i; $T_{proc(i)}$ is a linear combination of all of the times listed so far
T_n	one-hop delay (latency)
$\mathscr{B}$	maximum throughput of a network link
$\overline{M}$	average message size
Z	the thinking time of a client

Table 6.1: Parameters used for modelling; all of the parameters depend on the average message size, and denote a value-per-message.

The effect of batching and optimizations. Protocol designers often use different optimizations, such as batching, to improve the performance [Castro and Liskov, 1999]. Such optimizations reduce the cost-per-request of an operation. We need to capture such behaviour, and in our model we do so by appropriately changing the affected parameters. For example, a with batching factor b, the cost of MAC generation at the primary in Zyzzyva reduces by b, along with the same factor improvement (per request) for the cost of sending.

Assessing the throughput of BFT protocol. As explained in the previous paragraph, we focus on the critical path of a request when modelling performance. Based on resources involved in the processing of a request on the critical path, we obtain the total processing time. Request handling encompasses different operations, making the total processing time a linear combination of values presented in Table 6.1, and this linear combination depends on the protocol. Additionally, for each of the requests we calculate the *visiting rates* (denoted θ) — the number of times a request has visited each of the servers on the critical path.

[8]To be more precise, this is the inverse of half of the latency, because between any two replicas in one direction, there are two queues representing the network links: one outgoing and one incoming.

6.3 Modelling Ring

In the previous section, we have given a detailed description of our model, followed by a description of the modelling process. In this section, we give an example of the modelling process applied onto Ring. Ring relies on batching and piggybacking optimizations, and we include this knowledge in the parametrization phase.

Modelling Ring is a complex task, due to the following two design features of Ring: (1) the clients may contact any of the replicas, and (2) each of the requests makes two rounds around the ring. Since replicas in Ring handle requests in a complex manner, using two round of communication, we resort to modelling Ring using a multi-class queueing network [Bolch et al., 2005; Dijk, 1993]. As we cannot represent all of the possible interactions with a multi-class queueing network, we model only an approximate behaviour of Ring. For example, requests in queueing theory model are invariant, in the sense that there is no accurate way of modelling piggybacking of an acknowledgement. Nevertheless, we still achieve good accuracy despite resorting to approximations, as the results in Chapter 7 show.

Classes and the routing matrix. When representing Ring in our model, we use the concept of classes to handle the aforementioned design features of Ring. First, the routing probability from the *IS* station, representing all clients, to any replica is set to $q_{1,1}^{x,1} = \frac{1}{n} = \frac{1}{3f+1}$. By this we represent the fact that any client can access any of the replicas, under the assumption of uniform load. Second, to track the progress of a request, the request changes its class in transit between each replica. Thus, a class contains information on the "count" of the number of steps that each request took through its lifetime. We use deterministic routing: the probabilities of a request going from one replica to another[9] is either 1 or 0. Once the request reaches the last step of processing, the probability of moving to the queue of another replica is 0, while the probability of moving to the *IS* station, representing all of the clients, is 1. After forming the routing matrix, we determine the visit rates at each of the queues, and use them as the input to the MVA algorithm, in order to compute the mean occupancy times.

In order to model piggybacking, we resort to an approximation of this behaviour. The role of piggybacking is to reduce the used bandwidth. Hence, piggybacked data is negligible, and, as the name implies, this data is piggybacked onto another request. Thus, we can think of piggybacked requests as requests that do not use any bandwidth, but spend some time in transit over links. Consequently, we form the routing matrix such that an acknowledgement goes around any FIFO queue of stations representing the network links. However, an acknowledgement will still go through the *IS* station, as that station models link delays. As evaluation in Chapter 7 suggests that this approximation adds only a small error to the final calculation.

We use 9 classes to denote each of the processing stages of Ring in our queueing model. The replica are represented as in Figure 6.2. Figure 6.3 shows the interconnections among replicas,

[9]Or, more precisely, to their corresponding queues.

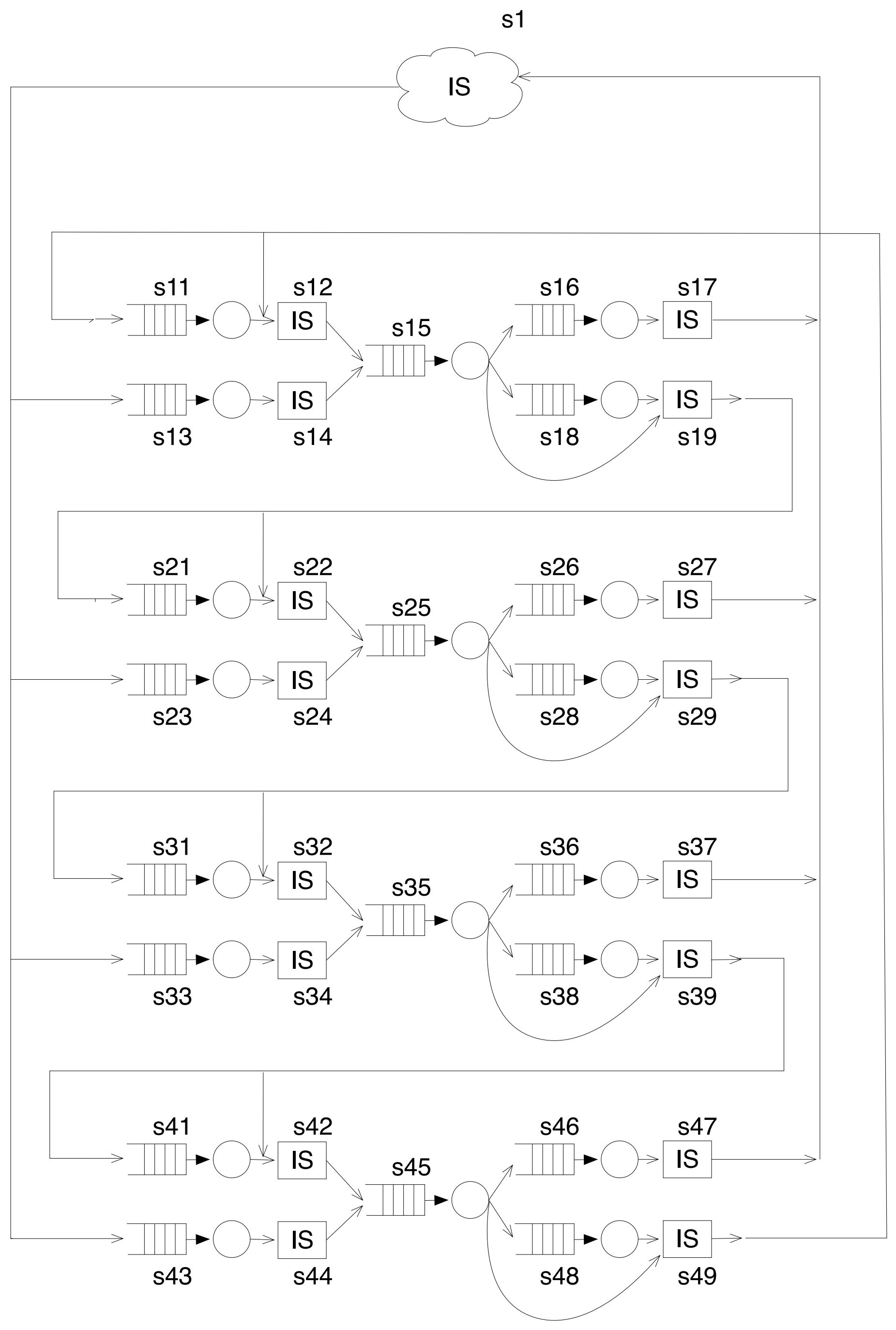

Figure 6.3: Representation of Ring in our model.

for $f = 1$. There are in total 20 FIFO queues (16 for modelling network links bandwidths, 4 for modelling processing at each replica), and 17 IS stations. The routing matrix (or, more precisely, the routing probabilities) are presented in Figure 6.4.

$$q_{1,2}^{13,14} = q_{1,2}^{23,24} = q_{1,2}^{33,34} = q_{1,2}^{43,44} = 1$$
$$q_{2,2}^{14,15} = q_{2,2}^{24,25} = q_{2,2}^{34,35} = q_{2,2}^{44,45} = 1$$
$$q_{2,2}^{15,18} = q_{2,2}^{25,28} = q_{2,2}^{35,38} = q_{2,2}^{45,48} = 1$$
$$q_{2,2}^{18,19} = q_{2,2}^{28,29} = q_{2,2}^{38,39} = q_{2,2}^{48,49} = 1$$
$$q_{2,2}^{19,21} = q_{2,2}^{29,31} = q_{2,2}^{39,41} = q_{2,2}^{49,11} = 1$$

$$q_{2,3}^{21,22} = q_{2,3}^{31,32} = q_{2,3}^{41,42} = q_{2,3}^{11,12} = 1$$
$$q_{3,3}^{22,25} = q_{3,3}^{32,35} = q_{3,3}^{42,45} = q_{3,3}^{12,15} = 1$$
$$q_{3,3}^{25,28} = q_{3,3}^{35,38} = q_{3,3}^{45,48} = q_{3,3}^{15,18} = 1$$
$$q_{3,3}^{28,29} = q_{3,3}^{38,39} = q_{3,3}^{48,49} = q_{3,3}^{18,19} = 1$$
$$q_{3,3}^{29,31} = q_{3,3}^{39,41} = q_{3,3}^{49,11} = q_{3,3}^{19,21} = 1$$

$$q_{3,4}^{31,32} = q_{3,4}^{41,42} = q_{3,4}^{11,12} = q_{3,4}^{21,22} = 1$$
$$q_{4,4}^{32,35} = q_{4,4}^{42,45} = q_{4,4}^{12,15} = q_{4,4}^{22,25} = 1$$
$$q_{4,4}^{35,38} = q_{4,4}^{45,48} = q_{4,4}^{15,18} = q_{4,4}^{25,28} = 1$$
$$q_{4,4}^{38,39} = q_{4,4}^{48,49} = q_{4,4}^{18,19} = q_{4,4}^{28,29} = 1$$
$$q_{4,4}^{39,41} = q_{4,4}^{49,11} = q_{4,4}^{19,21} = q_{4,4}^{29,31} = 1$$

$$q_{4,5}^{41,42} = q_{4,5}^{11,12} = q_{4,5}^{21,22} = q_{4,5}^{31,32} = 1$$
$$q_{5,5}^{42,45} = q_{5,5}^{12,15} = q_{5,5}^{22,25} = q_{5,5}^{32,35} = 1$$
$$q_{5,5}^{45,48} = q_{5,5}^{15,18} = q_{5,5}^{25,28} = q_{5,5}^{35,38} = 1$$
$$q_{5,5}^{48,49} = q_{5,5}^{18,19} = q_{5,5}^{28,29} = q_{5,5}^{38,39} = 1$$
$$q_{5,5}^{49,12} = q_{5,5}^{19,22} = q_{5,5}^{29,32} = q_{5,5}^{39,42} = 1$$

$$q_{5,6}^{12,15} = q_{5,6}^{22,25} = q_{5,6}^{32,35} = q_{5,6}^{42,45} = 1$$
$$q_{6,6}^{15,19} = q_{6,6}^{25,29} = q_{6,6}^{35,39} = q_{6,6}^{45,49} = 1$$
$$q_{6,6}^{19,22} = q_{6,6}^{29,32} = q_{6,6}^{39,42} = q_{6,6}^{49,12} = 1$$

$$q_{6,7}^{22,25} = q_{6,7}^{32,35} = q_{6,7}^{42,45} = q_{6,7}^{12,15} = 1$$
$$q_{7,7}^{25,29} = q_{7,7}^{35,39} = q_{7,7}^{45,49} = q_{7,7}^{15,19} = 1$$
$$q_{7,7}^{29,32} = q_{7,7}^{39,42} = q_{7,7}^{49,12} = q_{7,7}^{19,22} = 1$$

$$q_{7,8}^{32,35} = q_{7,8}^{42,45} = q_{7,8}^{12,15} = q_{7,8}^{22,25} = 1$$
$$q_{8,8}^{35,36} = q_{8,8}^{45,46} = q_{8,8}^{15,16} = q_{8,8}^{25,26} = 1$$
$$q_{8,8}^{36,37} = q_{8,8}^{46,47} = q_{8,8}^{16,17} = q_{8,8}^{26,27} = 1$$
$$q_{8,1}^{37,1} = q_{8,1}^{47,1} = q_{8,1}^{17,1} = q_{8,1}^{27,1} = 1$$

$$q_{1,1}^{1,13} = q_{1,1}^{1,23} = q_{1,1}^{1,33} = q_{1,1}^{1,43} = \frac{1}{4}$$

$$q_{s,c}^{s',c'} = 0 \; \textit{otherwise}$$

Figure 6.4: Routing probabilities for the representation of Ring. The station numbers correspond to stations in Figure 6.3.

Visiting rates. The next step in modelling Ring is the calculation of the visiting rates. The per-resource, per-class visiting rates for *closed* queueing networks, based on the routing matrix, up to a multiplicative constant, are defined, by the Equation (6.1), as:

$$\theta_c^s = \textstyle\sum_{s',c'} \theta_{c'}^{s'} q_{c',c}^{s',s} \tag{6.1}$$

Similarly, Equation (6.2) defines the per-resource, per-chain visiting rates:

$$\theta^s_{\mathscr{C}} = \sum\nolimits_{c\in\mathscr{C}} \theta^s_c \tag{6.2}$$

If there is only a single chain in the network, we can omit the subscript from Equation (6.2).

Being a chain is an equivalence relation between classes, and two classes are *chain-equivalent* if they are the same, or a customer of one class may become a customer of the other class. As such, chains simplify calculations related to visiting rates, and, consequently, the metrics of interest. In Ring, all of the classes belong to the same chain, since each of the requests starts with class "0" and finishes in the client substation with class "8".

