REDUCING COSTS OF BYZANTINE FAULT TOLERANT DISTRIBUTED APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Chi Ho

August 2011

© 2011 Chi Ho

ALL RIGHTS RESERVED

REDUCING COSTS OF BYZANTINE FAULT TOLERANT DISTRIBUTED APPLICATIONS

Chi Ho, Ph.D.

Cornell University 2011

Byzantine fault tolerance (BFT) is a powerful technique for building software that tolerates arbitrary failures. The technique has been developed since the 70s and has a rich research literature. Yet no production system has adopted the technique, despite an increasing need from today's large and complex systems. An important reason is that a BFT system incurs significantly higher costs than a crash-tolerant counterpart. The costs include developmental costs, as a BFT system is harder to design and implement correctly, and operational costs, as a BFT system requires to run on more machines, employs more expensive cryptographic operations, and sends more and larger messages.

This dissertation attempts to reduce both developmental and operational costs of BFT distributed applications. In the first part, we propose an approach to translate existing crash-tolerant systems to Byzantine-fault-tolerant. Our approach makes fewer assumptions about the source crash-tolerant distributed applications, and thus it is applicable to a larger set of applications than existing approaches. We propose and prove correct our basic approach. Then we extend the basic approach to large scale distributed applications and propose additional mechanisms to deal with practical problems in such settings—namely, replication and host churns. We evaluate our scalable translation approach by a simulation and case studies.

The second part of the dissertation presents a novel Byzantine replication

approach that focuses on reducing resource costs (but can also be used as a translation). By leveraging an external reconfiguration service, our replication approach requires only $t + 1$ replicas and t witnesses (lighter hosts) to tolerate t Byzantine faults. Our approach also uses inexpensive HMAC signatures and imposes a chain communication pattern between the replicas to reduce CPU and network consumptions. We also propose a variation of the approach, in which Byzantine hosts do not forge CRC checksums, to further reduce the resource costs. In particular, it uses $t + 1$ replicas, CRC checksums, and t fewer rounds of communication. We evaluate the performance of a prototype of each variation in the common case, when it runs without failures.

BIOGRAPHICAL SKETCH

Chi Ho, or Hồ Hải Chí (in Vietnamese[1]), was born to Hồ Ngọc Ẩn and Trần Ái Trung in Hồ Chí Minh city, Vietnam, on September 8, 1978. His childhood memories were filled with warm and joyous moments with his parents and siblings, Hồ Ngọc Ý Hạnh, Hồ Ngọc Chi Lâm, and Hồ Thiên Tứ . Throughout his studies, Chí was fortunate to have the full support from his parents and, possibly, the blessing from his deceased brother, Hồ Ngọc TuấnTriết. In the early years, Chí had the opportunity to attend one of the finest middle schools and high schools in Vietnam from their opening days, Trường Chuyên Quận I (later renamed Trường Chuyên Nguyễn Du) and Hệ Chuyên Toán-Tin Học Trường Đại Học Tổng Hợp TP. HCM (later renamed Trường Phổ Thông Năng Khiếu), respectively. There he learned his first lessons in computer science and learned to be humble as he was exposed to many brilliant minds from a very early age. Chí's fondest childhood memories were the summer vacations in Đà Lạt with his family and the countless soccer games with friends after classes.

After finishing high school, Chí and his siblings started a family business to make a living. Founded on November 30, 1996, their computer service was one of the first, if not the first, businesses in Ho Chi Minh city that rented computers with multimedia games. The business went well, but Chí had to drop out from college to run the business.

In the hot August of 1999, Chí and his family moved to Austin, Texas to start a new life. Empty-handedly starting a new life is difficult, especially in a country where people speak a different language. Fortunately, America often generously offers opportunities and support for hard-working people to pursue their dreams. Chí's dream was to continue his interrupted education and to find

[1] In Vietnamese we write family name first, then middle name, which usually is one or two words, and given name last.

business ideas in technology. And he started his pursuit by studying English while working at a convenient store to support himself. After a year, he enrolled in Austin Community College, and he transferred to the University of Texas at Austin (UT-Austin) in the next year.

In the summer of 2002, Chí met Phan Đỗ Thủy Linh, the most beautiful girl he had ever seen. Two and a half years later, they got married. Their wedding was held in a municipal court east of Austin, and his younger brother, Tứ , was the only guest. It was and still is Chí's happiest day.

At UT-Austin, Chí met Professor Lorenzo Alvisi and was inspired to pursue distributed computing in graduate studies. In the summer of 2005, Chí graduated with highest honors from the department of Mathematics and department of Computer Sciences, with special departmental honors in Computer Sciences. Following Professor Alvisi's footsteps, Chí joined the department of Computer Science at Cornell University in the fall of 2005. Under the guidance of Dr. Robbert van Renesse, Chí graduated from his Ph.D. program at Cornell University in August, 2011.

During his undergraduate and graduate studies, Chí spent several semesters working in industry. In the summer of 2003 and the spring of 2004, he worked at National Instruments, where he built the NI-CAN Driver Development Toolkit and dug into the NI-DNET firmware to find and fix a 5-year-standing bug. In the summer of 2004, he worked at Microsoft, where he fixed Windows bugs and released hotfixes. In the summer and fall of 2008, he worked at Google, where he prototyped a fault tolerance solution for a telephony system. Since April, 2011, Chí has started his career at Amazon, where he helps build a highly reliable cloud computing service.

This dissertation is dedicated to my parents.

ACKNOWLEDGMENTS

Getting a Ph.D. would not have been possible for me without many sources of inspiration and supports in life and in academia. First and foremost, I am deeply indebted to my parents for giving me life with a healthy body and for encouraging me to strive for lasting values. Without their unconditional support, interferences from life would have derailed me from my dream.

My dream of going to graduate school had been fuzzy until I met Lorenzo Alvisi. Through his lectures, I started to realize the importance and challenges of distributed computing in today's and tomorrow's everyday life and became attracted to the area. Lorenzo kindly accepted my independent-study request. And during my senior year, he and his student, Jean-Philippe Martin, patiently guided me in my first steps as a researcher. I am very thankful to both Lorenzo and JP for their guidance.

I graduated from UT-Austin, but Lorenzo's positive influence did not stop there. Lorenzo wrote one of my letters of recommendation that helped me get into graduate school. He continued helping me as a member of my thesis committee and giving me wise advice whenever I needed it. I will never forget our chats during my impromptu visits when I came back home in Austin. His real-life story and experience as a graduate student at Cornell helped me to get through the rough and tumble in my early career as a graduate student.

The best part about Lorenzo is that I am not his exception. Lorenzo has always attended to everyone and everything with great care. His passion for research, dedication for teaching, and genuine interest in his students' success have always been an endless source of inspiration to me. In my heart he is always "Professor Alvisi".

My living situation almost prevented me from going to Cornell. I received Cornell's admission shortly after Linh and I got married. At the time, Linh had just transferred to UT-Austin, and, given our financial situation, transferring again to Cornell was not an option. I seriously considered staying at UT-Austin with Linh. It would not have been a bad choice at all. In fact, I would get to continue studying with Lorenzo. The only downside was that I would follow the same school of thought as my undergraduate studies. Linh understood, and she encouraged me to go to Cornell to broaden my perspective. Without her encouragement, I would have forgone the opportunity to study at Cornell. Her understanding and sacrifice have been the sweetest gifts to me.

I had an invaluable experience studying at Cornell and with my advisor, in particular. My advisor, Robbert van Renesse, has been an advisor that any graduate student would wish for. He let me explore my own research interests, while keeping an eye on my research to keep me from heading into a wrong direction. Robbert has always given me advice and support whenever I needed them. I could discuss with him any technical problems, from high level concerns such as why existing Byzantine fault tolerance is not attractive in cloud computing production, to fine implementation details such as whether to handle client requests by a single or multiple threads. Studying under his guidance, I was able to balance theory and practice accordingly to my belief—that is, complex systems must work based on working principles, and that theory only makes impact when it is applied to practice.

My gratitude to Robbert is beyond academia. During 2008 and 2009, my personal life hit an upheaval that caused me almost to quit my Ph.D. program. Despite my lack of progress, Robbert was very supportive and shed a positive spirit over my darkest days. It helped me get back on my feet and continue my

studies. I would like to thank Robbert with utmost sincerity.

Besides Robbert, distributed systems research at Cornell is led by Ken Birman and Hakim Weatherspoon. I am thankful to Ken, Robbert, and Hakim for their leadership. And even though I did not get to work on a research project with Ken or Hakim, I am thankful for the various advice I received from Ken and Hakim, ranging from public speaking to career direction.

Fred Schneider generously supported me two semesters at Cornell, even when I was not working for him. I also had a chance to serve as a teaching assistant for Fred during the last three semesters I was resident at Cornell. His care for students, his perspective on the importance of matters, and his consistent aim for excellence are a model that I have been trying to follow. I am grateful to Fred's support and truly appreciate the lessons from him.

Joe Halpern and David Easley kindly accepted my invitation to join my thesis committee. From Joe I learned the importance of clear and precise definitions of problems. And from David I learned my first lessons in economics—a little later than I should have, but it is better late than never. I am grateful to Joe and David.

Financial support from Cornell has been generous. It made the airplane tickets I purchased to visit my wife more affordable. I would like to thank the university and the department for their support. And in particular, I would like to thank E. Gun Sirer and Rafael Pass for financial support.

Studying and research would not go as smoothly as they did without support from many staff members in the department. I would like to thank them all: Bruce Boda, Tammy Gardner, and others for helping me to settle in my new office and resolve computer-related issues; Maria Witlox, Angela Downing, Laurie Buck, and others for helping me with teaching-related issues; and Becky Stewart and Stephanie Meik for their help with any questions regarding the Ph.D.

program. I am especially grateful to Becky. In a way, Becky has been like a big sister that looks after us. Without her, we, graduate students, would easily get lost in the administrative tasks and procedures.

My biggest regret is that I was not smart enough to shorten my Ph.D. studies. During the long separation that Linh stayed at home in Texas and I stayed at Cornell, we have drifted apart. I started questioning every value, goal, and the meaning of life. The years of 2008 and 2009 were very depressing. Yet during this time, I was fortunate to have good friends by my side. Nguyễn Hoàng Nam, Nguyễn Tiến Thành, and Nguyễn Đức Linh would keep me companied. Trần Nguyễn Nhị Thừa, Võ Đình Cát Thanh, and Nguyễn Thị Hồng Nhung would often make me feel better by their uplifting stories. Over the time, I regained my composure and gained a fresh perspective over my problem. I highly appreciate their friendship, sympathies, and encouragement.

In the frigid winters of Ithaca, I was fortunate to find a home away from home. Thanks to all my friends in VITCO (Vietnamese IThaca COrnell group), I would always look forward to our traditional Tết Nguyên Đán every year. Those were the most joyful and warm events that anyone could find deep in Ithaca's winter. On a smaller scale, I always enjoyed hanging out with my close friends, to name a few, Nguyễn Hoàng Nam, Nguyễn Tiến Thành, Nguyễn Đức Linh, Trần Nguyễn Nhị Thừa, Trần Lâm Kim Tú, Vương Văn Thu, Nguyễn Thị Hồng Nhung, Lâm Ngọc Hạnh, Trần Thị Thủy Tiên, La Thị Minh Thúy, Bùi Thị Thúy Hồng, and Nguyễn Thị Minh Châu. Chatting with them over lunch or dinner was my main means to de-stress. And on Friday evenings, we got over our homesick together by cooking some Vietnamese dishes and playing bowling or karaoke. I will miss them all.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As the world becomes more and more connected, we become increasingly dependent on online services. We use traditional digital services such as data storage to store our files (e.g., [7, 4, 18]), photos (e.g., [14, 28, 23]), video clips (e.g., [30, 5, 17]), etc., so that we can access them anywhere and share them with friends and family all over the world. We also depend on online services for other work, ranging from mundane tasks such as ordering pizzas (e.g., [21, 6]) and getting directions (e.g. [16, 13]), to important tasks such as banking (e.g., [3, 27, 15]), payment processing (e.g., [12, 22, 1]), trading stocks (e.g., [11, 24, 8]), applying for jobs (e.g., [19]), and governmental document processing (e.g., [25, 20, 26]). There are more online services today than ever, and the number of such services keeps increasing rapidly.

Online services not only grow in numbers, but also grow in size. By the time this dissertation is being written, Facebook [9], Yahoo! Mail [29], and Amazon [2] have reached more than 500 million, 250 million, and 81 million users, respectively [10, 74, 126]. To accommodate the great numbers of users, online services need to spread the load over many machines. Service providers run distributed applications that coordinate the machines to implement the services.

To be competitive, service providers need to keep users' data safely and keep their services up and running as much as possible despite machine and software failures. However, building a fault tolerant distributed application is expensive, especially when the failures can be *arbitrary* (a.k.a. *Byzantine*). In this dissertation, we will look into techniques that lower the costs of developing and running distributed applications when facing arbitrary failures.

This chapter introduces the problem. Section 1.1 discusses the challenges that motivated this research. Section 1.2 presents our fault tolerance approach to distributed applications. The approach helps us to tackle the challenges. Then Section 1.3 briefly outlines the contributions of this dissertation. And Section 1.4 outlines the organization of the dissertation.

1.1 The Challenges

Byzantine fault tolerance was invented in the late '70s,[1] when NASA initiated a project to build computers that are reliable enough to fly aircraft [131]. Today Byzantine fault tolerance continues to be used in flying aircraft and space shuttles [45], but it also finds new application in distributed applications. As mentioned earlier, distributed applications that implement online services could scale to millions of users. When a distributed application scales up, it suffers from increasingly frequent and complex failures due to the increased system and code sizes, respectively. Studies (e.g., [100, 55, 79, 122]) have shown that most failures exhibit behavior that is more complicated than halting (a.k.a. *crash failures*). Spontaneous bit flips, disk and memory corruptions, misconfigurations, Heisenbugs [78], and malicious compromise are examples among such failures. A scalable distributed application therefore had better tolerate failures beyond crashes.

Yet, most fault-tolerant distributed applications deployed today (e.g., [73, 59, 66, 50, 88]) are limited to tolerating crashes. It is not that we do not know how to build Byzantine fault tolerant (*BFT*) distributed applications. We know

[1] The name *Byzantine* was not coined until 1982 [97].

how to build practical scalable data stores that tolerate Byzantine failures, such as [53, 33, 38, 130]. We know how to build BFT peer-to-peer protocols (e.g., [39, 52, 82, 87, 101]). And we also know the state machine replication approach (SMR) [93, 121], which can be used to develop a BFT version of any deterministic protocol or distributed application. Yet, again, no distributed application in production tolerates Byzantine failures.

A natural question arises at this point: Why is Byzantine fault tolerance not good enough for general distributed applications, while it has been used for some specific applications such as aircraft control for a few decades?

We identify two challenges that impede the deployment of BFT distributed applications in practice:

- BFT protocols are notoriously hard to design and implement correctly. Because arbitrary failures can occur, the applications need considerably more transitions to handle all types of failures and considerably more state to keep track of the inputs. In addition, subtle mistakes in the protocol design or in the development may leave vulnerabilities in the resulting applications. The SMR approach can help to alleviate this difficulty, but it will incur prohibitive costs.
- BFT systems implemented by the SMR approach have cost-of-ownership much higher than that of crash-tolerant systems. The costs come from two sources: replication and computation.
  - Replication: In an asynchronous environment, both crash-tolerant and BFT systems use the SMR approach. But BFT systems require t more replicas to tolerate t failures. In a synchronous environment,

crash-tolerant systems can leverage the primary-backup approach and save $2t$ more replicas than BFT systems.[2]

– Computation: BFT protocols require some cryptographic primitives such as HMAC [92] and/or RSA [120] that are necessarily computationally expensive in order to prevent brute-force attacks that try to forge invalid messages. Furthermore, BFT protocols exchange more messages than crash-tolerant protocols. When comparing with primary-backup protocols, BFT protocols exchange a quadratic number (in t, the maximum number of failures that can be tolerated) of messages to order inputs in the worst case [67], while primary-backup protocols only exchange a linear number of messages w.r.t. t. The performance of BFT protocols is thus lower than their crash-tolerant counterparts, as BFT protocols exchange more messages, and each message takes a longer time to process.

The two challenges are not large concerns in the original use of Byzantine fault tolerance. First, there are only a small number of applications running in an aircraft control system. Second, the applications running in an aircraft control system are small scale. And so, the cost-of-ownership of such applications is manageable. Distributed applications that implement online services, in contrast, could scale to millions of users. Their scales amplify the challenges above.

In this dissertation we will tackle these two challenges.

[2] In a synchronous environment, BFT systems can trade low CPU consumption for low replication cost by using digital signatures. Authenticated BFT systems only require $t + 1$ replicas in a synchronous environment.

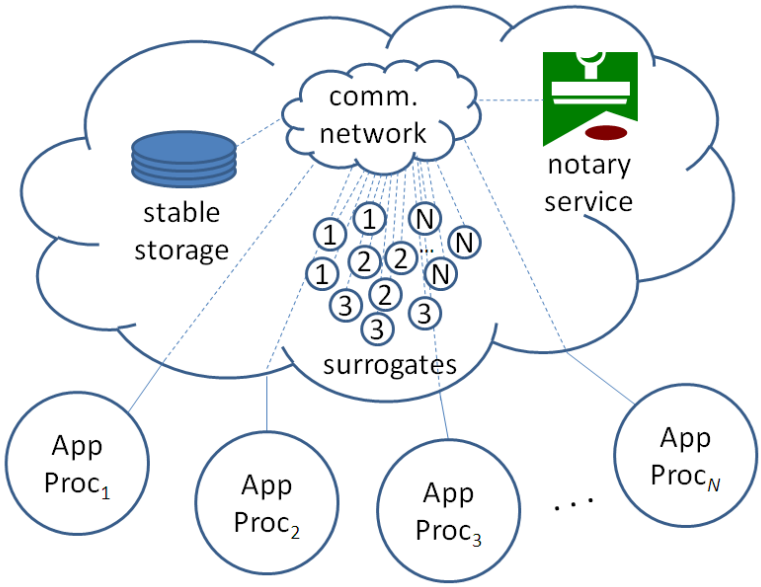


Figure 1.1: A Fault-Tolerance Substrate.

1.2 Our Approach

This section informally presents our approach toward fault tolerant distributed applications. We start with informally describing a model of such systems.

A *distributed system* comprises a set of application processes that implement a *distributed application* and a set of client processes that use the application. Each application process implements a *deterministic state machine*.[3] The processes communicate with one another by passing messages over a *fault-tolerance substrate*.

A fault-tolerance substrate comprises a *communication network* that transmits messages, a *stable storage* module that keeps data retrievable even when failures occur, a *notary service* that certifies messages not being forged by faulty

[3]Non-determinism, such as random values and clock values, can be modeled as inputs to deterministic transitions.

processes, and a set of *surrogates* that are replicas of the processes. Figure 1.1 illustrates the model.

When a client or an application process wants to send a request to another application process, it first uses the notary service to convert the request to a *self-verifiable* request. A self-verifiable request can by itself convince a receiver that it is not forged by a faulty process. We call such messages *valid*—a more formal definition follows in Chapters 3 and 5. When a process receives a request, it also uses the notary service to verify the request's validity.

When an application process receives and successfully verifies a request, it can execute the request and send outputs upon a condition: the distributed application sends outputs to the clients only when its causal past has been stored in stable storage. Stable storage stores self-verifiable data, such as messages, checkpoints, or incremental state changes. Stable storage preserves two invariants: (1) if a piece of data has been stored and has not been deleted, it will be retrievable; and (2) it is always possible to list the data that have been stored and have not been deleted.

Surrogates are optional in our model. When a distributed application needs to mask failures without interruption to the normal operation, a set of surrogates are associated with each application process. Surrogates monitor the stable storage for updates, verify the updates using the notary service, and apply the updates to bring themselves up-to-date with the application process. Surrogates can actively participate in the normal operation of the application, or sit passively as backups for the application process.

By modeling a distributed system this way, we can add, remove, and op-

timize each module separately. This flexibility is the key we use to tackle the challenges described in the previous section.

1.3 Contributions of the Dissertation

The dissertation presents a novel approach to fault tolerance, which separates the concerns in fault tolerance in a new way. The approach leads to two main concrete contributions:

- A translation approach that can be used to transform any crash-tolerant distributed applications to Byzantine-tolerant ones. We present two versions of the approach. The basic one is helpful in designing new BFT protocols based on existing crash-tolerant protocols, but it does not scale. The scalable version is applicable to large-scale distributed applications.
- An affordable replication protocol that leverages an external configuration service to use $t + 1$ replicas to tolerate t Byzantine failures. We present two variations of the protocol. The variations show a tradeoff between the assumptions we make about the adversary and the costs.

We also contribute the following building blocks:

- An ordered broadcast protocol that guarantees that the correct receivers will receive the broadcast messages in the same order, even when the sender is malicious.
- A credit-based flow control protocol that forces the replicated state machines to process inputs fairly or halt.

- An illustration of how to apply the basic translation approach to derive a BFT protocol from a crash-tolerant one. We derive the PBFT protocol [53] from the Viewstamped Replication protocol [115].

1.4 Outline of the Dissertation

The dissertation is organized as follows. Chapter 2 gives an overview of the related background and puts the dissertation into context. Chapter 3 presents the basic version of our translation approach. Chapter 4 presents the scalable version of our translation approach and its evaluation. Chapter 5 presents the affordable replication protocol and its evaluation. Finally Chapter 6 concludes the dissertation with discussion of future directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

The literature contains a rich body of work on fault tolerance. Some approaches are general enough to provide fault tolerance to many types of distributed applications, while others target specific applications such as data storage and message transmission. In this chapter we first discuss well-known fault tolerance approaches and map them to our model. Then we discuss related work on techniques that automatically strengthen distributed protocols and systems. And finally, we discuss related work on techniques that reduce replication costs in fault tolerant systems.

2.1 Fault-Tolerance Approaches

General fault tolerance approaches, when mapped to our model, implement one, two, or all three components in the fault tolerance substrate (all but the communication network). Their fault tolerance capability and performance depends on what they implement.

The rollback recovery approaches (R/R) such as message logging and checkpointing [69] assume the existence of stable storage and optionally employ a notary service [34], but do not have surrogates. Application processes save state and/or messages in stable storage while running. When an application process crashes, a new one loads the state and messages from stable storage and recover the system to a consistent state. Since there are no surrogates, these approaches can take considerable time to recover from failures.

The primary-backup approach (P/B) [37, 49] models each application process as a primary and implements stable storage that stores the state of the primary using t backups. The t backups also serve as surrogates, which can take over from the primary when it fails. The only component missing in the P/B approach is a notary service. Similar to R/R approaches, the absence of a notary service helps the P/B approach to incur low overhead in normal runs, but prevents it from tolerating Byzantine failures. Unlike R/R approaches, the P/B approach recovers quickly from failures, because the surrogates (backups) are kept up-to-date with the primary.

R/R and P/B approaches rely on synchrony conditions to decide when to replace an application process by a new one (in R/R) or by a surrogate (in P/B). These approaches use ad-hoc methods to handle failure detection mistakes happening when the synchrony conditions are violated.

The state machine replication approaches (SMR) [93, 121] work in both synchronous and asynchronous environments. SMR replicates each application process to implement the stable storage abstraction, the surrogates, and, in the case of Byzantine fault tolerance, the notary service abstraction. There are two variations of SMR: symmetric and leader-based. The two variations share the same characteristics that (1) messages rather than state are stored in stable storage, and (2) replicas actively execute the messages to bring themselves up-to-date.

Symmetric SMR protocols such as those using randomization [42] employ replicas that play the same role. Such protocols make no distinction between the application process and its surrogates. And so they can mask failures without interrupting the application execution. However, because the replicas in SMR

need to agree on the order of the input messages in order to implement the stable storage abstraction, the symmetry of the replicas incur higher overhead than P/B in normal runs to order the inputs.

Leader-based SMR protocols such as [94, 95, 58] are similar to P/B in that they designate a replica to act as the leader. The leader plays the role of the application process, while the others play the role of surrogates. Thus leader-based SMR protocols are similar to P/B in that they incur low overhead in normal runs and require a quick recovery to overcome leader failures (in contrast to masking failures on the fly as in symmetric protocols).

Some other approaches in the literature are not originally designed for general fault tolerant distributed applications, but can be used to build all or part of a fault tolerant distributed application.

The transaction processing approach (T/P) [127, 75, 77] is designed to maintain a consistent state when executing transactions in distributed databases. Each participant in a transaction can execute multiple operations, and the operations may differ among the participants. When we treat the execution of each input as a transaction, T/P can be used to implement fault tolerant distributed applications. More specifically, we can replicate each application process, and designate one replica to serve as the coordinator. Input messages are submitted to the coordinator, which coordinates the replicas to handle each input message as a transaction. When putting this into our model, the replicas implement the stable storage abstraction. Similar to R/R, T/P leverage stable storage to tolerate failures. But since T/P does not employ surrogates and notary services, it does not tolerate Byzantine failures and takes considerable time to recover from failures.

The quorum system approach (Q/S) [84, 104, 32] implements a fault tolerant variable by maintaining multiple copies of the variable. Each operation on the variable is applied to a large-enough quorum of replicas and brings the replicas to the same state. But in contrast to SMR, the replicas in Q/S are not required to go through the same sequence of operations. This approach implements stable storage. And when the stable storage is used to store the application state, Q/S can implement fault tolerant distributed applications. Fault tolerant distributed applications based on Q/S can mask failures and can tolerate Byzantine failures if they also implement surrogates and a notary service. The difference in the number of replicas and the size of the quorum determines the number of surrogates used. Q/S masks failures until there is no surrogate left. A notary service can be implemented in Q/S by using a larger number of replicas and a larger quorum size, and/or by using cryptographic primitives.

Various works in coding theory such as CRC checksum [118], hashed message authentication codes (HMAC) [92], RSA digital signatures [120] enable data integrity verification and can be used to implement a notary service under different adversarial strengths. In a Byzantine environment (strong adversary), these techniques can be used only for authentication but not for message validity verification. Malicious processes can forge invalid messages with valid signatures. Such hostile environments require stronger methods for building a notary service. Examples are using replicas along with digital signature, and using proof-carrying data [61].

Table 2.1 summarizes the approaches for fault tolerant distributed applications.

By separating fault tolerance concerns from applications, we are able to ma-

Notary Service	Surrogates	Approaches
No	No	R/R, T/P
No	Yes	P/B, SMR, Q/S
Yes	No	BFT R/R, Shuttle (Chapter 5)
Yes	Yes	BFT SMR, BFT Q/S

Table 2.1: General approaches for fault tolerant distributed applications. All these approaches require stable storage.

nipulate the concerns in isolation. In particular, our translation and replication approaches were resulted from exploring how to lower the costs of Byzantine fault tolerance (BFT):

- our translation approach lowers developmental costs of BFT applications by leveraging the existing fault tolerance capability of crash-tolerant applications and only implementing the missing notary service;
- our replication approach lowers the operational costs of BFT applications by eliminating the surrogates in BFT applications and reduce the number of replicas to $t + 1$ for t Byzantine faults.

Below we discuss related work on the two approaches.

2.2 Translation Techniques

The idea of automatically translating crash-tolerant systems into BFT systems can be traced back to the mid-eighties. Gabriel Bracha presents a translation mechanism that transforms a protocol tolerant of up to t crash failures into one that tolerates t Byzantine failures [46]. Brian Coan also presents a translation [64] that is similar to Bracha's. These approaches have two important

restrictions. One is that input protocols are required to have a specific style of execution, and in particular they have to be round-based with each participant awaiting the receipt of $n - t$ messages before starting a new round. Second, the approaches have quadratic message overheads and as a result do not scale well. Note that these approaches were primarily intended for a certain class of consensus protocols, while we are pursuing arbitrary protocols and distributed systems.

Toueg, Neiger and Bazzi worked on an extension of Bracha's and Coan's approaches for translation of synchronous systems [40, 41, 111]. Mpoeleng et al. [109] present a scalable translation that is also intended for synchronous systems, and transforms Byzantine faults to so-called *signal-on-failure* faults. They replace each host with a pair, and assume only one of the hosts in each pair may fail. In the Internet, making synchrony assumptions is dangerous. Byzantine hosts can easily trigger violations of such assumptions to attack the system.

Our translation approach makes fewer assumptions about the distributed applications than previous ones and can be applied to any deterministic distributed applications. In particular, our translation does not require the distributed applications to have any form. Furthermore, it is not based on time bounds and consensus to be correct and, hence, applicable to asynchronous distributed applications. Yet for each application message the translation terminates in a constant number of communication steps. Hence when synchrony is present, it is time bounded and applicable to synchronous distributed applications as well. Finally, the scalable version of our translation only incurs linear message overhead and scale well with the size of a distributed application.

2.3 Cost Reduction Approaches

In practice, Byzantine replication is used exclusively in synchronous environments [97], although there has been a large body of research on Byzantine replication in asynchronous or partially synchronous environments as well. One of the first systems to consider asynchronous Byzantine replication for practical deployment is PBFT [53], which demonstrates the practicality of a BFT version of NFS [113]. PBFT exploits HMACs for increased performance, but the cost in terms of hardware is high, requiring $3t + 1$ hosts to tolerate t Byzantine failures. Various approaches have since tried to lower this cost.

A notable series of optimization follows the principle of separation of concerns. [134] separates agreement from execution and observes that it is not necessary to have $3t + 1$ full replicas. The authors suggest a model with $2t + 1$ *execution nodes* (similar to replicas) and $3t + 1$ *agreement nodes* (inexpensive machines, similar to witnesses), where an agreement node can co-locate with an execution node—resulting in a total cost of $2t + 1$ full replicas and t inexpensive machines. UpRight Cluster Services [62], on the other hand, distinguishes malicious failures from crashes and assumes that crashes occur more often while it is cheaper to tolerate. The authors apply a hybrid model [96] to practical settings. The model requires $2u + r$ replicas (or $u + r + 1$ replicas and u witnesses) to tolerate u failures, among which at most r are malicious. When malicious failures are rarer than crashes ($r \leq u$), the hybrid model results in a lower replication cost.

ZZ [132] further optimizes [134] by noticing that replicas can be turned off and treated as "slow" in normal runs. To tolerate t Byzantine failures, a ZZ

system employs $t + 1$ execution nodes (replicas) and $3t + 1$ agreement nodes in normal runs. When failures occur, ZZ wakes up the other replicas, which are on hibernated virtual machines. ZZ demonstrates that on-demand state transfer makes the approach practical.

It is also possible to weaken consistency guarantees to lower costs. For example, [102] allows $2t$ out of $3t + 1$ hosts to fail, but the system does not guarantee linearizable behavior [85] in that case. The behavior is linearizable if and only if at most t hosts fail.

Yet another approach to lower cost is to add self-checking to crash-tolerant systems, in order to achieve fault tolerance to a greater class of failures than just crashes. For example, in Paxos Made Live [56] the authors use checksums to verify the integrity of a disk-based log in their Chubby [50] implementation. This implementation uses $2t + 1$ replicas for t failures.

Our affordable replication approach, called *Shuttle*, reduces the replication cost by relying on an external configuration service for liveness when failures occur. The Shuttle protocol uses only $t + 1$ replicas and t witnesses to make progress when failures do not occur while preserving safety against up to t Byzantine failures. However, when it is possible to make a stronger assumption about the adversary, we can eliminate witnesses and use $t+1$ replicas alone to preserve safety. With the stronger assumption, we can use a cheaper cryptographic primitive and reduce the computational cost of the protocol.

Even though Shuttle shares some mechanisms with Byzantine checkpointing [34], such as using $t + 1$ replicas when there are no failures and relying on an external service when failures occur, they are fundamentally different. The

Byzantine checkpointing approach aims for scientific computation, where it is acceptable to roll back a computation. Shuttle never rolls back. It aims for distributed applications running on the cloud environment.

CHAPTER 3

STRENGTHENING CRASH-TOLERANT DISTRIBUTED APPLICATIONS

Distributed applications have to deal with failures, because failures are common when a distributed application runs on multiple machines. Dealing with failures makes developing distributed applications difficult, especially when the environment is asynchronous and failed machines may exhibit arbitrary behavior. Yet, this is a problem that many software developers face today. While we know how to build replicated data stores that tolerate Byzantine behavior (*e.g.*, [53]), most applications go well beyond providing a data store. Tools like Byzantine consensus may help developing such applications, but most software developers find BFT is extremely challenging. Instead, they often make simplifying assumptions like a crash failure model, relying on careful monitoring to detect and fix problems that occur when such assumptions are violated.

Because most distributed applications already tolerate crash failures, can we leverage their existing fault tolerance capability to simplify the developing of Byzantine fault tolerance? This question intrigued us to study techniques that automatically transform crash-tolerant distributed applications into BFT ones that do not require careful monitoring and repair.

We develop such a technique in this chapter and make the following contributions. First we present a novel ordered broadcast protocol that we will use as a building block. The protocol is an extension of the Srikanth and Toueg authenticated broadcast protocol often used in Byzantine consensus protocols [124], adding consistent ordering for messages from the same sender even in the face of Byzantine behavior. Second, we present a new way of translating a distributed application that is tolerant of crash failures into one that tolerates

the same number of Byzantine failures, while imposing fewer restrictions on how the application is constructed than previous approaches. Third, we show how a version of the Castro and Liskov Practical Byzantine Replication protocol [53] can be derived from the Oki and Liskov Viewstamped Replication protocol [115] using our translation technique, something not possible with previous approaches.

We describe a system model in Section 3.1 and introduce three mechanisms used for translation: Authenticated Reliable broadcast (Section 3.2), Ordered Authenticast Reliable broadcast (Section 3.3), and the translation mechanism itself (Section 3.4). In Section 3.5 we demonstrate the translation mechanism.

3.1 System Model

In order to be precise we present a simple model to talk about machines, processes, and networks. The model consists of *agents* and *links*. An agent is an active entity that maintains state, receives messages on incoming links, performs some processing based on this input and its state, possibly updating its state and producing output messages on outgoing links.

Links are abstract unidirectional FIFO channels between two agents. Agents can interact across links only. In particular, an agent can *enqueue* a message on one of its outgoing links, and it can *dequeue* messages from one of its incoming links (assuming a message is available there).

We use agents and links to model various activities and interactions. Processes that run on hosts are agents, but the network is also an agent—one that forwards messages from its incoming links to its outgoing links according to

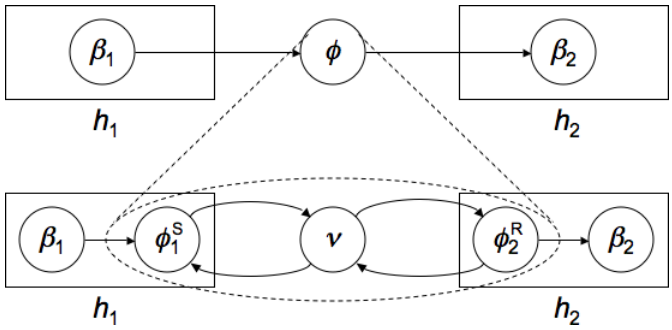


Figure 3.1: An agent model and a refinement.

some policy. Agents are named by lower-case Greek letters $\alpha, \beta, ...$ For agents that are processes, we will use subscripts on names to denote which hosts they run on. For example, β_i is an agent that runs on host h_i.

Hosts are containers for agents, and they are also the unit of failure. Hosts are either *honest*, executing programs as specified, or *Byzantine* [97], exhibiting arbitrary behavior. We also use the terms *correct* and *faulty*, but not as alternatives to honest and Byzantine. A correct host is honest and always eventually makes progress when it is given an input. A faulty host is a Byzantine host or an honest host that has crashed or will eventually crash. Honest and Byzantine are mutually exclusive, as are correct and faulty. However, a host can be both honest and faulty.

We do not assume timing bounds on execution of agents. Latency in the network is modeled as execution delay in a network agent. Note that this prevents hosts from accurately detecting crashes of other hosts.

Figure 3.1 depicts an example of an agent model and a *refinement*. Agents are represented by circles, links by arrows, and hosts by rectangles. The top half models two application agents β_1 and β_2 running on two hosts h_1 and h_2

communicating using a FIFO network agent ϕ. The bottom half refines the FIFO network using an unreliable network agent ν and two protocol agents $\phi_1^{\mathbf{S}}$ and $\phi_2^{\mathbf{R}}$ that implement ordering and retransmission using sequence numbers, timers, and acknowledgment messages. This kind of refinement will be a common theme throughout this chapter.

3.2 The ARcast Mechanism

The first mechanism we present is Authenticated Reliable broadcast (ARcast). This broadcast mechanism was suggested by Srikanth and Toueg, and they present an implementation that does not require public-key digital signatures in [124]. Their implementation requires $n > 3t$. As shown below, it is also possible to develop an implementation that uses public-key digital signatures, in which case n only has to be larger than $2t$.

3.2.1 ARcast Definition

Assume $\beta_i, ...$ are agents communicating using ARcast on hosts $h_i, ...$ Then ARcast provides the following properties:

1. *bc-Persistence.* If two hosts h_i and h_j are correct, and β_i sends a message m, then β_j delivers m from β_i;
2. *bc-Relay.* If h_i is honest and h_j is correct, and β_i delivers m from β_k, then β_j delivers m from β_k (host h_k is not necessarily correct);

3. *bc-Authenticity.* If two hosts h_i and h_j are honest and β_i does not send m, then β_j does not deliver m from β_i.

Informally, ARcast ensures that a message is reliably delivered to all correct receivers in case the sender is correct (bc-Persistence) or in case another honest receiver has delivered the message already (bc-Relay). Moreover, a Byzantine host cannot forge messages from an honest host (bc-Authenticity).

3.2.2 ARcast Implementation

We assume there is a single sender β_i on h_i. We model ARcast as a network agent ξ_i, which we refine by replacing it with the following agents (see Figure 3.2):

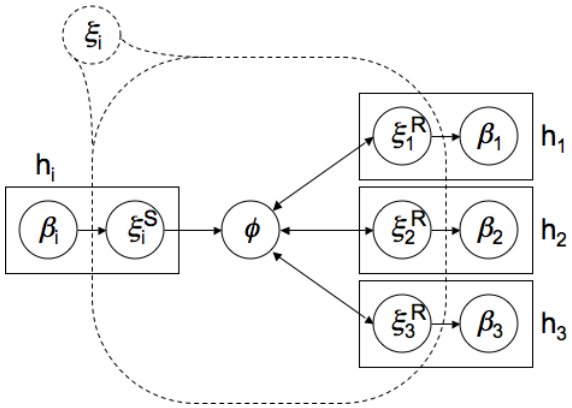


Figure 3.2: Architecture of the ARcast implementation if the sender is on host h_i.

ξ_i^{S} *sender* agent that is in charge of the sending side of the ARcast mechanism;

ξ_*^{R} *receiver* agents that are in charge of the receive side;

ϕ *FIFO network* agent that provides point-to-point authenticated FIFO communication between agents.

The mechanism has to be instantiated for each sender. The sending host h_i runs the ARcast sender agent $\xi_i^{\mathbf{S}}$. Each receiving host h_j runs a *receiver* agent $\xi_j^{\mathbf{R}}$. There have to be at least $2t+1$ receiving hosts, one of which may be h_i. When $\xi_i^{\mathbf{S}}$ wants to ARcast a message m, it sends $\langle \mathsf{echo}\ m, i \rangle_i$, signed by h_i using its public key signature, to all receivers. A receiver that receives such an `echo` message for the first time forwards it to all receivers. On receipt of $t+1$ of these correctly signed echoes for the same m from different receivers (it can count an echo from itself), a receiver delivers m from i.

3.2.3 ARcast Correctness

Theorem 3.2.1. *ARcast satisfies bc-Persistence.*

Proof. Assume h_i and h_j are correct, and say sender $\xi_i^{\mathbf{S}}$ sends m_i. Because h_i is correct, all correct receivers receive $\langle \mathsf{echo}\ m, i \rangle_i$ from $\xi_i^{\mathbf{S}}$. Because there are at least $2t+1$ receivers, at least $t+1$ of the receivers must be correct. As all these receivers send an echo message to $\xi_j^{\mathbf{R}}$, $\xi_j^{\mathbf{R}}$ will receive at least $t+1$ copies and deliver m from $\xi_i^{\mathbf{R}}$. □

Theorem 3.2.2. *ARcast satisfies bc-Authenticity.*

Proof. Enforced using the digital signature on the `echo` message sent by the honest sender. □

Theorem 3.2.3. *ARcast satisfies bc-Relay.*

Proof. Say h_j is honest and receiver $\xi_j^{\mathbf{R}}$ delivers m from $\xi_i^{\mathbf{S}}$. Because $\xi_j^{\mathbf{R}}$ awaited $t+1$ echoes for m, at least one of the echoes must have come from a correct receiver.

As this correct receiver sends a copy to all receivers, all correct receivers obtain a copy and will forward the message if they have not done so already. There are at least $t + 1$ correct receivers, and therefore each correct receiver eventually receives at least $t + 1$ echoes, and delivers the message. □

3.3 The OARcast Mechanism

ARcast does not provide any ordering. Even messages from a correct sender may be delivered in different orders at different receivers. Next we introduce a broadcast mechanism that is like ARcast, but adds delivery order for messages sent by either honest or Byzantine hosts.

3.3.1 OARcast Definition

OARcast provides, in addition to the ARcast properties, the following:

4. *bc-FIFO.* If two hosts h_i and h_j are honest and β_i sends m_1 before m_2, and β_j delivers m_1 and m_2 from β_i, then β_j delivers m_1 before m_2;
5. *bc-Ordering.* If two hosts h_i and h_j are honest and β_i and β_j both deliver m_1 from β_k and m_2 from β_k, then they do so in the same order (even if h_k is Byzantine).

As a result of bc-Ordering, even a Byzantine sender cannot cause two honest receivers to deliver OARcast messages from the same source out of order. bc-FIFO ensures that messages from honest hosts are delivered in the order sent.

OARcast does not guarantee any order among messages from different sources, and is thus weaker than consensus.

3.3.2 OARcast Implementation

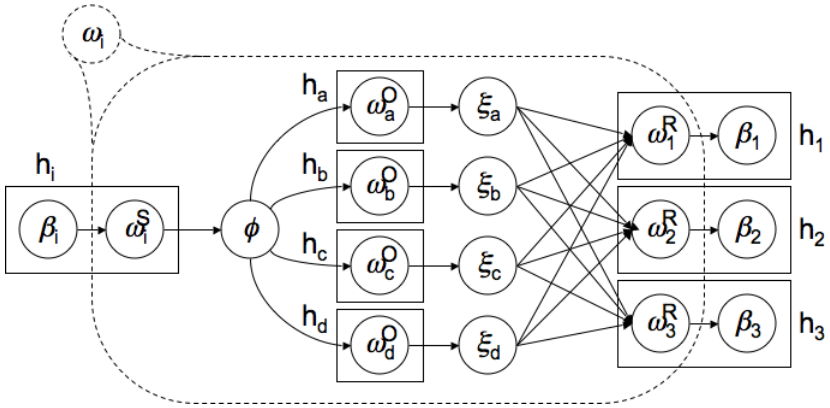


Figure 3.3: Architecture of the OARcast implementation if the sender is on host h_i.

We describe how OARcast may be implemented using ARcast. Again, we show the implementation for a single sender β_i on host h_i. With multiple senders, the implementation has to be instantiated for each sender separately. We refine the OARcast network agent ω_i by replacing it with the following agents (see Figure 3.3):

ω_i^{S} *sender* agent that is in charge of the sending side of the OARcast mechanism;

ω_*^{O} *orderer* agents that are in charge of ordering;

ω_*^{R} *receiver* agents that are in charge of the receive side;

ϕ *FIFO network* agent that provides point-to-point authenticated FIFO communication from the sender agent to each orderer agent;

ξ_* *ARcast network* agents each provides ARcast from a particular orderer agent to all receiver agents.

We need to run $3t + 1$ orderers on separate hosts, of which no more than t may fail. A host may end up running a sender, a receiver, as well as an orderer. A receiver $\omega_j^{\mathbf{R}}$ maintains a sequence number c_j, initially 0. An orderer $\omega_k^{\mathbf{O}}$ also maintains a sequence number, t_k, initially 0.

To OARcast a message m, $\omega_i^{\mathbf{S}}$ sends m to each orderer via ϕ. When an orderer $\omega_k^{\mathbf{O}}$ receives m from $\omega_i^{\mathbf{S}}$, it ARcasts $\langle \mathsf{order}\ m, t_k, i \rangle$ to each of the receivers, and increments t_k. A receiver $\omega_j^{\mathbf{R}}$ awaits $2t+1$ messages $\langle \mathsf{order}\ m, c_j, i \rangle$ from different orderers before delivering m from $\omega_i^{\mathbf{S}}$. After doing so, the receiver increments c_j.

3.3.3 OARcast Correctness

Lemma 3.3.1. *Say h_i and h_j are honest and m is the c^{th} message that $\omega_j^{\mathbf{R}}$ delivers from $\omega_i^{\mathbf{S}}$, then m is the c^{th} message that $\omega_i^{\mathbf{S}}$ sent.*

Proof. Say m is not the c^{th} message sent by $\omega_i^{\mathbf{S}}$, but it is the c^{th} message delivered by $\omega_j^{\mathbf{R}}$. $\omega_j^{\mathbf{R}}$ must have received $2t + 1$ messages of the form $\langle \mathsf{order}\ m, c-1, i \rangle$ from different orderers. Because only t hosts may fail, and because of bc-Authenticity of ARcast, at least one of the order messages comes from a correct orderer. Because communication between $\omega_i^{\mathbf{S}}$ and this orderer is FIFO, and because the sender does not send m as its c^{th} message, it is not possible that the orderer sent $\langle \mathsf{order}\ m, c-1, i \rangle$. □

Lemma 3.3.2. *Say m is the c^{th} message that a correct sender $\omega_i^{\mathbf{S}}$ sends. Then all correct*

receivers receive at least $2t + 1$ *messages of the form* $\langle \mathsf{order}\ m, c-1, i\rangle$ *from different orderers.*

Proof. Because the sender is correct, each of the correct orderers will deliver m. As all links are FIFO and m is the c^{th} message, it is clear that for each orderer $\omega_k^{\mathbf{o}}$, $t_k = c - 1$. Each correct orderer $\omega_k^{\mathbf{o}}$ therefore sends $\langle \mathsf{order}\ m, c-1, i\rangle$ to all receivers. Because at least $2t+1$ of the orderers are correct, and because of ARcast's bc-Persistence, each correct receiver receives $2t + 1$ such order messages. □

Theorem 3.3.3. *OARcast satisfies bc-Persistence.*

Proof. Assume the sending host, h_i, is correct, and consider a correct receiving host h_j. The proof proceeds by induction on c, the number of messages sent by $\omega_i^{\mathbf{s}}$. Consider the first message m sent by $\omega_i^{\mathbf{s}}$. By Lemma 3.3.2, $\omega_j^{\mathbf{R}}$ receives $2t + 1$ messages of the form $\langle \mathsf{order}\ m, 0, i\rangle$. By Lemma 3.3.1 it is not possible that the first message that $\omega_j^{\mathbf{R}}$ delivers is a message other than m. Therefore, $c_j = 0$ when $\omega_j^{\mathbf{R}}$ receives the order messages for m and will deliver m.

Now assume that bc-Persistence holds for the first c messages from $\omega_i^{\mathbf{s}}$. We show that bc-Persistence holds for the $(c + 1)^{st}$ message sent by $\omega_i^{\mathbf{s}}$. By Lemma 3.3.2, $\omega_j^{\mathbf{R}}$ receives $2t + 1$ messages of the form $\langle \mathsf{order}\ m, c, i\rangle$. By the induction hypothesis, $\omega_j^{\mathbf{R}}$ will increment c_j at least up to c. By Lemma 3.3.1 it is not possible that the c^{th} message that $\omega_j^{\mathbf{R}}$ delivers is a message other than m. Therefore, $c_j = c$ when $\omega_j^{\mathbf{R}}$ receives the order messages for m and will deliver m. □

Theorem 3.3.4. *OARcast satisfies bc-Authenticity.*

Proof. This is a straightforward corollary of Lemma 3.3.1. □

Theorem 3.3.5. *OARcast satisfies bc-Relay.*

Proof. By induction on the sequence number. Say some correct receiver $\omega_j^{\mathbf{R}}$ delivers the first message m from $\omega_i^{\mathbf{S}}$. By the OARcast protocol, $\omega_j^{\mathbf{R}}$ must have received $2t+1$ messages of the form $\langle \mathsf{order}\ m, 0, i\rangle$ from different orderers when $c_j = 0$. Because of the bc-Relay property of ARcast, all correct receivers receive the same order messages from the orderers. By Lemma 3.3.1 it is not possible that a correct receiver $\omega_{j'}^{\mathbf{R}}$ delivered a message other than m, and therefore $c_{j'} = 0$ when $\omega_{j'}^{\mathbf{R}}$ receives the order messages. Thus $\omega_{j'}^{\mathbf{R}}$ will also deliver m.

Now assume the theorem holds for the first c messages sent by $\omega_i^{\mathbf{S}}$. Say some correct receiver $\omega_j^{\mathbf{R}}$ delivers the $(c+1)^{st}$ message m from $\omega_i^{\mathbf{S}}$. By the OARcast protocol, $\omega_j^{\mathbf{R}}$ must have received $2t+1$ messages of the form $\langle \mathsf{order}\ m, c, i\rangle$ from different orderers when $c_j = c$. Because of the bc-Relay property of ARcast, all correct receivers receive the same order messages from the orderers. Because of the induction hypothesis, the correct receivers deliver the first c messages. By Lemma 3.3.1 it is not possible that a correct receiver $\omega_{j'}^{\mathbf{R}}$ delivered a message other than m, and therefore $c_j = c$ when $\omega_{j'}^{\mathbf{R}}$ receives the order messages containing m. Thus $\omega_{j'}^{\mathbf{R}}$ will also deliver m. □

Lemma 3.3.6. *Say m is the c^{th} message that an honest receiver $\omega_j^{\mathbf{R}}$ delivers from $\omega_i^{\mathbf{S}}$, and m' is the c^{th} message that another honest receiver $\omega_{j'}^{\mathbf{R}}$ delivers from $\omega_i^{\mathbf{S}}$. Then $m = m'$ (even if h_i is Byzantine).*

Proof. Say not. $\omega_j^{\mathbf{R}}$ must have received $2t+1$ messages of the form $\langle \mathsf{order}\ m, c-1, i\rangle$[1] from different orderers, while $\omega_{j'}^{\mathbf{R}}$ must have received $2t+1$ messages of the form $\langle \mathsf{order}\ m', c-1, i\rangle$ from different orderers. As there are only $3t+1$ orderers,

[1]Note that the counters start from 0. $\langle \mathsf{order}\ m, c-1, i\rangle$ means m is the c^{th} message being ordered.

at least one correct orderer must have sent one of each, which is impossible as correct orderers increment their sequence numbers for each new message. □

Theorem 3.3.7. *OARcast satisfies bc-Ordering.*

Proof. Corollary of Lemma 3.3.6. □

Theorem 3.3.8. *OARcast satisfies bc-FIFO.*

Proof. Evident from the FIFOness of messages from senders to orderers and the sequence numbers utilized by orderers and receivers. □

3.4 The Translation Mechanism

In this section, we describe how an *arbitrary* protocol tolerant only of crash failures can be translated into a protocol that tolerates Byzantine failures.

3.4.1 Definition

Below we use the terms *original* and *translated* to distinguish the system before and after translation, respectively. The original system tolerates only crash failures, while the translated system tolerates Byzantine failures as well. The original system consists of n hosts, each of which runs an *actor agent*, $\alpha_1, \ldots, \alpha_n$. Each actor α_i is a state machine that maintains a running state s^i, initially s^i_0, and, upon receiving an input message m, executes a deterministic *state transition function* F^i: $(\overline{m_o}, s^i_{c+1}) := F^i(m, s^i_c)$ where

- c indicates the number of messages that α_i has processed so far;
- s_c^i is the state of α_i before processing m;
- s_{c+1}^i is the next state of s_c^i as a result of processing m (called $F^i(m, s_c^i).\texttt{next}$);
- $\overline{m_o}$ is a finite, possibly empty set of output messages (called $F^i(m, s_c^i).\texttt{output}$).

The state transition functions process one input message at a time and may have no computational time bound.

Actors in the original system communicate via a FIFO network agent ϕ. Each actor maintains a pair of input-output links with the FIFO network agent. When an actor α_i wants to send a message m to another actor α_j (may be itself), α_i formats m (detailed below) and enqueues it on α_i's output link. We call this action α_i *sends* m *to* α_j. ϕ dequeues m from the link and places it into the message buffer that ϕ maintains. Eventually ϕ removes m from its buffer and enqueues m on the input link of α_j. When α_j dequeues m we say that α_j *delivers* m *from* α_i. The original system assumes the following of the network:

1. α-*Persistence.* If two hosts h_i and h_j are correct and α_i sends m to α_j, then α_j delivers m from α_i.
2. α-*Authenticity.* If two hosts h_i and h_j are honest and α_i does not send m to α_j, then α_j does not deliver m from α_i.
3. α-*FIFO.* If two hosts h_i and h_j are honest and α_i sends m_1 before m_2, and α_j delivers m_1 and m_2 from α_i, then α_j delivers m_1 before m_2;

Note that in the original system all hosts are honest. However, for the translation we need to be able to generalize these properties to include Byzantine hosts.

Messages in the original system are categorized as internal or external. *Internal messages* are sent between actors and are formatted as $\langle d, i, j \rangle$, where d is the data (or payload), i indicates the source actor, and j indicates the destination actor. *External messages* are from clients to actors and are formatted as $\langle d, \perp, j \rangle$, similar to the format of internal messages except the source actor is empty ($\perp$). Internal and external messages are in general called α-*messages*, or simply *messages* when the context is clear.

In the original system all actors produce output messages by making transitions based on input as specified by the protocol. We call such output messages *valid*. We formalize validity below.

External messages are assumed to be valid. For example, we may require that clients sign messages. An internal message m sent by actor α_i is valid if and only if there exists a sequence of valid messages $m_1^i, \ldots, m_c^i$ delivered by α_i such that

$$m \in F^i(m_c^i, F^i(m_{c-1}^i, F^i(\ldots, F^i(m_1^i, s_0^i).\texttt{next}\ldots).\texttt{next}).\texttt{next}).\texttt{output}$$

The expression means that actor α_i sends m after it has processed the first c input messages, be they internal or external. Note that external input forms the base case for this recursive definition, as actors produce no internal messages until at least one delivers an external message.[2]

[2]We model periodic processing not based on input by external *timer* messages.

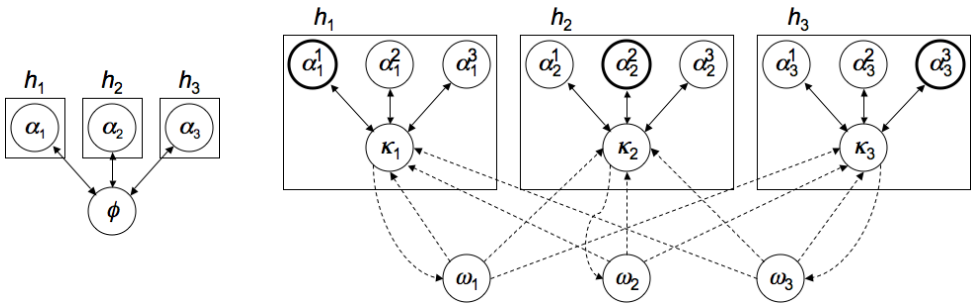


Figure 3.4: Translation: the original system (left) is simulated at each host in the translated system (right). Dark circles are master actors. Dashed lines represent OARcast communication.

In order for the original system to work correctly, each actor needs to make transitions based on valid input. More formally, the system needs to preserve the following property:

4. α-*Validity.* If h_i is honest and α_i delivers m from α_j, then m is valid.

The property is granted to the original system by default, because it is in an environment where faulty hosts follow the protocol faithfully until they crash.

Besides the four α–properties, the original system requires no other assumptions about communication among actors. However, the original system may require non-communication assumptions such as "up to t hosts can fail."

The Translation mechanism transforms a crash-tolerant system in which all hosts require the four α-properties into a Byzantine-tolerant system that preserves the α-properties.

3.4.2 Implementation

In the original system, each actor α_i runs on a separate host h_i. In the translated system each host simulates the entire original system (see Fig. 3.4). That is, a host runs a replica of each of the n actors and passes messages between the actors internally using a simulated network agent, called *coordinator*, that runs on the host. We denote the coordinator running on host h_i as κ_i.

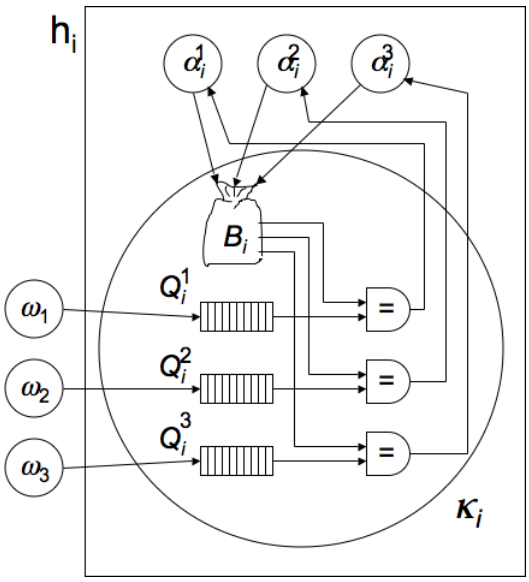


Figure 3.5: Anatomy of host h_i in the translated system.

To ensure that the different hosts stay synchronized, the coordinators agree on the order in which messages are delivered to replicas of the same actor. The replica of α_i on host h_j is called α_j^i. We designate α_i^i as the *master* replica and α_j^i $(i \neq j)$ as *slave* replicas. On honest hosts, the replicas of each actor start in the same initial state.

Each coordinator replaces ϕ of the original system by OARcast, i.e., OARcast is used to send messages. OARcast guarantees that coordinators agree on the

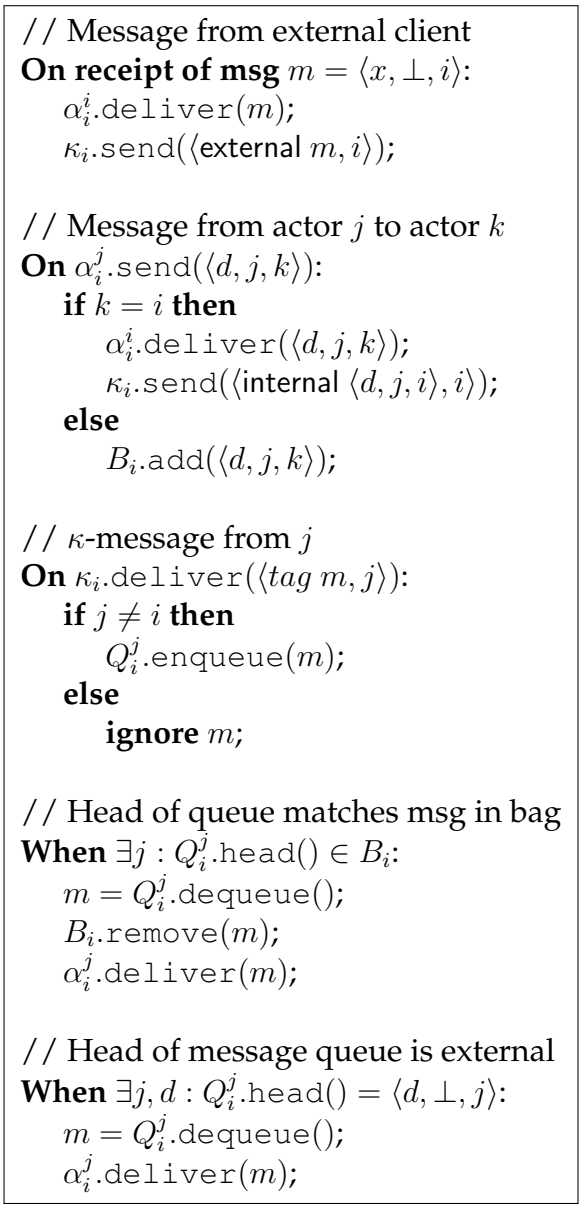


Figure 3.6: Pseudo-code of the Translation Mechanism for coordinator κ_i. When an actor delivers a message m, it also processes m and produces outputs atomically.

delivery of messages to replicas of a particular actor. Coordinators wrap each α-message in a *κ-message*. κ-messages have the form $\langle tag\ m, i\rangle$, where tag is either internal or external, m is an α-message, and i indicates the destination actor.

Each coordinator maintains an unordered *message bag* and n per-actor-replica *message queues*. By B_i we denote the message bag at host i and by Q_i^j we denote the message queue for actor α_i^j at host i (see Figure 3.5). The pseudo-code for a coordinator κ_i appears in Figure 3.6. κ_i intercepts messages from local actors, and it receives messages from remote coordinators. κ_i places α-messages sent by local actor replicas in B_i, and places α-messages received within κ-messages from κ_j in Q_i^j. When there is a match between a message m in the bag and the head of a queue, the coordinator enqueues m for the corresponding actor.

The translated system guarantees α-Persistence, α-Authenticity, α-FIFO, and α-Validity to all master actors on honest hosts. The next subsection contains a proof of correctness.

3.4.3 Correctness

We prove correctness of the Translation mechanism assuming the bc-properties. In particular, we show that the collection of coordinators and slave replicas that use the Translation mechanism preserves the α-properties: α-*Persistence*, α-*Authenticity*, α-*FIFO*, and α-*Validity*, for the master replicas $\{\alpha_i^i\}$.

For convenience, we combine bc-Relay and bc-Ordering to state that coordinators on correct hosts deliver the same sequence of κ-messages from any κ_k, even if h_k is Byzantine. This is put more formally in the following lemma:

Lemma 3.4.1. *For any i, j, and k, if h_i and h_j are correct, then κ_i and κ_j deliver the same sequence of messages from κ_k.*

Proof. bc-Relay guarantees that κ_i and κ_j deliver the same set of messages from

κ_k. bc-Ordering further guarantees that the delivery order between any two messages is the same at both κ_i and κ_j. □

In the proof we need to be able to compare states of hosts. We represent the state of host h_i by a vector of counters, $\Phi_i = (c_i^1, \ldots, c_i^n)$, where each c_i^k is the number of messages that (the local) actor α_i^k has delivered. As shown below, within an execution of the protocol, replicas of the same actor deliver the same sequence of messages. Thus from c_i^k and c_j^k we can compare progress of replicas of α_k on hosts h_i and h_j.

Lemma 3.4.2. *Given are that hosts h_i and h_j are correct, α_i^k delivers $m_1, \ldots, m_c$, and α_j^k delivers $c' \leq c$ messages. Then the messages that α_j^k delivers are $m_1, \ldots, m_{c'}$.*

Proof. By the Translation mechanism, the first c' messages that α_i^k and α_j^k deliver are the contents of the first c' κ-messages that κ_i and κ_j delivered from κ_k, resp. By Lemma 3.4.1, the two κ-message sequences are identical. This and the fact that links from coordinators to actors are FIFO imply that the first c' messages that α_i^k and α_j^k deliver are identical. □

In the remaining proof we use the following definitions and notations:

- *h_i reaches* $\Phi = (c_1, \ldots, c_n)$, denoted $h_i \rightsquigarrow \Phi$, if $\forall_j\ c_i^j \geq c_j$;
- $\Phi = (c_1, \ldots, c_n)$ *precedes* $\Phi' = (c'_1, \ldots, c'_n)$, denoted $\Phi < \Phi'$, if $(\forall_i\ c_i \leq c'_i) \wedge (\exists_j\ c_j < c'_j)$;
- $\Phi = (c_1, \ldots, c_n)$ *produces* m if $m \in \bigcup_{i=1}^{n} \bigcup_{c=1}^{c_i} (F^i(m_c^i, s_{c-1}^i).\texttt{output})$, where m_c^i is the c^{th} message to α_i and s_{c-1}^i is the state of α_i after it processes the first $c-1$ input messages.

Corollary 3.4.3. *If Φ produces m on a correct host, Φ produces m on all correct hosts that reach Φ.*

Proof. By Lemma 3.4.2 and because replicas of the same actor start in the same state and are deterministic, if Φ produces m on a correct host, Φ produces m on all correct hosts that reach Φ. □

We now show that if a correct host *is* in a particular state then all other correct hosts will *reach* this state.

Lemma 3.4.4. *If there is a correct host h_i in state Φ, then, eventually, all correct hosts reach Φ.*

Proof. By induction on Φ. All correct hosts start in state $\Phi^0 = (0, \ldots, 0)$, and $\forall \Phi \neq \Phi^0 : \Phi^0 < \Phi$.

Base case: All correct hosts reach Φ^0 by definition.

Inductive case: Say that correct host h_i is in state $\Phi = (c_1, \ldots, c_n)$, and the lemma holds for all $\Phi' < \Phi$ (Induction Hypothesis). We need to show that any correct host h_j reaches Φ.

Consider the last message m that some actor replica α_i^p delivered. Thus, m is the c_p^{th} message that α_i^p delivered. The state of h_i prior to delivering this message is $\Phi' = (c_1, \ldots, c_p - 1, \ldots, c_n)$. It is clear that $\Phi' < \Phi$. By the induction hypothesis $h_j \rightsquigarrow \Phi'$.