Hence, the routing probabilities from Figure 6.4 yield the following, per-chain, visiting rates (when $n = 4$ since $n = 3f + 1, f = 1$):

$\theta^1 = 1$	*client station visiting rate, arbitrarily set*
$\theta^{11} = \theta^{21} = \theta^{31} = \theta^{41} = \frac{n-1}{n} = \frac{3}{4}$	*client-to-replica NICs, inbound bandwidth station*
$\theta^{12} = \theta^{22} = \theta^{32} = \theta^{42} = \frac{2n-2}{n} = \frac{6}{4}$	*client-to-replica NICs, inbound delay station*
$\theta^{13} = \theta^{23} = \theta^{33} = \theta^{43} = 1$	*replica-to-replica NICs, inbound bandwidth station*
$\theta^{14} = \theta^{24} = \theta^{34} = \theta^{44} = 1$	*replica-to-replica NICs, inbound delay station*
$\theta^{15} = \theta^{25} = \theta^{35} = \theta^{45} = \frac{2n-1}{n} = \frac{7}{4}$	*CPU*
$\theta^{16} = \theta^{26} = \theta^{36} = \theta^{46} = 1$	*client-to-replica NICs, outbound bandwidth station*
$\theta^{17} = \theta^{27} = \theta^{37} = \theta^{47} = 1$	*client-to-replica NICs, outbound delay station*
$\theta^{18} = \theta^{28} = \theta^{38} = \theta^{48} = \frac{n-1}{n} = \frac{3}{4}$	*replica-to-replica NICs, outbound bandwidth station*
$\theta^{19} = \theta^{29} = \theta^{39} = \theta^{49} = \frac{2n-2}{n} = \frac{6}{4}$	*replica-to-replica NICs, outbound delay station*

Parametrization. Finally, we calculate the processing times for all of the servers, as required by the MVA algorithm. In order to obtain the total processing time of a request, we must: (1) measure the parameters of the system for the network related stations, and (2) account for all of the processing for each class of requests at the CPU stations. Both of the steps are platform-dependent, and we present the corresponding measurements in Section 7.1.1. For the second step, we first make a breakdown of operations involved, by taking into account that:

- a replica computes the digest of the request 2 times (when receiving the request for the first time, and then again to compare the associated acknowledgement);

- the replica also computes a digest of a reply, when generating a reply to the client;
- the replica generates and verifies the MACs the same number of times per request;
- the replica executes the request exactly once;
- the replica sends the request twice, but we account for only one, to represent the fact that the requests are piggybacked. Moreover, the sending factor is reduced b times, as we assume that Ring batches b messages together;
- the replica sends the reply exactly once, and the batching does not have any effect on this operation;
- the replica allocates the memory for the request, the acknowledgement and the reply. Also, the replica allocates the memory once again, during the send operation.

Due to the symmetry of Ring, all of the processing times are equal across all of the replicas. Thus, we obtain the following formulas for the processing times, per request, given a batching factor b:

$$T_{proc}^{ring} = 2T_{digest}^{req} + T_{digest}^{reply} + \frac{1}{2}M_{avg}T_{macv} + \frac{1}{2}M_{avg}T_{macg} + T_{exec} + \frac{1}{b}T_{send}^{req} + T_{send}^{reply} + T_{h}^{ring}$$

In this equation, the term M_{avg} is calculated in Section 7.2.1, and represents the number of MAC operation per replica, per request. In Ring, MAC operations are symmetric, and any replica reads and writes the same number of MACs. All of the parameters (variables) are described in Table 6.1. In our model, we use the cost of memory allocation as an approximation for the cost of data structure handling. For each of the messages, the system allocates memory, and performs operations on a memory copy of the content of the messages. Hence, memory allocation correlates with the use of necessary data structures. The actual values for the lengths of these operations depend on the environment and we measure them in Section 7.1.1.

6.3.1 Calculating the Maximal Throughput

Next, we demonstrate how our analytic model could be used to obtain various properties of the system, even without resorting to measurements. Namely, we use our analytic model to calculate the maximal throughput that Ring can achieve. We validate our initial claim that the maximal throughput Ring could achieve is $\frac{n}{n-1}\mathscr{B}$, where $\mathscr{B}$ is the maximum throughput of the network (we assume that all of the links are equal).

If we denote by λ_c^s the flow of requests of class c at station s, and we denote by $\lambda_{\mathscr{C}}$ the throughput of chain $\mathscr{C}$, the relation between these flows is [Boudec, 2010]:

$$\lambda_c^s(\overrightarrow{K}) = \lambda_{\mathscr{C}}(\overrightarrow{K})\theta_c^s \tag{6.3}$$

The Equation (6.3) states that for any population of requests $(\overrightarrow{K})$ the flow of requests of class c, through some station s is equal to the product of the *per-chain flow* and the visit rate of class c

to resource s. The total throughput through a single station is equal to the sums of all of the *per-class* throughputs going through that station (given that there is only a single chain $\mathscr{C}$)) is:

$$\begin{aligned}\lambda^s(\overrightarrow{K}) &= \textstyle\sum_{c\in\mathscr{C}} \lambda_c^s(\overrightarrow{K}) \\ &= \textstyle\sum_{c\in\mathscr{C}} \lambda_{\mathscr{C}}(\overrightarrow{K})\theta_c^s \end{aligned} \tag{6.4}$$

Due to Equation (6.2), we finally obtain:

$$\lambda^s(\overrightarrow{K}) = \lambda_{\mathscr{C}}(\overrightarrow{K})\theta^s \tag{6.5}$$

where we drop the *per-chain* index on the visit rate.

Now, consider the throughput through the station representing clients: λ^1. This is the throughput of requests which enter the system, and protocol designers optimize in order to increase λ^1. Due to Equation (6.5), we have that:

$$\lambda^1 = \lambda_{\mathscr{C}}\theta^1 \quad \text{where} \quad \theta^1 = 1$$

Similarly, from Equation (6.5), we obtain the relation between the flow to a replica and the *per-chain* flow:

$$\lambda^{11} = \lambda_{\mathscr{C}}\theta^{11} \quad \text{where} \quad \theta^{11} = \frac{n-1}{n} \tag{6.6}$$

which leads to:

$$\lambda^{11} = \lambda^1\frac{n-1}{n} \tag{6.7}$$

If we consider the constraint that no flow to a replica exceeds the line rate $\mathscr{B}$:

$$\lambda^{11} = \lambda^{21} = \lambda^{31} = \lambda^{41} \leq \mathscr{B} \tag{6.8}$$

then, by using Equation (6.7), we obtain the following inequality for the maximal throughput through the ring:

$$\begin{aligned}&\lambda^{11} = \lambda^1\frac{n-1}{n} \\ &\implies \lambda^1\frac{n-1}{n} \leq \mathscr{B} && \textit{substituting from Equation } (6.8) \\ &\implies \lambda^1 \leq \frac{n}{n-1}\mathscr{B} \end{aligned} \tag{6.9}$$

The calculation in Equation (6.9) gives the maximum throughput in Ring, where we consider the throughput of requests going into replicas over the client-to-replica network. We prove that, indeed, the maximum throughput in Ring is $\frac{n}{n-1}\mathscr{B}$.

6.4 Performance Models of Other BFT Protocols

Now, we present the representations of other BFT protocols (which we use in the evaluation) in our performance model. Similarly to the previous presentation of Ring, we limit the analysis to the throughput.

We consider the following three state-of-the-art protocols: Chain, Zyzzyva, and PBFT. Although we have applied our performance modeling framework on Quorum-like protocols (such as Q/U [Abd-El-Malek et al., 2005], and HQ [Cowling et al., 2006]), we do not present them here, since quorum-like protocols only run in contention-free environments — a requirement which is in stark contrast to achieving high-throughput [Singh et al., 2008]. Chain, Zyzzyva, and PBFT rely on a dedicated replica, which receives requests, to order them and to forward them to other replicas. Again, it is important to note that all of these protocols require $3f+1$ replicas in order to be able to tolerate f faults (which is optimal, according to [Lamport, 2004]). For the simplicity of the presentation, here we will consider only the best-case execution scenarios.

6.4.1 Chain

As stated in Section 2.1.3, Chain relies on two distinct replicas: the *head* and the *tail*. All replicas are arranged in a chain (from which the protocol derives its name). A client sends a request to the *head*, which assigns a sequence number to the request. The *head* then forwards the request to the next replica in the chain. Each replica executes the request, appends it to its local history, and forwards the request until the request reaches the *tail*. Finally, the *tail* replies to the client. The last $f+1$ replicas include the digest of their history in the forwarded request, which the tail sends to the client. If these digests match, the client commits the request. Otherwise, the client resorts to a backup protocol to commit the request. We do not describe this backup protocol as it is not used in the standard case (synchronous network, no faults).

Contrary to the complex representation of Ring, the representation of Chain in our performance model is simple, as depicted on Figure 6.5. In Chain, all $3f+1$ replicas are on the critical path, and some of the links remain unused. Visiting times to any visited resource are exactly 1. There is no complex, probabilistic routing, and there is just one class. To simplify our presentation, we assume that $f=1$. All replicas send exactly one message. The head reads one MAC, and writes 2 MACs. All of the other (except the last) replicas read 2 MACs and write 2 MACs, while the last replica writes only one MAC.

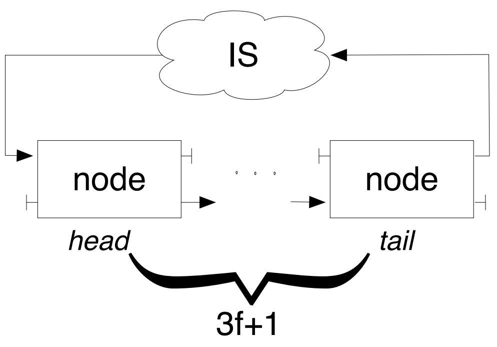

Figure 6.5: Queueing model representation of Chain. Each *node* box represents one instance of queues from Figure 6.2. The *IS* cloud represents a single delay station, modeling processing at the clients. Note that the replicas use one set of network links, while the clients connect to another set of network links, at the head and the tail.

$$T^{chain}_{proc(1)} = T_{digest(1)} + T_{macv} + 2T_{macg} + T_{exec} + T_{send} + T^{chain}_{h} \tag{6.10}$$

$$T^{chain}_{proc(2)} = T_{digest(2)} + 2T_{macv} + 2T_{macg} + T_{exec} + T_{send} + T^{chain}_{h} \tag{6.11}$$

$$T^{chain}_{proc(3)} = T_{digest(3)} + 2T_{macv} + 2T_{macg} + T_{exec} + T_{send} + T^{chain}_{h} \tag{6.12}$$

$$T^{chain}_{proc(4)} = T_{digest(4)} + 2T_{macv} + T_{macg} + T_{exec} + T_{send} + T^{chain}_{h} \tag{6.13}$$

6.4.2 Zyzzyva

In Zyzzyva, a client sends its request to the primary (a special, dedicated replica). The primary assigns a sequence number to the received request, and multicasts the request, along with its sequence number to other replicas (backup replicas). Each replica (including the primary) executes the request, appends the result to its local history, and sends the response to the client. Whenever faults occur, the replicas execute certain recovery mechanisms. Due to its speculative nature (replicas execute the request as soon as they receive it, without making sure that the sequence number is correct), Zyzzyva exhibits high performance when there are no faults.

Similarly to Chain, the representation of Zyzzyva is simple in our model, as there are only two replicas on the critical path, and the visiting times are exactly 1. Additionally, there is only just one class, and there is no complex routing. The first replica on the path is the primary. The primary performs several cryptographic operations: it verifies one MAC from the client, and generates $3f + 1$ MACs (one for each backup replica and one for the client). Moreover, the primary sends two messages: the request that it multicasts to other replicas, and the reply to the client. The primary does, thus, issue two `send()` calls. Consequently, the processing time of the primary is the following:

$$T^{zyzzyva}_{proc(1)} = T_{digest(1)} + T_{macv} + (3f+1)T_{macg} + T_{exec} + 2T_{send} + T^{zyzzyva}_{h}$$
$$T^{zyzzyva}_{proc(2)} = T_{digest(2)} + 2T_{macv} + T_{macg} + T_{exec} + T_{send} + T^{zyzzyva}_{h}$$

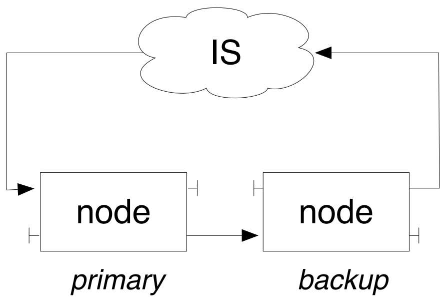

Figure 6.6: Queueing model representation of Zyzzyva.

As the load increases, one of the queues becomes the bottleneck, and we observe the queueing effects. The utilization of the queue (precisely, of its associated server) is equal to 1, and the throughput is limited by the inverse of the servicing time [Bolch et al., 2005]. The servicing time includes both the processing and the queueing time. Since every request visits each queue exactly once, we can obtain an upper bound on the throughput under contention, as follows:

$$\lambda \leq \min\{\frac{1}{T_{proc(1)}}, \frac{1}{T^{incoming}_{n(1)}}, \frac{1}{T^{outgoing}_{n(1)}}, \frac{1}{T_{proc(2)}}, \frac{1}{T^{incoming}_{n(2)}}, \frac{1}{T^{outgoing}_{n(2)}}\} \tag{6.14}$$

Naturally, this upper bound depends on the request size.

6.4.3 PBFT

Similarly to Zyzzyva, PBFT relies on a dedicated replica, called the *primary* to order the requests. To issue a request, a client has to send it to the *primary*, which appends a sequence number to the request and broadcasts a *PRE-PREPARE* message to all of the other replicas containing the ordered request. When a backup replica receives the *PRE-PREPARE* message, it acknowledges the message by broadcasting a *PREPARE* message to all of the replicas. As soon as a replica receives a quorum of $2f+1$ *PREPARE* messages, it promises to commit the request (at the sequence number appended to the request by the *primary*) by broadcasting a *COMMIT* message. Lastly, when any replica receives a quorum of $2f+1$ *COMMIT* messages, it executes the request and replies to the client.

PBFT uses an optimization regarding *COMMIT* messages, called *tentative execution* — if all of the requests which have a lower sequence number have been executed, replicas reply to the client before sending the commit message. Since this is the case in best-case execution, we include that behaviour in our representation of PBFT.

The client commits the request upon receiving $f+1$ matching replies. Otherwise, the client retransmits the request. If the request does not commit after a certain time, the protocol executes a leader election protocol, in order to change the primary. This part of the protocol is not executed in the common case (synchronous network, no faults), and for this reason we do not describe such a protocol in this section.

All protocols employ different kinds of optimization, with the predominant one being batching, where multiple requests from different clients are batched in order to reduce the overall processing time. Although the batching is, in general, configurable, in PBFT one can only control the batching of the requests. For all of the other stages, PBFT aggressively batches the messages itself, out of user control.

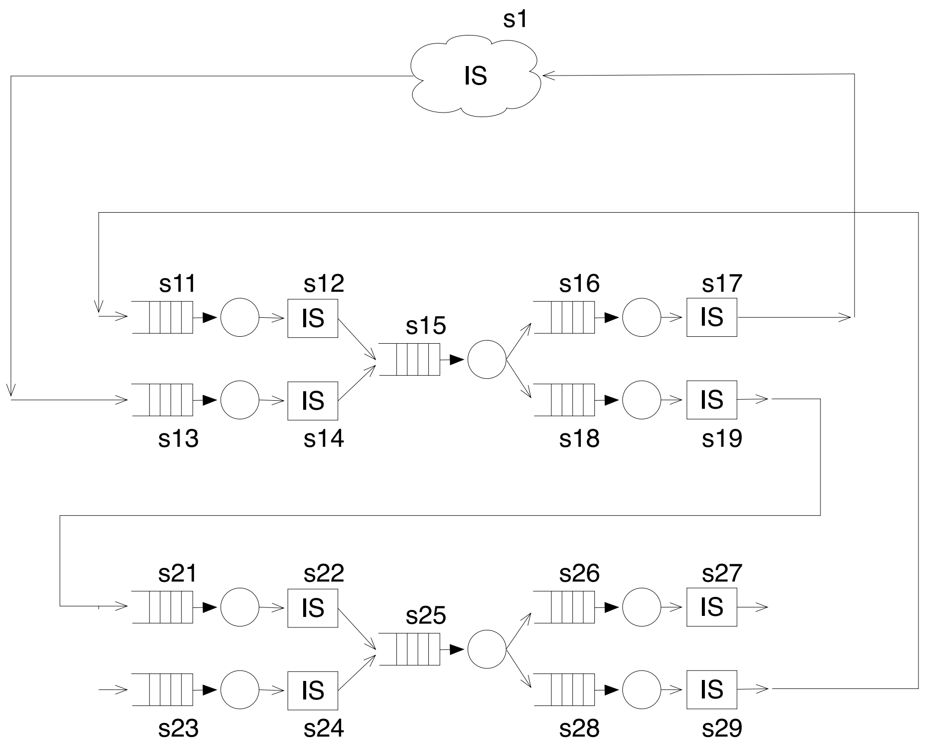

Figure 6.7: Queueing network representation of PBFT.

PBFT has 4 different message-exchange stages, with each stage having a different processing time on the same replica. Thus, we have resorted to modeling an approximate behaviour.

Our approximation matches the performance of PBFT well, as shown in Chapter 7. Our approximation of the behaviour of PBFT takes into consideration only two replicas in the system, and is as follows: 1) the client sends a request; 2) the primary processes the request, and sends it to a backup replica; 3) the backup receives the request, and forwards it back to the primary; 4) the primary receives the request, and sends a reply to the client. Each of these steps corresponds to one of the communication steps depicted in Figure 2.2. In every step, the request changes its class, similarly to the case of Ring. Using this approach, we are able to emulate multicast communication which occurs at every stage.

If we denote the station representing the clients as *station 1*, the primary as *station 2*, and the backup replica as *station 3*, the routing probabilities are:

$$\begin{aligned} q_{1,1}^{1,11} &= q_{1,1}^{11,12} = q_{1,1}^{12,15} = q_{1,1}^{15,16} = q_{1,1}^{16,17} = \\ &= q_{1,2}^{17,23} = q_{2,2}^{23,24} = q_{1,1}^{24,25} = q_{2,2}^{25,26} = q_{2,2}^{26,27} = \\ &= q_{2,3}^{27,13} = q_{3,3}^{13,14} = q_{3,3}^{14,15} = q_{3,3}^{15,18} = q_{3,3}^{18,19} = \\ &= q_{3,1}^{19,1} = 1 \\ q_{s,c}^{s',c'} &= 0 \textit{ otherwise} \end{aligned}$$

From the given routing matrix, we obtain that each of the requests visits the CPU on the primary 2 times, and visits any other resource exactly once:

$$\begin{aligned} &\theta^1 = 1 && \textit{arbitrarily set} \\ &\theta^{15} = 2 \\ &\theta^{11} = \theta^{12} = \theta^{13} = \theta^{14} = 1 \\ &\theta^{16} = \theta^{17} = \theta^{18} = \theta^{19} = 1 \\ &\theta^{21} = \theta^{22} = \theta^{28} = \theta^{29} = 1 \\ &\theta^{23} = \theta^{24} = \theta^{26} = \theta^{27} = 0 && \textit{these stations are not used} \end{aligned}$$

Finally, by simply accounting for all of the the processing for each of the classes of requests arriving at a particular server, we obtain:

$$T_{proc(2)}^{pbft} = T_{digest(2)} + 7T_{macv} + (2 + \frac{6f}{b})T_{macg} + T_{exec} + (1 + \frac{2}{b})T_{send} + T_h^{pbft} \tag{6.15}$$

$$T_{proc(3)}^{pbft} = T_{digest(3)} + T_{macv} + (1 + \frac{2f}{b})T_{macg} + T_{exec} + (1 + \frac{1}{b})T_{send} + T_h^{pbft} \tag{6.16}$$

6.5 Summary

In this chapter, we have given an overview of queueing theory, necessary for the building of our performance model. Next, we have outlined the assumptions that we take, the modelling process and our performance model. Further, we demonstrated the use of our model, and built a representation of Ring. Using this representation, we have analytically shown that, indeed, the maximum throughput of Ring is $\frac{n}{n-1}\mathscr{B}$. Finally, we presented how our performance model applies to other, state-of-the-art BFT protocols.

We will present the evaluation of our performance model, along with an analytic comparison of different BFT protocols in Chapter 7.

7 Performance Evaluation

In this section, we report on the results of both the practical and the analytic performance evaluation of Ring, in comparison with the three state-of-the-art protocols: PBFT, Chain, and Zyzzyva (described in Section 2.1.3).

PBFT is taken from http://www.pmg.lcs.mit.edu/bft/bft.tar.gz, and slightly modified to run our version of benchmarks. Zyzzyva was taken from http://research.microsoft.com/en-us/people/kotla/. However, we had problems running Zyzzyva on our platform, and thus we used our implementation of Zyzzyva, called Zlight. Zlight has the same communication pattern as Zyzzyva, but achieves higher performance [Guerraoui et al., 2010a]. We implemented Chain in the context of the ABSTRACT framework [Guerraoui et al., 2010a].

Similarly to these protocols, Ring is also implemented in C++. The replicas and the clients communicate over a TCP connections. In order to be able to handle a large number of client connections, we use the `epoll` event-notification mechanism. We observe that `epoll` is more efficient than the `select` mechanism, as claimed by Gammo et al. [2004]. Moreover, in order to prevent malicious participants from exhausting all of the network resources, Ring uses a token bucket [Shenker and Wroclawski, 1997] mechanism for establishing fairness among TCP flows. In our implementation, the token bucket splits the incoming throughput between the predecessor and (all) client traffic, using the ratio $3f:1$.

We begin this section by giving a description of the experimental setup we have used, along with the discovery of the system parameters, needed for our analytic model. Next, we show that, unlike existing protocols, Ring equally balances both the CPU utilization on the various replicas, and the network utilization on the various network links. Afterwards, we present an exhaustive performance comparison of Ring and the state-of-the-art protocols. More precisely, we show that Ring significantly outperforms other protocols in terms of throughput (+27 %) when the network is the bottleneck, and that it achieves up to 14 % lower response time than state-of-the-art protocols when a large number of clients issue requests. Finally, the last part of the evaluation is reserved for the assessment of the accuracy of our performance model. The experimental comparison shows that, for the most cases, the relative error is below 5 %.

7.1 Experimental Setup

We have performed all of the experiments on the Emulab [White et al., 2002] testbed. In each of the experiments, we have used *pc3000* machines – Dell PowerEdge 2850s systems, with a single 3 GHz Xeon processor, 2 GiB of RAM, and 2 NICs. The replicas are systematically running on their own, separate machine, while the clients are collocated on a total of 40 machines. Finally, we use a topology in which the replicas belong to one LAN, and clients communicate with replicas over a second LAN.

We use the benchmarks described in PBFT [Castro and Liskov, 1999], where the clients perform requests in a closed-loop manner. In closed-loop benchmarks, clients issue only one outstanding request and wait until they have received the response. Such a behaviour models the synchronous, blocking model of programming, present in nearly all of the POSIX systems. Benchmarks have the option to vary the size of the request issued by the clients and the size of the replies produced by the replicas. The request and reply size could be set to any of the values from the range 8 to 16’000 B. We use 8 B replies, unless stated otherwise. Each of the experiments was repeated three times, and we report the average of these three executions.

7.1.1 System Parameters

Our model, based on queueing theory, requires some measurements for different data sizes of the underlying system, such as link latencies, throughput, processing speeds for different request-handling operations, and the time required to actually send the message from the application.

Network parameters In order to parametrize network-related substations in our model, we measure the propagation time for different protocols (TCP, UDP, and IP multicast) of the underlying Fast Ethernet network, while varying the message size. In this experiment, we set two machines, a server and a client, to exchange messages. The client issues 100 batches of 100’000 messages, and reports the response time for each of the batches. We also measure the time of local delivery (i.e., the time it takes to send the same message on the same machine), in order to exclude the effects of the underlying operating system. We report the link propagation time as one quarter of the difference between the total propagation time and the local delivery time. Figure 7.1 illustrates the link propagation time for all of the three used protocols (TCP, UDP, and IP multicast).

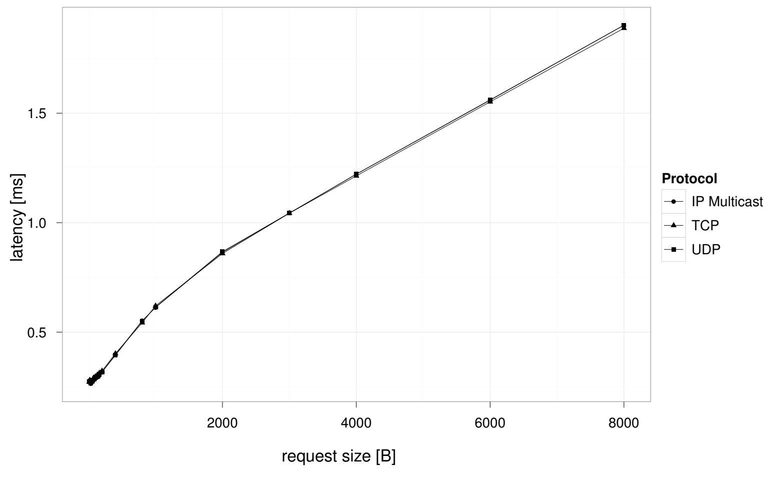

Figure 7.1: Link propagation times for different message sizes, for different protocols.

As Figure 7.1 suggests, Emulab has a sub-millisecond latency for small requests. For large requests[1], the latency is between 1 ms and 1.5 ms.

We use the `iperf` tool [Tirumala et al., 2010] to measure the throughput. Table 7.1 contains our findings. Expectedly, UDP achieves higher throughput due to less protocol overhead. However, the difference is less than 1.5 %.

	Throughput (Mbps)
UDP	93.704
TCP	90.947

Table 7.1: The maximum observed throughput on the testbed.

CPU parameters Next, we measure the speed of various operations generally involved in request processing (such as the generation and verification of digests and MACs, the time needed to send a message, and memory allocation). To obtain these protocol-agnostic measurements, we wrote a set of simple benchmarks which mimic the common behaviour of many BFT protocols. Our benchmark ensures that the caches are invalidated before processing each of the requests, since we attempt to replicate conditions of running under a high load, where caches are often evicted. We use the `rdtsc` syscall to measure the time needed to perform each of the operations. The benchmark executes each of the operations 100'000 times, for a set of different message sizes, and reports the average number of ticks per operation. We repeat each experiment 100 times, to obtain statistically significant means.

[1]In context of BFT replicated protocols.

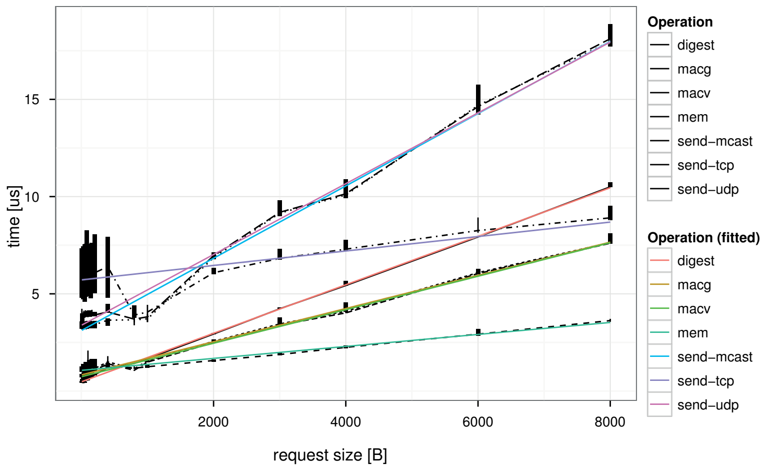

Figure 7.2: Processing times for different operations.

Figure 7.2 summarizes the processing times for different operations. The figure plots all of the data points produced by the benchmark. We also plot the linear regression model which fits the data, per operation. The list of parameters we use to parametrize the servers associated with the replicas is given in Table 7.2. This table contains a single-dimension linear regression model of all of the parameters, dependent on the request size. Lengths of all of the operations are linearly dependent on the request size, although an extension to the model where operation lengths depend on multiple parameters is straightforward. The reported values are presented in seconds.

	Parameter fit	
T_{digest}	4.750579×10^{-7}	$+\ 12.47527\ \times10^{-10}\times M$
T_{macg}	8.293719×10^{-7}	$+\ 8.536017\times10^{-10}\times M$
T_{macv}	7.362421×10^{-7}	$+\ 8.617151\times10^{-10}\times M$
T_{mem}	$10.63922\ \times10^{-7}$	$+\ 3.079504\times10^{-10}\times M$
$T_{\text{send-mcast}}$	$31.09838\ \times10^{-7}$	$+\ 18.59765\ \times10^{-10}\times M$
$T_{\text{send-tcp}}$	$57.15264\ \times10^{-7}$	$+\ 3.717047\times10^{-10}\times M$
$T_{\text{send-udp}}$	$33.80395\ \times10^{-7}$	$+\ 18.20803\ \times10^{-10}\times M$

Table 7.2: The parameters used for parametrization of different CPU operations; We use a linear model, where lengths of all of the operations depend only on the request size (denoted by M). The reported values are based on the linear fit of the data from Figure 7.2. The unit which was used is the second.