By the Translation mechanism we know that $\langle tag\ m, p \rangle$ (for some tag) is the c_p^{th} κ-message that κ_i delivers from κ_p. Lemma 3.4.1 implies that $\langle tag\ m, p \rangle$ must also be the c_p^{th} κ-message that κ_j delivers from κ_p. Since $h_j \rightsquigarrow \Phi'$, α_j^p delivers the

first $c_p - 1$ α-messages, and thus κ_j must have removed those messages from Q_j^p. Consequently, m gets to the head of Q_j^p. (1)

Now there are two cases to consider. If m is external, then κ_j will directly remove m from Q_j^p and enqueue m on the link to α_j^p. Because α_i^p delivered m after delivering the first $c_p - 1$ messages (from the Induction Hypothesis and the definition of m), and α_i^p and α_j^p run the same function F^p, α_j^p will eventually deliver m as well, and therefore $h_j \rightsquigarrow \Phi$.

Consider the case where m is internal. By definition, $\Phi' = (c_1, \ldots, c_p - 1, \ldots, c_n)$ *produces* m at host h_i. By Corollary 3.4.3, Φ' produces m at host h_j. Thus, eventually κ_j places the message in the message bag B_j. (2)

(1) and (2) provide the matching condition for κ_j to enqueue m on its link to α_j^p. Using the same reasoning for the external message case, $h_j \rightsquigarrow \Phi$. □

We can now show the communication properties.

Theorem 3.4.5. *(α-Persistence.) If two hosts h_i and h_j are correct and α_i^i sends m to α_j, then α_j^j delivers m from α_i.*

Proof. Suppose h_i is in state Φ_i when α_i^i sends m to α_j. By Lemma 3.4.4, $h_j \rightsquigarrow \Phi_i$. Thus, α_j^i sends m to α_j as well. By the Translation mechanism, κ_j places m in B_j and OARcasts $\langle \mathsf{internal}\ m, j \rangle$. By bc-Persistence, κ_j delivers $\langle \mathsf{internal}\ m, j \rangle$ (from itself) and places m on its queue Q_j^j. (1)

By the Translation Mechanism, each external message at the head of Q_j^j is dequeued and delivered by α_j^j. (2)

Let us consider an internal message m' at the head of Q_j^j. Since h_j is correct,

the Translation mechanism ensures that κ_j has delivered $\langle \mathsf{internal}\ m', j\rangle$ (the κ-message containing m' and from κ_j). bc-Authenticity ensures that κ_j has indeed sent the κ-message. By the Translation mechanism, κ_j always puts a copy of m' in B_j before sending $\langle \mathsf{internal}\ m', j\rangle$. Thus, m' in Q_j^j matches a copy in B_j, and α_j^j delivers m'. This together with (2) show that α_j^j delivers all internal messages in Q_j^j. (3)

(1) shows that m sent by α_i^i arrives in Q_j^j, and (3) shows that α_j^j delivers all internal messages in Q_j^j. Together they show that α_j^j delivers m from α_i. □

Theorem 3.4.6. *(α-Authenticity.) If two hosts h_i and h_j are honest and α_i^i does not send m to α_j, then α_j^j does not deliver m from α_i.*

Proof. Assume α_j^j delivers m from α_i, but α_i^i did not send m to α_j. By the Translation mechanism, α_j^j only delivers m when α_j^i sends m. There are two cases.

If $i = j$, we get a contradiction to the assumption that α_i^i did not send m.

If $i \neq j$, note that actors do not generate outputs without inputs. There must exists a message m' that α_j^i delivers right before it sends m. By the Translation mechanism, α_j^i ($i \neq j$) delivers m' only if κ_j has delivered $\langle \mathsf{internal}\ m', i\rangle$. By bc-Authenticity of OARcast, κ_i must have OARcast $\langle \mathsf{internal}\ m', i\rangle$. Before sending $\langle \mathsf{internal}\ m', i\rangle$, κ_i must have delivered m' to α_i^i. And therefore α_i^i must have processed m' and generated outputs (including m) before κ_i sends $\langle \mathsf{internal}\ m', i\rangle$. In other words, α_i^i must have sent m, contradicting the assumption. □

Theorem 3.4.7. *(α-FIFO.) If two hosts h_i and h_j are honest, α_i^i sends m_1 before m_2, and α_j^j delivers m_1 and m_2 from α_i, then α_j^j delivers m_1 before m_2.*

Proof. Suppose that α_i^i sends m_1 and m_2 after it delivers m_1' and m_2', respectively.

Note that m'_1 and m'_2 can be the same. By the Translation mechanism, κ_i OAR-casts m'_1 before m'_2. By bc-FIFO, κ_j delivers m'_1 before m'_2. By the Translation mechanism, κ_j lets α^i_j deliver m'_1 before m'_2, as α^i_i does. As the actors are deterministic, α^i_j sends m_1 before m_2, similar to α^i_i. By the Translation mechanism, κ_j delivers m_1 and m_2 to α^j_j in that order, as desired. □

We introduce a lemma that helps us show α-*Validity*:

Lemma 3.4.8. *Actor replicas on honest hosts only send valid messages.*

Proof. Suppose not. Let m sent by α^i_j be the first invalid message sent by an actor replica on an honest host. Since h_j is honest, there must be a sequence of messages $m^i_1, \ldots, m^i_c$ that α^i_j delivered, such that

$$m \in F^i(m^i_c, F^i(m^i_{c-1}, F^i(\ldots, F^i(m^i_1, s^i_0).\texttt{next}\ldots).\texttt{next}).\texttt{next}).\texttt{output}$$

Since m is the first invalid message sent by an actor replica, all internal messages in the sequence $m^i_1, \ldots, m^i_c$ must be valid. Moreover, external messages are valid by definition. Thus, all messages $m^i_1, \ldots, m^i_c$ are valid. But then, m is valid by definition, contradicting the assumption. □

Theorem 3.4.9. *(α-Validity.) If h_i is honest and α^i_i delivers m from α_j, then m is valid (even if $j \neq \perp$ and h_j is faulty.)*

Proof. If m is an external message, then it is valid and unforgeable by definition.

If m is an internal message, the fact that α^i_i delivers m from α_j implies that α^j_i has sent m to α_i. By Lemma 3.4.8, m is valid. □

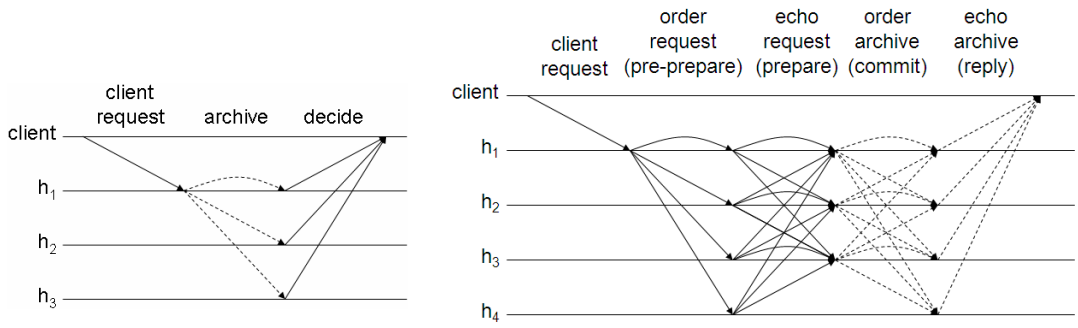
3.5 Illustration: PBFT

In 1999 Castro and Liskov published "Practical Byzantine Fault Tolerance," a paper about a Byzantine-tolerant replication protocol (PBFT) for an NFS file system [53]. The paper shows that PBFT is indeed practical, adding relatively little overhead to NFS. In this section we show, informally, that a protocol much like PBFT can be synthesized from the Viewstamped Replication protocol by Oki and Liskov [115] and the transformations in this chapter. The main difference is that our protocol is structured, while PBFT is largely monolithic. In our opinion, the structure simplifies understanding and enhances the ability to scrutinize the protocol. The PBFT paper addresses several practical issues and possible optimizations that can be applied to our scheme as well, but omitted for brevity.

Viewstamped Replication is a consensus protocol. A normal case execution is shown in Figure 3.7(a).[3] A client sends a request to a server that is elected primary. The primary server sends an `archive` message to each server in the system. If a quorum responds to the client, the request is completed successfully. In the case of failures, a possibly infinite number of rounds of this consensus protocol may be necessary to reach a decision.

If we were to apply translation literally as described, we would end up with a protocol that sends significantly more messages than PBFT. This is because our translation does nothing to group related information from a particular sender to a particular receiver in single messages. Instead, all pieces of information go out, concurrently, in separate small messages. While explicit optimizations could eliminate these, FIFO protocols like TCP automatically aggregate concurrent traffic between a pair of hosts into single messages for efficiency, obviating

[3] Slightly optimized by sending `decide` messages back to the client instead of the primary.



(a) before translation (b) after translation

Figure 3.7: A normal case run of (a) the original system and (b) the translated system. Dashed arrows indicate the `archive` message from the primary. Between brackets we indicate the corresponding PBFT message types.

the need for any explicit optimizations. Note that while these techniques reduce the number of messages, the messages become larger and the number of rounds remains the same.

Figure 3.7(b) shows a normal run of the translated Viewstamped protocol for $t = 1$.[4] The figure shows only the traffic that is causally prior to the reply received by the client and, thus, essential to the latency that the client perceives. In this particular translation we used t additional hosts for OARcast only, but a more faithful translation would have started with $3t + 1$ servers. Nevertheless, the run closely resembles that of a normal run of PBFT (see Figure 1 of [53]).

Viewstamped Replication solves *uniform consensus* [60]. Our translation maintains uniformity[5] because of bc-Relay, which requires that if an honest host delivers a message, then all correct hosts have to do the same. In particular,

[4]Slightly optimized by sending the echoed `archive` messages directly to the client. The client will be able to derive a set of `decide` messages from the echoed `archive` messages.

[5]In Byzantine environment, uniform consensus guarantees that if two honest servers decide on an update, they decide on the same update.

ARcast maintains the uniformity in the original protocol by having a receiver await $t + 1$ copies of a message before delivery. Doing so makes sure that one of the copies was sent by a correct receiver that forwards a copy to all other correct receivers as well. When uniformity is not necessary, a receiver can deliver the message as soon as the first copy is received. If the receiver is correct, it will forward the message to all other correct receivers, and (non-uniform) bcRelay is preserved. With this optimization, the `echo` traffic can be piggybacked on future traffic. Furthermore, the translated Viewstamped protocol with this optimization is similar to PBFT with tentative execution.

3.6 Discussion

We presented a mechanism to translate a distributed application that tolerates only crash failures into one that tolerates Byzantine failures. Few restrictions are placed on the application, and the approach is applicable not only to consensus but to a large class of distributed applications. The approach makes use of a novel broadcast protocol. We have illustrated how the approach may be used to derive a version of the Castro and Liskov Practical Byzantine Replication protocol, showing that our translation mechanism is pragmatic and more powerful than previous translation approaches.

CHAPTER 4

NYSIAD: STRENGTHENING LARGE-SCALE CRASH-TOLERANT DISTRIBUTED APPLICATIONS

Scalable distributed systems have to tolerate nondeterministic failures from causes such as Heisenbugs and Mandelbugs [79], aging related or bit errors (*e.g.*, [110]), selfish behavior (*e.g.*, freeloading), and intrusion. While all these failures are prevalent and it would seem that large distributed systems have sufficient redundancy and diversity to handle such failures, developing software that delivers scalable Byzantine fault tolerance has proved difficult, and few such systems have been built and deployed. Distributed systems and protocols like DNS [106, 107], BGP [103], OSPF [108], IS-IS [116], as well as most P2P communication systems tolerate only crash failures. Secure versions of such systems aim to prevent compromise of participants. While important, this issue is orthogonal to tolerating Byzantine failures as a host is not faulty until it is compromised.

We know how to build practical scalable Byzantine-tolerant data stores (*e.g.*, [33, 38, 130]). Various work has also focused on Byzantine-tolerant peer-to-peer protocols (*e.g.*, [39, 52, 82, 87, 101]). However, the only known and general approach to developing a Byzantine version of a protocol or distributed system is to replace each host by a Replicated State Machine (RSM) [98, 121]. As replicas of a host can be assigned to existing hosts in the system, this does not necessarily require a large investment of hardware.

This chapter presents Nysiad, a technique that uses a variant of RSM to make distributed systems tolerant of Byzantine failures in asynchronous environments, and evaluates the practicality of the approach. Nysiad leverages

that most distributed systems already deal with crash failures and, rather than masking arbitrary failures, translates arbitrary failures into crash failures. Doing so avoids having to solve consensus [70] during normal operation. Nysiad invokes consensus only when a host needs to communicate with new peers or when one of its replicas is being replaced.

Instead of treating replicas as symmetric, Nysiad's replication scheme is essentially primary-backup with the host that is being replicated acting as primary. Different from RSM's original specification [98], Nysiad allows the entire RSM to halt in case the host does not comply with the protocol. A voting protocol ensures that the output of the RSM is valid. A credit-based flow control protocol ensures that the RSM processes all its inputs (including external input) fairly. As a result of combining both properties, the worst that the Byzantine host can accomplish is to stop processing input, a behavior that the original system will treat as a crash failure and recover accordingly.

Nysiad is an extension to the translation approach in the previous chapter. Both approaches translate Byzantine failures to crash failures. But unlike the previous approach, Nysiad aims for scalable distributed applications that could run over a long period of time. Nysiad reduces the replication cost and makes it independent from the system size. Also, Nysiad reconfigures the system to tolerate more failures that will occur in long runs.

We believe that the cost of Nysiad, while significant, is within the range of practicality for mission-critical applications. End-to-end message latencies grow by a factor of 3 compared to message latencies in the original system. The overhead caused by public key cryptography operations are manageable. Most alarmingly, the total number of messages sent in the translated system per end-

to-end message sent in the original system can grow significantly depending on factors such as the communication behavior of the original system. However, the message overhead does not grow significantly as a function of the total number of hosts, and grows only linearly as a function of the number of failures to be tolerated. Most of the additional traffic is in the form of control messages that do not carry payload.

The chapter

- evaluates for the first time the overheads involved when applying RSM to scalable distributed systems;
- shows that RSM does not require solving consensus if the original system is already tolerant of crash failures;
- presents a novel technique that forces hosts to process input fairly in a Byzantine environment, or leave;
- demonstrates how automatic reconfiguration can be supported in a Byzantine-tolerant distributed system.

Section 4.1 describes an execution model and introduces terminology. The Nysiad design is presented in Section 4.2. Section 4.3 provides notes on the implementation. Section 4.4 evaluates the performance of systems generated by Nysiad using various case studies. Section 4.5 discusses limitations and concludes the chapter.

4.1 Model

A *system* is a collection of hosts that exchange messages as specified by a *protocol*. Below we will use the terms *original* and *translated* to refer to the systems before and after translation, respectively. The original system tolerates only crash failures, while the translated system tolerates Byzantine failures as well. For simplicity we will assume that each host runs a deterministic state machine that transitions in response to receiving messages or expiring timers. (Nysiad may handle nondeterministic state machines by considering nondeterministic events as inputs.) As a result of input processing, a state machine may *produce* messages, intended to be sent to other hosts. The system is assumed to be asynchronous, with no bounds on event processing, message latencies, or clock drift.

The hosts are configured in an undirected *communication graph* (V, E), where V is the set of hosts and E the set of communication links. A host only communicates directly with its adjacent hosts, also called *neighbors*. The graph may change over time, for example as hosts join and leave the system. We will initially assume that the graph is static and known to all hosts. We later weaken this assumption and address topology changes.

The Nysiad transformation requires that the communication graph has a *guard graph*. A t-guard graph of (V, E) is a directed graph (V', E') with the following requirements:

1. Each host in V has a (directed) edge to at least $3t + 1$ hosts in V' (including itself) called the *guards* of the host.
2. For each two neighboring hosts in V, the two hosts have edges to at least

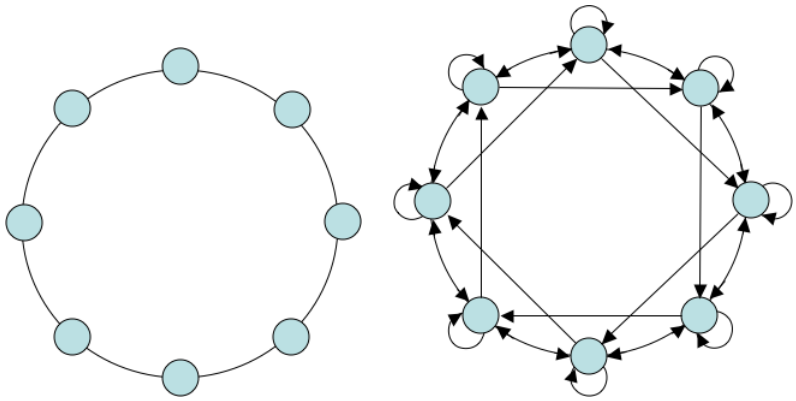


Figure 4.1: A communication graph (left) and a possible guard graph (right) for $t = 1$. In this particular case, each host has exactly $3t+1$ guards, and each set of neighbors exactly $2t+1$ monitors.

$2t + 1$ common guards in V' (including themselves). We call such guards the *monitors* of the two hosts.

We assume that for each host in V at most t of its guards are Byzantine. We also assume that messages between correct guards of the same host are eventually delivered (using an underlying retransmission protocol that recovers from message loss). Note that a monitor of two hosts is a guard of both hosts, and that neighbors in V are each other's guards. Moreover, each host is one of its own guards.

Within the constraints of these requirements, Nysiad works with any guard graph. For efficiency it is important to create as few guards per host as possible, as all guards of a host need to be kept synchronized. However, the requirements on guards and monitors may produce guard graphs with some of the hosts needing more than $3t + 1$ guards.

If $V = V'$, no additional hosts are added to the system and hosts guard one another. Figure 4.1 presents an example communication graph and a possible

guard graph for $t = 1$ where no additional hosts were added. Some deployments may favor adding additional hosts for the sole purpose of guarding hosts in the original system.

In the current implementation of Nysiad, the guard graph is created and maintained by a logically centralized, Byzantine-tolerant service called the *Olympus*, described in Section 4.2.4. The Olympus certifies the guards of a host, and is involved only when the communication graph changes as a result of host churn or new communication patterns in the original system. The Olympus does not need to be aware of the protocol that the original system employs.

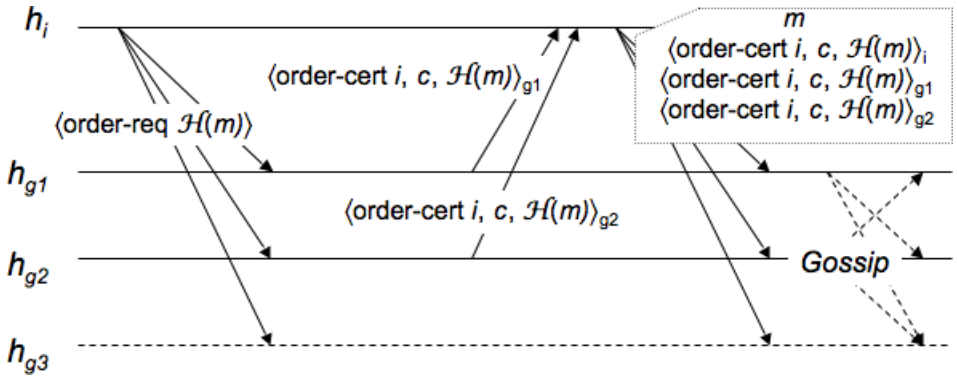


Figure 4.2: Host h_i initiates an OARcast execution for $t = 1$. The time diagram shows all guards of h_i, where only h_{g3} is faulty.

4.2 Design

Nysiad translates the original system by replicating the deterministic state machine of a host onto its guards. Nysiad is composed of four subprotocols. The *replication protocol* ensures that guards of a host remain synchronized. The *attestation protocol* guarantees that messages delivered to guards are messages pro-

duced by a valid execution of the protocol. The *credit protocol* forces a host to either process all its input fairly, or to ignore all input. Finally, the *epoch protocol* allows the guard graph to be bootstrapped and reconfigured in response to host churn. The following subsections describe each of these protocols. The final subsection describes how Nysiad deals with external I/O.

4.2.1 Replication

The state machine of a host is replicated onto the guards of the host, together constituting a Replicated State Machine (RSM). It is important to keep in mind that we replicate a host only for ensuring integrity, not for availability or performance reasons. After all, the original system can already maintain availability in the face of unavailable hosts.

Say α_j^i is the replica of the state machine of host h_i on guard h_j. A host h_i broadcasts input events for its local state machine replica α_i^i to its guards. A guard h_j delivers an input event to α_j^i when h_j receives such a broadcast message from h_i. In order to guarantee that the guards of h_i stay synchronized in the face of Byzantine behavior, the hosts use OARcast (see Section 3.3) for communication.

Using OARcast a host can *send* a message that is intended for all its guards. When a guard host h_j *delivers a message* m *from* h_i it means that h_j received m, believes it came from h_i, and delivers m to α_j^i, the replica of h_i's state machine on guard h_j. OARcast guarantees the following properties:

- *Relay.* If h_j and h_k are correct, and h_j delivers m from h_i, then h_k delivers m from h_i (even if h_i is Byzantine);
- *Ordering.* If two hosts h_j and h_k are correct and h_j and h_k both deliver m_1 from h_i and m_2 from h_i, then they do so in the same order (even if h_i is Byzantine);
- *Authenticity.* If two hosts h_i and h_j are correct and h_i does not send m, then h_j does not deliver m from h_i;
- *Persistence.* If two hosts h_i and h_j are correct, and h_i sends a message m, then h_j delivers m from h_i;
- *FIFO.* If two hosts h_i and h_j are correct, and h_i sends a message m_1 before m_2, then h_j delivers m_1 from h_i before delivering m_2 from h_i.

Relay guarantees that all correct guards deliver a message if one correct guard does. *Ordering* guarantees that all correct guards deliver messages from the same host in the same order. These two properties together guarantee that the correct replicas of a host stay synchronized, *even if the host is Byzantine*. *Authenticity* guarantees that Byzantine hosts cannot forge messages of correct hosts. *Persistence* rules out a trivial implementation that does not deliver any messages. *FIFO* stipulates that correct guards deliver messages from a correct host in the order sent.

These properties are not as strong as those for asynchronous consensus [70] and indeed consensus is not necessary for our use, as only the host whose state is replicated can issue updates (*i.e.*, there is only one *proposer*). If that host crashes or stops producing updates for some other reason, no new host has to be elected to take over its role, and the entire RSM is allowed to halt as a

result. Indeed, unlike consensus, the OARcast properties can be realized in an asynchronous environment with failures, as we shall show next.

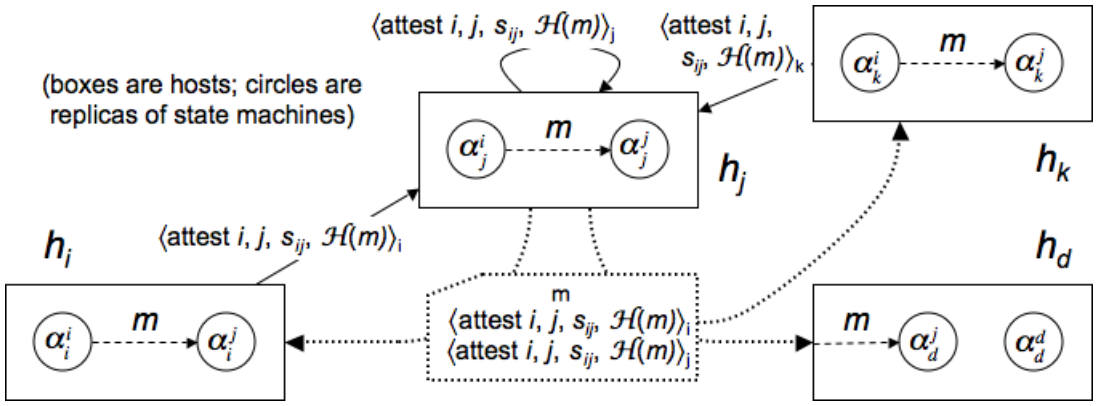


Figure 4.3: Normal case attestation when $t = 1$. Here the state machine of h_i sends a message m to the state machine of h_j. The guards of h_j are h_i, h_k, h_d, and h_j itself, and each run a replica of h_j's state machine. Hosts h_i, h_j, and h_k monitor h_i and h_j. h_j collects attestations for m and OARcasts the event to its guards. In this case only h_d needs the attestations.

The implementation of OARcast used in this chapter is as follows. Say a sending host $h_i \in V$ wants to OARcast an input message m to its n_i guards in V', where ($n_i > 3t$). Each guard h_j maintains a sequence number c on behalf of h_i. Using private (MAC-authenticated) FIFO point-to-point connections, h_i sends $\langle \texttt{order-req}\ \mathcal{H}(m) \rangle$ to each guard, where $\mathcal{H}$ is a cryptographic one-way hash function. On receipt, h_j sends an *order certificate* $\langle \texttt{order-cert}\ i, c, \mathcal{H}(m) \rangle_j$ back to h_i, where the subscript j means that h_j digitally signed the message such that any host can check its origin.

As at most t of h_i's guards are Byzantine, h_i is guaranteed to receive `order-cert` messages from at least $n_i - t$ different guards with the correct sequence number and message hash. We call such a collection of `order-cert` messages an *order proof for* (c, m). Byzantine orderers cannot generate conflict-

ing order proofs (same sequence number, different messages) even if h_i itself is Byzantine, as two order proofs have at least $t + 1$ `order-cert` messages in common, one of which is guaranteed to be generated by a correct guard [104].

h_i delivers m locally to α_i^i and forwards m along with an order proof to each of its guards. On receipt, a guard h_j checks that the order proof corresponds to m and is for the next message from h_i. If so, h_j delivers m to α_j^i.

To guarantee the Relay property, h_j gossips order proofs with the other guards. A similar implementation of OARcast is proved correct in Chapter 3 (also [86]). That chapter also presents an implementation that does not use public key cryptography, but has higher message overhead.

Figure 4.2 shows an example of an OARcast execution. Optimizations are discussed in Section 4.3. Not counting the overhead of gossip and without exploiting hardware multicast, a single OARcast to $3t$ guards uses at most $9t$ messages. Gossip can be largely piggybacked on existing traffic.

4.2.2 Attestation

While the replication protocol above guarantees that guards of a host synchronize on its state, it does not guarantee that the host OARcasts *valid* input events, because the sending host h_i may forge arbitrary input events. We consider two kinds of input events: message events and timer events. Checking validity for each is slightly different.

First let us examine message sending. Say in the original system host $h_i \in V$ sends a message m to a host $h_j \in V$. Because h_j is a neighbor of h_i it is also

a guard of h_i, and thus in the translated system α_j^i will produce m as an input event for α_j^j. Accordingly h_j OARcasts m to its guards, but the guards, not sure whether to trust h_j, need a way to verify the validity of m before delivering m to local replicas. To protect against Byzantine behavior of h_j, we require that h_j includes a proof of validity with every OARcast in the form of a collection of $t + 1$ *attestations* from guards of h_i.

Because the guards of h_i implement an RSM, each (correct) guard $h_k \in V'$ has a replica α_k^i of the state of h_i that produces m. Each guard h_k of h_i (including h_i and h_j) sends $\langle \texttt{attest}\ i, j, s_{ij}, \mathcal{H}(m) \rangle_k$ to h_j. s_{ij} is a sequence number for messages from i to j, and prevents replay attacks. h_j has to collect t of these attestations in addition to its own and include them in its OARcast to convince h_j's guards of the validity of m. Again, correct guards have to gossip attestations in order to guarantee that every correct guard receives them in case one does.

There are two important optimizations to this. First, as h_j only needs $t + 1$ attestations, only the monitors of h_i and h_j need to send attestations to guarantee that h_j gets enough attestations. This not only reduces traffic, but the monitors are neighbors of h_j and thus no additional communication links need be created. Second, h_j does not need the attestations until the last phase of the OARcast protocol, thus h_j can request order certificates before it has received the attestations. This way ordering and attestation can happen in parallel rather than sequentially. Both these optimizations are exploited in the implementation (Section 4.3). Figure 4.3 shows an example of the flow of traffic when using attestations.

In case of a timer event at a host h, h needs to collect t additional attestations of its own guards in addition to its own attestation. This prevents h from pro-

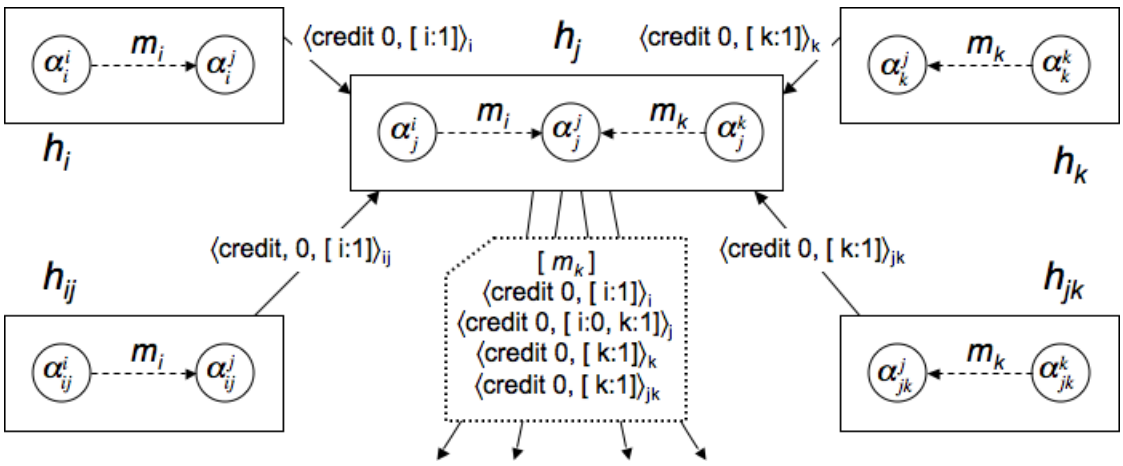


Figure 4.4: Credit mechanism with $t = 1$. h_i and h_k are neighbors of h_j, each sending a message to h_j. h_j tries to order the message from h_k while ignoring the message from h_i. The credit mechanism renders the OARcast illegal.

ducing timer events at a rate higher than that of the fastest correct host. While theoretically this may appear useless in an asynchronous environment, in practice doing so is important. Consider, for example, a system in which a host pings its neighbors in order to verify that they are alive. Without timer attestation, a Byzantine host may force a failure detection by not waiting long enough for the response to those pings. While in an asynchronous system one cannot detect failures accurately using a pinging protocol, timer attestation ensures in this case that a host has to wait a reasonable amount of time. Also, because hosts only wait for t attestations from more than $2t$ guards, Byzantine guards cannot delay or block timer events emitted by correct hosts.

4.2.3 Credits

While attestation prevents a host from forging invalid input events, a host may still selectively ignore input events and fail to produce certain output events. For example, in the pinging example above, a host could respond to pings, avoiding failure detection, but neglect to process other events. In a multicast tree application a host could accept input but neglect to forward output to its children (freeloading). Such a host could even deny wrongdoing by claiming that it has not yet received the input events or that the output events have already been sent but simply not yet delivered by the network—after all, we assume that the system is asynchronous.

We present a credit-based approach that forces hosts either to process input from all sources fairly and produce the corresponding output events, or to cease processing altogether. The essence of the idea is to require that a host obtain credits from its guards in order to OARcast new input events, and a guard only complies if it has received the OARcast from the host for previous input events. As such, credits are the flip-side of attestations: while attestations prevent a host from producing bad output, credits force a host to either process all input or process none of it. If a host elects to process no input, it cannot produce output and will eventually be considered as a crashed host by the original system.

We will exploit that a single OARcast from a host can order a sequence of pending input events for its state machine, rather than one input message at a time. The OARcast's order certificate binds a single sequence number to the ordered list of input events. We say that the OARcast *orders* the events in the list.

A credit is a signed object of the form $\langle \texttt{credit}\ j, c, \vec{v}_{i,j} \rangle_i$, where h_i has to be a guard of h_j. h_i sends such a credit to h_j to approve delivery of the c^{th} OARcast message from h_j, provided a certain ordering condition specified by $\vec{v}_{i,j}$ holds. Including c prevents replay attacks. The ordering constraint $\vec{v}_{i,j}$ is a vector that contains an entry for each state machine that h_i and h_j both guard. Such an entry contains how many events (possibly 0) the corresponding state machine replica on h_i has produced for α_i^j.

For each neighbor h_k of h_j, h_j has to collect at least $t + 1$ credits for OARcast c from monitors of h_j and h_k. However, h_j can only use a credit for an OARcast if the OARcast orders all messages specified in the credit's ordering constraint that were not ordered already by OARcasts numbered less than c. These two constraints taken together guarantee that an OARcast contains a credit from a correct monitor for each of its neighbors, and prevents h_j from ignoring input messages that correct monitors observe while ordering other input messages.