Interestingly, we note that, for a wide range of message sizes, it is more expensive to issue a `send()` system call, then to perform any of the computational operations. The only exception are TCP messages larger than 6 KiB, for which computing digests is more costly than the sending operation. Thus, in this case, batching should improve performance to an extent.

Moreover, Figure 7.2 suggests that a single MAC operation is less expensive than performing a digest. However, performing three or more MAC operations (which is the case for all of the protocols, for $f \geq 1$), is more expensive than performing the digest operation. Finally, Figure 7.2 shows that sending a TCP packet is less expensive than sending an UDP message, but only for large messages. With small messages, UDP performs more than 50 % better, at the protocol level.

7.2 CPU Utilization

Figure 7.3 illustrates CPU utilization for Ring, alongside the values of CPU utilization for other protocols (previously shown in Figure 3.1, in Section 3). We observe that all of the replicas in Ring are equally loaded. This balance is a consequence of the fact that there is no asymmetry in replica processing: all of the replicas perform virtually the same computations, and each of the replicas receives the same amount of client requests[2]. Consequently, all of the replicas in Ring become the bottleneck at the same time.

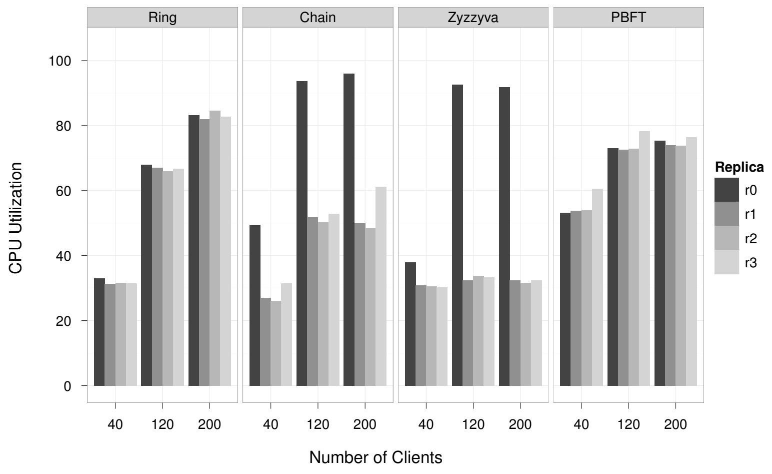

Figure 7.3: CPU utilization of Ring (and other protocols).

7.2.1 MAC Operations

Since message authentication/verification is a CPU-intensive task, the number of MAC operations represents a good indicator of the likely bottlenecks in the system — the replicas which execute the most operations will become bottlenecks first. In Ring, all of the replicas are equal, given the balanced load. Nevertheless, we calculate the average number of MAC operations per request in a steady, balanced state, as a comparison to other protocols.

[2]Provided that clients uniformly balance their requests across different replicas, which is trivially achieved by having clients choose the entry replica in a round-robin manner.

As described in Chapter 4, each of the request in Ring takes two rounds. In the first round, the replicas forward the request. In the second round, the replicas forward the corresponding acknowledgement. Each of the replicas first verifies (*reads*) a set of MACs before processing the requests. If these MACs match the request, the request is executed (or just processed, if the sequence number is still not known), and forwarded later along the ring. Before forwarding the request, the replica authenticates (*writes*) the request, by writing a set of MACs. If R_i^r (W_i^r) denotes the number of MAC read (respectively, write) operations at replica i within Ring, the account of performed MAC operations throughout the lifetime of the request on all $n = 3f + 1$ replicas is[3]:

$$
\begin{aligned}
R_0^r &= (1) + (f+1) & W_0^r &= (f+1) + (f+1) \\
R_1^r &= (2) + (f+1) & W_1^r &= (f+1) + (f+1) \\
&\dots & &\dots \\
R_{f-1}^r &= (f) + (f+1) & W_{f-1}^r &= (f+1) + (f+1) \\
R_f^r &= (f+1) + (f+1) & W_f^r &= (f+1) + (f+1) \\
&\dots & &\dots \\
R_{2f}^r &= (f+1) + (f+1) & W_{2f}^r &= (f+1) + (f+1) \\
R_{2f+1}^r &= (f+1) + (f+1) & W_{2f+1}^r &= (f+1) + (f) \\
&\dots & &\dots \\
R_{3f-1}^r &= (f+1) + (f+1) & W_{3f-1}^r &= (f+1) + (2) \\
R_{3f}^r &= (f+1) + (f+1) & W_{3f}^r &= (f+1) + (1)
\end{aligned}
$$

In Ring, every replica takes every role for the same amount of time, on average (i.e., is an entry replica, a sending replica, or a replica in the middle of processing chain). Thus, the per-replica average number of MAC operations (presented separately for reads and writes), per request, is:

$$
\begin{aligned}
R_{\text{avg-req}}^r &= \frac{1}{3f+1} \sum_0^{3f} R_i^r = \frac{1}{3f+1}\left((3f+1)(f+1) + (2f+1)(f+1) + \frac{f(f+1)}{2}\right) \\
&= \frac{(f+1)(\frac{11f}{2}+1)}{3f+1} \\
&= \frac{1}{2}(f+1)\left(4 - \frac{f+2}{3f+1}\right) \\
W_{\text{avg-req}}^r &= \frac{1}{3f+1} \sum_0^{3f} W_i^r \\
&= \frac{1}{2}(f+1)\left(4 - \frac{f+2}{3f+1}\right)
\end{aligned}
$$

[3]A term in the first (second) parenthesis represents the number of performed MAC operations in the first (respectively, second) round.

Hence, the average number of MAC operations (denoted M^r) per request is:

$$M^r_{\text{avg-req}} = R^r_{\text{avg-req}} + W^r_{\text{avg-req}} = (f+1)\left(4 - \frac{f+2}{3f+1}\right) \tag{7.1}$$

Ring uses piggybacking to improve performance. Hence, we extend Equation (7.1) to account for the MACs introduced by this optimization. The piggybacked acknowledgement introduces $f+1$ additional read MAC operations, and $f+1$ write MAC operations. In addition, there are two requests in the same flow (the carrier and the piggybacked acknowledgement):

$$\begin{aligned} M^r_{\text{avg-pb}} &= \frac{1}{2}\left(M^r_{\text{avg}} + \frac{(f+1)+(f+1)}{3f+1}\right) = \frac{1}{2}\left((f+1)\left(4-\frac{f+2}{3f+1}\right) + 2\frac{f+1}{3f+1}\right) \\ &= (f+1)\left(4 - \frac{f}{3f+1}\right) \\ &= \frac{11}{3}f + \frac{34}{9} + \frac{2}{9(3f+1)} \end{aligned} \tag{7.2}$$

Ring also uses batching as another form of optimization. Apart from reducing the number of sent messages, batching b messages together reduces the total number of MAC operations. A replica needs to read a MAC written by a client for each of the requests in the batch, and it needs to write a MAC for the client for each of the requests in the batch. Otherwise, when there are no MACs written by a client, the replicas generate a MAC for the whole batch, as if it were a single request. Thus, the average number of MACs operations, for a batch of b requests, is:

$$\begin{aligned} M^r_{\text{avg-b}} &= \frac{1}{b}\frac{1}{3f+1}(R^r_{\text{avg-b}} + W^r_{\text{avg-b}}) \\ &= \frac{1}{b}\frac{f+1}{3f+1}\big((10+b)f+2\big) \end{aligned} \tag{7.3}$$

where

$$R^r_{\text{avg-b}} = W^r_{\text{avg-b}} = b\frac{f(f+1)}{2} + (2f+1)(f+1) + (3f+1)(f+1)$$

Finally, we obtain the average number of operations, when both piggybacking and batching are used. To do so, we apply the same process as in obtaining Equation (7.2) on Equation (7.3) and Equation (7.2), and make use of the fact that we also piggyback a batch of b requests:

$$\begin{aligned} M^r_{\text{avg}} &= \frac{1}{2b}\left(bM^r_{\text{avg-b}} + b\frac{(f+1)+(f+1)}{3f+1}\right) \\ &= \frac{1}{2b}\frac{f+1}{3f+1}\big((10+b)f+2+2b\big) \end{aligned} \tag{7.4}$$

Table 7.3 reports the number of per-request MAC operations performed at different replicas for PBFT, Zyzzyva, Chain and Ring. Additionally, the table also reports the number of per-request MAC operations when batching is used, assuming that the batch size is b. Table 7.3 shows that PBFT and Ring do not differentiate replicas, in terms of the number of MAC operations. Zyzzyva exhibits a highly asymmetric behaviour, but compensates this CPU-bound asymmetry with a small number of MAC operations.

	Number of MAC operations			
	No batching		**Batching**	
Protocol	**Primary**	**Other**	**Primary**	**Other**
PBFT	$12f+2$	$12f+2$	$2+\frac{8f+1}{b}$	$2+\frac{5f+1}{b}$
Zyzzyva	$3f+2$	2	$2+\frac{3f}{b}$	$\frac{2}{b}$
Chain	$f+2$	$2f+2$	$1+\frac{f+1}{b}$	$2\frac{f+1}{b}$
Ring	$\frac{11}{3}f+\frac{34}{9}+\frac{2}{9(3f+1)}$		$\frac{1}{2b}\frac{f+1}{3f+1}\big((10+b)f+2+2b\big)$	

Table 7.3: Comparison of the number of MAC operations on different replicas for different protocols. For the row marked “Chain”, the column Primary contains the number of MAC operations at the sequencer, while the column Other contains the number of MAC operations at the bottleneck (middle) replica. The number of MAC operations for Ring represents an average, since replicas in Ring can take any role, with regard to request processing.

7.3 Network Utilization

In Figure 7.4, we show the number of bytes which are sent/received by Ring replicas during replica-to-replica communication (let us recall that each of the replicas has two network interfaces: one for client-to-replica communications, and one for replica-to-replica communications). The clients issue 1 KiB requests. Similarly to Figures 3.2, 3.3, and 3.4, in Figure 7.4 we present the number of bytes which are sent (or received) for each byte received from a client. The bars *in* (*out*) denote the normalized amount of data on the incoming (respectively, outgoing) links to the replica.

The first observation that we can make is that network utilization is perfectly balanced across the different links: each of the replicas equally uses its incoming and outgoing links. The reason for such a balance stems from the fact that each of the replicas sends/receives, on average, the same number of messages. This is a consequence of the fact that each of the replicas acts, on average, the same number of times as an “entry replica”, and, also, as an “exit replica”. Consequently, when considering network utilization in Ring, each of the replicas has the same (indistinguishable) “role” in the protocol.

The second observation that we can make is that for every 1 B transmitted by a client, a replica only transmits (receives) 0.78 B on its outgoing (respectively, incoming) link. This is explained by the fact that there are 4 replicas-to-replicas links, and only 3 of them are used to disseminate request payloads (the link from the exit replica to the entry replica is not used). As any replica

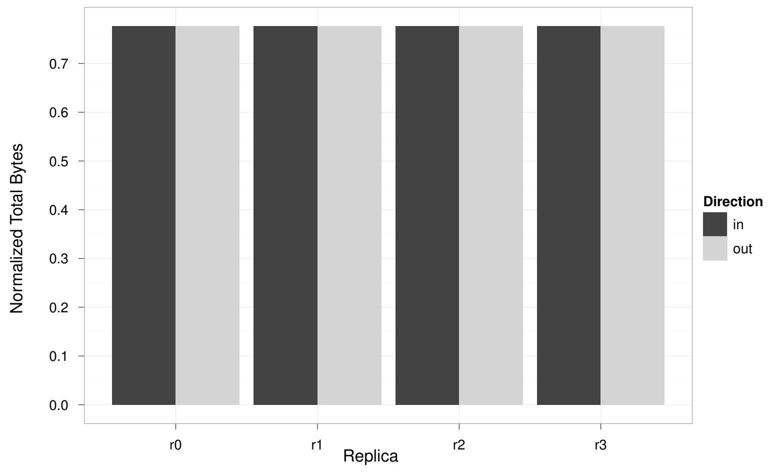

Figure 7.4: Network link utilization in the Ring protocol.

can have the role of the "entry replica" for each of the requests, then each of the replicas has the same probability to have one of its links not used. Consequently, the average number of bytes which is transmitted on each of the links should be $\frac{3}{4} = 0.75\,\mathrm{B}$, which is very close to the 0.78 B, that we observe. The slight difference comes from the fact that messages have headers and that an acknowledgement is produced for every message, thus increasing the number of bytes that are transmitted over the network links.

7.4 Microbenchmarks

The previous two sections show that all of the replicas in Ring perform similar processing and send/receive similar number of bytes. In this Section, we evaluate the impact of this balanced CPU and network utilization on the overall performance of the protocol.

First, we evaluate Ring using a standard set of microbenchmarks [Castro and Liskov, 1999], when $f = 1$. These microbenchmarks resemble various workloads, and each of them stresses different parts of the system. Then, using the same set of microbenchmarks, we assess the peak throughput that each of the protocols attains, as a function of the message size. Next, we evaluate the effects of the client load on the protocol throughput, for 1 KiB requests. Finally, we evaluate the fault scalability of Ring.

7.4.1 4/0 Microbenchmark

We first evaluate the throughput of the different protocols in the so-called 4/0 microbenchmark. In this benchmark, the clients issue large requests (4 KiB), while the expected responses are small (8 B). This benchmark exactly emulates the network-bottleneck conditions which Ring was designed for. Moreover, the 4/0 microbenchmark models write-intensive applications, similar to a networked storage, or a networked `syslog` facility. Figure 7.5 shows the results for throughput, while Figure 7.6 shows a response time vs. throughput curve for this particular benchmark.

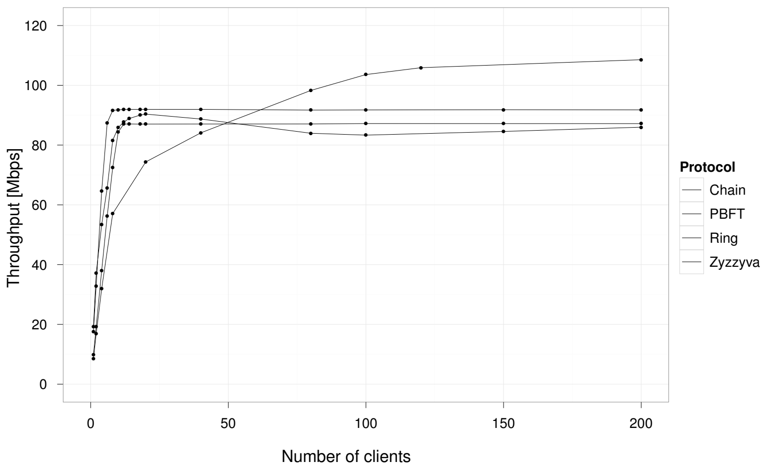

Figure 7.5: Throughput in the 4/0 microbenchmark.

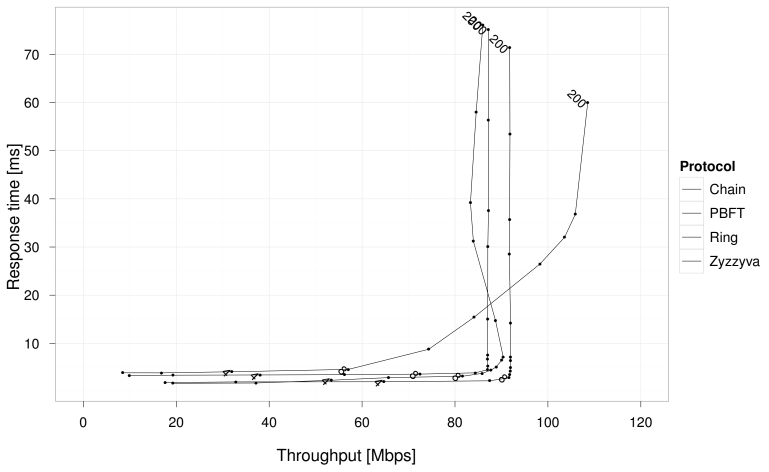

Figure 7.6: Response time–throughput curve for the 4/0 microbenchmark. Some points are annotated with the number of clients for which the given parameters were measured.

The main observation is that Ring, indeed, outperforms all of the other protocols, with as few as 60 clients. Other protocols reach a plateau of 93 Mbps, with only as few as 10 clients, with up to 7 % difference in maximum throughput. On the other hand, Ring surpasses 100 Mbps with 90 clients, and reaches 114.6 Mbps with 200 clients.

The difference between Chain and Zyzzyva rises from the fact that Chain has larger message headers, which take up the bandwidth. With more than 20 clients, the performance of PBFT starts to degrade slightly, due to its multiple communication rounds. Additionally, we observe the ill-effect of IP multicast, which congests the inter-replica LAN.

Despite having a lengthy communication pattern, Figure 7.6 shows that Ring actually achieves a lower response time than the other protocols, once it has reached its maximal throughput. With 200 clients, Ring's response time is 60 ms, while the other fastest protocol, Zyzzyva, is 20 % slower. The reason for this low response time in Ring is the fact that the queueing (waiting) time at a replica is inversely proportional to the throughput. Hence, the requests in Ring wait for less time before being processed, than in other protocols. Due to the reduced queueing time, Ring achieves an lower overall response time, even though messages take more communication steps.

7.4.2 0/4 **Microbenchmark**

The 0/4 microbenchmark models read-intensive applications, such as networked data-indexes and storage. The clients issue small requests, while the replicas generate large replies. Due to its format, this benchmark evaluates the efficiency of replica-to-client communication. Although the 0/4 microbenchmark resembles a read-intensive workload, we turn off read-optimizations in all of the protocols, and measure the performance of a more costly write operation.

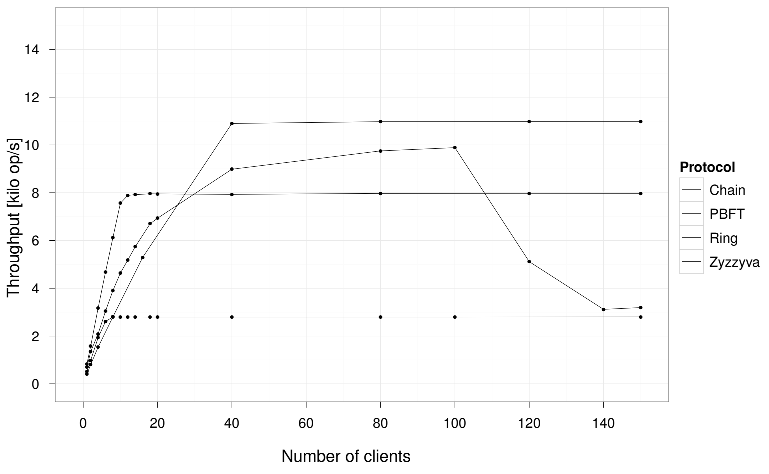

Figure 7.7: Throughput in the 0/4 microbenchmark.

Figure 7.7 (Figure 7.8) presents the throughput (respectively, the response time vs. throughput curve) for this benchmark. In Ring, different replicas reply to different clients[4], thus reducing the stress on the replica-to-client communication. Figure 7.7 supports this fact, as Ring outperforms other protocols, and reaches around 11 kops (kilo-operations per second). Moreover, Ring surpasses the other protocols with as few as 27 clients.

[4]Only one replica replies to one client.

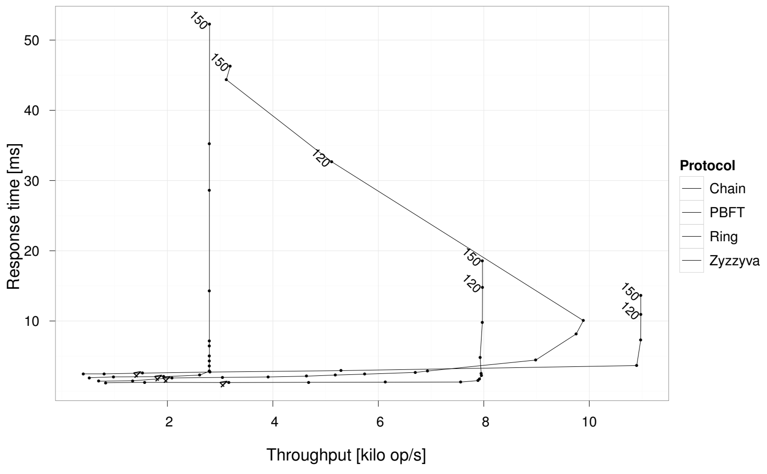

Figure 7.8: Response time–throughput curve for the 0/4 microbenchmark. Some points are annotated with the number of clients for which the given parameters were measured.

We note that, in this microbenchmark, Chain exhibits low performance. In this respect, the performance of Chain in the 0/4 microbenchmark is similar to its behaviour in the 4/0 microbenchmark, where the head was the bottleneck, due to large requests. Here, however, the tail is the bottleneck, due to large replies, which saturate the replica-to-client link. Zyzzyva and PBFT achieve a higher throughput than Chain, because different replicas may send the full reply to the client[5]. However, both Zyzzyva and PBFT perform worse than Ring. As is shown in Figure 7.7, Ring effectively utilizes all of the outgoing links, having 4 times the throughput of Chain. Zyzzyva (and PBFT) reach 8 kops (respectively, 10 kops) due to the fact that every replica replies to every client. Moreover, PBFT reaches a *congestion collapse* with more than 100 clients.

Similarly to the 4/0 microbenchmark, we observe a lower response times in Ring, too. Chain immediately enters saturation, as Figure 7.8 shows — with only a few clients, the response time vs. throughput curve becomes a vertical line on the figure. PBFT experiences a sharp increase in response times after the congestion collapse, due to frequent retransmissions, which we observe on the same figure, as the line going from the lower right corner to the upper left corner.

7.4.3 4/4 Microbenchmark

The 4/4 microbenchmark stresses all of the network links, as both the clients send large messages and the replicas issue large replies. This benchmark resembles a high-throughput P2P file-sharing service, for instance, a video on demand service. Figure 7.9 shows the results

[5]Some protocols use this optimization, in which only a certain, random replica sends the full reply, while other replicas send only a digest of the reply.

obtained for throughput, while Figure 7.10 shows a response time vs. throughput curve for this particular benchmark.

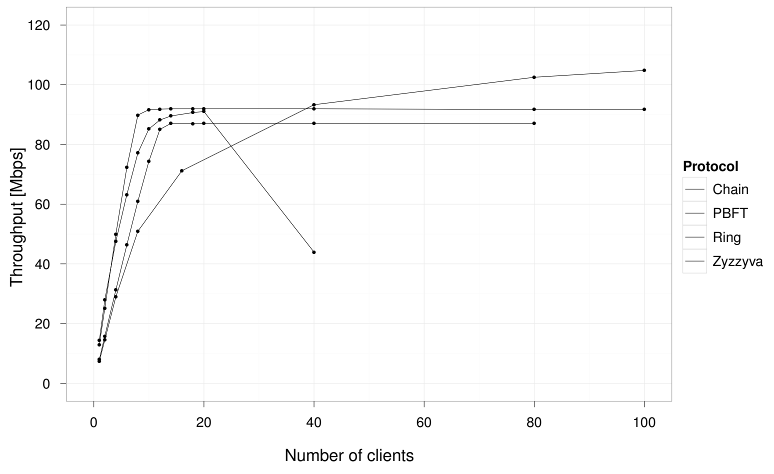

Figure 7.9: Throughput in the 4/4 microbenchmark.

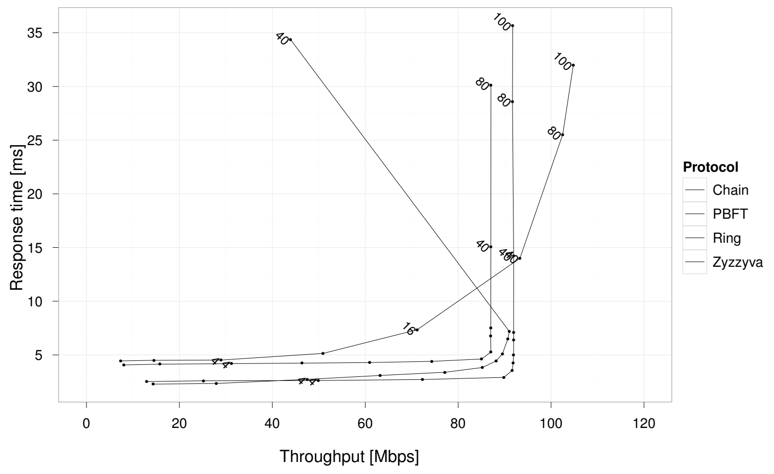

Figure 7.10: Response time–throughput curve in the 4/4 microbenchmark. Some points are annotated with the number of clients for which the given parameters were measured.

Similarly to the 4/0 microbenchmark, Ring outperforms other protocols as soon as there are more than 40 clients. We also observe that even with 80 clients, Ring has a notably lower response time.

Due to large replies, PBFT reaches a congestion collapse after 20 clients. We could not obtain measurements for PBFT with more than 40 clients, due to constant crashes. However, even without crashes, PBFT would not achieve better performance than Zyzzyva [Kotla et al., 2007].

Other protocols (except Ring) also crash with more than 100 clients. Therefore, we report results for up to 100 clients.

7.4.4 0/0 Microbenchmark

The 0/0 microbenchmark models a CPU-intensive workload, as it uses small messages and small requests. Thus, context switching when processing a network message, request handling, and cryptographic operations dominate the total cost of operations. Due to the small request size, this microbenchmark favours protocols with short communication paths (because it takes less to propagate a short message), and a small number of cryptographic operations.

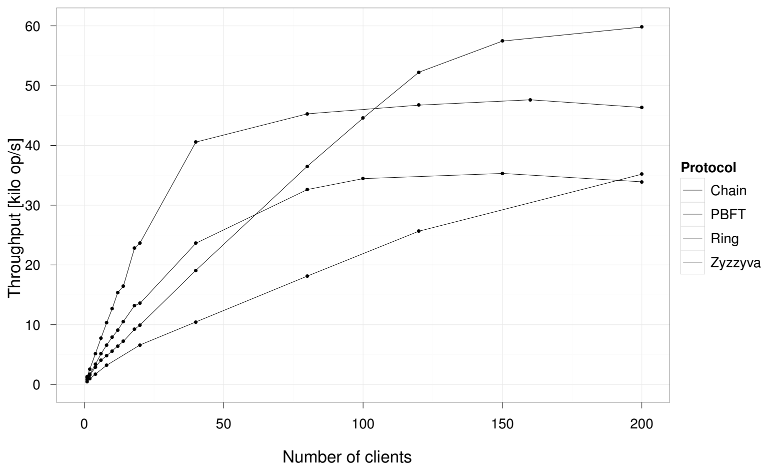

Figure 7.11: Throughput in the 0/0 microbenchmark.

Figures 7.11 and 7.12 summarize the results of the 0/0 microbenchmark. As expected, Zyzzyva dominates when there is a small number of clients, as the system is not in the bottleneck condition, and throughput depends mainly on the response times. The response time of Zyzzyva is less than that of Chain (shown in Figure 7.11), as the requests in Zyzzyva take 3 communication steps, while the communication in Chain spans over 5 separate steps. Once the CPU becomes the bottleneck, Chain achieves the highest throughput, although the number of MAC operations is the same as with Zyzzyva ($2f+2$ vs. $3f+1$). The reason for this is that the primary in Zyzzyva spends more time processing requests, as it needs to answer to the client, while handling many client requests. The response time of Ring is quite high, as each of the requests takes 9 communication steps, and the number of MAC operations is higher than in other protocols. However, we note that with over 200 clients, Ring overtakes PBFT, as the latter enters saturation, as shown in Figure 7.12. Moreover, all of the protocols, except for Ring, reach saturation with less than 200 clients. The reason for this is the inherent symmetry of Ring, as each of the replicas takes one fourth of clients. Thus, Ring stays away from the CPU bottleneck zone, even with a large number of clients present.

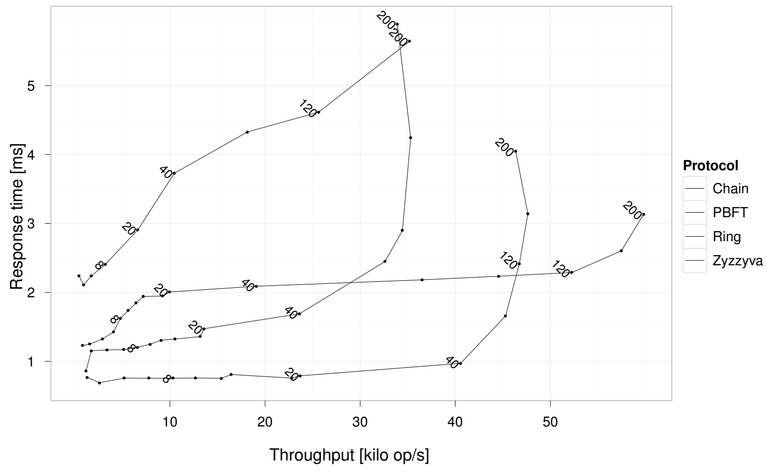

Figure 7.12: Response time–Throughput curve in the 0/0 microbenchmark. Some points are annotated with the number of clients for which the given parameters were measured.