For example, consider Figure 4.4, showing five hosts. h_i and h_k are neighbors of h_j. h_{ij} is a monitor for hosts h_i and h_j, while h_{jk} is a monitor for h_j and h_k. Assume $t = 1$. Consider a situation in which h_j has not yet sent any OARcasts, but α_i^i has produced a message m_i for h_j on hosts h_i and h_{ij}, while α_k^k has produced a message m_k for h_j on hosts h_k and h_{jk}. Each guard of h_j sends credit for the first OARcast that reflects the messages produced locally for h_j.

Now assume that h_j is Byzantine and trying to ignore messages from h_i but process messages from h_k. h_j has to include a credit from either h_i or h_{ij}. Because h_j is Byzantine and $t = 1$, both h_i and h_{ij} have to be correct and will not collude with h_j. If h_j tries to order only m_k as shown in the figure, receiving hosts will note that at least one credit requires that a message from h_i has to

be ordered and will therefore ignore the OARcast (and report the message to authorities as proof of wrongdoing).

As with other credit-based flow control mechanisms, a *window* w may be used to allow for pipelining of messages. Initially, each guard of h_j sends credits for the first w OARcasts from h_j, specifying an empty ordering constraint. Then, on receipt of the c^{th} OARcast, a guard sends a credit for OARcast $c + w$, using an ordering constraint that reflects the current set of produced messages for h_j. If $w = 1$, the next OARcast cannot be issued until it has been received by at least $t + 1$ monitors of each neighbor and the new credits have been communicated to h_j. If $w > 1$ pipelining becomes possible, but at the expense of additional freedom for h_j. In practice we found that $w = 2$ enables good performance while monitors still have significant control over the order of messages produced by the hosts they guard.

4.2.4 Epochs

So far we have assumed that the communication graph (V, E) and its t-guard graph (V', E') are static and well-known to all hosts. This is necessary, because when a host receives an OARcast it has to check that the order certificates, the attestations, and the credits were generated by qualified hosts. In particular, order certificates and credits have to be generated by a guard of the sending host of an OARcast message, and each attestation of a message has to be generated by monitors of the source and destination of the message. Also, the receiving host of an OARcast has to know how many guards the sending host has in order

to check that a message contains a sufficient number of ordering certificates and credits.

While Nysiad, in theory, could inspect the code of the state machines, it has no good way of determining which hosts will be communicating with which other hosts, and so in reality even the communication graph (V, E) is initially unknown, let alone its guard graph. Making matters worse, such a communication graph often evolves over time.

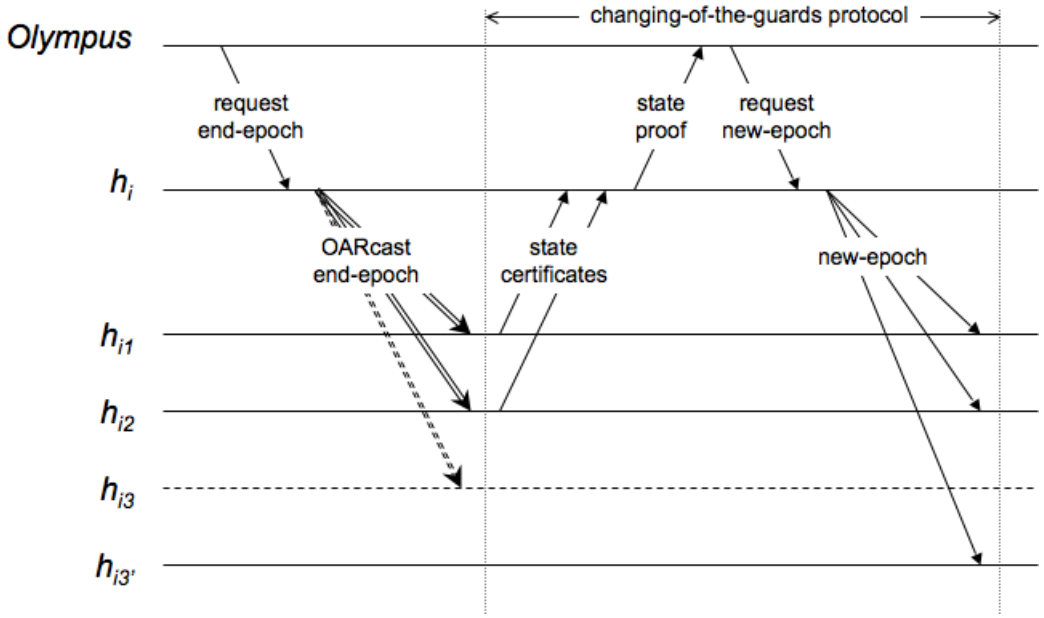


Figure 4.5: Example of an execution of the reconfiguration protocol. h_{i1}, h_{i2}, and h_{i3} are guards of h_i. When the Olympus suspects that h_{i3} has failed, it requests the current epoch of h_i to be concluded and installs a new set of guards, replacing h_{i3} with $h_{i3'}$.

In order to handle this problem, we introduce a logically centralized (but Byzantine-replicated [53]) trusted certification service that we call the *Olympus*. The Olympus is not involved in normal communication, but only in charge of tracking the communication graph and updating the guard graph accordingly. The Olympus produces signed *epoch certificates* for hosts, which contain suffi-

cient information for a receiver of an OARcast message to check its validity. In particular, an epoch certificate for a host h_i describes

- the host identifier (i);
- the set of the identifiers of all h_i's guards;
- the *epoch number* (described below);
- a hash of the final state of the host in the previous epoch.

The Olympus does not need to know the protocol that the original system uses. Initially, the Olympus assigns $3t$ guards to each host arbitrarily, in addition to the host itself. Each guard starts in epoch 0 and runs the state machine starting from a well-defined initial state. Order certificates and credits have to contain the epoch number in order to prevent replay attacks of old certificates in later epochs. Next we describe a general protocol for changing guards and how this protocol is used to handle reconfigurations.

Changing-of-the-guards

While the Olympus assigns guards to hosts, the *changing-of-the-guards* protocol starts with the guards themselves. In response to triggers that we will describe below, each guard of h_i sends a *state certificate* containing the current epoch number and a secure hash of its current state to h_i. After the guard receives an acknowledgment from h_i it is free to clean up its replica, unless the guard is h_i itself. However, in order to avoid replay attacks the guard needs to remember that this epoch of h_i's execution has terminated.

When h_i has received $n_i - t$ such certificates (typically including its own) that correspond to its own state, h_i sends the collection of state certificates to the Olympus. $n_i - t$ certificates together guarantee that there are at most t correct guards and t Byzantine guards that are still active, not enough to order additional OARcast messages. Effectively, the collection certifies that the state machine of h_i has halted in the given state.

In response, the Olympus assigns new guards to h_i and creates a new epoch certificate using an incremented epoch number and the state hash, and sends the certificate to h_i. On receipt, h_i sends its signed state and the new epoch certificate to its new collection of guards. Recipients check validity of the state using the hash in the epoch certificate and resume normal operation.

Reconfiguration

One scenario in which the changing-of-the-guards protocol is triggered is when guards of h_i produce a message m for another host h_j for the first time. Each correct guard sends a state certificate to h_i when it produces the message. *The state has to be such that the message m is about to be produced, so that when the state machine is later restarted, possibly on a different guard, m is produced and processed the normal way.* The state certificate also indicates that a message for h_j is being produced, so that the Olympus may know the reason for the invocation.

h_i collects $n_i - t$ state certificates, and sends the collection to the Olympus. The Olympus, now convinced that h_i has produced a message for h_j, requests h_j to change its guards as well. h_j does this by OARcasting a special `end-epoch` message, triggering the changing-of-the-guards protocol at each guard in the

same state. (Should h_j not respond then it is up to the Olympus to decide when to declare h_j faulty, block h_j's guards, and restart h_i.)

Assuming the Olympus has received the state certificates for both h_i and h_j, the Olympus can assign new guards to each in order to satisfy the constraints of the guard graph. The Olympus then sends new epoch certificates to both h_i and h_j, after which each sends its certificate to its new guards. These guards start in a state where they first produce m, which can now be processed normally.

The Olympus also undertakes reconfiguration when it determines that a guard of a host has failed. In order to detect crash failures, the Olympus may periodically ping all guards to determine responsiveness. (A more scalable solution is described in [87]. Note that while a false positive may introduce overhead, it is not a correctness issue.) Also, guards send proof of observable Byzantine behavior to the Olympus. In response to detection of a failure of a guard of a host other than the host itself, the Olympus requests the host to OARcast an `end-epoch` message to invoke the changing-of-the-guards protocol. Figure 4.5 shows an example.

Should a host $h_i \in V$ be detected as faulty then the Olympus sends a message to all h_i's guards, requesting them to block further OARcasts from h_i. Once the Olympus has received acknowledgments from $n_i - t$ guards, the Olympus knows that h_i can no longer produce input for other hosts successfully.

4.2.5 External I/O

So far we have assumed that Nysiad translates a system in its entirety. However, often such a system serves external clients that cannot easily be treated in the same way. We cannot expect to be able to replicate those clients onto multiple hosts, and it becomes impossible to verify that the clients send valid data using a general technique. To wit, a Byzantine-tolerant storage service does not verify the validity of the data that it stores, nor does a Byzantine-tolerant multicast service check the data from the broadcaster. The usual assumption, from the system's point of view, is to *trust* clients.

In Nysiad, we treat external clients as trusted hosts. Such hosts may crash or leave, but there is no need to replicate their state machines, nor to attest the data they generate. However, when a trusted host h_i sends a message to an untrusted host h_j, we do want to make sure that h_j treats the input fairly with respect to other inputs that it receives. Vice versa, when h_j sends a message to h_i, h_i has to collect attestations in order to verify the validity of the message. We also want to prevent h_j from withholding messages for h_i.

The methodology we developed so far can be adapted to achieve these requirements. We assign the pair (h_i, h_j) $2t + 1$ *half-monitors*. Each half-monitor runs a full replica of h_j's state machine, but for h_i only keeps track of the messages that h_i sends. Unlike normal monitors, h_i itself does not run a half-monitor, but h_j does.

When h_i wants to send a message to h_j, it sends a copy of the message to each half-monitor using authenticated FIFO channels. (The half-monitors gossip the receipt of this message with one another to ensure that either all or

none of the correct half-monitors receive the message in a situation in which h_i crashes during sending.) Like normal monitors, half-monitors generate attestations for messages from h_i so that h_j can convince others of the validity of that input. More importantly, half-monitors generate credits for h_j forcing h_j to treat h_i's messages fairly with respect to its other inputs. In a similar manner, half-monitors generate attestations for messages from h_j to h_i so that h_i can verify the validity of those messages. Should h_j itself fail to send messages to h_i then the half-monitors can provide the necessary copy.

4.3 Implementation Details

In order to evaluate overheads we implemented Nysiad in Java. In this section we provide details on how we construct guard graphs and how we combine the various subprotocols into a single coherent protocol.

Given a communication graph and a parameter t many different guard graphs are often possible. For efficiency and fault tolerance it is prudent to minimize the number of guards per host (see Section 4.1). We are not aware of an optimal algorithm for determining such a graph. We devised the following algorithm to create a t-guard graph of the communication graph. It runs in two phases. In the first phase the algorithm considers each pair of neighbors (h_i, h_j). Initially h_i and h_j are assigned as monitors. The algorithm then determines the hosts that are 1 hop away from the current set of monitors, and adds, randomly, such hosts to the set of monitors until there are no such hosts left or until the number of monitors has reached $2t + 1$. This step is repeated until the set of monitors has reached the required size. Note that the monitors are guards to

both h_i and h_j. In the next phase, the algorithm considers all hosts individually. If a host has fewer than $3t + 1$ guards then the closest hosts in terms of hop distance are added, randomly as before, until the desired number of guards is reached.

While best understood separately, the OARcast, attestation, and credit protocols combine into a single replication protocol. Doing so reduces message and CPU overheads significantly, while also simplifying implementation. Consider the c^{th} OARcast from some host h_i, and assume h_i has the necessary credits and has produced the messages required by those credits. At this point h_i creates an `order-req`, containing a list of hashes of the messages that it has produced but not yet ordered in previous OARcasts, and sends the request to each of its n_i guards.

On receipt, each guard signs a *single* certificate that contains the credit for OARcast $c+w$, an order certificate for OARcast c, and any attestations that it can create for messages in OARcast c. This way the signing and checking costs of all certificate types can be amortized. The guard sends the resulting certificate back to h_i. h_i awaits $n_i - t$ certificates, which collectively are guaranteed to contain the necessary order certificates and attestations for completing the current OARcast, and the necessary credits for OARcast $c + w$.

In the third and final round, h_i sends these aggregate certificates to its guards. On receipt, a guard has to check the signatures on all certificates except its own. The end-to-end latency consists of three network latencies, plus the latency of signing (done in parallel by each of the guards) and checking $n_i - 1$ certificates (executed in parallel as well). The more messages can be ordered by a single OARcast, the more these costs can be amortized.

An execution of OARcast requires $3 \cdot (n_i - 1)$ FIFO messages. Since $n_i > 3t$, the minimum number of FIFO messages per OARcast is $9t$. In order to further reduce traffic, Nysiad also tries to combine messages for different OARcasts—if two FIFO messages are sent at approximately the same time between two different hosts, they are combined in a manner similar to back-to-back messages in the TCP protocol.

4.4 Case Studies

While one cannot test if a system tolerates Byzantine failures, it is possible to measure the overheads involved. In this section we report on two case studies: a point-to-point link-level routing protocol and a peer-to-peer multicast protocol. We applied Nysiad to each and ran the result over a simulated network to measure network overheads and overheads caused by cryptographic operations.

For the point-to-point routing protocol we selected Scalable Source Routing (SSR) [71]. SSR is inspired by the Chord overlay routing protocol [125], but can be deployed on top of the link layer. (SSR is similar to Virtual Ring Routing [51], which applies the same idea to Pastry.)

The basic idea of SSR is simple. Each host initially knows its own (location-independent) identifier and those of the neighbors it is directly connected to. The SSR protocol organizes the hosts into a Chord-like ring by having each host discover a source route to its successor and predecessor. This is done as follows. Initially a host h_i sends a message to its best guess at its successor. Should this tentative successor host know of a better successor for h_i, or discover one

later, then the successor host sends a source route for the better successor back to h_i. On receipt h_i sends a message to its new best guess at its successor, and so on. This protocol converges into the desired ring and terminates. Once the ring is established routing can be done in a Chord-like manner, whereby a message travels around the ring, but taking shortcuts whenever possible. In our simulations we measure the ring-discovery protocol, not the routing itself.

The multicast protocol is even simpler. Here we assume that the hosts are organized in a balanced binary tree, and that each host forwards messages from its parent to its children (if any). We call this protocol MCAST. We measured the overhead of sending a message from the root host to all hosts.

We considered two network graph configurations. In the first, Tree, the network graph is a balanced binary tree. In the second, Random, we placed hosts uniformly at random on a square metric space, and connected each host to its k closest peers.

We report on three configurations:

- **SSR/Random** The SSR protocol on top of a Random graph;
- **SSR/Tree** The SSR protocol on a Tree graph;
- **MCAST/Tree** The MCAST protocol on a Tree graph.

For the evaluation we developed a simple discrete time event network simulator to evaluate message overheads. The fidelity of the simulation was kept low in order to scale the simulation experiments to interesting sizes. While the simulator models network latency, we assume bandwidth is infinite. The public key signature operations were replaced by simple hash functions. We focus our

evaluation on the failure-free "normal case" executions. We vary the number of hosts and t, and in the case of the Random graph we also vary k, the (minimum) number of neighbors of each host. In all experiments, the credits window w was chosen to be 2.

By and large, the increase in latency is close to a factor of 3 for all experiments, independent of what parameters are chosen. (No graphs shown.) This amount of increase was expected as the OARcast protocol consists of three rounds of communication (see Section 4.3). This can be decreased to two rounds by having the guards broadcast certificates directly to each other, but this results in a message overhead that is quadratic in t rather than linear.

When measuring message overhead, we report on the ratio between the number of FIFO messages sent in the translated protocol and the number of FIFO messages sent in the original protocol. We call this the *message overhead factor*, and report the minimum, average, and maximum over 10 executions. We ignore messages sent on behalf of the gossip protocol that implement the Relay property of OARcast. These messages do not require additional cryptographic operations and contribute only a small and constant load on the network.

For measuring CPU overhead, we report only the number of public key signing and checking operations per message per guard. Such operations tend to dominate protocol processing overheads. We found the variance for these measurements to be low, the minimum and maximum usually being within 1 operation from the average number of operations, and so we report only the averages.

In the first set of experiments, we used the SSR/Random configuration using a Random graph with $k = 3$. In Figure 4.6(a) we show the message overhead

factor for $t = 1, 2, 3$. As we described in Section 4.3, an OARcast to n guards uses at most $3n$ messages, and we see that this explains the trends well. There is an increase in overhead as we increase the number of hosts due to an increase in the average number of guards per host and reduced opportunity for aggregation as traffic becomes less concentrated due to the larger graph. Small graphs necessitate more sharing of guards, which reduces overhead.

Figure 4.6(b) reports, per guard the average number of public key sign and check operations per message in the original system. Due to aggregation, the number of sign operations message in the original system per guard is always less than 1 and does not significantly depend on t, as can be understood from Section 4.3. However, guards have to check each other's signatures and The number of check operations per message per guard may exceed $3t$ because a host may have more than $3t + 1$ guards, and, as stated above larger graphs tend to have more guards. Nonetheless, these graphs should also reach an asymptote.

Next, for the same SSR/Random configuration, we fix $t = 2$ and range k from 3 to 6. We show the message and public key signature overhead measurements in Figure 4.7. Even though t is fixed, an increase in the number of neighbors per host requires additional monitors, and thus the average number of guards per host tends to increase beyond the required $3t + 1$, causing additional message and CPU overhead. It is thus important for overhead of translation and indeed for fault tolerance to configure the original protocol to use as sparse a graph as possible. This tends to increase the diameter of the communication graph, and thus a suitable trade-off has to be designed.

In the final experiments, we compare the three different configurations for

$t = 1$. For the Random graph we chose $k = 3$. In the case of a Tree graph, the average number of neighbors per host is approximately 2, internal hosts having 3 neighbors, leaf hosts having 1 neighbor, and the root host having 2 neighbors. We report results in Figure 4.8.

MCAST suffers most message overhead. This is because there is no opportunity for message aggregation in the experiment—each host receives only one message (from its parent). However, when multiple messages are streamed, the opportunity for message aggregation is excellent—any backlog that builds up can be combined and ordered using a single OARcast operation—and thus throughput is not limited by this overhead. Even if messages cannot be aggregated, order certificates, attestations, and credits still can, and thus signature generation and checking overheads are still good.

SSR performs significantly better on the Tree graph than on the Random graph. Because communication opportunities are more limited in the Tree graph with fewer neighbors to choose from, many messages can be aggregated and ordered simultaneously. For such situations the message overhead can indeed completely disappear.

Finally, note that if hardware multicast were available the overhead of Nysiad could be significantly reduced (from $9t$ point-to-point messages for an OARcast in the best case to $3t$ point-to-point messages and 2 multicasts).

4.5 Discussion

Nysiad can generate a Byzantine-tolerant version of a system that was designed to tolerate only crash failures. This comes with significant overheads. When developing a Byzantine-tolerant file system, such overheads are easily masked by the overhead of accessing the disk and large data transfers. When applied to message routing protocols where there is no disk overhead and payload sizes are relatively small, overheads cannot be masked as easily.

In practice, Nysiad may be used to generate a *first cut* at a Byzantine-tolerant protocol or distributed system, and then apply application-specific optimizations that maintain correctness. For example, if it is possible to distinguish the retransmission of a data packet from the original transmission, then it may be possible for the original transmission to be routed unguarded. Doing so could potentially mask most overhead of Nysiad.

But even if such optimizations are not possible, some applications may choose robustness over raw speed. Byzantine fault tolerance can be a part of increasing security, but it does not solve all security problems. Nysiad is not intended to defend against intrusion, but to tolerate intrusions. Defense against intrusion involves authentication and authorization techniques, as well as intrusion detection, and these are essential to guarantee that there is sufficient diversity among guards and no more than a small fraction are compromised. In the face of a limited number of successful intrusions Nysiad maintains integrity and availability of a system, but it does not provide confidentiality of data. Worse still, the replication of state complicates confidentiality. Hosts cannot trust their

guards for confidentiality, and confidential data has to be encrypted in an end-to-end fashion.

Another possibility is to run some of the mechanisms that Nysiad uses inside secured hosts that are more difficult to compromise than hosts "in the field." Such secured hosts may have reduced general functionality and use their resources to guard a relatively large number of state machines.

Nysiad makes strong assumptions about how many hosts can fail using the threshold value t. But what happens if more than t guards of a host become Byzantine? Now the host can in fact behave in a Byzantine fashion and break the system. As a system becomes larger it becomes more likely that a host has more than t Byzantine guards, and thus t should the chosen large enough to handle the maximum system size. If N is the maximum system size, then t should be chosen $O(\log N)$ in order to keep the probability that any host in the system has more than t Byzantine guards sufficiently low. As [87] demonstrates, a value for t of 2 or 3 is probably sufficient for most applications. It is also important that, as much as possible, proofs of observed Byzantine behavior are sent to the Olympus immediately so that faulty hosts can be removed quickly [133].

Nysiad exploits diversity and is defenseless against deterministic bugs that either cause a host to make an incorrect state transition or allow an attacker to compromise more than t host. The use of configuration wizards, high-level languages, and bug-finding tools may help avoid such problems. Similarly, Nysiad is helpless in the face of link-level Denial-of-Service attacks. These should be controlled by network-level anti-DoS techniques.

Nysiad in its current form uses the Olympus, a logically centralized service,

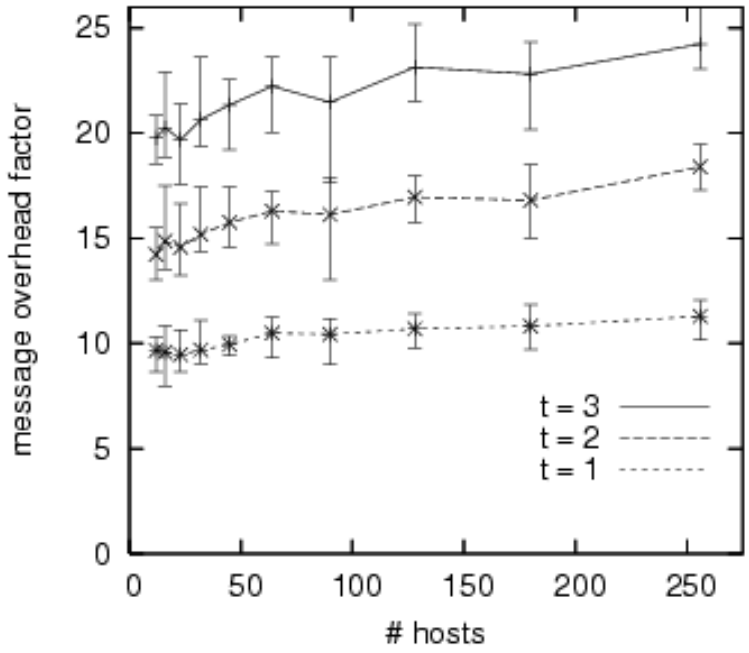
to handle configuration changes. Because the Olympus is not invoked during normal operation, the load on the Olympus is likely sufficiently low for many practical applications. This architecture does not deal well with high churn, nor does the translated protocol handle network partitions well: hosts that cannot communicate with the Olympus are excluded from participating.

We have evaluated the use of Nysiad for systems where each host has a relatively small number of neighbors with which it communicates actively. Figure 4.7 shows that overhead grows as a function of the number of neighbors. In systems where hosts have many active neighbors the overhead of the Nysiad protocols could be substantial. One can consider a variant of Nysiad where not all neighbors of a host are guards in order to contain overhead. The challenge is how to ensure that the set of guards contains $2t + 1$ monitors for each link.

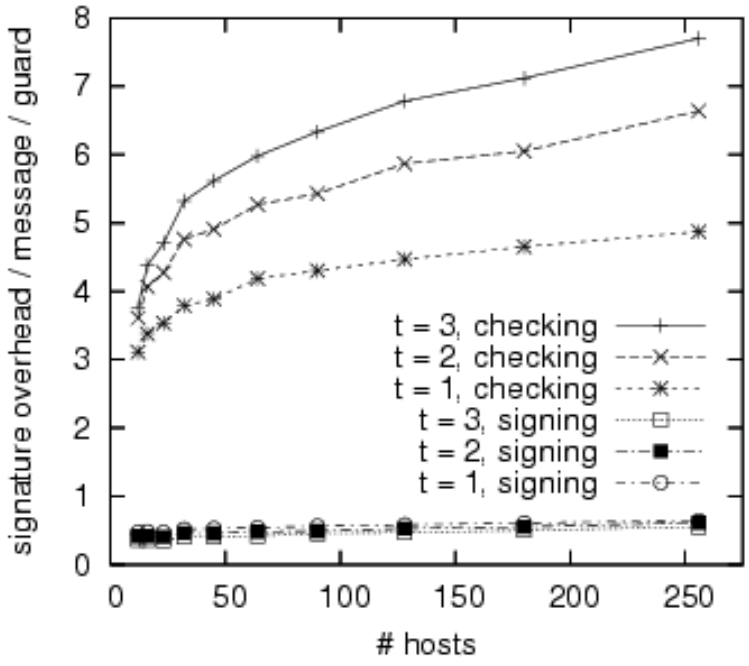
Closely related to Nysiad is the PeerReview system [81], providing accountability [133] in distributed systems. PeerReview detects those Byzantine failures that are observable by a correct host. Like Nysiad, PeerReview assumes that each host implements a protocol using a deterministic state machine. PeerReview maintains incoming and outgoing message logs and, periodically, runs incoming logs through the state machines and checks output against outgoing logs. PeerReview can only detect a subclass of Byzantine failures, and only after the fact. Like reputation management and intrusion detection systems, accountability deters intentionally faulty behavior, but does not prevent or tolerate it.

Nysiad is a general technique for developing scalable Byzantine-tolerant systems and protocols in an asynchronous environment that does not require consensus to be solved. Starting with a system tolerant of crash failures only, Nysiad assigns a set of guards to each host that verify the output of the host

and constrain the order in which the host handles its inputs. A logically centralized service assigns guards to hosts in response to churn in the communication graph. Simulation results show that Nysiad may be practical for a large class of distributed systems.

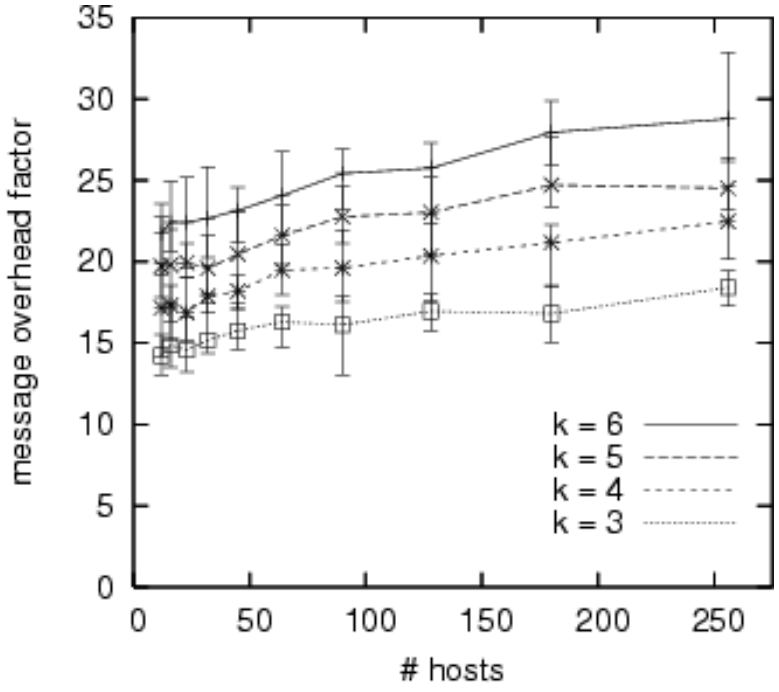


(a)

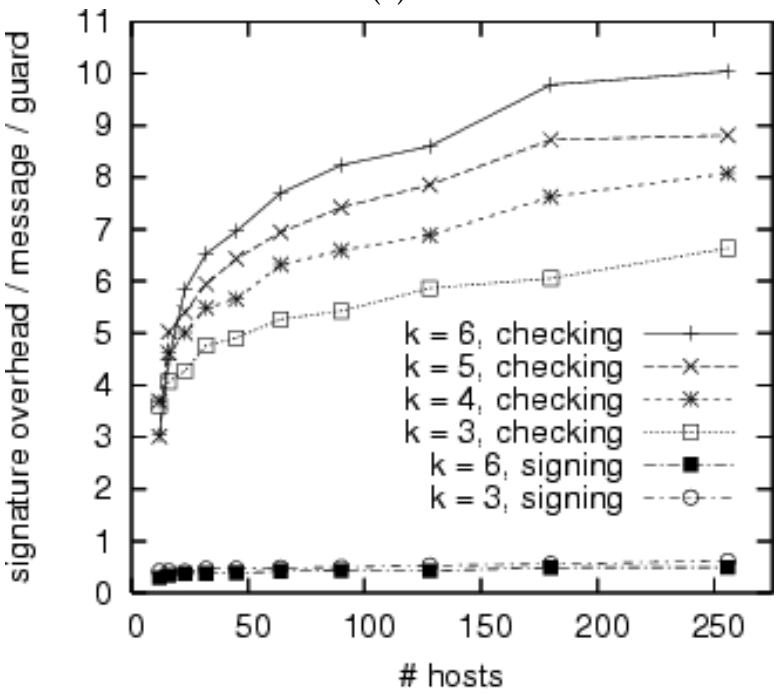


(b)

Figure 4.6: Message overhead factor (a) and public key signing and checking overheads (b) as a function of the number of hosts for running SSR on a Random graph using $k = 3$ and various t.

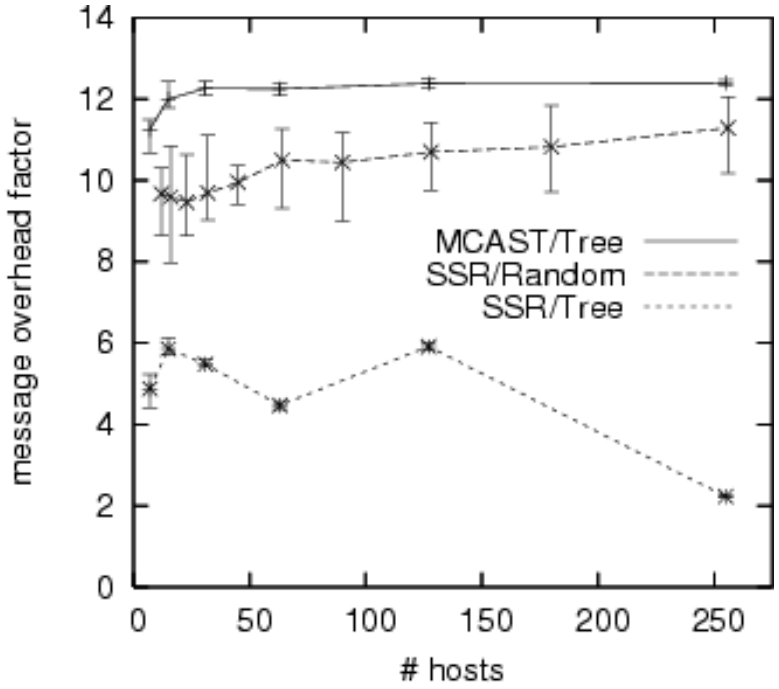


(a)

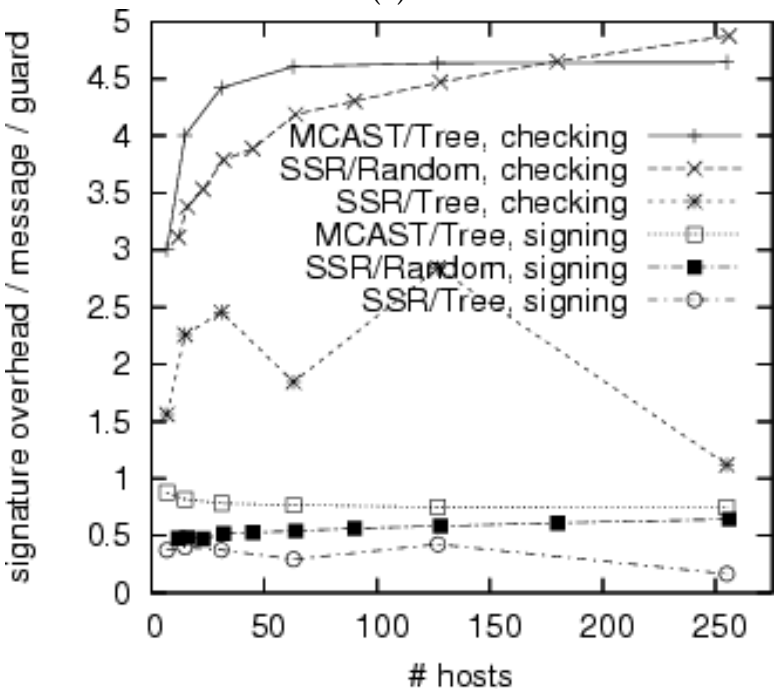


(b)

Figure 4.7: Message overhead (a) and public key signing and checking overheads (b) as a function of the number of hosts for the SSR protocol on a Random graph using $t = 2$ and various k, the minimum number of neighbors per host.