7.4.5 Mixed Workload Microbenchmark

So far, all microbenchmarks used the constant request size. However, real applications issue request of various sizes. In order to assess the impact of variable request size, we modified the clients to issue requests of sizes: 8 B, 512 B, 2048 B, 256 B and 4096 B, in that order[6].

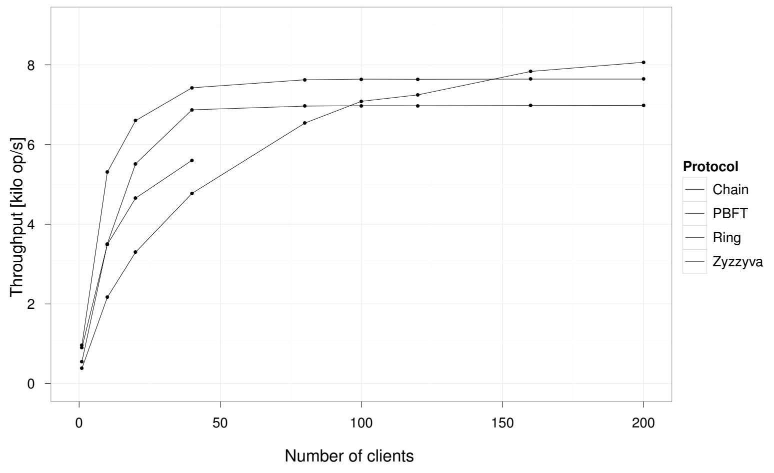

Figure 7.13: Throughput in the mixed workload microbenchmark.

Figures 7.13 and 7.14 summarize the results of the mixed workload microbenchmark. Note that we express the throughput as *kilo ops*, since requests are not of an uniform size. Due to

[6]We require deterministic order, to ensure repeatability

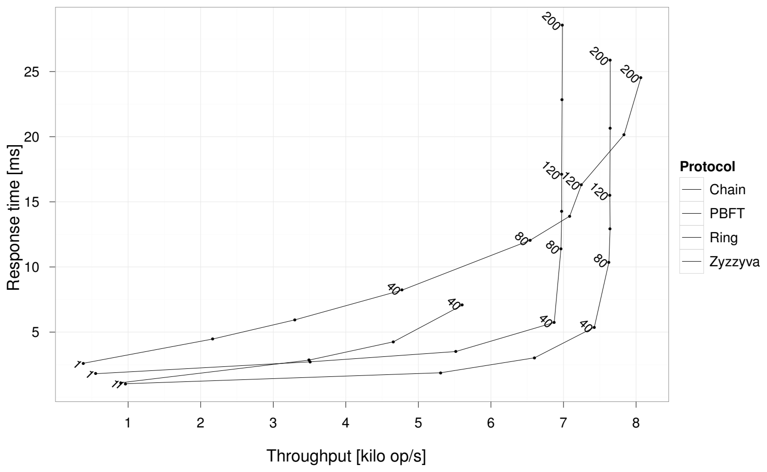

Figure 7.14: Response time–Throughput curve in the mixed workload microbenchmark. Some points are annotated with the number of clients for which the given parameters were measured.

a bug in the implementation of PBFT, we could not obtain measurements for more than 40 clients.

We observe that with a low load, Zyzzyva has the best performance, since it behaves the best with small requests. However, with more than 150 clients, Ring begins to dominate, and improves by 6% upon Zyzzyva. Moreover, the trend shows that, unlike other protocols, Ring does not reach saturation. The reason is the same as for the 4/0 microbenchmark: Ring can utilize more links to accept incoming requests, thus enabling higher total throughput. Similarly to 4/0 microbenchmark, we observe that, with 200 clients, Ring again has lower response time than other protocols with much shorter communication path.

7.5 The Impact of the Request Size

Next, we study how the throughput of different protocols is affected by the size of the requests issued by the clients. In Figure 7.15, we show the peak throughput per protocol. We have varied the size of the requests and the number of clients, while measuring the throughput. For every considered request size, we report the maximal throughput that we have observed. Note that the *x*-axis on the figure uses a logarithmic scale.

The first observation that we make from Figure 7.15 is that the behavior of protocols is similar to the simulated behaviour reported by Singh et al. [2008]: PBFT and Zyzzyva perform very similarly. The network setting that we use for these experiments influences the behaviour of Chain and Zyzzyva, as observations differ from those reported by Guerraoui et al. [2010a]: Zyzzyva and Chain exhibit negligible differences with large messages. The difference is due

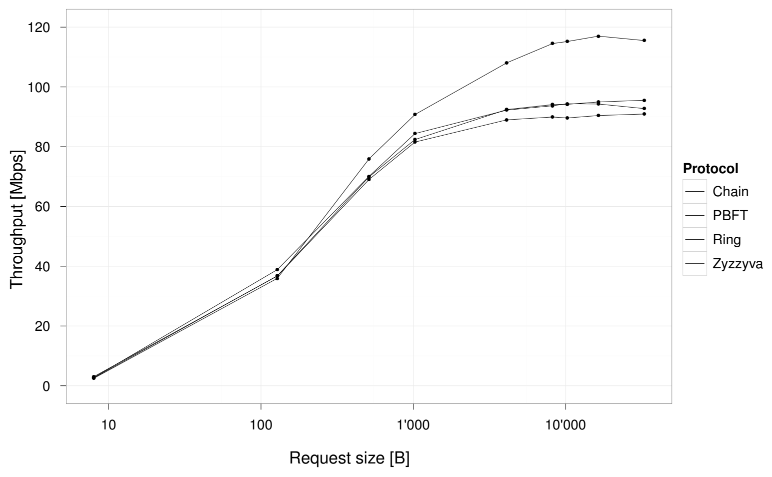

Figure 7.15: Peak throughput as a function of the size of the request.

to the fact that clients communicate with replicas using a separate, dedicated LAN. This setup reduces the number of IP multicast packet drops in Zyzzyva, effectively increasing its performance.

It can further be seen from Figure 7.15 that, with small requests (below 1000 B), all of the protocols perform similarly, which we later confirm with our performance model. With larger requests, Ring significantly outperforms other protocols. More precisely, state-of-the-art protocols have a peak throughput ranging between 90 Mbps for PBFT and 93 Mbps for Zyzzyva and Chain. Ring, on the other hand, has a peak throughput of about 118 Mbps, which represents a 27 % performance improvement over the most efficient state-of-the-art BFT protocols. Ring achieves a throughput of 118 Mbps on a Fast Ethernet network because the replicas in Ring only send/receive 0.78 B for every 1 B of a client request. To conclude, we are safe to say that, with large messages, the throughput of Ring is very close to the optimal replication throughput that can be achieved on a Fast Ethernet: 124 Mbps.

7.6 The Impact of the Number of Clients

Network bottleneck conditions may be reached when either a certain number of clients issues large requests, or when a large number of clients issues requests of modest size. Either way, it is important for BFT protocols to be able to graciously handle traffic from all of the clients, and, furthermore, a robust[7] protocol should be able to retain good performance as the number of clients increases.

Similarly to the microbenchmarks in Section 7.4, Figures 7.16 and 7.17 illustrate the performance of different protocols, as the number of clients varies from 1 to 2000. Figure 7.16

[7]In terms of client scalability.

shows the throughput as the number of clients varies, while Figure 7.17 shows the dependency between the response time and the throughput. In this experiment, the clients issue 4 KiB requests, while the replicas send 8 B replies.

Note that we do not issue 16 KiB requests (which yields the best results for all protocols, as illustrated in Figure 7.15) because both Zyzzyva and PBFT were crashing when being stressed with a large number of clients (> 120) issuing 16 KiB requests. Moreover, even with 4 KiB requests, PBFT crashes with more than 200 clients, while Chain and Zyzzyva crash with more than 1000, and 1200 clients, respectively. Ring does not crash, even with 2000 clients present.

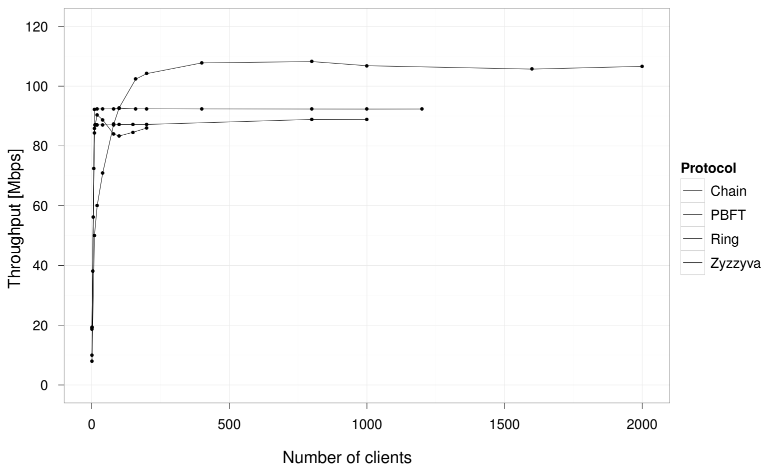

Figure 7.16: Throughput, as the number of client increases toward 2000, for 4 KiB requests and small replies.

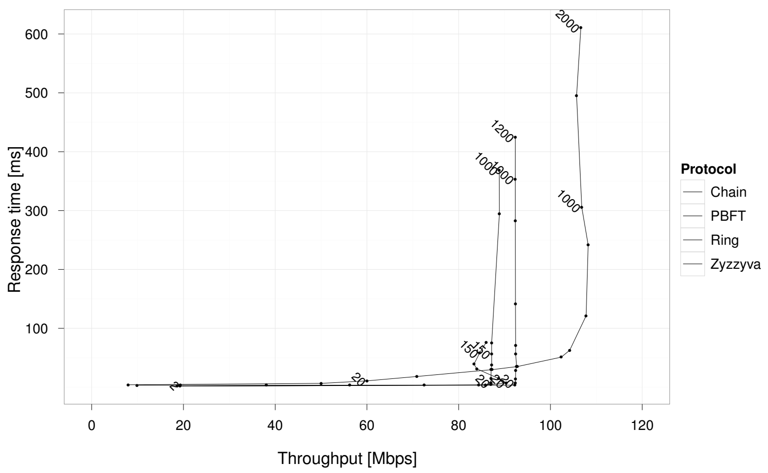

Figure 7.17: Response time–throughput curve in the case in which the system is handling a large number of clients, for 4 KiB requests and small replies.

We note that all of the protocols reach saturation after 20 clients have started using the system. Also, as the number of clients increases, all of the protocols exhibit slight performance variations, due to the high load. However, we do not observe any significant performance drops, except in the case of PBFT. One reason for such a steady performance of Ring stems from its design — the clients connect to different replicas, thus reducing the overall load.

7.7 Fault Scalability

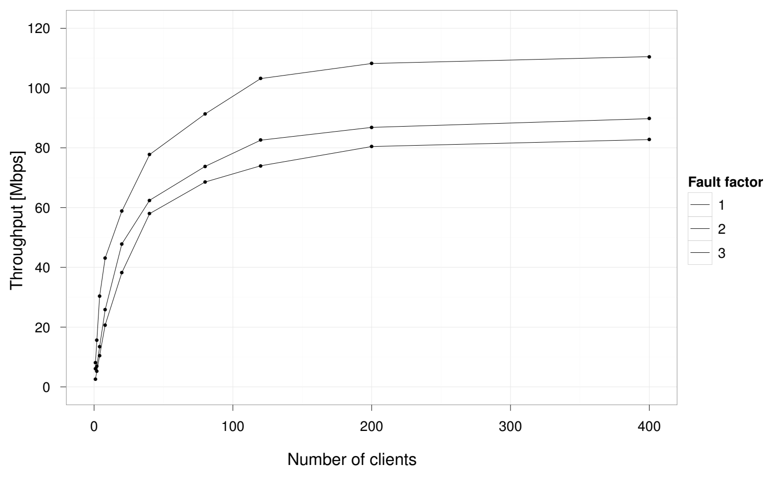

Figure 7.18: Throughput as a function of the resilience (f), with 4 KiB requests and small replies.

An important characteristic of BFT protocols is the level of their performance, as the number of tolerated faults increases. Figure 7.18 illustrates the throughput of Ring in the 4/0 microbenchmark, when f varies between 1 and 3. Kotla et al. [2007] reports that the peak throughput of PBFT and Zyzzyva slightly drops with the increase of f. Similarly, Guerraoui et al. [2010a] reports the same finding for Chain. We, too, observe a noticeable drop in performance in Ring. The reason is that the peak throughput in Ring is dependent on f, and amounts to $\mathrm{b}_{max} = \frac{3f+1}{3f}\mathscr{B}$. As is suggested in Figure 7.18, with the increase of f, the peak throughput in Ring approaches the link limit, which is, incidentally, the peak throughput of other protocols. Note that the maximum reported throughput of Ring in this experiment is 115 Mbps, as we use 4 KiB requests.

7.8 Accuracy of the Performance Model

This section discusses the accuracy of the performance model presented in Chapter 6.

Figures 7.19, and 7.20 compare the modeled versus the observed throughput for small and large requests, respectively. We obtain the modeled throughput using the MVA algorithm.

All of the models are parametrized with the values obtained in Section 7.1.1. The observed throughput represents the throughput of a *nil* service — a service with no execution time. The same service is also used in the presented microbenchmarks in Section 7.4.

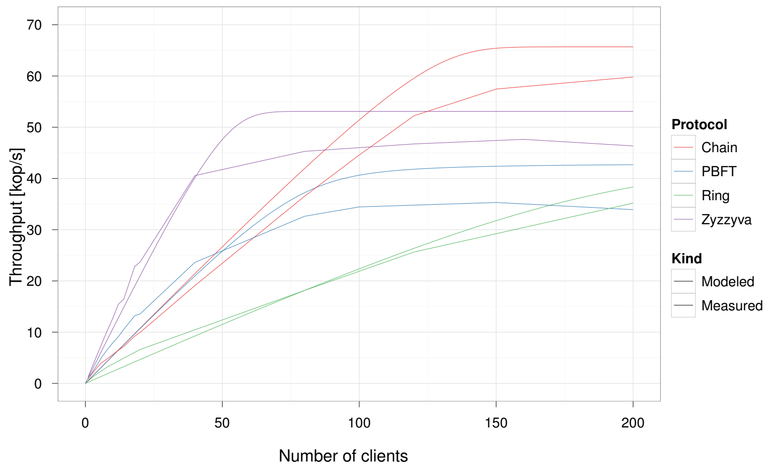

Figure 7.19: Comparison of the modeled and the measured (Section 7.4.4) throughput, as the number of load changes, with small requests and small replies.