(a)



(b)

Figure 4.8: Message overhead factor (a) and public key signing and checking overheads (b) as a function of the number of hosts for various protocols and graphs using $t = 1$ and $k = 3$.

CHAPTER 5

SHUTTLE: AFFORDABLE BYZANTINE REPLICATION

Nysiad (Chapter 4) is designed for strengthening crash-tolerant distributed applications to Byzantine fault tolerant (BFT). The technique is applicable to scalable systems with reasonable overhead on replication and performance. Nonetheless, the overhead is still far from appealing to practitioners:

- *Replication cost.* To tolerate t Byzantine faults, Nysiad requires $2t + 1$ *monitors* (i.e., replicas) for each link, with a minimum of $3t + 1$ replicas total for each host. This requirement may result in high replication cost, where hosts have more than $3t + 1$ replicas.
- *Computational cost.* Nysiad relies on expensive RSA public-key digital signatures for authenticating messages. Though the average numbers of signing and verifying operations per application message can be small in practice (see, for examples, Figures 4.6(b), 4.7(b), and 4.8(b)) the computational cost of RSA operations still imposes significant overhead on the performance of the application.

In this chapter we propose *Shuttle*, a technique that improves on Nysiad. First, Shuttle reduces the replication costs in Nysiad by (1) eliminating the links' monitors in Nysiad and using a retransmission and an acknowledgment mechanisms to prevent send and receive omission, and (2) reducing the number of replicas in Nysiad to $t + 1$ for tolerating t Byzantine failures. Second, Shuttle reduces computational cost by using cheaper cryptographic primitives (e.g., HMAC [92] and CRC [118] vs. RSA [120]). And third, Shuttle strength-

ens Nysiad so that it can translate *any* distributed applications (not just crash-tolerant ones) to BFT ones.

Shuttle is optimized by reducing its bandwidth consumption. The optimization follows the communication approach in Chain Replication [128]. Compared to the broadcast communication pattern (e.g., in Primary-Backup replication [49]), the chain communication pattern significantly reduces message complexity while pipelining supports high throughput.

Shuttle is further optimized by considering a stronger assumption about adversaries, which restricts certain faulty behavior. The optimization is rooted in our observation that Byzantine fault tolerance cannot actually tolerate *every* type of failure. For example, if a machine fails because its vulnerabilities are exploited by a hacker, the hacker can compromise other machines that have the same vulnerabilities as well. [1] Also, we observe that Byzantine fault tolerance was invented (in the SIFT project [131]) for arbitrary failures that behave more in a random fashion than as if being controlled by hackers. The stronger assumption still fits those failures, while it reduces the replication cost, which in turn improves the latency and throughput of the system.

The chapter is organized as follows. First we describe a model of a distributed system in Section 5.1 and the problem we want to solve in Section 5.2. Then we introduce our solution in Section 5.3. The next few sections deal with the details of our solution:

- Section 5.4 develops a solution.
- Section 5.5 proves the correctness of the solution.

[1] N-version programming approaches such as BASE [54] can help, but they do not scale.

- Section 5.6 enhances the solution in practice.
- Section 5.7 evaluates the solution.

Finally, Section 5.8 closes the chapter.

5.1 A Distributed System Model

A *distributed system* comprises a set of N server agents that implement a *distributed application* and a set of client agents. The agents communicate with one another by message passing over a reliable FIFO network. Each server agent, or server for short, implements a *deterministic state machine*,[2] specified in Figure 5.1. For simplicity of exposition, the state of a server p includes the full history of requests made to that server (called $Hist^p$). In practice implementations would likely maintain only a compact *running* state. Each server p also maintains $InputSet^p$, the set of requests that the server has received, and a set of output messages $OutputSet^p$. A message is a triple $\langle src, dst, payload \rangle$ of its source agent *src*, its destination agent *dst*, and a payload.

The specification also lists *transitions*. Each transition gives a *predicate*, and a list of *actions* to be taken when the predicate becomes true. In S1T1 in Figure 5.1, when a new request message r arrives at a server p, the predicate $r \in InputSet^p \wedge r \notin Hist^p$ becomes true. Server p performs three actions. First, it produces the output messages by evaluating $\texttt{execute}^p(r, Hist^p)$. Second, it puts the output messages into the output set. And third, it appends the request to the history. Note that $\texttt{execute}^p(.,.)$ is the application function for server p. Once a message

[2]Non-determinism, such as random values and clock values, can be modeled as inputs to deterministic transitions.

State:
$\mathit{Hist}^p, \mathit{InputSet}^p, \mathit{OutputSet}^p$ // initially empty

Transitions:

S1T1. On $r \in \mathit{InputSet}^p \wedge r \notin \mathit{Hist}^p \rightarrow$
$\mathit{results} := \texttt{execute}^p(r, \mathit{Hist}^p)$
$\mathit{Hist}^p := \mathit{Hist}^p :: r$
$\mathit{OutputSet}^p := \mathit{OutputSet}^p \cup \mathit{results}$

S1T2. On $m = \langle p, \mathit{dst}, \mathit{payload} \rangle \wedge$
$m \in \mathit{OutputSet}^p \wedge m \notin \mathit{InputSet}^{\mathit{dst}} \rightarrow$
$\mathit{InputSet}^{\mathit{dst}} := \mathit{InputSet}^{\mathit{dst}} \cup \{m\}$

Figure 5.1: A high-level specification of a server p.

is in the output set, it is eventually added into the destination's input set by the transition S1T2 in Figure 5.1. This models message transmission across the network.

The distributed system is *asynchronous*: there is no bound on how long it takes before a continuously enabled transition is performed.

5.2 Problem Statement

Given a set of N distributed application functions $\texttt{execute}^1(.,.)$, $\texttt{execute}^2(.,.)$, ..., $\texttt{execute}^N(.,.)$, we are interested in the design and implementation of a *correct*, *low-cost*, and *high-throughput* distributed system where each server implements the specification in Figure 5.1 and works in the following settings.

The system supplies an abundant number of *processes*, which are deterministic state machines. They are units of failure. A process can be *correct*, when it faithfully follows the protocols' steps, or *failed*. When a process fails, it can behave *arbitrarily*, or in a *Byzantine* way. For instance, it can stop, omit send-

ing/receiving messages, or produce unspecified messages. However, a Byzantine process is limited in what it can forge. In this chapter, we consider two assumptions, which will lead to different replication and computation costs.

> **Strong Adversary Assumption**: Byzantine processes do not forge MAC signatures on messages sent between two correct processes. Also, they do not forge RSA signatures generated by a correct process.
>
> **Weak Adversary Assumption**: Byzantine processes do not forge CRC checksums generated by a different process and *do not compute CRC checksums on corrupted data*, such as truncated request histories.

The first assumption is frequently stated in the BFT literature. The second assumption is considerably stronger. It models systems where failures are accidental and in a random fashion, such as bit flips and Mandelbugs [79]. In practice, we can implement this assumption by using ECC memory (similar to SafeMem [119]) or keeping multiple copies of the history. When a server receives a request for its history, the server returns the checksum of every copy. As failures are accidental and occur in a random fashion, it is highly unlikely that all copies are incorrectly modified in the same way. Different checksums of a history indicate that the history is corrupted or truncated.

The processes communicate over a *reliable FIFO network*.[3] The transmission of messages between any two correct processes is guaranteed.

The system is *partially synchronous* (as introduced in [68]). It starts with no bounds on the network's transmission time and processes' computational time.

[3] In practice, a reliable FIFO network can be implemented over a *fair* network by sequence numbers and retransmission.

But there exists a *Global Stabilization Time (GST)* [68], after which the network's transmission time and processes' computational time are bounded, all failed processes are detected or suspected, and no more processes fail.

In order to define the correctness of a system, we first define what it means for a message to be *valid*. All messages from clients are assumed to be valid.[4] A message m from a server p is valid if and only if there exists a sequence of valid messages $m_1, m_2, \ldots, m_c$ destined for p such that

$$m \in \texttt{execute}^p(m_c, [m_1, m_2, \ldots, m_{c-1}]).$$

For correctness, each server in the system must implement the specification in Figure 5.1. In addition, each server needs to keep state consistent over time and needs to make progress. More formally, a server p needs to preserve the following properties:

- **Safety.** $Hist^p$, $InputSet^p$, and $OutputSet^p$ must contain only valid requests. Also,
  - If $H = Hist^p$ at time T and $H' = Hist^p$ at time $T' > T$, then H is a prefix of H'.
  - If $I = InputSet^p$ at time T and $I' = InputSet^p$ at time $T' > T$, then $I \subseteq I'$.
  - If $O = OutputSet^p$ observed at time T and $O' = OutputSet^p$ at time $T' > T$, then $O \subseteq O'$.
- **Liveness.** After GST, every transition that is continuously enabled will happen.

[4] We consider Byzantine clients later.

5.3 A Refinement Approach

We approach a solution to our problem by refining the server specification in Figure 5.1 in two steps. The first refinement introduces *attestation* and *verification*, while the second refinement results in an executable replication protocol. We also discuss how to deal with failures in the final refinement of the protocol.

Attestation and Verification

Figure 5.1 specifies how a server should work. It models a server in an ideal environment, where there is no failure. In this first refinement, we still assume that servers do not fail, but they may have *external adversaries* that try to interfere by forging their messages. To protect the servers from external adversaries, we introduce two steps to their specification:

- *Attestation.* Servers attest each message they send by supplying a *validity proof* of the message.
- *Verification.* Servers verify each message they receive by verifying the validity proof associated with the message.

The attestation and verification have the following properties:

- *AV1.* A server p attests a message $m = \langle p, q, \textit{payload} \rangle$ only if m is valid.
- *AV2.* A server q successfully verifies a message $m = \langle p, q, \textit{payload} \rangle$ only if server p has attested m.
- *AV3.* The attestation and verification steps terminate.

State:
$\mathit{Hist}^p, \mathit{InputSet}^p, \mathit{OutputSet}^p, \mathit{Unverified}^p$

Transitions:

S2T1. $\texttt{On}\ (r, \mathit{proof}) \in \mathit{Unverified}^p \wedge r \notin \mathit{InputSet}^p \rightarrow$
$\quad \texttt{if}\ \texttt{verify}^p(r, \mathit{proof})\ \texttt{then}$
$\quad\quad \mathit{InputSet}^p := \mathit{InputSet}^p \cup \{r\}$

S2T2. $\texttt{On}\ r \in \mathit{InputSet}^p \wedge r \notin \mathit{Hist}^p \rightarrow$
$\quad \mathit{results} := \texttt{execute}^p(r, \mathit{Hist}^p)$
$\quad \mathit{Hist}^p := \mathit{Hist}^p :: r$
$\quad \mathit{OutputSet}^p := \mathit{OutputSet}^p \cup \{(m, \texttt{attest}^p(m)) \mid m \in \mathit{results}\}$

S2T3. $\texttt{On}\ \mathit{sm} = (\langle p, \mathit{dst}, \mathit{payload}\rangle, \mathit{proof}) \wedge$
$\quad\quad \mathit{sm} \in \mathit{OutputSet}^p \wedge \mathit{sm} \notin \mathit{Unverified}^{\mathit{dst}} \rightarrow$
$\quad \mathit{Unverified}^{\mathit{dst}} := \mathit{Unverified}^{\mathit{dst}} \cup \{\mathit{sm}\}$

Figure 5.2: First refinement of a server p.

Figure 5.2 presents the refinement of the high-level specification in Figure 5.1 with attestation and verification. Hist^p, $\mathit{InputSet}^p$, and $\mathit{OutputSet}^p$ are as in the high-level specification, except that p attests its output messages (in S2T2) before putting into the output set. Transition S2T2 corresponds to transition S1T1. Also, messages are no longer directly transmitted from output sets to their destinations' input sets as in S1T2. Instead, messages intended for p are first added to a set $\mathit{Unverified}^p$ (in S2T3). Transition S2T3 corresponds to a no-op in the high-level specification. Finally, server p verifies those messages before adding them to its input set (in S2T1). In this case, transition S2T1 corresponds to transition S1T2. But when a message fails the verification, it is ignored. In this case, transition S2T1 corresponds to a no-op in the high-level specification.

Safety and Liveness

Here we informally and briefly argue why properties AV1-3 are desirable for the attestation and verification steps. In particular, AV1-3 preserve Safety and

Liveness when there are external adversaries. The full proof of correctness is deferred until the second refinement.

Liveness is preserved because the attestation and verification steps are non-blocking (by AV3) and because the network is reliable. Safety is preserved first because each server p only adds a request into its input set and history (S2T1 and S2T2) after the request has been verified (S2T1). AV2 ensures that p successfully verified the request only if the sender of the request has attested it. This happens only if the request is valid, as AV1 guarantees. Second, Safety is preserved because a server p only adds a request into its output set after it has attested the request. AV1 ensures that the output set contains only valid requests. And third, Safety is preserved because no transition takes requests out of any input set, output set, or history.

5.4 The Shuttle Replication Protocol

So far we have assumed that servers do not fail. This assumption is not realistic, especially in a large scale distributed environment. In this section we refine the specification in Figure 5.2 so that the system tolerates server failures.

We will first describe our approach at a high level. Then we will discuss each aspect of the approach in detail, starting with a reasoning about how to replicate a server at a minimal cost. The discussion includes a refinement of the specification (Figure 5.2) and the component protocols.

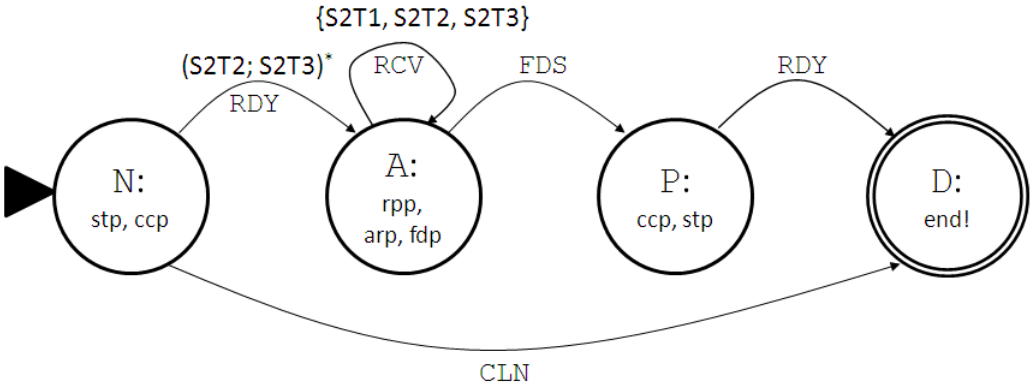


Figure 5.3: Overview of the Shuttle approach. `N`, `A`, `P`, and `D` are states of a configuration; and stp, arp, fdp, and ccp are protocols that run in the states. `RDY`, `RCV`, `FDS`, and `CLN` are events input to the states. Transitions are labeled S2T1, S2T2, and S2T3 if they correspond to those spec transitions (in Figure 5.2); otherwise, they correspond to no-op.

5.4.1 Overview

Various types of failure may occur. Some failures, such as crashes, are impossible to detect accurately in an asynchronous system. We can at best suspect them when the system becomes unresponsive. Some other failures we can readily detect and pinpoint to a particular process. Yet other failures we can detect, but we cannot prove which process is the culprit. For the Shuttle approach, we categorize failures as *provable* and *unprovable*. When a process detects a failure, if it can compile a *proof of misbehavior* [36] to convince other processes about the failure, then the failure is provable.[5] Other failures detected or suspected are unprovable.

We implement each server by a group of processes, so that the application

[5]We do not discuss how to detect failures and make them provable. Interested readers may refer to PeerReview [81] for details.

state persists over process failures. The approach is known as *state machine replication* [93, 121]. Each server p becomes a *replicated server* (or *group*), denoted by $\mathcal{G}_p$, that contains multiple processes playing different roles. The set of processes, an assignment of roles (e.g., replica, witness, leader, etc.), and the value of t (which can differ from group to group) are called a *configuration* of the replicated server. Each configuration has a unique *configuration identifier* (CID). A replicated server replaces its configuration to overcome failures. A configuration goes through four states (not to be confused with state of the application): `N`, `A`, `P`, and `D`. The encoding of a configuration is called a *configuration representation*. For example, a configuration representation may include a CID, a set of process identifiers, a role assignment, and a value for t. Each replicated server may have its own size (i.e., different t value). But for simplicity of the discussion, we will let n be the size of a replicated server unless explicitly stating otherwise.

Figure 5.3 gives an overview of our approach. A configuration begins in a *new* state (state `N`) and constructs its application state using a *state transfer protocol* (*stp*). After transferring the state, the new configuration becomes *ready* (event `RDY`) and makes a transition to an *active* state (state `A`). The transition refines a sequence of zero or more pairs of transitions S2T2 and S2T3. In state `A`, the configuration processes messages it *receives* (event `RCV`), using a *request processing protocol* (rpp). Most transitions S2T1, S2T2, and S2T3 occur during request processing. While being active, the configuration also runs an *acknowledgment-retransmission protocol* (*arp*) to prevent omission, and a *failure detection protocol* (*fdp*) to monitor its own health. When the failure detection protocol *detects or suspects a failure* (event `FDS`), the configuration makes a transition to a *passive* state (state `P`). In this state, the configuration does not modify its application state. It triggers a *change-of-configuration protocol* (*ccp*) to determine a new configuration

for the replicated server, and it serves state-transfer read-only requests for transferring the application state to the new configuration. Note that a new configuration may have failed processes before it becomes `active`. In this case, the current configuration will repeat the change-of-configuration protocol to determine another configuration. When a new configuration becomes ready (event `RDY`), the change-of-configuration protocol issues a `clean` command (event `CLN`) to terminate all new configurations that have failed to become active. Finally, the current configuration and those that received a `clean` command make a transition to a *done* state (state `D`)—where the processes terminate.

5.4.2 Replication

Traditionally, a replicated server would need $3t + 1$ replicas to tolerate t Byzantine failures [117]. Later work [134] separates agreement from execution, reducing t of the replicas to *witnesses*. Witnesses do not hold application state nor perform application operations. And thus they can run on cheaper machines. Still, overall a replicated server requires $3t + 1$ members. This cost is high, especially in large scale distributed systems.

In order to reduce the replication cost, it is important to understand the rationale behind traditional approaches. A replicated server requires $3t+1$ members, because it is designed to operate uninterruptedly up to t failures. Informally, we show why this is the case. For a formal treatment, see [47]. Any coordination protocol must synchronize no more than $n - t$ replicas in each step. Requiring a higher number will lead to blocking when t failures occur. Because some correct members may lag behind others, only $n - 2t$ correct members are guar-

anteed to have the most up-to-date state necessary for recovery from failures. In a Byzantine environment, some necessary but *unsignable* information[6] (such as the recency of state) needs to be recovered from at least $t+1$ members to ensure its authenticity. Thus, $n - 2t \geq t + 1$, or $n \geq 3t + 1$.

We observe that the design of Byzantine fault tolerant systems is rooted in an effort to build aircraft control systems [131]. In such an environment, masking failures is important as the operation of an aircraft control system must not be interrupted. However, when applying the same technique to asynchronous systems, such as online services, the high replication cost is unjustified and unnecessary. Even worse, the high replication cost is amplified with the system size. *In large scale asynchronous systems, it is more important to reduce the replication cost than masking failures on the fly.*

By trading off the failure masking capability in traditional state machine replication approaches, we save t replicas in each replicated server. A replicated server synchronizes *all* its members in every step. When some members fail, the replicated server will block and needs to be reconfigured. But reconfigurations are indistinguishable from delays in an asynchronous environment. Because a replicated server synchronizes all members, at least $n-t$ correct members will have the most up-to-date state for recovery. By similar reasoning, we need $n - t \geq t + 1$, or $n \geq 2t + 1$.

Among the $2t + 1$ members, $t + 1$ are replicas and t are witnesses. Recall that for simplicity we represent the application state a server p maintains as $Hist^p$. To recover $Hist^p$, we can first find the length of the most recent history, then find each request within that length. The length of the most recent history

[6]We informally call a piece of information D *unsignable* if there is no mapping $f()$ such that if $S = f(D)$ at time T, then $S = f(D)$ at any time $T' > T$.

Approaches	#replicas	#witnesses	Total
Traditional BFT SMR (e.g., PBFT [53])	$3t+1$	0	$3t+1$
Separating Agreement from Execution [134]	$2t+1$	t	$3t+1$
Shuttle w/ Strong Adversary	$\boldsymbol{t+1}$	$\boldsymbol{t}$	$\boldsymbol{2t+1}$
Shuttle w/ Weak Adversary	$\boldsymbol{t+1}$	$\boldsymbol{0}$	$\boldsymbol{t+1}$

Table 5.1: Replication cost in BFT state machine replication approaches. Bold entries high-light the approaches in this chapter.

is unsignable information and needs to have $t+1$ members vouching for it. Thus all $2t+1$ members need to store the length of the most recent history. The requests, on the other hand, can be signed. And so they may be recovered from a single member. Thus, the replicas keep a copy of $Hist^p$, while the witnesses only keep the length of the most recent history.

We can reduce this cost further under the weak adversary assumption. Exploiting the weak adversary assumption that processes do not truncate their request histories, we eliminate the witnesses. As replicas, even faulty, do not truncate their histories, the recovered history is the most recent one. Thus under the weak adversary assumption, a replicated server comprises only $t+1$ replicas.

Table 5.1 summarizes the replication cost of representative traditional approaches and Shuttle. All the listed approaches are BFT.

5.4.3 Digital Signatures and Proofs

Shuttle employs proofs to ensure that each protocol step only takes place if its precondition holds. Each proof is constructed from digital signatures of a collection of processes. We use d_i to denote some data d signed by a process i. If pro-

Type (Protocol)	**Precondition**
Validity (rpp)	the message is valid
Acceptance (rpp)*	other members consider the proof valid
Order (rpp)	no correct members process the message in a different order
Acknowledgment (arp)	the receiver has received and processed the message
Checkpoint (gcp)*	the checkpoint is recoverable
Misbehavior (fdp)	the process has failed
Failure (ccp)	configuration is to be replaced
Ready (ccp)	configuration is ready

Table 5.2: Summary of proofs. * These proofs are used in the optimized request processing protocol (Section 5.6).

cess i is correct, then it is the only one that produces d_i. We use $\langle S, \{d_i \mid i \in S\}\rangle$ to denote a *proof* created by processes in S for some data d. We call processes in S the *signers*.

Validity proofs, introduced in Section 5.3, are a type of proof. A correct member checks a validity proof $\langle \mathcal{G}_p, \{m_i \mid i \in \mathcal{G}_p\}\rangle$ for the precondition that m is valid. Other types of proof will be introduced when necessary. Table 5.2 summarizes them for readability.

5.4.4 Data Structures and State Refinement

Each correct replica i in a replicated server $\mathcal{G}_p$ maintains a history copy $Hist_i^p$, containing *order proofs* $\langle \mathcal{R}_p, \{\langle r, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ where $\mathcal{R}_p$ is the set of replicas in $\mathcal{G}_p$, r is an input request, and c is an ordering number that is the index into the

history. It also maintains an order counter ord_i^p (initially 0), and a variable $state_i^p$ that can assume a value in $\{\texttt{new}, \texttt{active}, \texttt{passive}, \texttt{done}\}$.

Each correct witness i in a replicated server $\mathcal{G}_p$ maintains a variable $latest_i^p$, which stores the latest order proof that the witness has seen, and a variable $state_i^p$ that can assume a value in $\{\texttt{new}, \texttt{active}, \texttt{passive}, \texttt{done}\}$.

To keep the state refinement concise, we use the following terms. A history copy is *consistent* if (1) it has only order proofs of valid requests, and (2) the ordering numbers in the order proofs form a continuous sequence of integers starting from 0. A replica i (or $Hist_i^p$) *vouches* for a prefix H of a history copy if $Hist_i^p$ is a prefix of H. A witness j (or $latest_j^p$) *vouches* for a prefix H of a history copy if $latest_j^p$ appears in H.

The state of a server p in Figure 5.2 is refined as follows:

- $Hist^p$ is defined as the shortest prefix H of a history copy among all consistent $Hist_i^p$ such that there are $n - t$ correct members vouching for H.[7]
- $Unverified^p$ is defined as the set of requests that any member of $\mathcal{G}_p$ received;
- $InputSet^p$ is defined as the set of requests that have passed the verification step run by $\mathcal{G}_p$ (defined in Section 5.4.5);
- $OutputSet^p$ is defined as the set of requests that have passed the attestation step run by $\mathcal{G}_p$ (defined in Section 5.4.5).

The state of a configuration of a replicated server $\mathcal{G}_p$ is defined as follows:

- *new*: if every member i has $state_i^p = \texttt{new}$;

[7] We ignore the difference between a history of requests and a history of order proofs that contain requests.

- *active*: if at least one member i has $state_i^p$ = `active`, and every member j has $state_j^p$ be either `new` or `active`;
- *passive*: if at least one member i has $state_i^p$ = `passive`, and no member j has $state_j^p$ = `done`;
- *done*: if at least one member i has $state_i^p$ = `done`.

As indicated in Figure 5.3, when the state of a member is `new`, the member participates in the state transfer and change-of-configuration protocols; when the state is `active`, the member participates in the request processing, failure detection, and acknowledgment-retransmission protocols; when the state is `passive`, the member participates in the change-of-configuration and state transfer protocols; and when the state is `done`, the member terminates.

5.4.5 Transition Refinement and Request Processing Protocol

The specification in Figure 5.2 contains three transitions to be refined (italic verbs denote actions):

- S2T1: replicated server $\mathcal{G}_p$ *verifies* a request and *adds* it to $InputSet^p$ if successful;
- S2T2: replicated server $\mathcal{G}_p$ *executes* a request in $InputSet^p$, *appends* the request to $Hist^p$, and *attests* and *adds* the results to $OutputSet^p$;
- S2T3: a request from $OutputSet^p$ is *added* to $Unverified^q$, where q is the destination of the request.

Note that each transition may comprise more than one action. Actions in the same transition need to appear atomic. Also, attestation and verification need to preserve properties AV1-3 defined in the first refinement. And finally, these actions are performed by replicated servers. We call them *group actions*.

In this section we present a request processing protocol that refines the transitions in the common case, when there is no failure. The request processing protocol defines the group actions based on *individual actions*, which are actions performed by individual replicas or witnesses. For readability, we highlight actions to be defined in bold, and protocols in bold and italic. When the context is clear, we simply mention the actions, without "group" or "individual". Without loss of generality, we define actions performed by a replicated server $\mathcal{G}_p$.

The simplest actions are adding a request to a set. By the definitions of $InputSet^p$ and $OutputSet^p$, **adding a request** (to $InputSet^p$ or to $OutputSet^p$) is done automatically by verification or attestation, respectively. **Adding a request in $OutputSet^p$ to $Unverified^q$**, where q is the destination of the request, is having a member of $\mathcal{G}_p$ send the request to a member of $\mathcal{G}_q$.

Verification of a request m from $\mathcal{G}_q$ is defined as the first correct *individual verification* of the request—that is, the first correct member accepts that the request comes with a validity proof $\langle \mathcal{G}_q, \{m_i \mid i \in \mathcal{G}_q\} \rangle$. The validity proof is, similar to the first refinement, to enforce the precondition that the message is valid.

The actions in S2T2 (executing, appending an input request, and attesting an output message), require correct members to produce consistent results and state changes. Because $\texttt{execute}^p(.,.)$ is deterministic, this requirement trans-

lates to a precondition that no correct members process the input message in a different order. We enforce this precondition by having each correct member check an *order proof* (defined below) before processing any m. The following ordering protocol orders input requests and produces order proofs. Ordering is a no-op in the specification.

The ***ordering protocol*** follows the two-phase commit approach [99]. [8] A replica, say l, is designated as a leader, which will receive and order the requests. When l receives a request m, it sends $\langle m, ord_l^p \rangle$ to the other replicas, asking them to agree on the assigned order of m, and increments ord_l^p. When a replica i receives $\langle m, c \rangle$ with a valid m and an ordering number $c = ord_i^p$, replica i agrees with the assigned order of m. It then signs the ordered request, increments ord_i^p, and returns the signed and ordered request to the leader. The leader compiles an *order proof*, which is a tuple $\langle \mathcal{R}_p, \{\langle m, c \rangle_i \mid i \in \mathcal{R}_p\} \rangle$. Note that since correct replicas increment their order counters every request, no two order proofs of different requests share the same ordering number.

For brevity we say that an order proof $\langle S, \{\langle m_1, c_1 \rangle_i \mid i \in S\} \rangle$ is *newer* than an order proof $\langle S, \{\langle m_2, c_2 \rangle_i \mid i \in S\} \rangle$ if $c_1 > c_2$, and vice versa (for *older*). Similarly, $\langle S, \{\langle m_1, c_1 \rangle_i \mid i \in S\} \rangle$ is *at least as new as* $\langle S, \{\langle m_2, c_2 \rangle_i \mid i \in S\} \rangle$ if $c_1 \geq c_2$, and vice versa (for *at least as old as*).

Execution of a request m is defined as that the replica i, such that $Hist_i^p$ is the shortest consistent history copy that is vouched for by $t + 1$ correct members, evaluates the application function $\texttt{execute}^p(m, Hist_i^p)$ to produce the results.[9] We say that replica i *individually executes* m. Note that each replica that receives

[8] Unlike traditional approaches, we cannot use consensus protocols, such as [94, 57, 42], to order requests, because using them requires each replicated server to have $3t + 1$ members [97].

[9] Note that correct replicas execute m in the same order and produce the same result. By defining this way, we match the history-copy parameter with $Hist^p$.

an order proof $\langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ will individually execute m if it has executed all requests with a lower ordering number.

Appending a request m into *Hist*^{p} at index c is defined by one correct replica i enqueueing an order proof *oproof* $= \langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ into *Hist*$_i^p$ and $t+1$ correct members j keeping *oproof* in *Hist*$_j^p$ (if replica), or in *latest*$_j^p$ (if witness). We say that the members *individually append* m. We require that (1) replicas and witnesses only do so when they have individually appended every message with an older order proof, and (2) witnesses only append m after replicas have done so.

Attestation of an output m is defined by each member individually attesting m. A replica i *individually attests* m by signing m if (1) it receives both m and the order proof *oproof* from which m was produced, (2) it has produced m, and (3) it has *oproof* in *Hist*$_i^p$. A witness j *individually attests* m by signing m if (1) it receives both m and the order proof *oproof* from which m was produced, and (2) *latest*$_j^p$ holds *oproof* or a newer order proof. If all members are correct, $\mathcal{G}_p$ will produce a validity proof for m after the attestation finishes. Note that:

- The signatures from the $t+1$ replicas are to ensure that the request is produced by a correct replica—and so, it is valid. We will prove this claim in Section 5.5.
- The additional signatures from the witnesses are to ensure that m is reproducible, when there are failures in the group, by making the causal past [93] of m recoverable after a validity proof of m is formed.

Put together, the ***request processing protocol*** is as follows. $\mathcal{G}_p$ elects a leader among the replicas. In the common case, when there are no failures, the leader

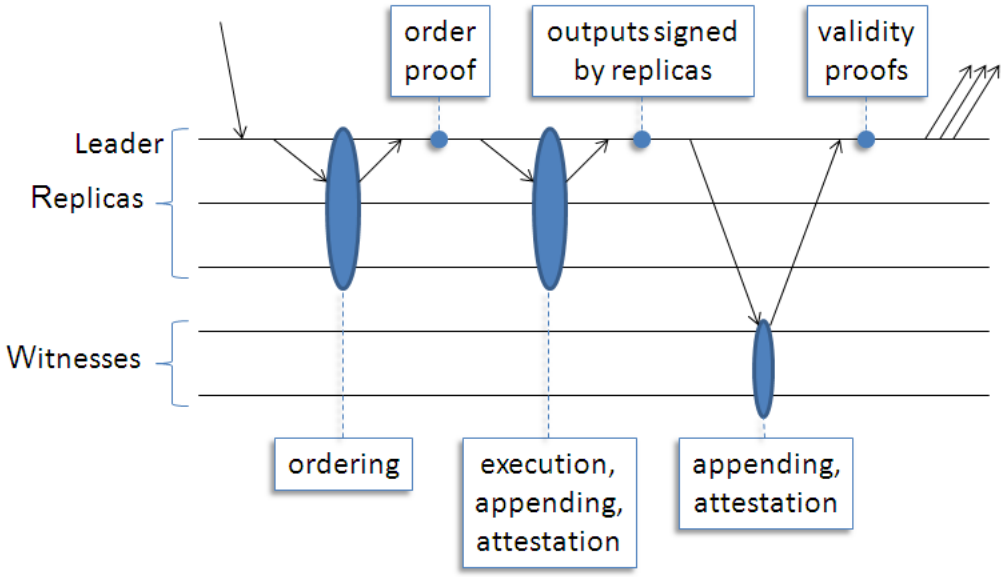


Figure 5.4: The request processing protocol for $t = 2$. The witnesses only exist under the strong adversary assumption.

receives requests and initiates the request processing protocol. When the leader receives a request r, it verifies r. If verification succeeds, the leader orders r to form an order proof $\textit{oproof} = \langle \mathcal{R}_p, \{\langle r, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$. The leader executes r and produces a set of output messages M. It sends r, M and *oproof* to all replicas. Each replica verifies r, executes r (if verification succeeds), appends *oproof* to its history copy, attests each output message in M, and returns the signed output messages to the leader. The leader compiles M', which is a set of messages in M, each being signed by all replicas. The leader then sends *oproof* and M' to all witnesses. Each witness stores *oproof* (i.e., appends r) and attests the messages in M' if they are signed by all replicas. Note that this check is to ensure that witnesses only append r after all replicas have done so. The witnesses then return the output messages in M', each has been signed, to the leader. The leader compiles a validity proof for each output message and sends it to the destination.

Figure 5.4 illustrates the request processing protocol. Note that while execution, appending and attestation interleave, they appear atomic because of the local order of performing the individual actions. The use of outputs attested by the replicas in the appending action at the witnesses is only an optimization that enforces a precondition (witnesses only append a request after replicas have done so) while avoiding an additional type of proof.

5.4.6 Configuration Management

A replicated server deals with failures by changing its configuration. If a failure is provable and $t \geq 1$, we either need to remove the failed member from the group and decrement t or need to replace the failed member by a new process. In case of unprovable failures, we have to replace a subset of members that have been suspected of failing. Configurations are managed by a BFT service called *Olympus*, denoted $\mathcal{O}$. Note that $\mathcal{O}$ is also a group of processes.

Olympus is responsible for assigning processes to each replicated server initially. When a failure occurs, Olympus is also responsible for assigning a new configurations to the failed group as long as it has t or fewer failed members. More formally, Olympus

- maintains a list of processes available in the system,
- maintains a map from *replicated server identifiers (RSIDs)* to configuration representations,
- implements an *Olympus function* that computes, signs, and outputs a configuration representation for an RSID, a CID, and optionally a proof of misbehavior.

Note that Olympus needs to verify that an input comes from the replicated server with the given RSID.

Our *configuration management mechanism* follows the publish/subscibe approach [114]. Olympus manages the configurations of all replicated servers. Each replicated server subscribes to its neighbors' configuration representations, posted by Olympus. A member may publish a *failure detection or suspicion* event (`FDS`) when this event happens in the failure detection protocol (Section 5.4.7). We call such a publication a *failure publication*. Also, members of a replicated server may publish their state recovered from the state transfer protocol. We call this publication a *ready publication*. Upon receiving a publication, Olympus verifies that the publication is correctly signed (i.e., by one member in a failure publication and by all members in a ready publication, and all signatures are valid). For a ready publication, Olympus also verifies that the members recover the same state. Upon success, Olympus creates a *failure proof* $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ RSID, CID\rangle_i \mid i \in \mathcal{O}\}\rangle$ for a failure publication, or a *ready proof* $\langle \mathcal{O}, \{\langle \texttt{READY}\ RSID, CID\rangle_i \mid i \in \mathcal{O}\}\rangle$ for a ready publication and propagates them to all their subscribers through their leaders. A *`RDY` event* occurs when Olympus propagates a ready proof.

5.4.7 Failure Detection Protocol

A member runs the *failure detection protocol* when it is `active`. The member monitors messages and raises a detection or suspicion (`FDS`) upon an unexpected event. In many cases, failures are unprovable. The lack of a proof of misbehavior enables any process (possibly Byzantine) to raise an `FDS` event.

We contain that malicious behavior by limiting that *a process can only publish an* `FDS` *event of its own group*. Two types of unexpected events may happen:

1. Corrupted messages: A message that has been authenticated to have been sent from a sender but its contents is not what it should be. For example, a message carrying a malformed order proof and verified to have been sent by the leader indicates that the leader has failed. Note that a message that cannot be authenticated does not raise a detection or suspicion, as it may have been sent from an attacker in the network.
2. Lost messages: A member expects some particular messages in each step of the protocol. When the member times out on waiting for a message, it raises a suspicion. For example, during the processing of a message m,
   - a member can time out on waiting for an order proof $\langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ after it has signed $\langle m, c\rangle$;
   - a leader can time out while collecting signatures for an order proof $\langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$;
   - and so on...

When a member i of $\mathcal{G}_p$ raises an `FDS` event, it changes $state_i^p$ from `active` to `passive`. The replicated server progresses no further. And eventually other members will change their state from `active` to `passive` as well.

5.4.8 Acknowledgment-Retransmission Protocol

We devise an *acknowledgment-retransmission protocol* for two purposes. First, it is to prevent omission, which is caused by malicious leaders. And second, it is to enforce the following property (for Liveness):

- *AR*. If $\mathcal{G}_p$ sends m to $\mathcal{G}_q$, then either m is in $Hist^q$ or $\mathcal{G}_p$ will resend m to $\mathcal{G}_q$.

Suppose, without loss of generality, that $\mathcal{G}_p$ in configuration S is sending m to $\mathcal{G}_q$ in configuration D. The protocol runs on both sender and receiver sides.

On the sender side, each correct member sets a timer when it (individually) attests m. The member waits for either an *acknowledgment proof*, $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m \rangle_i \mid i \in \mathcal{G}_q\} \rangle$, or a failure proof $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ q, D \rangle_i \mid i \in \mathcal{O}\} \rangle$. The acknowledgment proof ensures that *the receiver has received and processed the message* (a precondition) before the sender stops sending m.

An active $\mathcal{G}_p$ forces $\mathcal{G}_q$ to return either an acknowledgment proof or a failure proof by retransmission: The leader of $\mathcal{G}_p$ sets a timer waiting for $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m \rangle_i \mid i \in \mathcal{G}_q\} \rangle$ after sending m. When it times out, the leader broadcasts m to $t+1$ members in $\mathcal{G}_q$. Then the leader waits again (but without timeout) for either $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m \rangle_i \mid i \in \mathcal{G}_q\} \rangle$ or $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ q, D \rangle_i \mid i \in \mathcal{O}\} \rangle$. At least one correct member among the $t + 1$ will either acknowledge the message or publish an `FDS` event (details below).

If the leader of $\mathcal{G}_p$ receives an acknowledgment proof $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m \rangle_i \mid i \in \mathcal{G}_q\} \rangle$ or a failure proof $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ q, D \rangle_i \mid i \in \mathcal{O}\} \rangle$, it will forward the proof to the members. They will cancel their timer upon receipt.

If $\mathcal{G}_p$ receives $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ q, D\rangle_i \mid i \in \mathcal{O}\}\rangle$, it will wait for a ready proof $\langle \mathcal{O}, \{\langle \texttt{READY}\ q, D'\rangle_i \mid i \in \mathcal{O}\}\rangle$ (D' is the configuration identifier of a new configuration of $\mathcal{G}_q$) and retry sending m as if it is sending m the first time.

If a member in $\mathcal{G}_p$ times out before receiving $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m\rangle_i \mid i \in \mathcal{G}_q\}\rangle$ or $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ q, D\rangle_i \mid i \in \mathcal{O}\}\rangle$, it suspects that a failure has occurred in its own group—for instance, the leader omitted sending the output. The member triggers an `FDS` event. After a new configuration S' becomes active, the new leader re-initiates the attestation for each output and resends the outputs with their validity proofs to the destinations.[10] If a destination has already processed a request, it will just acknowledge the request without executing it.

On the receiver side, if $\mathcal{G}_q$ is active, its leader will create $\langle \texttt{ACK}\ m\rangle$ when executing m, collect signatures from the members during the appending step, and return $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m\rangle_i \mid i \in \mathcal{G}_q\}\rangle$ to the sender's leader.

When a member receives m, it individually verifies the request. If m passes the member's verification, the member forwards m to the leader and sets a timer. The leader handles the request as if it has come directly from $\mathcal{G}_p$ until the last step. Instead of sending $\langle \mathcal{G}_q, \{\langle \texttt{ACK}\ m\rangle_i \mid i \in \mathcal{G}_q\}\rangle$ to $\mathcal{G}_p$, the leader forwards it to all the members from which it received m. Those members will cancel their timer and forward the acknowledgment proof to the leader of $\mathcal{G}_p$. If, instead, a member times out, it will publish an `FDS` event.

[10] For efficiency, one may want to keep track of the outputs that have not been acknowledged and only re-attest and resend those.

5.4.9 State Transfer Protocol

The *state transfer protocol* is to transfer the application state (i.e., the history) from a passive configuration to a new configuration, such that the following property is preserved (for Safety):

- *ST.* If H is a history of $\mathcal{G}_p$ at time T, then H is *recoverable* at any time $T' > T$—that is, the state transfer protocol will read a history H' out of the configuration of $\mathcal{G}_p$ at time T' such that H is a prefix of H'.

Without loss of generality, suppose the current and passive configuration is C, and the new configuration is C'. The leader in C' queries the state of all members in C. Replicas in C reply with their history copy. Each witness in C replies with the last order proof it has witnessed. The leader in C' waits for $n-t$ pair-wise *non-conflicting* responses and picks the longest history copy among them. Two responses are *non-conflicting* if (1) they have no history copy that is not consistent, (2) they do not contain any two order proofs of the same request but different orderings, and (3) a history copy (response from a replica) must be vouched for by a latest order proof (response from a witness). The longest history copy is the state of the new configuration.

The leader propagates the $n - t$ non-conflicting responses to all members. From the $n - t$ non-conflicting responses, the replicas derive the new history, and the witnesses derive the latest order proof. Each member computes its signature on the hash of the $n - t$ responses and returns the signature to the leader in a `ready` message. The leader publishes the hash and the signatures from all members in a ready publication. The hash and the signatures ensure that Olym-

pus only sends a ready proof after a all members in a new configuration have started from the same state.

5.4.10 Change-of-Configuration Protocol

The *change-of-configuration protocol* links the failure detection and state transfer protocols together to help a replicated server deal with failures. The protocol starts in a passive configuration and produces a new configuration. Suppose a replicated server $\mathcal{G}_p$ that is in a configuration C triggers the protocol:

- A member (called a *detector*) publishes an `FDS` event by sending Olympus its RSID p, CID C, and optionally a proof of misbehavior. Olympus verifies the data to make sure that the publication comes from a process in configuration C. It then creates and propagates a failure proof $\langle \mathcal{O}, \{\langle \texttt{FAILURE}\ p, C\rangle_i \mid i \in \mathcal{O}\}\rangle$ to all subscribers of $\mathcal{G}_p$.
- Based on p, C and optionally a proof of misbehavior submitted by the detector, Olympus allocates processes for a new configuration C'. It signs and sends the representation of configuration C' to the leader of configuration C'.
- Configuration C' transfers the state from configuration C using the state transfer protocol with an additional detail. The leader of C' passes the configuration presentation of C' (signed by Olympus) to the other members along with the recovered state (i.e., the $n-t$ non-conflicting responses from C). Each member in configuration C' verifies the configuration presentation to ensure it is in C' before adopting the state.

- When Olympus gathers all n `ready` messages from C' that contain the same hash value, Olympus creates and propagates a `ready` proof $\langle \mathcal{O}, \{\langle \texttt{READY}\ p, C' \rangle_i \mid i \in \mathcal{O}\} \rangle$ to all $\mathcal{G}_p$'s subscribers, who will then resume normal operation of the application. When Olympus sends a `ready` proof, we say that a *`RDY` event occurs*.
- Olympus sets a timer after computing configuration C'. If it times out before receiving a `ready` publication, it will allocate another configuration and repeat the change-of-configuration protocol. When Olympus propagates a `ready` proof, it also cleans up configurations that have failed to become ready by sending a `clean` command to all processes in that configuration. We say that a *`CLN` event occurs*. The processes that receive a `CLN` command will terminate.

5.4.11 Timeouts

We use different timeouts for different purposes, and they need to be set properly, otherwise the system will be unable to make progress. In particular, the leader of a sender group has a *retransmission timeout* Δ_T after which it broadcasts the un-acknowledged request. Each member in a receiver group that receives a retransmitted request maintains a *receive-omission timeout* Δ_R after which it publish a failure suspicion event. And each member in a sender group maintains a *send-omission timeout* Δ_S after which it publishes a failure suspicion event. When the system is asynchronous, we cannot guarantee any timing. But after GST, we need to ensure that the leader allows sufficient time for the receiver to process a request and return an acknowledgment. Also, a receiving member in a receiver group must allow enough time for its group to process the request and return

an acknowledgment. And a member in a sender group must allow enough time for the leader to retry and for the receiver group to publish an failure suspicion event.

After GST, if we let

- δ_P be the *request processing time*, from when a process (leader of a sender group or a receiving member of a receiver group) sends a request until it receives an acknowledgment;
- δ_F be the *failure publication time*, from when a process publishes an `FDS` event until all interested processes receive a failure proof;
- δ_A be the *attestation time*, from when a member in the sender group individually attests an output until the output leaves the sender group;

then we need

$$\Delta_T > \delta_P \tag{5.1}$$

$$\Delta_R > \delta_P \tag{5.2}$$

$$\Delta_S > \Delta_T + \Delta_R + \delta_F + \delta_A + \epsilon \tag{5.3}$$

where ϵ accounts for the time from when the retransmission timeout Δ_T expires (at leader) until $t + 1$ members in the receiver group receive the request.

When equations 5.1, 5.2, and 5.3 hold, the following property holds:

- *TO.* After GST, there will be no failure suspicion or detection in $\mathcal{G}_p$, unless some members of $\mathcal{G}_p$ itself fail.

5.5 Correctness

We need to show that Shuttle preserves Safety and Liveness even when there are failures. Throughout the proof we assume that *each replicated server contains at most t failed members at any time*.

5.5.1 Safety.

Lemma 5.5.1. *Correct replicas produce valid messages.*

Proof. We will show the lemma by induction on the number of messages produced by correct replicas (of all replicated servers).

Base case: We need to show that if m is the first message produced by a correct replica, then m is valid.

Suppose m is produced by a replica of a replicated server $\mathcal{G}_p$ upon an input m'. Because m is the first message produced by a correct replica, m' cannot be produced by any correct replica. m' cannot be produced by a faulty replica, either, because in such case m' would not have enough signatures to form a validity proof and would fail the verification. Thus, m' must be an external input.

By assumption, all external inputs are valid. And since m' is valid, $m = \texttt{execute}^p(m', [])$ is valid by definition.

Induction hypothesis (IH): Suppose the first c messages produced by correct replicas are valid, where $c \geq 1$.

Inductive case: We need to show that the $(c+1)^{\text{th}}$ message is valid. Let us call it m, and suppose that m is produced by replica i of p, based on an input m'. Let H be the history copy of replica i. We claim that H contains only valid messages.

Proof of claim: Suppose not, let m'' be an invalid message in H. By definition, m'' cannot be an external message (*). Before appending m'' into H, replica i must have checked that m'' has a validity proof (**). (*) and (**) imply that some correct replica has produced m''. But this contradicts to the IH, which implies that all messages produced by correct replicas before m is valid. Hence the claim is correct.

There are three cases: m' is external, m' is produced by a correct replica, and m' is produced by a faulty replica but not by a correct one.

- m' is external. As m' is valid and H contains only valid messages, $m = \texttt{execute}^p(m', H)$ is valid by definition.
- m' is produced by a correct replica. Because m' is produced before m, m' is valid by the IH. Then by definition, $m = \texttt{execute}^p(m', H)$ is valid.
- m' is produced by a faulty replica, but not by a correct one. This means correct replicas do not sign m'. And so m' does not have a validity proof to pass the verification. Thus this case cannot happen.

We conclude that m is valid. And by induction, correct replicas produce valid messages. □

Lemma 5.5.2 (AV1)**.** *A replicated server $\mathcal{G}_p$ attests a message $m = \langle p, q, payload \rangle$ only if m is valid.*

Proof. By definition, m is attested only if all members in replicated server $\mathcal{G}_p$ have individually attested m. That is, a correct replica in group $\mathcal{G}_p$ must have produced m. By Lemma 5.5.1, m is valid. □

Lemma 5.5.3 (AV2)**.** *A message $m = \langle p, q, payload \rangle$ is successfully verified only if m is attested by replicated server $\mathcal{G}_p$.*

Proof. This is correct by the construction of verification: verification only succeeds when m has a validity proof $\langle \mathcal{G}_p, \{m_i \mid i \in \mathcal{G}_p\}\rangle$—meaning that m has been attested by replicated server $\mathcal{G}_p$. □

Lemma 5.5.4. *$Hist^p$ contains only valid requests.*

Proof. By definition, $\mathcal{G}_p$ appends a request r into $Hist^p$ only if every correct replica i has individually appended r to $Hist^p_i$.

A correct replica i individually appends r to $Hist^p_i$ only if r comes with an order proof $\langle \mathcal{R}_p, \{\langle r, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$, for some number c. In the ordering instance that produces $\langle \mathcal{R}_p, \{\langle r, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$, each replica individually verifies r before producing a signature for $\langle r, c\rangle$. If r is external, it is valid by definition. If r is from another replica q, it must have been attested by replicated server $\mathcal{G}_q$ by Lemma 5.5.3. And by Lemma 5.5.2, $\mathcal{G}_q$ attests r only if r is valid. □

Lemma 5.5.5. *$InputSet^p$ contains only valid requests.*

Proof. By definition, a request m from a replicated server q is added into $InputSet^p$ if a correct member in group $\mathcal{G}_p$ has individually verified m.

A correct member individually verifies m only if m has a validity proof $\langle \mathcal{G}_q, \{m_i \mid i \in \mathcal{G}_q\}\rangle$. This means a correct replica in group $\mathcal{G}_q$ must have produced and signed m. By Lemma 5.5.1, m is valid. □

Lemma 5.5.6. *OutputSet^p contains only valid requests.*

Proof. By definition, $\mathcal{G}_p$ adds a request m into *OutputSet^p* only if all members of group $\mathcal{G}_p$ has been individually attested m. One of them must be a correct replica.

A correct replica individually attests m only if it has produced m. By Lemma 5.5.1, m is valid. □

Lemma 5.5.7. *If there are two order proofs $\langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ and $\langle \mathcal{R}_p, \{\langle m', c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$, then $m = m'$.*

Proof. There must be a correct replica in $\mathcal{R}_p$ that signs both $\langle m, c\rangle$ and $\langle m', c\rangle$. Because a correct replica i only signs some pair $\langle r, c\rangle$ when $ord_i^p = c$, and it increments ord_i^p after signing, m must be the same as m'. □

Lemma 5.5.8. *If i and j are correct replicas in the same configuration of a replicated server $\mathcal{G}_p$, then either $Hist_i^p$ is a prefix of $Hist_j^p$ or vice versa.*

Proof. By the state transfer protocol, in order for a configuration to become ready all members have to produce a signature of the same initial state ($t + 1$ non-conflicting responses). This means when a configuration becomes active, all correct replicas start from the same state—i.e., $Hist_i^p = Hist_j^p$. During the request processing protocol, a replica enqueue the c^{th} request, say m, into its history only if it receives an order proof $\langle m, c\rangle$. And by Lemma 5.5.7, there is no valid order proofs $\langle m, c\rangle$ and $\langle m', c\rangle$ where $m \neq m'$. Thus replicas i and j cannot enqueue different requests to $Hist_i^p$ and $Hist_j^p$ at the same position. And so the lemma holds. □

For convenience, in the following proofs we will use *correct history* to call a copy of the $Hist^p$ at a correct replica (under the strong adversary assumption) or a copy of $Hist^p$ that contains only valid requests (under the weak adversary assumption).

Lemma 5.5.9 (ST)**.** *If H is a history of $\mathcal{G}_p$ at time T, H is recoverable at any time $T' > T$.*

Proof. We need to show that the state transfer protocol run at time T' will retrieve a history H' such that H is a prefix of H'. Let C be the configuration at time T. It suffices to show that the state transfer protocol retrieves such H' from C. In case the configuration at time T' is not C, the result still holds by transitivity.

Recall that the state transfer protocol retrieves $n-t$ non-conflicting responses from C and lets H' be the longest history copy among the responses. By the definition of non-conflicting, the H' is vouched for by the $n - t$ responses. And by the definition of a history, H is shorter than H'. Finally, by Lemma 5.5.8, H is a prefix of H', as desired. □

Lemma 5.5.10. *If $H = Hist^p$ at time T and $H' = Hist^p$ at time T', where $T' > T$, then H is a prefix of H'.*

Proof. There are two cases: H and H' are from the same configuration, and not.

Case 1: By definition, H is the shortest correct history at time T and H' is the shortest correct history at time $T' > T$. This and Lemma 5.5.8 imply that H is a prefix of any correct history at time T. And since the request processing protocol only enqueues requests into history copies, no correct history is shortened from

time T to time T'. Thus, H is also a prefix of any correct history at time T', and this includes H'.

Case 2: By Lemma 5.5.9, any history value of a configuration is a prefix of the initial history of the next configuration, as it is recovered by the state transfer protocol. Applying this repeatedly over the configurations we have H is a prefix of the initial history of the configuration where H' is read. By this, case 1, and the fact that the prefix quality is transitive, H is a prefix of H'. □

Lemma 5.5.11. *If $I = InputSet^p$ at time T and $I' = InputSet^p$ at time T', where $T' > T$, then $I \subseteq I'$.*

Proof. By definition, $\mathcal{G}_p$ puts a request into $InputSet^p$ after it has verified the request. Also, the protocols do not remove any request out of $InputSet^p$. Thus, $InputSet^p$ always expands, as desired. □

Lemma 5.5.12. *If $O = OutputSet^p$ at time T and $O' = OutputSet^p$ at time T', where $T' > T$, then $O \subseteq O'$.*

Proof. Similar to the proof of Lemma 5.5.11. □

Theorem 5.5.13. *Shuttle preserves Safety.*

Proof. The theorem holds by Lemmas 5.5.4, 5.5.5, 5.5.6, 5.5.10, 5.5.11, 5.5.12. □

5.5.2 Liveness

Recall that the system model we are interested in has the following assumptions:

- *There is a Global Stabilization Time (GST) after which the system is synchronous and no more processes fail.*
- *There are at most t failures that have occurred in any configuration.*

Lemma 5.5.14. *Correct members in a configuration that starts start from the same state.*

Proof. This follows from the state transfer protocol. Before a configuration becomes active, it is required to collect a signature from each member on the same $n - t$ responses from which it computes its state. □

Lemma 5.5.15 (AR)**.** *If a replicated server $\mathcal{G}_p$ sends a valid output message m to $\mathcal{G}_q$, either m is in Hist^q or $\mathcal{G}_p$ will resend m to $\mathcal{G}_q$.*

Proof. A replicated server $\mathcal{G}_p$ only sends m if it has appended r to *Hist*^{p}, where $m = \texttt{execute}^p(r, H)$ for some history value H of *Hist*^{p}. By Lemma 5.5.9, $[r \mid H]$ is recoverable. And so when $\mathcal{G}_p$ is reconfigured, it will re-execute r and resend m. (*)

Now suppose m is not in *Hist*^{q}, we need to show that $\mathcal{G}_p$ will resend m. When m is not in *Hist*^{q}, $\mathcal{G}_q$ must have not acknowledged m. By the acknowledgment-retransmission protocol, $\mathcal{G}_p$ will broadcast m and set a timer. When it times out and $\mathcal{G}_q$ has not acknowledged m, $\mathcal{G}_p$ will be reconfigured, and it will resend m by (*). □

Lemma 5.5.16 (TO)**.** *After GST, there will be no failure suspicion or detection in $\mathcal{G}_p$, unless some members of $\mathcal{G}_p$ itself fail.*

Proof. Suppose that after GST all members in $\mathcal{G}_p$ are correct. And suppose that a member i in $\mathcal{G}_p$ publishes an `FDS` event.

Member i must have timed out after Δ_R, waiting for an acknowledgment from the leader, or after Δ_S, waiting for an acknowledgment from another replicated server, say $\mathcal{G}_p$. It cannot be the former case, as $\Delta_R > \delta_P$ (Equation 5.2) allows enough time for $\mathcal{G}_p$ to process the request and send the acknowledgment to i. It cannot be the latter case, either, as $\Delta_S > \Delta_T + \Delta_R + \delta_F + \epsilon$ (Equation 5.3). In the worst case ($\mathcal{G}_q$ does not return an acknowledgment), Δ_T allows the leader to rebroadcast the request; ϵ allows enough time for the retransmitted request reaches $t+1$ members in $\mathcal{G}_q$; Δ_R allows a correct member in $\mathcal{G}_q$ to publish an `FDS` event, and δ_F allows the failure publication to propagate to member i. Thus, neither case can happen, contradicting the assumption. □

Lemma 5.5.17. *After GST, a reconfiguration triggered by a member will terminate with a configuration containing only correct processes.*

Proof. Reconfiguration involves the following steps: failure publication, configuration assignment, state transfer, and ready publication. Failure publication terminates because sending a message to Olympus terminates after GST. Configuration assignment terminate by the assumption about Olympus. State transfer involves pulling the state from $n - t$ members of a configuration and passing it to other members. The pulling step terminates because there are at most t failures in any configuration. The passing step terminates because the system become synchronous after GST. Finally, ready publication terminates because (1) the system becomes synchronous after GST, and (2) all members in the new configuration start from the same state (Lemma 5.5.14) and their `ready` messages trigger Olympus to send a ready proof. So all the steps terminate. Furthermore, Olympus in the configuration assignment step assigns all correct processes to the new configuration (as after GST no more process fails, and all

failed processes are detected.) Thus reconfiguration terminates with a configuration containing all correct processes. □

Lemma 5.5.18. *After GST, a correct replicated server $\mathcal{G}_p$ that is not reconfigured will succeed in ordering a request (i.e., the leader will be able to compile a complete order proof of the request.)*

Proof. First, from Lemma 5.5.14 we have that any active configuration of $\mathcal{G}_p$ starts with $ord_i^p = ord_j^p$, for all replicas i and j.

Second, we need to show that the request processing protocol preserves an invariant that when a replica i in a correct configuration of $\mathcal{G}_p$ receives a pair $\langle m, c\rangle$, then $ord_i^p = c$. This is true because (1) the leader sends pairs $\langle m, c\rangle$ to the replicas in the order of c, (2) the links transmit messages in FIFO order, and (3) the replicas increment their order counter after each receipt.

Therefore, when the leader of $\mathcal{G}_p$ sends any pair $\langle m, c\rangle$ to the replicas, the replicas will compute their signatures over $\langle m, c\rangle$ and return it to the leader. The leader collects enough signatures to compile a order proof $\langle \mathcal{R}_p, \{\langle m, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$, and the ordering finishes. □

Lemma 5.5.19 (AV3)**.** *After GST, if $\mathcal{G}_p$ is in an active configuration, then its attestation and verification steps terminate.*

Proof. When $\mathcal{G}_p$ is correct, verification is a local action at the leader of $\mathcal{G}_p$. Local actions terminate because it is after GST.

Attestation involves a number of message transmissions and local computations, which terminate after GST. The only condition is that the replicas produce the same messages. This is guaranteed because the replicas start from the same

state (Lemma 5.5.14), ordering terminates (Lemma 5.5.18), and the replicas execute requests in the order enforced by order proofs. □

Lemma 5.5.20. *After GST, if $\mathcal{G}_p$ is in an active configuration, then it terminates on the appending steps.*

Proof. The proof is similar to the proof of attestation. □

Lemma 5.5.21. *After GST, if $\mathcal{G}_p$ is in an active configuration, $r \in$ InputSet^p, and $r \notin$ Hist^p, then $\mathcal{G}_p$ will execute r, add the results to OutputSet^p, and append r to Hist^p.*

Proof. When $r \in$ *InputSet*^{p}, r must be valid by Lemma 5.5.5. Lemma 5.5.15 implies that once $r \in$ *InputSet*^{p}, $\mathcal{G}_p$ will receive r again even after it is reconfigured. Thus it is sufficient to show that if $\mathcal{G}_p$ is in an active configuration and receives a valid r, it will execute r, add the results to *OutputSet*^{p}, and append r to *Hist*^{p}.

First, by Lemma 5.5.16, $\mathcal{G}_p$ will not be reconfigured. So, it will go through the processing steps as follows. First, r is sent to the leader of $\mathcal{G}_p$. The leader will successfully verifies r, as r is valid and the leader is correct. The leader initiates the ordering protocol, which will succeed by Lemma 5.5.18 and yield an order proof $\mathit{oproof} = \langle \mathcal{R}_p, \{\langle r, c\rangle_i \mid i \in \mathcal{R}_p\}\rangle$ for some c. The leader then sends *oproof* to all replicas, which will individually execute r—hence $\mathcal{G}_p$ executes r. The other request processing steps, namely attesting, and appending, succeed by Lemma 5.5.19, and Lemma 5.5.20, respectively. Thus, the lemma holds. □

Lemma 5.5.22. *If $r \in$ InputSet^p and $r \notin$ Hist^p, then $\mathcal{G}_p$ will execute r, add the results to OutputSet^p, and append r to Hist^p.*

Proof. Suppose r is from $\mathcal{G}_q$. Assume by contradiction, the lemma does not hold. Since $\mathcal{G}_p$ does not append r to *Hist*^{p}, $\mathcal{G}_q$ will resend r to $\mathcal{G}_p$ (by Lemma 5.5.15).

By the acknowledgment-retransmission protocol, the resends alternate between sending r to a leader of a new configuration and retransmitting r to $t + 1$ members of the same configuration. Because $\mathcal{G}_p$ does not acknowledge r, a correct member in $\mathcal{G}_p$ will publish an `FDS` event. Similar reconfigurations are repeated as long as $\mathcal{G}_p$ does not acknowledge r. But after GST, $\mathcal{G}_p$ will get a configuration with all correct processes (Lemma 5.5.17). And the assumption contradicts to Lemma 5.5.21. □

Lemma 5.5.23. *If $m = \langle p, q, payload \rangle$, $m \in OutputSet^p$, and $m \notin InputSet^q$, then $\mathcal{G}_q$ will add m to its $InputSet^q$.*

Proof. Assume by contradiction that the lemma does not hold. By definition, m is only in $OutputSet^p$ if $\mathcal{G}_p$ has attested m. Since m is not in $InputSet^q$, $\mathcal{G}_q$ has not acknowledged m. Some correct member in $\mathcal{G}_p$ will time out and publish an `FDS` event. These steps repeat as long as the lemma does not hold. But after GST, $\mathcal{G}_p$ will get a configuration will all correct processes (Lemma 5.5.17). This configuration will send m with a validity proof $\langle \mathcal{G}_p, \{m_i \mid i \in \mathcal{G}_p\} \rangle$ to $\mathcal{G}_q$ when $m \in OutputSet^p$.

By Lemma 5.5.15, either (1) $\mathcal{G}_q$ appends m to $Hist^q$ or (2) $\mathcal{G}_p$ retransmits m. In case 1, $\mathcal{G}_q$ must have verified m before appending it to $Hist^q$—meaning $\mathcal{G}_q$ must have added m to $InputSet^q$. In case 2, the leader of $\mathcal{G}_q$ retransmits m by broadcasting to $t + 1$ members of $\mathcal{G}_q$. A correct one will time out and $\mathcal{G}_q$ will get reconfigured. This can only repeat until GST, when $\mathcal{G}_q$ gets a configuration of all correct processes and the system is synchronous. We get back to case 1, where $\mathcal{G}_q$ adds m to $InputSet^q$.

Both cases contradict to the assumption. Hence the lemma must hold. □

Theorem 5.5.24. *Shuttle preserves Liveness.*

Proof. Liveness holds by Lemmas 5.5.22 and 5.5.23. □

5.6 Practical Considerations

The Shuttle protocol presented in Section 5.4 achieves our goal of reducing the replication cost. But the protocol still involves high computation and communication costs. In this section we consider a few optimizations and implementation issues that make the protocol more practical.

5.6.1 Chain Communication

The Shuttle protocol involves several rounds of group communication, in which a message is passed from the leader to every member in the group. Usually broadcast implements such communication. However, given that we have a small group size (usually $t = 1$ or $t = 2$) communication in a chain pattern reduces the bandwidth consumption to approximately half compared to broadcasting, while still maintaining acceptable latency.