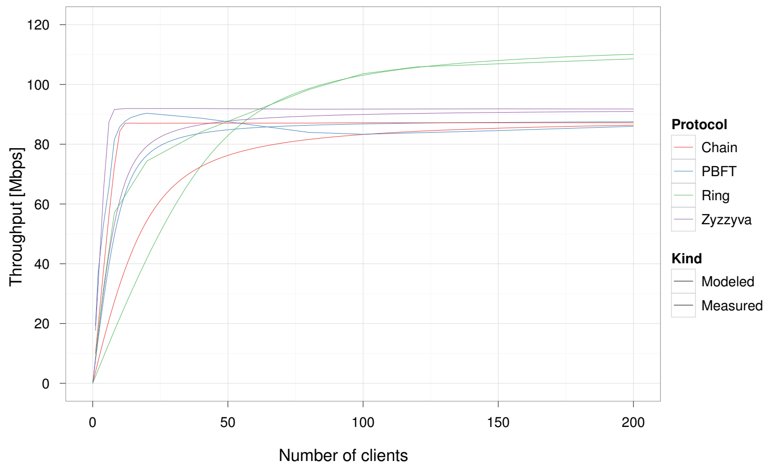

Figure 7.20: Comparison of the modeled and the measured (Section 7.4.1) throughput, as the number of load changes, with 4 KiB requests and small replies.

As Figure 7.19 suggests, the difference between the modeled and the measured data may sometimes be as much as 30 %. The reason for such a discrepency is twofold: (1) our model is quite simple, and (2) we can not capture all of the possible behaviours of the system, especially memory interactions, which matter in cases of CPU-constrained workload. Nevertheless, we note that the modeled data allows for reaching the same conclusions as the measured data. Namely:

- Chain achieves the highest throughput;
- Zyzzyva achieves a higher throughput than Chain, when the number of clients is small; and
- Ring overtakes PBFT with higher loads, by a small margin.

On the other hand, in Figure 7.20, the difference between the maximum throughputs is shown to be lower than 3 %. Under a low client load, there are some discrepancies. For example, the “knee” of the performance curve in the modeled data is reached with a higher number of clients, when compared to the measured data. This difference in behaviour is due to the fact that our model assumes a uniform operation cost over all of the possible loads. However, under a low client load, all of the operations take less time to execute. We chose not to take this dependency into account, as it would complicate the model. Nevertheless, including load-dependent processing into the model remains a possibility [Bolch et al., 2005].

7.8.1 The Impact of the Request Size

The next step in the evaluation of our performance model is the analysis of the impact of the request size on the model’s accuracy.

Figure 7.21 illustrates the relative difference between the predicted and the observed maximum throughput, as the request size varies. In this experiment, we replicate the *nil* service — a service which does not incur any overhead.

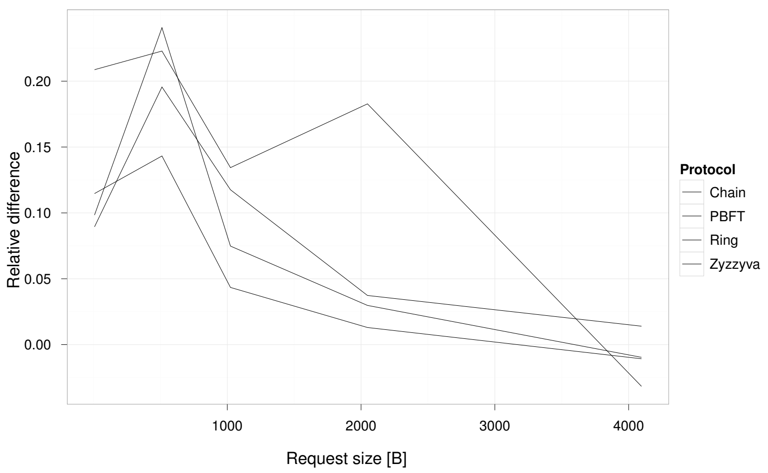

Figure 7.21: Relative difference between the predicted and the measured maximum throughput, as the request size changes.

We observe that the relative difference between the predicted and the observed maximum throughput declines, as the request size increases. The reason is that, with larger requests, processing times are greater, hence the errors in measuring the overall process time have a small effect on the total processing time.

However, for some of the values of the request size, we can see an increase in the relative difference, especially for 512 B requests. The reason for such an increase is a consequence of the inaccuracy in calculated costs per operation. As Figure 7.2 shows, we use a simple linear fit for all of the system parameters, and yet, network-related system parameters are not linearly dependent on the request size, for some values of the request size. That is exactly what we observe for 512 B requests, where the send operation incurs 40 % more overhead than the fitted model predicts. Such a large difference affects the subsequent calculations, as Figure 7.21 suggests.

The error in predicting the maximum throughput is generally[8] below 10 %, except in the case of PBFT. PBFT is presented through a simplified model that does not take all of the interactions (and optimizations) into account. Therefore, we observe an around 20 % difference when predicting the performance of PBFT.

7.8.2 The Impact of the Execution Time

In this section, we address how varying the execution time affects the accuracy of the model. So far, we have conducted all of the measurements using the *nil* service — a service which takes 0 s to process a request. However, in real deployments, the service time is significant, probably higher than the total agreement time [Clement et al., 2009b]. In this experiment, we instruct clients to issue small requests, while replicas issue small replies. We focus on small requests and small replies, because such a workload is CPU-intensive. An increase in the execution time also affects the CPU load. Moreover, as the previous section implies, our model contains the highest error in this setting. Thus, we can observe the effect of the increase in the execution time more clearly.

In Figure 7.22, we show the mean relative difference as the execution time changes, with a 95 % confidence interval. The mean relative difference (MRD) represents an average of magnitudes of relative differences between the modeled and the measured throughput, for all of the measured points:

$$\text{MRD} = \frac{1}{n}\sum\nolimits_{1}^{n} \frac{\left|x^i_{\text{measured}} - x^i_{\text{model}}\right|}{x^i_{\text{measured}}}$$

where n is the sample size and x^i is the ith sample. In our experiments, the sample size is 6 — we sample the throughput when the load is 20, 40, 80, 120, 160 and 200 clients.

Besides the fitted mean relative difference, in Figure 7.22 are also displayed the individual per-protocol relative differences, for the sake of reference. We observe that with short execution

[8]For a large range of values.

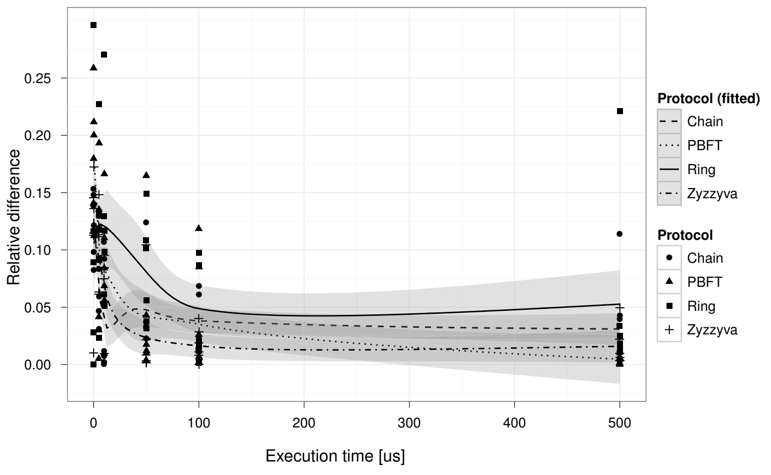

Figure 7.22: The mean relative difference as the execution time changes.

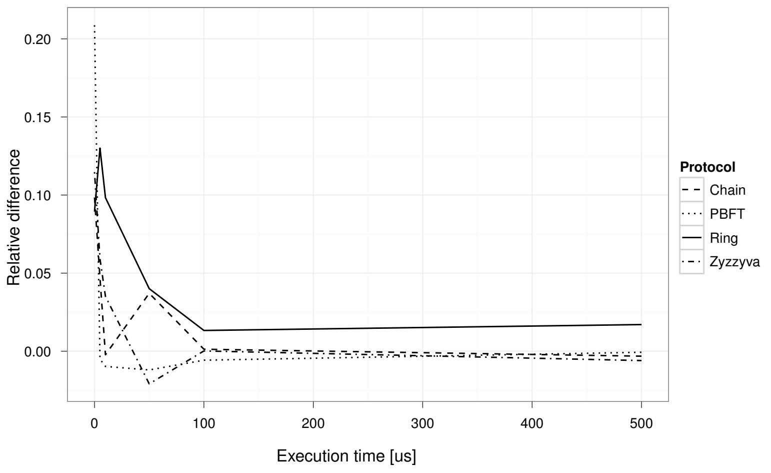

Figure 7.23: The relative difference between the predicted and the measured maximum throughput, as the execution time changes.

times, an individual relative difference could be as high as 30 % for some client load. However, as the execution time grows, the mean relative difference rapidly drops, and drops below 5 % for a large range of the values.

Similarly to the previous section, in Figure 7.23 we report the relative difference in the modeled and the observed maximum throughput, as the execution time of the replicated service varies. Both Figure 7.22 and 7.23 illustrate the same global trend — as the execution time increases, the error in prediction rapidly drops. For example, as soon as the execution time grows over 10 µs, the error in prediction is less than 5 %, for all of the protocols. Moreover, for execution times longer than 100μs, the prediction error is less than 1 % for most of the protocols, and only 2 % for Ring.

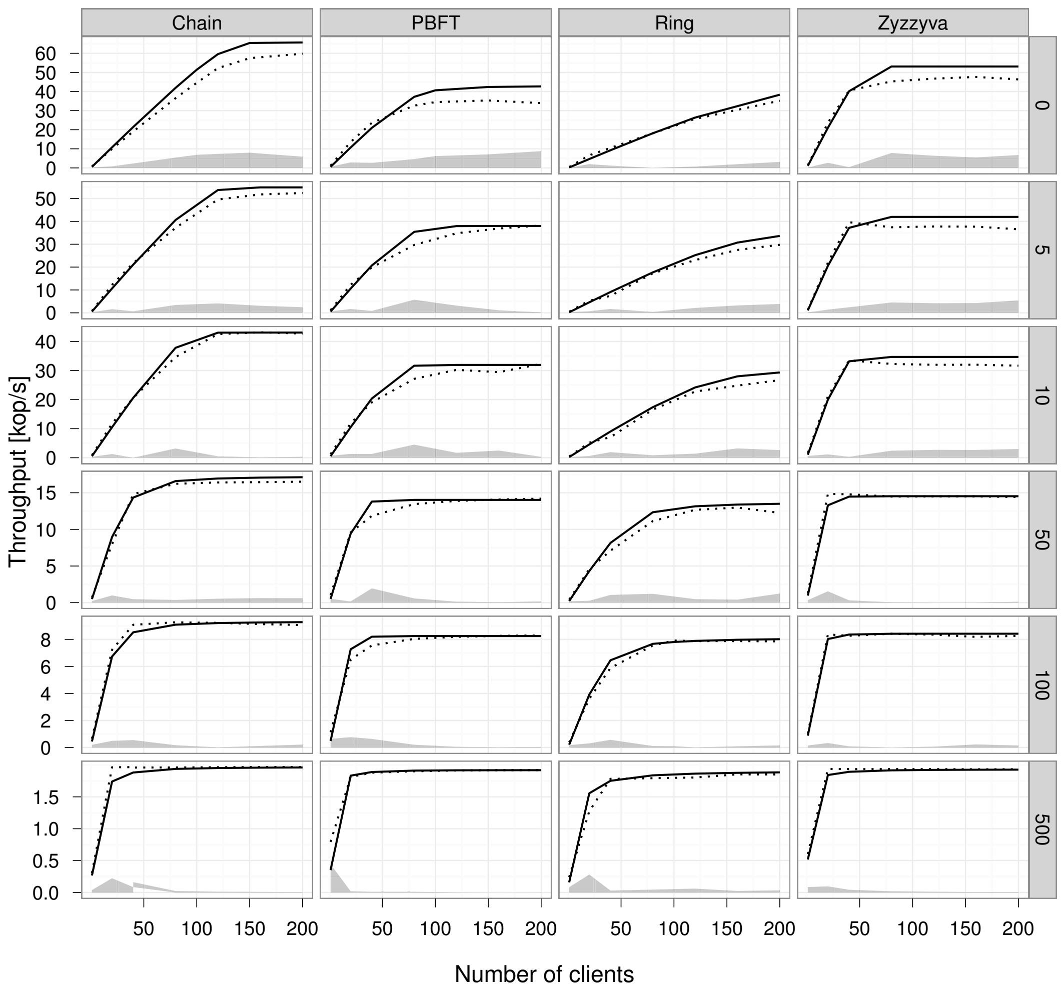

Figure 7.24: Comparison of the modeled and the observed performance, for all of the protocols, as the load and the execution time change, for small requests and small replies. The solid lines represent modeled performance, while the dotted lines represent observed performance. The greyed area at the bottom of each of the facets denotes the absolute difference between the model and the observed measurement.

In addition, Figure 7.24 displays, side-by-side, the differences for all protocols, as the execution time varies. This figure gives a visual overview of how well the model and the actual measurements match. Figure 7.24 supports the observation that the accuracy improves as the execution time increases. The reason is that the execution time becomes the dominant factor in the overall processing time. In turn, the total error in assessing the overall processing time decreases, increasing the accuracy of the model. Finally, we can conclude that our simple model achieves good accuracy, as we have rarely observe differences higher than 10 %.

7.9 Summary

The results in this chapter show that, under the network-bottleneck conditions, Ring outperforms the state-of-the-art protocols (Chain and Zyzzyva).

The results obtained from microbenchmarks show that Ring has up to 27 % higher throughput than other protocols, in the majority of the experiments. In these microbenchmarks, somewhat surprisingly, Ring achieves the best response time, although each of the requests takes 9 communications steps. The reason for such a good performance lies in the fact that, in the bottleneck conditions, the response time is inversely proportional to the throughput.