A group, or replicated server, is organized into a chain, called *group chain*. A group chain comprises a chain of replicas, called *replica chain*, followed by a chain of witnesses, called *witness chain*. Each chain begins with a *head* and ends with a *tail*. Note that within a group, the head of the replica chain is also the head of the group chain, and the tail of the witness chain is also the tail of the group chain. The head of a group chain also serves as the leader of the group.

Processes communicate by exchanging *shuttles*. A shuttle comprises multiple fields, including one or more proofs. Some of the proofs may be *incomplete*, meaning that they do not carry a signature from every signer. As a shuttle passes through a chain, the members may check the shuttle's contents and add signatures to the proofs. Proofs will be *complete*, meaning that they carry a signature from every signer, when their shuttle reaches the end of the chain. Shuttles passing between groups contain no incomplete proofs.

5.6.2 Digital Signatures

A digital signature needs to guarantee that it is verifiable by correct processes and unforgeable by faulty ones. Depending on the failure assumption, different methods can implement a signature. For example:

Assumption	Public-key	Secret-key
Strong adversary	RSA	MAC
Weak adversary	CRC	N/A

Public-key signatures provide the guarantees we need. For weak adversaries, CRC efficiently implements public-key signatures. But for strong adversaries, RSA may be too expensive for frequent use. Message authentication code (MAC) is more efficient, but it only implements secret-key signatures. Unlike public-key, secret-key signatures are not verifiable by all correct processes.

MAC signatures, however, can efficiently simulate RSA signatures for messages sent between two groups in Shuttle. Each process instead of generating an RSA signature generates a vector of MAC signatures, one for each process

in the destination group. A proof changes from a vector of RSA signatures, one per each signer, to a matrix of MAC signatures, where each source process contributes a row, and each destination process verifies a column.

Because a vector of MACs may be verified by some processes but not others, so may a proof be. In general, it requires to have an external service to simulate equivalent properties as RSA signatures' using MACs, as proposed in [35, 62]. The external service, depending on how it is deployed, may incur extra communication and/or replication costs. In Shuttle we avoid these costs by weakening the property. We only require a correct member be able to verify a proof when another correct member can and the replicated server is correct. If the replicated server fails, correct members may disagree on the validity of a proof. In this case, a member will trigger a reconfiguration to fix the replicated server.

Proofs can be used internally to a group (*internal proofs*) or used between groups (*external proofs*). For internal proofs, namely order proofs, each member verifies every MAC signature in its column and triggers a reconfiguration if any of them is invalid. The protocol is the same as when public-key signatures are used.

For external proofs, namely validity proofs and acknowledgment proofs, we need to add a *pre-verification* step to ensure that *a correct member accepts an external proof if others accept the proof*. This precondition is enforced by *acceptance proofs*. To ensure that every external proof is verified by at least one correct member, we require $t + 1$ members to participate in the pre-verification step. The **pre-verification** step is as follows:

- Each external proof (i.e., validity or acknowledgment proof) is signed by

every member in the source group for some $t + 1$ members in the destination group,[11] containing n rows and $t + 1$ columns of MAC signatures.

- Each of the $t+1$ members in the destination subgroup individually verifies all n MAC entries in its column. Upon success, the member produces $t + 1$ MAC signatures for the replicas (in case of validity proofs) or n MAC signatures for all the members (in case of acknowledgment proofs).

In the verification step, each participating member verifies all its $t + 1$ MAC entries in the acceptance proof.

Note that because pre-verification lets any member unilaterally discard external proofs, using MAC signatures in retransmitted validity proofs would disable the retransmission-acknowledgment protocol from exposing omission failures in the destination. Therefore we keep using RSA in retransmitted validity proofs. [12] Retransmission is triggered by a member (rather than the leader), who sends a shuttle to collect RSA signatures from other members.

5.6.3 Early Request Processing

In the Shuttle protocol, order proofs help to enforce a precondition that no correct members process inputs in a different order. In general this property requires all replicas to complete an order proof before letting any member act on it.

However, when using the chain pattern for communication, a member can

[11] Any $t + 1$ members work. But a particular subgroup may be chosen for optimization purposes.

[12] Alternatively, an external signature service approach [35, 62] also works.

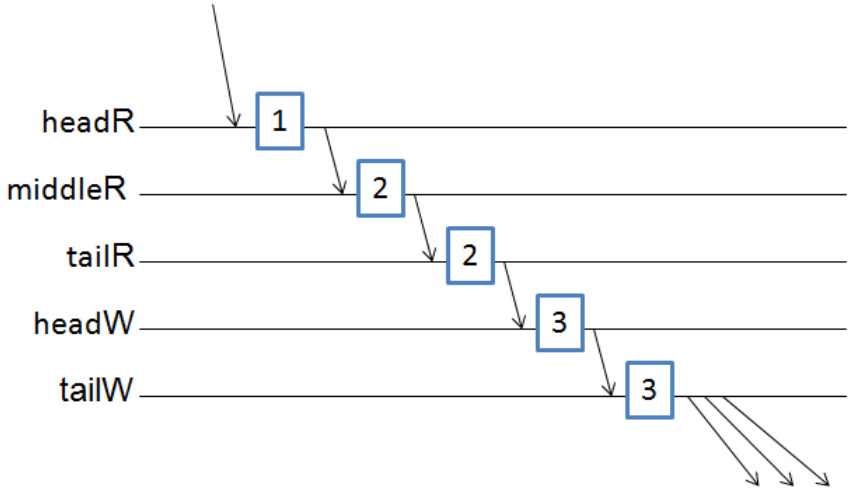


Figure 5.5: The optimized request processing protocol for $t = 2$. The witnesses only exist under the strong adversary assumption.

let its successors know the order of its execution. Thus an incomplete order proof still enforces the same precondition if it carries a signature of each preceding replica. A member can process an input with an incomplete order proof as it is passing through the chain.

Figure 5.5 illustrates the optimized request processing protocol. The numbers present different steps that the processes take:[13]

1. A request, say m, comes to the head h of the chain. The head

   - verifies the validity proof that comes with m;
   - executes m and produces outputs;
   - creates and signs an order proof $\langle m, ord_h^p \rangle$, increments ord_h^p, creates

[13]CRC, RSA, and internal validity proof are all called validity proof for convenience. Under the weak adversary assumption, these are CRC validity proofs. Under the strong adversary assumption, these are acceptance proofs for the first attempt and are RSA validity proofs for retries.

and signs an acknowledgment proof for m, and creates and signs a validity proof for each output.

2. Each non-head replica i

   - verifies the validity proof that comes with m;
   - verifies that the order proof contains a counter equal to ord_i^p and a valid signature from each replica preceding i;
   - executes m and produces outputs;
   - signs the order proof, increments ord_i^p, signs the acknowledgment proof of m, and signs the validity proof of each output.

3. Each witness i

   - verifies that the order proof contains a counter equal to ord_i^p and a valid signature from each replica;
   - increments ord_i^p, signs the acknowledgment proof of m, and signs the validity proof of each output.

The tail of the group chain will send the acknowledgment proof to the head of the source group of m and send the outputs along with their validity proofs to their destinations.

It is straightforward that the optimized request processing protocol refines the server specification in Figure 5.2 in the same way as the request processing protocol in Section 5.4.5 does. Furthermore, each member enforces the same precondition before performing an individual action:

- Before executing a request, appending it to the local history copy, and attesting resulting outputs, a member ensures that (1) the request is valid

(by a validity proof as before), (2) no correct members do so in a different order (by an incomplete order proof, as explained above);

- A witness only individually appends a request if all replicas have done so (by the chain ordering).

Therefore the optimized protocol also works.

5.6.4 Tolerating Client Failures

We have been assuming that requests from clients are valid. While it is unnecessary for a client to submit validity proofs with its requests, it still needs to sign its requests so that they cannot be forged by faulty processes. For this purpose, we treat each client as a singleton group. Under the weak adversary assumption, the clients sign each of their requests by a CRC checksum. Under the strong adversary assumption, the clients sign each of their requests by a vector of $t + 1$ MAC signatures. The request processing protocol works as it is now. Note that the pre-verification step ensures that a request coming with a vector of MAC signatures would consistently pass or consistently fail the individual verification performed at the correct members.

5.6.5 Large Requests

Large requests consume much bandwidth, and their transmissions easily become the bottleneck in distributed applications. To minimize bandwidth consumption, we limit sending large requests only to where it is absolutely necessary.

On the sender side, we prevent each replica from sending a full request to another replica, because the other replica also produces the request. Only the tail replica needs to send full requests, but it has two options. It sends the requests either directly to their destinations, or to the head witness. The former option consumes lower bandwidth overall, but it requires the destination to garbage collect full requests with incomplete validity proofs. We choose the latter option as it is simpler, even though it consumes more bandwidth (at the witnesses, which are not the bottleneck).

On the receiver side, only replicas need full requests for execution. We limit sending full requests to each replica once (under fault-free conditions), by selecting the replicas as the subgroup that performs the pre-verification step. During the pre-verification step, the replicas extract the full request. They compute the acceptance and order proofs based on the hashes of the full requests, which is actually faster than based on the requests themselves.

Figure 5.6 presents the request processing protocol with the pre-verification step. Note that this step only exists under the strong adversary assumption, when MAC signatures are used.

5.6.6 Garbage Collection

Until now the Shuttle protocol keeps an infinite history as state. The garbage collection protocol presented in this section helps to keep state manageable by replacing the history with a checkpoint and a log.

A log is, similar to a history, a queue of requests. Unlike a history, a log has

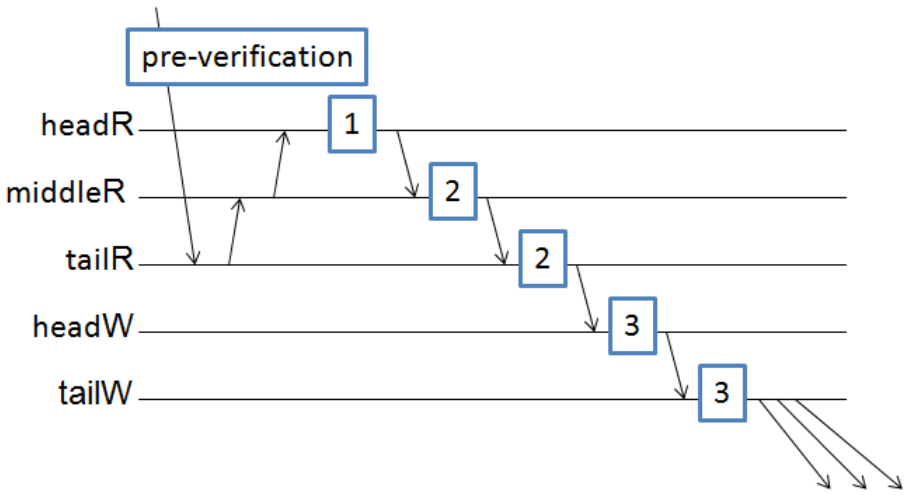


Figure 5.6: The request processing protocol for $t = 2$. The pre-verification step presented here is necessary only under the strong adversary assumption.

a bounded length. A log stores a suffix of a history. When a log is full, new requests are suspended.

A replica computes a checkpoint every K requests and truncates its log when it learns that other replicas have computed the same checkpoint.

For the protocol below, we call a request *stable* if all its resulting outputs have been acknowledged. The last acknowledgment proof that makes a request r stable is called the *stabilizing acknowledgment proof* of r.

Each correct replica computes a checkpoint every K requests. A **garbage-collection protocol** starts at the head when the first K requests in its log have become stable. The head computes a hash value of the checkpoint, then creates and signs a *checkpoint proof* that contains the hash. It then sends the incomplete checkpoint proof down the chain to the replica tail along with the last stabilizing acknowledgment proof. Each non-head replica computes a hash

value of its own checkpoint, and adds a signature of the hash to the checkpoint proof. The replica tail drops the checkpoint proof from the shuttle carrying the acknowledgment proof to the witness chain (as part of the acknowledgment-retransmission protocol).

The tail also verifies the checkpoint proof. Upon success, it removes the first K requests from its log and discards any older checkpoints it may have. The tail then sends the complete checkpoint proof along the replica chain to the head. Each replica en route repeats the same thing to truncate its log and discard any checkpoints older than the one in the checkpoint proof.

5.6.7 Reconfiguration

Reconfiguration faces three concerns:

- *Olympus implementation:* a straightforward approach to implement the Olympus service is to use an external BFT service, such as PBFT [53] or Zyzzyva [91].
- *Secret-key verification:* during a state transfer under the strong adversary assumption, the new members of a new configuration receive order proofs signed by the secret keys of the current configuration. We need to find a mechanism for the new members to verify those signatures.
- *Configuration Assignment:* as we cannot precisely determine a failed member in an unprovable failure, we have to replace more than just the faulty members. How many members a reconfiguration replaces in an unprovable failure directly affects the efficacy of Shuttle.

In this section we will address the latter two concerns. Implementing Olympus efficiently is a topic for our future work.

Secret-key Verification

During a state transfer, members of the new configuration need to verify secret-key signatures in the order and checkpoint proofs used internally in the current configuration. Unless a member is carried over from the current configuration, it does not have the necessary secret keys to verify the signatures. We call such members *new members*.

When a new member wants to verify a secret-key signature, it broadcasts the signature to all members of the current configuration and waits for $t + 1$ approvals. A current member approves a signature only if it is valid, and it sends the approval to the new member. When the new member receives $t + 1$ approvals, it accepts the signature. The number of approvals ensures that at least one correct member in the current configuration has verified the signature. Note that while the protocol does not terminate if the signature is invalid, the state transfer is guaranteed to terminate since there are at least $t + 1$ valid state-transfer responses.

Configuration Assignment

Shuttle monitors and detects/suspects failures when it observes message corruption or loss. In many cases, the failures are unprovable, leading to the need of replacing more than one member. Here we present a heuristic that lever-

ages the chain communication pattern to limit the number of members being replaced in a reconfiguration to two in the common case.

Assume that we have a coordinator and that m is the message being corrupted or lost. The coordinator runs an **interrogation protocol** with the current configuration to identify which members are to be replaced. Starting from the member that published the `FDS` event back to the head, the coordinator asks each member in the chain for m. If a member has received and individually verified m, it replies with m. Otherwise, the member replies with a `no`. The interrogation stops with the first reply of m. There are three cases:

- If the coordinator successfully verifies a response that contains m, then the member that replied and its successor (in the chain order, from head to tail) are replaced. In case the member is the one that published the `FDS` event, then it is the only member to be replaced.
- If the coordinator fails to verify m, then the member that replied is the only member to be replaced.
- If all of the replies (including the one from the head) are `no`, then the head is the only member to be replaced.

The interrogation protocol does not guarantee to replace every failed member, nor does it guarantee to replace only failed members. But it guarantees to replace at least one failed member. If a new configuration fails to become ready, the next reconfiguration will replace the entire group to make sure that all failed members are eliminated.

The role of a coordinator can be played by Olympus or, if we can assume that *newly allocated processes stay correct during a reconfiguration*, a new process.

The result of the interrogation protocol is passed to Olympus to determine the new configuration.

5.7 Evaluation

In this section we evaluate the overhead of the Shuttle protocol under both strong and weak adversary assumptions. The evaluation is done by both theoretical analysis and experimental measurements. We also compare Shuttle with existing protocols.

5.7.1 Analysis

The bulk of overhead in Shuttle, similar to other fault tolerant protocols, lies in the computational overhead of cryptographic operations and the communication overhead of the incurred extra messages. The overhead of processing a message differs depending on the types of adversaries as well as the source and destination of the message.

Number of Cryptographic Operations

We measure the computational overhead by the number of cryptographic operations performed in the processing of each request. Because we use MAC signatures under the strong adversary assumption, and CRC signatures under the weak adversary assumption, signing and verifying incur the same cost and are counted together. Tables 5.3 and 5.4 summarize the numbers of cryptographic

Steps	Strong Adversary Client-Server	Strong Adversary Server-Server
Attestation	$t_r + 1$	$2t_st_r + 2t_s + t_r + 1$
Pre-verification	$t_r^2 + 2t_r + 1$	$2t_st_r + t_r^2 + 2t_s + 2t_r + 1$
Verification	$t_r^2 + t_r$	$t_r^2 + 2t_r$
Ordering	$3t_r^2 + 3t_r$	$3t_r^2 + 3t_r$
Acknowledgment	$4t_r + 2$	$4t_st_r + 4t_s^2 + 6t_s + 4t_r + 2$
Total	$5t_r^2 + 11t_r + 4$	$8t_st_r + 4t_s^2 + 5t_r^2 + 10t_s + 12t_r + 4$
Bottleneck (at any replica)	$4t_r + 2$	$3t_s + 4t_r + 2$

Table 5.3: The number of MAC operations per request under the strong adversary assumption when there are no failures. t_s and t_r are the maximum number of failures that the sender and the receiver can tolerate, respectively.

operations performed in the processing of a request under the strong and weak adversary assumptions, respectively. The analysis considers the general case where the sender and the receiver may tolerate different numbers of failures. We use t_s for the sender and t_r for the receiver.

The first column in Table 5.3 shows the numbers for a request sent from a client to a replicated server under the strong adversary assumption:

- The client attests the request by computing $t_r + 1$ MAC signatures for the replicas in the destination group.
- The request goes through a pre-verification step at the destination. Each of the $t_r + 1$ replicas verifies a MAC signature and computes t_r MAC signatures for the other replicas. Overall, pre-verification performs $(t_r + 1)^2 = t_r^2 + 2t_r + 1$ MAC operations.

- In verification, each replica verifies t_r MAC signatures in the acceptance proof. Overall, verification performs $(t_r + 1)t_r = t_r^2 + t_r$ MAC operations.
- In ordering, each of the $t_r + 1$ replicas verifies the MAC signatures of the preceding members and computes a MAC signature for each succeeding member in the chain. Each of the t_r witnesses only verifies the $t_r + 1$ MAC signatures generated by the replicas. Overall, ordering performs $(t_r + 1)2t_r + t_r(t_r + 1) = 3t_r^2 + 3t_r$ MAC operations.
- Each of the $2t_r + 1$ members also computes a MAC signature to include in the acknowledgment proof that the replicated server returns to the client. And the client will verify all $2t_r + 1$ MAC signatures in the acknowledgment proof. Overall, acknowledgment performs $4t_r + 2$ MAC operations on both server and client sides.

The bottleneck, in term of cryptographic operations, can be at any replica, which performs $4t_r + 2$ MAC operations.

The second column in Table 5.3 differs from the first one in that the sender is a replicated server rather than a client. This leads to a different overhead in the protocol steps:

- Each of the $2t_s + 1$ members in the sender computes a MAC signature for each of the $t_r + 1$ replicas in the destination. Overall, attestation performs $(2t_s + 1)(t_r + 1) = 2t_s t_r + 2t_s + t_r + 1$ MAC operations.
- The request goes through a pre-verification step at the destination. Each of the $t_r + 1$ replicas verifies the $2t_s + 1$ MAC signatures in the validity proof and computes t_r MAC signatures for the other replicas. Overall,

pre-verification performs $(t_r + 1)((2t_s + 1) + t_r) = 2t_s t_r + t_r^2 + 2t_s + 2t_r + 1$ MAC operations.

- Verification is exactly as before. It performs $t_r^2 + 2t_r$ MAC operations.
- Ordering is exactly as before. It performs $3t_r^2 + 3t_r$ MAC operations.
- For acknowledgment, each of the $2t_r + 1$ members computes $t_s + 1$ MAC signatures to include in the acknowledgment proof that the receiver returns to the sender. The sender will go through a two-round procedure to verify the acknowledgment proof, similar to how the receiver verifies the validity proof.

  The first round is similar to the pre-verification step, except that each of the $t_s + 1$ pre-verifying members computes $2t_s$ MAC signatures for the other members (rather than t_s for the other replicas in pre-verification). This round performs $(t_s + 1)((2t_r + 1) + 2t_s) = 2t_s t_r + 2t_s^2 + 3t_s + 2t_r + 1$ MAC operations.

  The second round is similar to the verification step, except that all $2t_s + 1$ members participate (rather than $t_s + 1$ in verification). Also, members that participate in the first round verify only t_s signatures as they do not generate one for themselves, while others verify $t_s + 1$ signatures. This round performs $(t_s + 1)t_s + t_s(t_s + 1) = 2t_s^2 + 2t_s$ MAC operations.

  Overall, acknowledgment performs $4t_s t_r + 4t_s^2 + 6t_s + 4t_r + 2$ MAC operations.

The bottleneck in this case also can be at any replica, which performs $3t_s + 4t_r + 2$ MAC operations.

The first column in Table 5.4 shows the numbers of cryptographic operations

Steps	Weak Adversary Client-Server	Weak Adversary Server-Server
Attestation	1	$t_s + 1$
Verification	$t_r + 1$	$t_s t_r + t_s + t_r + 1$
Ordering	$\frac{(t_r+1)(t_r+2)}{2}$	$\frac{(t_r+1)(t_r+2)}{2}$
Acknowledgment	$2t_r + 2$	$t_s t_r + t_s + 2t_r + 2$
Total	$\frac{(t_r+1)(t_r+2)}{2} + 2t_r + 3$	$\frac{(t_r+1)(t_r+2)}{2} + 2t_s t_r + 3t_s + 3t_r + 4$
Bottleneck (at chain tail)	$t_r + 2$	$t_s + t_r + 2$

Table 5.4: The number of CRC operations per request under the weak adversary assumption when there are no failures. t_s and t_r are the maximum number of failures that the sender and the receiver can tolerate, respectively.

performed in the processing of a request from a client under the weak adversary assumption:

- The client attests the request by computing a CRC signature.
- Each of the $t_r + 1$ members of the receiver verifies the CRC signature on the request. This comprises $t_r + 1$ CRC operations.
- In the ordering, each of the $t_r + 1$ members verifies the CRC signatures of those preceding it and computes a CRC signature. This comprises $\frac{(t_r+1)(t_r+2)}{2}$ CRC operations.
- For acknowledgment, each of the $t_r + 1$ members computes a CRC signature to contribute to the acknowledgment proof. The client also performs t_r+1 CRC operations to verify the acknowledgment proof. This comprises $2t_r + 2$ CRC operations.

Unlike the protocol under the strong adversary, the replicas here do not per-

form the same number of cryptographic operations. Replicas closer to the tail need to check more signatures in an order proof than those closer to the head. The bottleneck is therefore at the tail of the chain, which performs $t_r + 2$ CRC operations.

The second column in Table 5.4 differs from the first column in that it is for a request from a replicated server. The computational overhead in the attestation and verification steps differs by a factor of $t_s + 1$, because the sender comprises $t_s + 1$ members rather than one. The acknowledgment step is simply the sum of the receiver's attesting the acknowledgment and the sender's verifying it. As for bottleneck, it is similar to the previous case. The tail performs $t_s + t_r + 2$ CRC operations.

Message Overhead

We measure message overhead in two metrics: (1) message factor, and (2) message latency. Message factor is the ratio between the number of messages sent in processing an application message in Shuttle and in a non-fault-tolerant system (which is two, one for the request, and the other for the response). Message latency is the number of hops measured from when a client or a head of a replicated server produces a request to when it receives a response.

Because we use a chain communication pattern, the number of messages is the same as the length of the processing path, which goes through the source chain in attestation, the destination replica chain in pre-verification (only under the strong adversary assumption), the destination group chain in ordering and execution, the source replica chain in pre-verification of an acknowledgment

	Message Factor	Latency
Strong Adversary (Client-Server)	$\frac{3t_r+2}{2}$	$3t_r + 2$
Strong Adversary (Server-Server)	$\frac{5t_s+3t_r+2}{2}$	$3t_s + 3t_r + 2$
Weak Adversary (Client-Server)	$\frac{t_r+2}{2}$	$t_r + 2$
Weak Adversary (Server-Server)	$\frac{2t_s+t_r+2}{2}$	$t_s + t_r + 2$

Table 5.5: Message overhead. t_s and t_r are the maximum numbers of failures that a sender and a receiver can tolerate, respectively.

(also only under the strong adversary assumption), and the source group chain in the verification of an acknowledgment. The length of the processing path also reflects message latency. But because a response (same as an acknowledgment) is delivered to the head before it is passed down the chain, the latency values are smaller than the path length by the length of the chain.

Table 5.5 shows the message overhead of the Shuttle protocol under strong and weak adversary assumptions. Note that the number of messages (message factor) is calculated per single application message. In practice we may aggregate the output messages in the attestation step with the input-processing message to lower the message factor.

5.7.2 Experimental Measurements

We are interested in the practical performance of Shuttle in the common case, when the system has no failure.

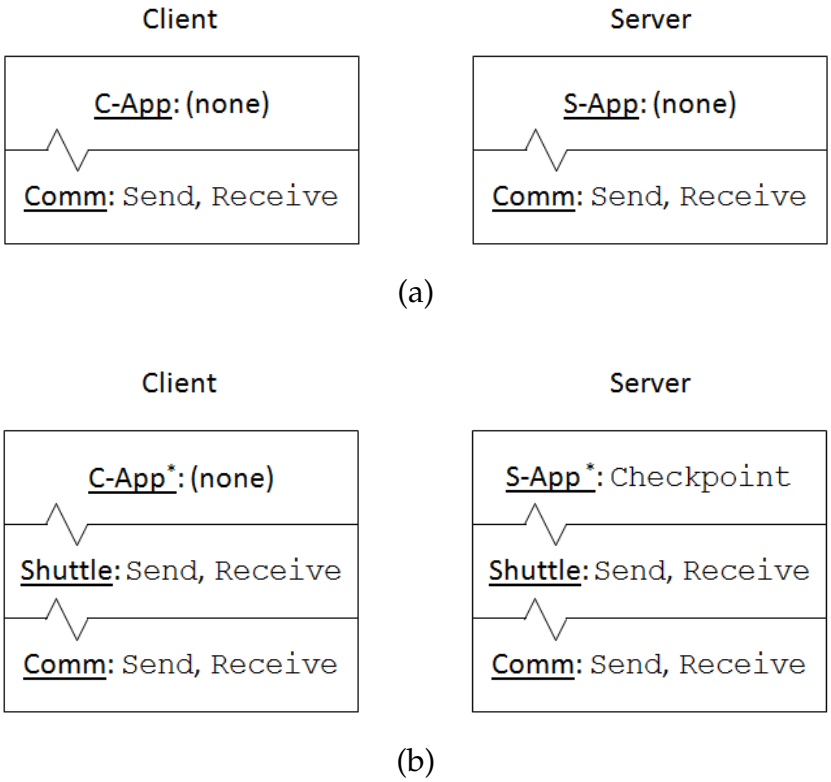


Figure 5.7: Module interfaces in a non fault tolerant system (a), and in a fault tolerant system (b).

Implementation

We have implemented a prototype of the Shuttle protocol in Erlang. Our implementation is approximately 2K lines of code under the strong adversary assumption, and approximately 1.2K lines under the weak adversary assumption. Olympus is implemented as an unreplicated external service.

Figure 5.7 describes the interfaces of the modules we implemented. The application clients (`C-App`) and servers (`S-App`) communicate over the network by calling `Send` and `Receive` functions supported by the network module

(`Comm`). The Shuttle protocol is implemented as a middle layer (`Shuttle`) that also supports `Send` and `Receive`.

Existing applications need a slight modification to work with Shuttle. The clients and servers need to be recompiled to use `Shuttle`'s `Send` and `Receive` rather than `Comm`'s. Furthermore, the servers need to support an additional function, `Checkpoint`, that computes a checkpoint of the state of a server, used in the garbage collection protocol.

As discussed earlier, the different adversary assumptions require different cryptographic tools. Under the strong adversary assumption, our Shuttle implementation uses `crypto:sha_mac/2` with 160-bit secret keys to generate MAC signatures, and RSA with 1024-bit public keys to sign retransmitted requests. Under the weak adversary assumption, the Shuttle protocol uses `erlang:crc32/1` to generate CRC signatures. We differentiate the variations of the Shuttle implementation by naming them based on their main cryptographic mechanism: *HMAC-Shuttle* is the variation that assumes a strong adversary, and *CRC-Shuttle* is the one that assumes a weak adversary.

Notes On Non-Optimizations

Our prototype implementation forgoes some optimizations usually employed in traditional approaches.

Batch Processing. Unlike traditional fault tolerant approaches, Shuttle does not benefit from processing requests in batches. There are a few reasons. First, batching is sometimes said to reduce the number of cryptographic operations per request, because it spreads the cost of a cryptographic operation on mul-

tiple messages. However, as Figure 5.8 shows, the computational cost of the HMAC and CRC operations we use increases linearly with the size of the message. So, batching does not reduce the computational cost of the cryptographic operations, even though it reduces the number of such operations. Second, we

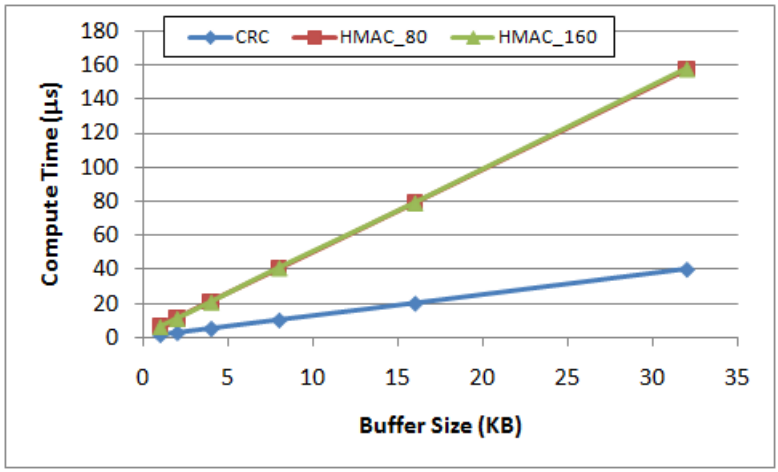


Figure 5.8: Computational costs of HMAC and CRC operations. HMAC_x indicates the HMAC function with a key of length x bits. The key lengths do not matter as they are insignificant relatively to the message lengths.

made the same observation that ZZ [132] does. Batching becomes less and less effective as the application's message processing time increases. And third, because of the chain communication pattern, we can pipeline requests and process one request in each round.

Garbage Collection. Our implementation keeps the garbage collection constant $K = 1$. The garbage collection protocol runs with the processing of every request. Doing so we evaluate not only the request processing overhead but also the overhead incurred by checkpointing and garbage collection. We keep the measured overhead application independent by keeping the state of the sam-

ple application small. The downside of running the garbage collection protocol frequently is that the performance we measured is conservative.

Experimental Setup

In order to evaluate the Shuttle protocol, we have implemented a simple application. The application models a distributed bank that has multiple branches. Each branch keeps a simple database of accounts and their balances. Below we focus on two of the supported operations:

> **Deposit**: increase the balance of an account by a specified amount and reply with the updated balance,
>
> **Transfer**: decrease the balance of a source account by a specified amount, reply with the updated balance of the source account, and issue a deposit request to increase the balance of a destination account at a possibly different branch by the same amount.

For testing purposes, we optionally attach a large buffer to requests. The buffer is copied in the reply. In the case of the transfer request, the buffer is also attached to the deposit request that the source branch sends to the destination branch.

The system is deployed on a cluster of 11 Dell 1950 Intel Xeon Quad-core 2.33GHz nodes connected by 1Gbps Ethernet. Nodes 0 through 8 have 8GB memory each. Nodes 9 and 10 each has 16GB. Each node runs Linux (kernel 2.6.9-55ELsmp), Erlang R14A, and OpenSSL 0.9.7d.

In the experiments below, servers are deployed on nodes 1 through 10; clients and the Olympus service are deployed on node 0. Each client waits for the reply of its request before issuing the next request, but multiple clients can issue requests concurrently.

In the experiments we try to answer three questions:

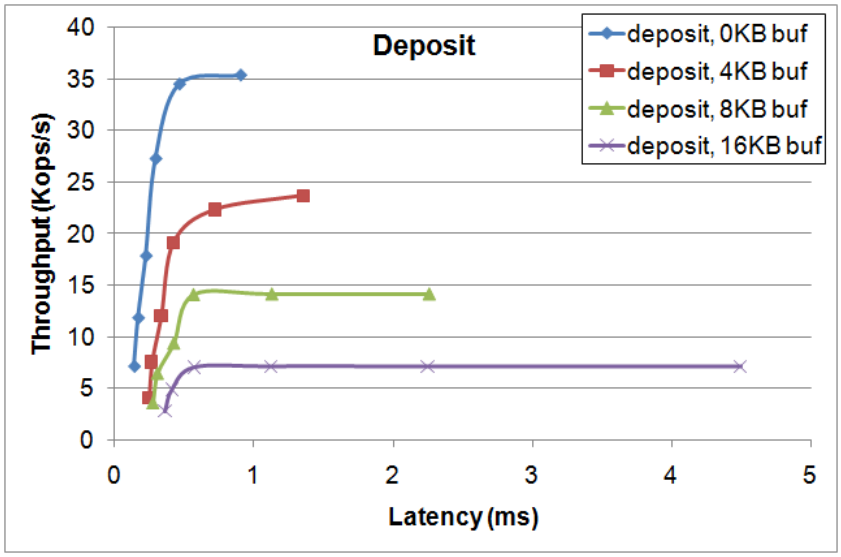
- How does our choice of programming language affect the performance of the system?
- What are the throughput and latency of Shuttle in processing requests?
- What can we improve in future implementations?