Ring achieves lower throughput (22 % less than that of Zyzzyva, and 30 % less than that of Chain), only in microbenchmarks in which operations have small arguments and result sizes. In those particular microbenchmarks, Ring generates more CPU load (on average) than other protocols, although the load is well balanced among the replicas. Ring achieves slightly higher performance than PBFT, even though (as shown in Section 7.2) Ring operates at a higher load, because with this higher load, the IP multicast incurs high message drops, and we observe lower performance in PBFT. The main reason for the lower performance of Ring in the 0/0 microbenchmark, conversely, stems from the long communication path of Ring — with small requests, transporting a request takes more time than its actual processing, which directly affects the throughput. As opposed to other protocols, Ring does not reach saturation with even 200 clients, precisely due to this long communication pattern.

The experimental results show that our analytic performance model is accurate: the absolute value of the relative prediction error for the throughput was below 10 % of the experimental results, in almost all of the experiments. Our analytic model achieves good accuracy although it requires only a small number of (protocol-agnostic) measurements.

8 Concluding Remarks

As our society relies deeply on computers, and as faults are still more the norm than the exception, highly-available fault-tolerant systems are a necessity. In the recent years, we have witnessed the emergence of Byzantine fault tolerant systems, aimed at the improvement of throughput under CPU-constrained workloads.

This thesis has focused on exploring a different design space — one in which throughput is scarce. The thesis has offered an analysis of scaling impediments in current, state-of-the-art protocols, for such throughput-constrained conditions. These impediments range from imbalances in resource utilization (e.g., when not all of the links are equally loaded), over protocol inefficiencies (e.g., the fact that IP multicast is fragile under high-load), to various implementation deficiencies, such as problems with handling large number of connections, and processing requests on multicore platforms.

Given the aforementioned analysis, we have proposed a design and an implementation of Ring — a BFT protocol which utilizes ring topology to achieve high-throughput. Ring is an agreement-based BFT protocol, with a specific replica in charge of imposing order on all of the requests. However, in Ring, that task is negligible to the extent that all of the replicas in Ring can safely be considered identical — a feature which shares a significant similarity with quorum-based BFT protocols. Furthermore, all of the replicas accept and reply to requests from clients, and it is this inherent symmetry that allows for a balanced usage of resources in Ring, thus overcoming one of the major highlighted impediments to performance scaling.

Such symmetrical processing, paired with a ring topology, allows a maximum theoretical throughput of $\frac{n}{n-1}\mathscr{B}$, where n is the number of replicas, and $\mathscr{B}$ is the link bandwidth. The throughput of all of the other protocols is limited to B. The evaluation, among other interesting properties, shows that our implementation of Ring approaches this limit, and that Ring outperforms all other, state-of-the-art protocols by 27 %.

In the last chapter of the thesis, we have presented a performance analysis framework, based on queueing theory. To the best of our knowledge, this is the first use of queueing theory in modelling the performance of BFT protocols. Our model uses protocol-agnostic measurements of the environment for parametrizing various parts of the system. Furthermore, the model is simple, and requires building the network of queues representing each processing stage *only once*, for *each* of the protocols. Given its simplicity, this performance model predicts the performance with, somewhat surprisingly, a below 5 % error for realistic execution times, and below 20 % for short (sub 10 ms) execution times.

The research on deterministic execution on multicore systems, along with the points raised in this thesis, opens a path toward efficient, scalable, and polymorphic BFT systems. Guerraoui et al., 2010a has laid out the foundation for moving correctly in-flight from one instance of a BFT protocol to another. In this thesis, we have described a protocol implemented in this framework, suitable for specific working conditions, in which it outperforms all state-of-the-art protocols. Thus, one could easily envision a system (implemented using guidelines presented in this thesis), which would monitor environment conditions, use our performance model to detect a better protocol to run, and switch to it by using the ABSTRACT framework. Ultimately, we would be able to achieve best performance for any given workload.

Finally, the main insight this thesis offers is that the advantage of symmetric systems is that all components become bottlenecks at the same time. In a non-symmetric system, some component will become the bottleneck with a much lower utilization (compared to a symmetric system), thus limiting the total utilization of the system. Thus, the power of symmetry lies in postponing the bottleneck condition to a much higher utilization, that in turn enables a higher total utilization.

Bibliography

[Abd-El-Malek et al., 2005] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. “Fault-scalable Byzantine Fault Tolerant services”. In: *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 2005.

[Aiyer et al., 2008] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. “Matrix Signatures: From MACs to Digital Signatures in Distributed Systems”. In: *Proceedings of the Conference on Distributed Computing (DISC)*. 2008.

[Amir et al., 2004] Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. *The Spread Toolkit: Architecture and Performance*. Tech. rep. Johns Hopkins University, 2004.

[Amir et al., 2008] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. “Byzantine replication under attack”. In: *Proceedings of the Conference on Dependable Systems and Networks (DSN)*. 2008.

[Aviram et al., 2010] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. “Efficient System-Enforced Deterministic Parallelism”. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[Banga and Mogul, 1998] Gaurav Banga and Jeffrey C. Mogul. “Scalable kernel performance for internet servers under realistic loads”. In: *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*. 1998.

[Baskett et al., 1975] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. “Open, Closed, and Mixed Networks of Queues with Different Classes of Customers”. In: *Journal of ACM* 22 [2 1975].

[Bergan et al., 2010] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. “Deterministic Process Groups in dOS”. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[Birman et al., 2009] Ken Birman, Gregory Chockler, and Robbert van Renesse. “Toward a cloud computing research agenda”. In: *ACM SIGACT News* 40.2 [2009].

[Bolch et al., 2005] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Shridharbhai Trivedi. *Queueing Networks and Markov Chains*. Wiley-Interscience, 2005.

[Boudec, 2010] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems.* EPFL Press, 2010.

[Bracha and Toueg, 1985] Gabriel Bracha and Sam Toueg. “Asynchronous consensus and broadcast protocols”. In: *Journal of ACM* 32 [1985].

[Cachin, 2000] Christian Cachin. “Distributing Trust on the Internet”. In: *Proceedings of the Conference on Dependable Systems and Networks (DSN).* 2000.

[Cachin, 2010] Christian Cachin. “State Machine Replication with Byzantine Faults”. In: *Replication: Theory and Practice.* Ed. by Bernadette Charron-Bost, Fernando Pedone, and André Schiper. Vol. 5959. Lecture Notes in Computer Science. 2010.

[Cachin and Poritz, 2002] Christian Cachin and Jonathan A. Poritz. “Secure Intrusion-tolerant Replication on the Internet”. In: 2002.

[Castro, 2001] Miguel Castro. “Practical Byzantine Fault Tolerance”. Ph.D. thesis. MIT, 2001.

[Castro and Liskov, 1999] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI).* Feb. 1999.

[Clement et al., 2009a] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. “Making Byzantine Fault Tolerant systems tolerate Byzantine faults”. In: *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI).* 2009.

[Clement et al., 2009b] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. “Upright cluster services”. In: *Proceedings of the Symposium on Operating Systems Principles (SOSP).* 2009.

[Cowling et al., 2006] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. “HQ replication: a hybrid quorum protocol for Byzantine Fault Tolerance”. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI).* 2006.

[Dijk, 1993] Niko M. Dijk. *Queueing networks and product forms: A system’s approach.* Wiley, 1993.

[Dobrescu et al., 2009] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. “RouteBricks: Exploiting Parallelism to Scale Software Routers”. In: *Proceedings of the Symposium on Operating Systems Principles (SOSP).* 2009.

[Duda and Czachórski, 1987] Andrzej Duda and Tadeusz Czachórski. “Performance Evaluation of Fork and Join Synchronization Primitives”. In: *Acta Informatica* 24 [1987].

[Dwork et al., 1988] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of ACM* 35 [2 1988].

[Egi et al., 2008] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. “Towards high performance virtual routers on commodity hardware”. In: *Proceedings of the ACM CoNEXT Conference.* 2008.

[Gammo et al., 2004] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. “Comparing and Evaluating epoll, select, and poll Event Mechanisms”. In: *Proceedings of the Ottawa Linux Symposium.* 2004.

[Godard, 2010] Sebastien Godard. *SYSSTAT utilities.* 2010. URL: http://sebastien.godard.pagesperso-orange.fr/.

[Gordon and Newell, 1967] William J. Gordon and Gordon F. Newell. “Closed Queuing Systems with Exponential Servers”. In: *Operations Research* 15.2 [1967].

[Guerraoui et al., 2010a] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. “The Next 700 BFT Protocols”. In: *Proceedings of the European conference on Computer systems (EuroSys).* 2010.

[Guerraoui et al., 2010b] Rachid Guerraoui, Ron Levy, Bastian Pochon, and Vivien Quéma. “Throughput Optimal Total Order Broadcast for Cluster Environments”. In: *Transactions on Computer Systems (TOCS)* 28.2 [2010].

[Intel Corporation, 2007] Intel Corporation. *Intel 5400 Chipset Memory Controller Hub (MCH).* 2007. URL: http://www.intel.com/Assets/PDF/datasheet/318610.pdf.

[Jackson, 1963] James R. Jackson. “Jobshop-Like Queueing Systems”. In: *Management Science* 50.12 [1963].

[Jalili Marandi et al., 2010] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. “Ring Paxos: A High-Throughput Atomic Broadcast Protocol”. In: *Proceedings of the Conference on Dependable Systems and Networks (DSN).* 2010.

[Kapritsos and Junqueira, 2010] Manos Kapritsos and Flavio P. Junqueira. “Scalable Agreement: Toward Ordering as a Service”. In: *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep).* 2010.

[Kegel, 2006] Dan Kegel. *C10K Problem.* 2006. URL: http://www.kegel.com/c10k.html.

[Kelly, 1979] Frank P. Kelly. *Reversibility and Stochastic Networks.* Willey, 1979.

[Kleinrock, 1975] Leonard Kleinrock. *Queueing Systems.* Vol. I: Theory. Wiley Interscience, 1975.

[Knežević et al., 2010] Nikola Knežević, Simon Schubert, and Dejan Kostić. “Towards a cost-effective networking testbed”. In: *ACM SIGOPS Operating Systems Review* 43 [2010].

[Kohler et al., 2000] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. “The Click Modular Router”. In: *ACM Transactions on Computer Systems* 18.3 [2000].

[Kotla et al., 2007] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. “Zyzzyva: speculative Byzantine Fault Tolerance”. In: *Proceedings of the Symposium on Operating Systems Principles (SOSP).* 2007.

[Lamport, 1978] Leslie Lamport. “Time clocks, and the ordering of events in a distributed system”. In: *Communications of ACM* 21 [7 1978].

[Lamport, 2004] Leslie Lamport. *Lower Bounds for Asynchronous Consensus.* 2004.

[Lamport et al., 1982] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems* 4 [3 1982].

[Libenzi, 2002] Davide Libenzi. *Improving (network) I/O performance.* 2002. URL: http://www.xmailserver.org/linux-patches/nio-improve.html.

[Lynch, 1996] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[Malkhi and Reiter, 1997] Dahlia Malkhi and Michael Reiter. "Byzantine quorum systems". In: *Proceedings of the ACM Symposium on Theory of Computing (STOC).* 1997.

[Menon et al., 2006] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. "Optimizing Network Virtualization in Xen". In: *USENIX Annual Technical Conference.* 2006.

[Pease et al., 1980] Marshall Pease, Robert Shostak, and Leslie Lamport. "Reaching agreement in the presence of faults". In: *Journal of the ACM* 2 [27 1980].

[Provos et al., 2000] Niels Provos, Chuck Lever, and Sun-Netscape Alliance. "Scalable Network I/O in Linux". In: *Proceedings of the USENIX Annual Technical Conference, FREENIX Track.* 2000.

[Reiser and Lavenberg, 1980] Martin Reiser and Stephen S. Lavenberg. "Mean-Value Analysis of Closed Multichain Queuing Networks". In: *Journal of ACM* 27 [2 1980].

[Rompel, 1990] John Rompel. "One-Way Functions are Necessary and Sufficient for Secure Signatures". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing.* 1990.

[Schneider, 1990] Fred B. Schneider. "Implementing fault-tolerant services using the state machine approach: a tutorial". In: *ACM Computer Survey* 22 [4 1990].

[Serafini et al., 2010] Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. "Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas". In: *Proceedings of the Conference on Dependable Systems and Networks (DSN).* 2010.

[Shenker and Wroclawski, 1997] Scott Shenker and John Wroclawski. *RFC 2215: General Characterization Parameters for Integrated Service Network Elements.* 1997.

[Singh et al., 2008] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. "BFT protocols under fire". In: *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI).* 2008.

[Tirumala et al., 2010] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. *Iperf – The TCP/UDP Bandwidth Measurement Tool.* 2010. URL: http://iperf.sourceforge.net.

[Veronese et al., 2009] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary". In: *Proceedings of International Symposium on Reliable Distributed Systems (SRDS).* 2009.

[Vigfusson et al., 2010] 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: *Proceedings of the European conference on Computer systems (EuroSys).* 2010.

[Vukolic, 2008] Marko Vukolic. "Abstractions for asynchronous distributed computing with malicious players". Ph.D. thesis. EPFL, 2008.

[Walrand, 1988] Jean Walrand. *An Introduction to Queueing Networks.* Englewood Cliffs, NJ: Prentice Hall, 1988.

[White et al., 2002] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. "An Integrated Experimental Environment for Distributed Systems and Networks". In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI).* 2002.

[Zhou et al., 2002] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. "COCA: A Secure Distributed On-line Certification Authority". In: *Transactions on Computer Systems (TOCS)* 20 [2002].

About the Author

Nikola Knežević was born in Belgrade, Serbia on January 17th, 1981. He completed Mathematical High School in Belgrade in the year 2000.

In the same year, he enrolled and began his studies at the Faculty of Electrical Engineering (ETF), University of Belgrade. In 2006, he obtained his graduated engineer (dipl.ing.) degree in Computer Engineering and Informatics, at the Department of Computer Engineering and Informatics, as the best student of his class.

During his undergraduate studies at ETF, he spent three months as an intern in the Networked Systems Laboratory at the School of Computer and Information Sciences of the École Polytechnique Fédérale de Lausanne (EPFL IC), in Switzerland, where he continued his education, by enrolling, also in 2006, into a doctoral programme, immediately after graduating from ETF. In 2009, he joined the Distributed Programming Laboratory at EPFL IC, to continue his doctoral studies under the supervision of Prof. Rachid Guerraoui.