Erlang Performance

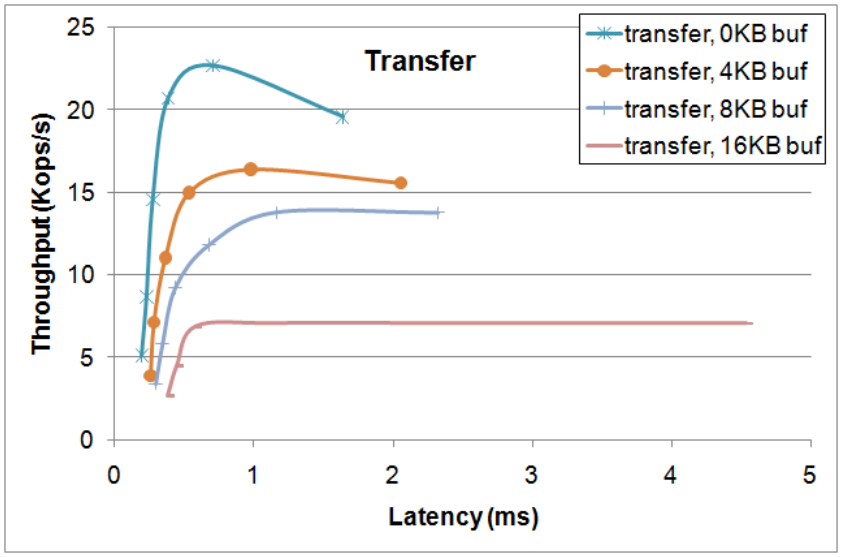
To understand how our choice of programming language affects the performance of the system, we evaluated its network performance and its computational performance.

For network performance, we measured the throughput and latency of the distributed bank application without replication. In the experiments we ran two unreplicated servers, for server-server transfer operations, and increased the number of clients until the server was saturated. Figure 5.9 shows the results. Each line in the graph contains six measurements, corresponding to 1, 2, 4, 8, 16, and 32 clients. The different lines correspond to different buffer sizes.

For deposit operations, the system achieves a peak throughput at 35.3 Kops/s, with 32 clients issuing deposit requests carrying an empty buffer. The



(a)



(b)

Figure 5.9: Throughput vs. Latency of the non-replicated bank application.

corresponding latency at this rate is about 0.9 milliseconds per request. The throughput yields an average of $28.3\mu s$ for the system to serve a request.

To factor out the network performance, we profiled the request execution at

the server. On average, it takes the server $20\mu s$ to process each deposit. And so less than $8.3\mu s$ can be attributed to the network.

For computational performance, we performed `erlang:crc32/1` and `crypto:sha_mac/2` on different buffer sizes. Figure 5.8 shows that we can compute CRC-32 at a rate of about 795 Mbytes/sec, and SHA1-based HMAC signatures at a rate of about 203 Mbytes/sec. This is, the CRC computation is almost 4x faster than the HMAC computation.

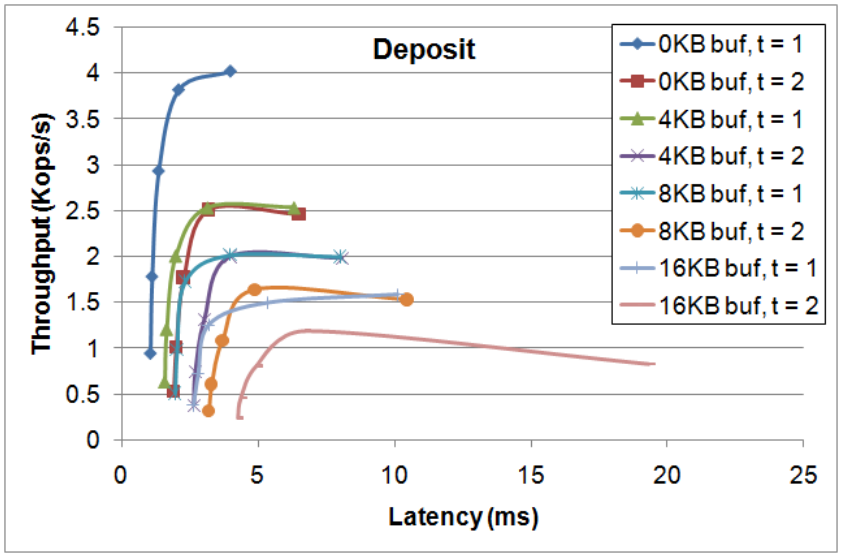
Throughput vs. Latency

We repeated the experiments, but with replicated servers with $t = 1$ and $t = 2$, to measure the throughput and latency of HMAC- and CRC-Shuttle. In these experiments, the servers are saturated with 16 clients rather than 32. Figures 5.10 and 5.11 show the results.

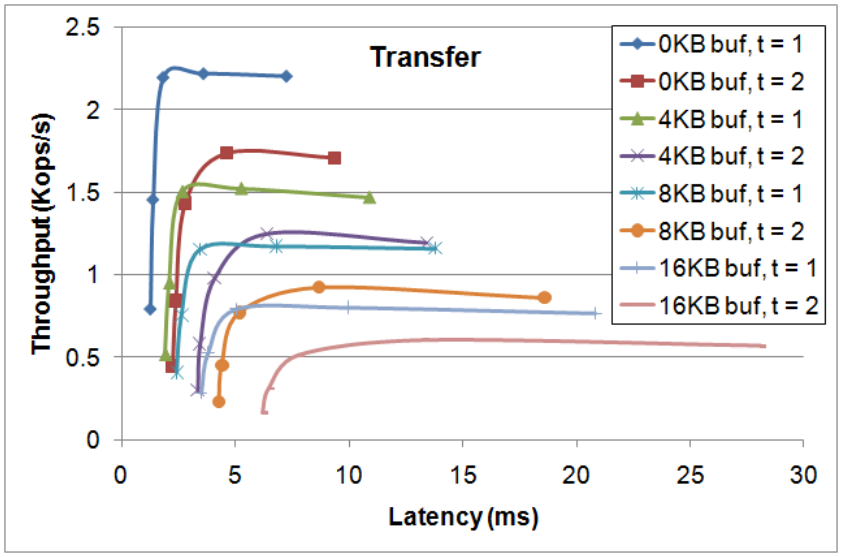
As the graphs show, Shuttle's performance is strongly affected by the message size. It shows in the decreased performance when either the buffer size is increased, or t is increased from 1 to 2, or both. When the value of t is increased, more signatures are included in a message. It does not only increase the message size, but also increases the computational time of signing and verifying the signatures.

Bottleneck and Future Improvements

We study the request processing profile of HMAC- and CRC-Shuttle to understand why their peak performances are 8x and 4x lower than the unreplicated



(a)

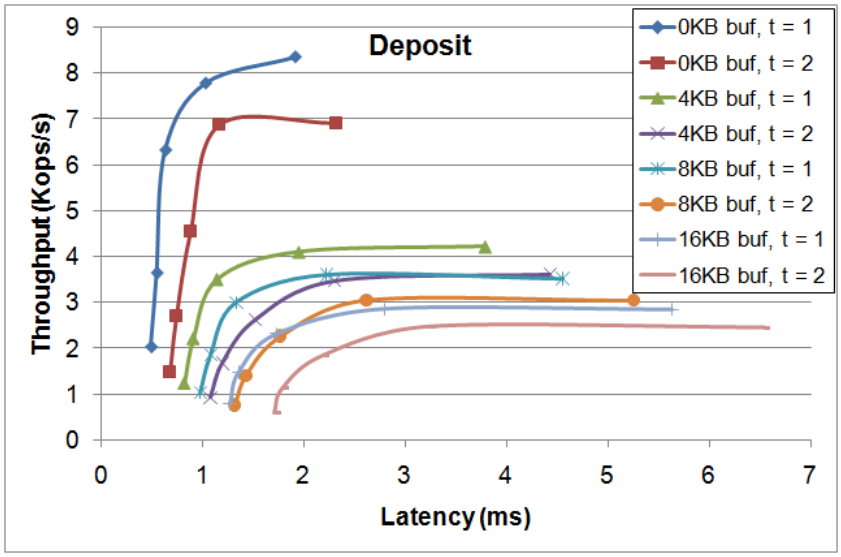


(b)

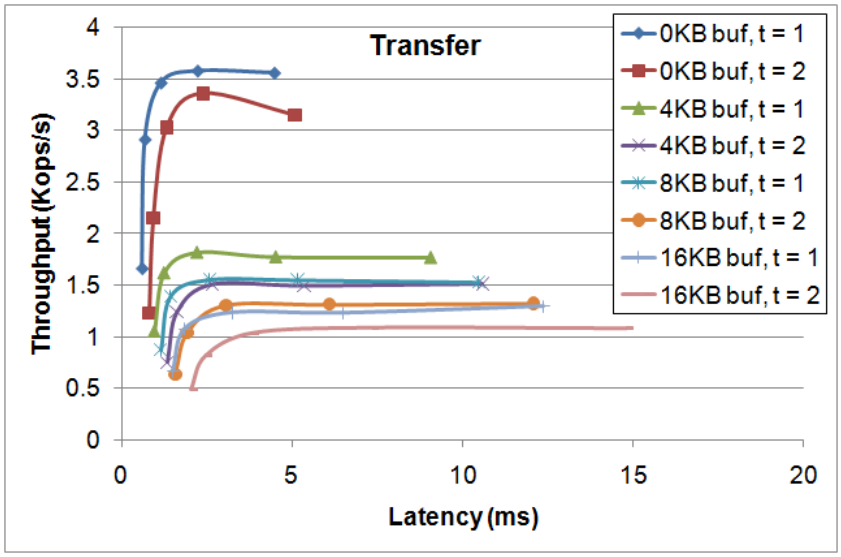
Figure 5.10: Throughput vs. Latency of the HMAC Shuttle protocol.

system. We identified where the bottleneck is and what could be done to improve performance in future implementations.

Figure 5.12 shows a sample profile of the processing a deposit under HMAC-Shuttle with $t = 1$. The figure shows where time is spent and also the size of



(a)



(b)

Figure 5.11: Throughput vs. Latency of the CRC Shuttle protocol.

messages generated for handling the request. The bottleneck in this case is the processing that happens at the tail of the replica chain, which first has to verify the validity proof and generate a new acceptance proof, and later it has to check the completed acceptance proof and the order proof, add its own signa-

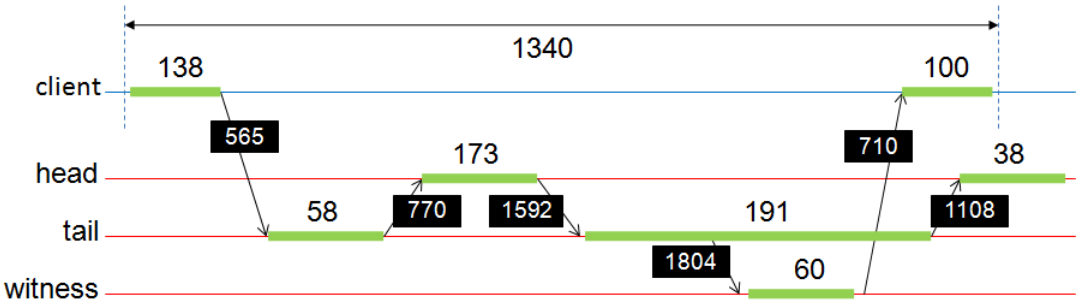


Figure 5.12: Sample request processing profile for HMAC Shuttle, $t = 1$, 0KB buffer. The numbers in black boxes are message sizes, in bytes. The other numbers are processing times, in microseconds.

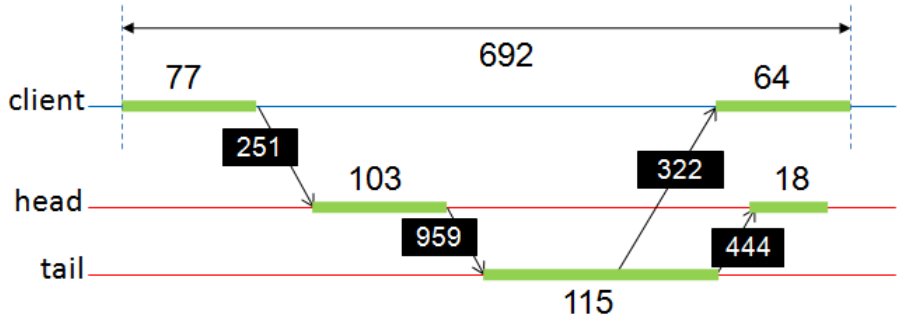


Figure 5.13: Sample request processing profile for CRC Shuttle, $t = 1$, 0KB buffer. The numbers in black boxes are message sizes, in bytes. The other numbers are processing times, in microseconds.

tures to the order proof and the acknowledgment proof, take a checkpoint, sign the checkpoint proof, and collect garbage. The bottleneck consumes $249\mu s$ for each deposit, which translates to about 4 Kops/s. It explains the 4 Kops/s peak throughput of HMAC-Shuttle in Figure 5.10(a) (first curve). When adding a 4KB buffer to each deposit request, the 1Gbps network bandwidth takes about $32\mu s$ extra for sending/receiving a request. The bottleneck (the replica tail) requires three such instances (receiving the buffer from the client, sending it to the head,

and sending it to the witness as part of the response). In addition, it takes approximately $19\mu s$ to compute a HMAC signature of 4KB data. There are two HMAC operations at the bottleneck involving the buffer (one for the request coming from the client, and the other is for the response sent to the witness). With the extra time, the bottleneck takes $383\mu s$ to process each deposit. This translates to a throughput of 2.6Kops/s. And it explains the peak bandwidth of the second curve in Figure 5.10.

Similarly, Figure 5.13 shows a sample profile of the processing a deposit under CRC-Shuttle with $t = 1$. Besides a shorter path, shorter processing times, and smaller messages, we can also see that processing is better balanced between the head and the tail of the chain. The bottleneck (even though it shows only a slight difference) is at the head, which takes $121\mu s$ to process a deposit. This translates to a throughput of slightly higher than 8Kops/s. It explains the peak throughput of the first curve in Figure 5.11. When attaching a 4KB buffer, the head takes an extra $32\mu s$ for receiving the buffer from the client, another $32\mu s$ for sending it to the tail, and one more $32\mu s$ for sending the buffer in the response. The head also takes an extra $19\mu s$ for computing the digest of the request to include in the order proof. The extra overhead caused by the 4KB increases the bottleneck to $236\mu s$, which yields about 4.2Kops/s as depicted by the second curve in Figure 5.11. The buffer included in the response from the head reveals an optimization left unimplemented.

The profiling studies suggest the following improvements for future implementations:

- In CRC-Shuttle, a CRC checksum could be used as the digest of a request as it is already good for serving as a digital signature.

- The reason that the request size strongly affects our prototype's performance is because it is single threaded in the critical path. In a future implementation, we could exploit the multi-core/hyperthreading processor architecture and the duplex feature of modern network interface cards to parallelize the request processing task at each member to four subtasks: receiving messages, Shuttle processing requests, application processing requests (only at replicas), and sending messages.

5.7.3 Comparison with Prior Work

The literature is rich with approaches for making Byzantine fault tolerance practical. Starting from PBFT [53], various projects [32, 65, 90, 123, 80] aim to reduce latency and increase throughput. Recently, more work has tried to address concerns beyond latency and throughput. For instance, Steward [38] and Mencius [105] address how one would scale a BFT service over a wide-area network. Aardvark [63] and Zyzzyvark [62] focus on sustainable performance rather than peak performance. And Separation[14] [134], BFT2F [102], and ZZ [132] aim for reducing the replication cost of BFT services.

Shuttle focuses mainly on reducing replication cost. But unlike existing work, Shuttle targets distributed services, in which the original, non-fault-tolerant services consist of multiple unreplicated servers. Considerable complexity of Shuttle rises from the interaction among servers, especially from the prevention of malicious processes from colluding across different replicated

[14]The authors did not give a short name to their paper, "Separating agreement from execution for Byzantine fault tolerant services." We take the liberty to call it "Separation" for short.

	PBFT	Zyzzyva	Aliph	Zyzzyvark	ZZ	HMAC Shuttle	CRC Shuttle
total #processes	$3t+1$ (7)	$3t+1$ (7)	$3t+1$ (7)	$3t+1$ (7)	$3t+1$ (7)	$2t+1$ (5)	$\mathbf{t+1}$ (3)
#replicas	$2t+1$ (5)	$2t+1$ (5)	$3t+1$ (7)	$2t+1$ (5)	$(1+\epsilon)t+1$ $(3+\epsilon)$	$\mathbf{t+1}$ (3)	$\mathbf{t+1}$ (3)
#crypto ops at bottleneck	$2+\frac{8t+1}{b}$ (17)	$2+\frac{3t}{b}$ (8)	$1+\frac{t+1}{b}$ (4)	$2t+\frac{3t}{b}+2$ (12)	$2+\frac{10t+3}{b}$ (25)	$2+\frac{4t}{b}$ (10)	$\mathbf{2+\frac{t}{b}}$ (4)
# message latencies	4	**3**	$3t+2$ (8)	4	4	$3t+2$ (8)	$t+2$ (4)
tolerates strong adversaries	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No
effects of faulty clients	Reconfiguration	RSA + Rollback	Rollback	**None**	Reconfiguration	**None**	**None**

Table 5.6: Properties of state-of-the-art BFT replication approaches that tolerate t failures, avoid RSA signatures, and use a batch size b. The numbers in parentheses are calculated with $t = 2$ and $b = 1$.

servers to harm the system. To the best of our knowledge, no existing work has ventured in the same direction.

Table 5.6 compares Shuttle with some related work. All the compared related work has strongly consistent semantics and avoids expensive RSA digital signatures. We include ZZ for its similar goal of reducing replication cost, Zyzzyvark for its similar goal of tolerating client failures, and Aliph for its similar chain communication pattern. We also compare Shuttle with Zyzzyva for its well known performance, and with PBFT for historical purposes. To put Shuttle into context, we adapt the numbers in tables 5.3, 5.4, and 5.5, taking only the client-server case and assuming that Shuttle uses batch processing with batch size b. For other approaches, we get the numbers published by the authors, except:

- the numbers of PBFT are taken from the Zyzzyva paper [90];
- the latency number of Aliph is the common case where there is contention from multiple clients rather than the best case where there is no contention;
- the numbers of Zyzzyvark are computed as follows (with $t = u = r$ to make it comparable):
  - the number of processes is $3t + 1$, assuming that request quorum, order, and execution groups co-locate;
  - the number of replicas is $2t + 1$, as it uses separation of agreement from execution;
  - the computational bottleneck is $2t + \frac{3t}{b} + 2$ MAC operations at the leader of the order group, where it needs to check $2t + 1$ MAC signa-

tures from the request quorum, and, similar to Zyzzyva, signs $\frac{3t}{b}$ for the other members plus 1 for the client;

  - the message latency is 4, as the critical path of a common-case request processing is client $\rightarrow$ request quorum $\rightarrow$ order leader $\rightarrow$ order members $\rightarrow$ client, assuming each replica co-locates with a member of the order group;

- the numbers of latency for ZZ are not published, but they are supposedly the same as PBFT because ZZ uses the same ordering protocol as PBFT.

As for the qualitative properties, all the listed approaches by design tolerate strong adversaries except CRC-Shuttle. The approaches maintain safety against faulty clients, but they do so at different costs:

- PBFT: as PBFT uses MAC signatures in all but view-change and new-view messages, a malicious client can send to a non-leader member a request with only one valid MAC signature for that member. The member will consider the message as valid, relay it to the leader (who discards it), and time out waiting for the execution of the request. The malicious client triggers a reconfiguration at a correct replicated server.
- Zyzzyva: a malicious client can send a request to the leader with a MAC vectors containing t valid entries and diverge the replicas' state. The request authentication protocol (presented in Kotla's dissertation [89]) will resolve in a few communication rounds using RSA signatures and a roll-back of the t replicas that have executed the request.
- Aliph: a malicious client can perform the same misbehavior as above. When a replica cannot verify its MAC entry, the execution stops. The client

	Zyzzyvark	HMAC Shuttle
Processors	Dual-core Pentium IV 3GHz	Xeon Quad-core 2.33GHz
NICs	1Gbps Ethernet	1Gbps Ethernet
Language	Java 1.6	Erlang R14A
Operating System	Linux 2.6	Linux 2.6
$t = 1$, small requests	3.3Kops/s, 73ms, 1B	4Kops/s, 4ms, 0B
$t = 2$, small requests	1.7Kops/s, 90ms, 1B	2.5Kops/s, 3.2ms, 0B
$t = 1$, large requests	$<$ 1Kops/s, $>$ 70ms, 10KB	1.58Kops/s, 10.2ms, 16KB

Table 5.7: Practical performance comparison between HMAC-Shuttle (client-server) and Zyzzyvark. The measurements are throughput, latency, and request size.

will trigger the rollback of the request by sending a `PANIC` message and collecting an abort history.

- Zyzzyvark: the request quorum helps to filter out harmful requests from malicious clients. Zyzzyvark only diverges from the common-case request processing protocol when there is a failure in its replicated server.
- ZZ: ZZ uses the same protocol as PBFT and suffers from the same weaknesses.

To shed some light on Shuttle's performance in practice, we relate it to Zyzzyvark for its recency (Zyzzyvark was published in SOSP'2009), its similar use of a high-level programming language, and its similar robustness against faulty clients. Zyzzyvark is implemented in Java, while Shuttle is prototyped in Erlang. For Zyzzyvark we consider the cases $t = u = r = 1$ and $t = u = r = 2$, where the request quorum, order, and execution nodes are co-located. Table 5.7 summarizes the results.

In Table 5.7, Shuttle's measurements are copied from our experiments. Zyzzyvark's measurements are copied from Figures 7 and 9 in the UpRight paper [62], with throughputs scaled up 8% and latencies scaled down 10% to take into account the difference between JS-Zyzzyvark (the read values) and J-Zyzzyvark (fairer comparison with Shuttle as both do not write to disk). These percentages are suggested by Figure 5 in the same UpRight paper. Note that Shuttle performs deposit operations (approximately $20\mu s$), while Zyzzyvark produces responses upon null requests.

While we are uncertain of the impact of the different processor architectures on performance, the rates of changes in the measurements suggest that Shuttle scales better with the request size and the group size. Another interesting observation is that while Shuttle has larger theoretical latencies than Zyzzyvark's, its actual latencies are about an order of magnitude lower than Zyzzyvark's.

5.8 Discussion

This chapter proposes and evaluates the Shuttle protocol, a novel Byzantine replication approach that uses only $t + 1$ replicas and t witnesses, the latter are necessary only if working under the strong adversary assumption. Our proposal makes the following main contributions:

- a consideration for distributed applications as a whole, which addresses collusion across replicated servers;
- a protocol design that emphasizes high throughput;
- a tradeoff between fault masking efficacy and replication cost;

- a tradeoff between fault tolerance strength and both replication cost and performance;
- a common-case evaluation of our approach in both analysis and experimentation.

Even though Shuttle implements an asynchronous distributed application, the protocol can be used to implement a synchronous or partially synchronous distributed application. The key is that after GST, a failed replicated server is reconfigured in bounded time and a correct replicated server processes an input in bounded time as well.

Our evaluation shows that our approach achieves a common-case performance comparable to existing state-of-the-art approaches, while it cuts the cost by one-third to a half. However, the evaluation is still in its preliminary stage. In future work, we aim to evaluate the reconfiguration performance as well as study cases of Shuttle being applied to some practical distributed applications.

CHAPTER 6

CONCLUSIONS

This chapter concludes the dissertation. We begin by reviewing the contributions in Section 6.1. Then we discuss future research directions in Section 6.2.

6.1 Contributions

The dissertation makes the following main contributions:

- A new approach toward building fault tolerant distributed applications. The approach is presented informally, with discussion of how to relate it to existing general fault tolerance approaches. We have solved the challenges in this dissertation using this new approach.
- An novel ordered broadcast protocol, called OARcast. OARcast ensures when a sender broadcast messages to a number of receivers, the receivers deliver the same sequence of messages from the sender. The property is preserved even when the sender is malicious.
- A translation technique that can be used to strengthen distributed applications automatically. The technique makes few assumptions about the distributed applications. And thus it can be applied in a larger class of distributed applications than previous approaches.
- A scalable translation technique that can be applied to large-scale distributed applications. The technique is adaptive to hosts joining and leaving and topology changes. It also ensures application processes handle inputs fairly.

- A replication protocol that uses $t + 1$ replicas to tolerate t Byzantine failures. When facing strong adversaries, the protocol may use t witnesses in addition to the replicas. But when the adversaries are weak, the protocol does not require the witnesses and uses a cheaper cryptographic primitive for reducing computational overhead.

6.2 Future Directions

6.2.1 Unifying Rollback Recovery and State Machine Replication Approaches

Rollback recovery approaches such as message logging and checkpointing were invented for saving parallel scientific applications from losing hours or days of computational work when failures occur. State machine replication was invented for fault tolerance in distributed systems, where the machines are more loosely synchronized to others compared to parallel systems that are targeted by rollback recovery. Rollback recovery approaches also make stronger assumptions about the environment, as parallel scientific applications are usually run in a cluster or a supercomputer that is specialized and isolated from other applications. State machine replication on the other hand makes fewer assumptions about the environment, as some of its target applications (e.g., aircraft controller) are required to work in hostile environments. The different targeted applications and environments lead to different characteristics in the resulting rollback-recovery and state-machine-replication protocols. And the two approaches have been considered fundamentally different.

In Section 1.2 we informally show how the fault tolerance approaches relate. Rollback recovery and state machine replication approaches seem to focus on different facets of the fault tolerance problem. In particular, rollback recovery focuses on how to retain consistent state over failures, while state machine replication further deals with validity of inputs. When mapped to our model, the state machine replication approach implements the notary service and surrogate abstractions, which do not exist in a rollback recovery system.

A possible future direction is to formalize our approach in Section 1.2. The final model of a fault tolerant distributed application would be similar to our informal model in that it (1) separates the concerns of retaining state over failures from preventing messages and state from being undetectably altered, and (2) separates the concern of masking failures on the fly from fault tolerance in general. But unlike our informal model, the formal model should identify the precise properties for each concern.

The formalized approach would unify rollback recovery and state machine replication approaches (possibly other fault tolerance approaches as well). The unification would allow us to bring advancements from one area to another. The next two subsections are examples.

6.2.2 Strengthening Rollback Recovery Approaches

Rollback recovery approaches such as message logging and checkpointing are restricted to tolerating only crash failures. This is due to two reasons: (1) application processes do not check input messages and recovery information saved into stable storage (e.g., input messages, checkpoints) for their authenticity and

validity, and (2) the stable storage may not be strong enough to make saved data retrievable when Byzantine faults occur.

Both problems can be overcome. The first one can be overcome by implementing a notary service that turns data (e.g., input messages, checkpoints) to self-verifiable data. The second problem is not a fundamental shortcoming of rollback recovery approaches, as they assume a stable storage abstraction rather than a specific implementation. One only needs to use an appropriate stable storage implementation.

While it is questionable that the overhead of the resulting BFT rollback recovery protocols will be practical, this question should be limited to the implementation rather than the abstraction. With future advancements in hardware and software systems, it is certainly possible that the overhead will become acceptable.

6.2.3 Optimizing State Machine Replication Protocols

As discussed in Section 1.2, the replicas of an application process in a state-machine-replication system implement stable storage. Similar to the pessimistic message logging approach, the application process stores each input message to stable storage before it sends any further message to another application process, or to the clients.

The pessimistic message logging approach is simple in recovery, but it is slow in the common case, when there is no failure. Alternatives have been proposed to make the common case faster. For example, the optimistic mes-

sage logging approach stores input messages to stable storage asynchronously, letting the application process send output messages to other application processes as soon as possible. The optimistic message logging approach achieves higher performance in the normal case, but it complicates recovery. Another example is causal logging, which makes sure that the *causal past* [93] of each process is either stable or available locally to the process. The causal message logging approach is balanced between the pessimistic and optimistic approaches.

Similar to pessimistic message logging, fault tolerant distributed applications using SMR protocols are known to be expensive, and they are rarely deployed in production environments. A possible future research direction is to optimize SMR protocols used in fault tolerant distributed applications, so that they can produce outputs before the causal input has been saved to stable storage. The distributed applications that use the optimized SMR protocols behave similarly to optimistic and causal message logging protocols.

The relaxing of synchrony of the protocol steps is similar to Virtual Synchrony [44, 43], and Speculator [112]. They however differ in scale. Speculator is at the operating system level and increases performance of individual file systems. Virtual Synchrony is at the process group level. It can help to implement faster stable storage. The proposed future work is at the distributed application level. It is orthogonal to Virtual Synchrony and Speculator and can benefit from them.

6.2.4 Byzantine Fault Tolerant Transaction Commit Protocols

The transaction commit problem is a central part of transaction processing systems. It guarantees that when the participants of a transaction need to make an irrevocable decision, they will make the same decision. Traditionally, the transaction commit problem assumes only crash failures. And as a result, transaction processing systems were limited to tolerate only crash failures.

Though the transaction commit and the Byzantine agreement problems are considered to be different [76], various works have proposed solutions to Byzantine fault tolerant transaction processing systems (e.g., [72, 129]). They apply the SMR approach to replicate each participant and solving the Byzantine agreement problem within each replicated participant. The main challenge this approach encounters is efficiency: (1) Byzantine agreement protocols involve high message complexity, and (2) the Byzantine agreement problem is not solvable when the environment is asynchronous. In practice, this means that the replicated participant may take arbitrarily long to send its vote to the coordinator.

Using our translation techniques, we can transform a transaction commit protocol from crash-tolerant to Byzantine-tolerant. The resulting protocol does not need to solve the Byzantine agreement problem, has lower message complexity (linear vs. quadratic), and therefore is potentially more practical. A possible future research project is to verify this observation.

BIBLIOGRAPHY

[1] Amazon Payments. https://payments.amazon.com.

[2] Amazon.com, Inc. http://www.amazon.com.

[3] Bank of America Corporation. https://www.bankofamerica.com.

[4] Box.net. http://box.net.

[5] Dailymotion. http://www.dailymotion.com.

[6] Domino's IP Holder LLC. http://express.dominos.com/order/olo.jsp.

[7] Dropbox. http://www.dropbox.com.

[8] E*TRADE Financial Corporation. https://www.etrade.com.

[9] Facebook. http://www.facebook.com.

[10] Facebook Statistics. http://www.facebook.com/press/info.php?statistics.

[11] FMR LLC. https://www.fidelity.com.

[12] Google Checkout. https://checkout.google.com.

[13] Google Maps. http://maps.google.com.

[14] Google Picasa. http://picasa.google.com.

[15] JPMorgan Chase & Co. https://www.chase.com.

[16] Mapquest, Inc. http://www.mapquest.com.

[17] Metacafe Inc. http://www.metacafe.com.

[18] Microsoft Sync. http://sync.live.com.

[19] Monster Worldwide Inc. http://www.monster.com.

[20] The New York State Department of Motor Vehicles. http://www.nydmv.state.ny.us/licrenew/default.html.

[21] Papa John's International, Inc. http://order.papajohns.com/index.html.

[22] Paypal. https://www.paypal.com.

[23] Photobucket Corporation. http://photobucket.com.

[24] Scottrade, Inc. https://www.scottrade.com/.

[25] Texas Department of Public Safety. http://www.txdps.state.tx.us/driverlicense/index.htm.

[26] The U.S. Social Security Administration. http://www.ssa.gov/onlineservices.

[27] Wells Fargo. https://www.wellsfargo.com.

[28] Yahoo! Flickr. http://www.flickr.com.

[29] Yahoo! Mail. http://mail.yahoo.com.

[30] Youtube, LLC. http://www.youtube.com.

[31] *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, Toronto, Ontario, August 1988. ACM SIGOPS-SIGACT.

[32] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In Herbert and Birman [83], pages 59–74.

[33] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX.

[34] A. Agbaria and R. Friedman. Overcoming Byzantine Failures Using Checkpointing. Technical report, University of Illinois at Urbana-Champaign, 2003.

[35] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix Signatures: From MACs to Digital Signatures in Distributed Systems. In Gadi Taubenfeld, editor, *DISC*, volume 5218 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2008.

[36] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In Herbert and Birman [83], pages 45–58.

[37] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the 2nd Int. Conf. on Software Engineering (ICSE'76)*, pages 627–644, San Francisco, CA, October 1976. IEEE.

[38] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. J. Zage. Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. In *DSN*, pages 105–114. IEEE Computer Society, 2006.

[39] B. Awerbuch and C. Scheideler. Group Spreading: A protocol for provably secure distributed name service. In *Proc. of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *Lecture Notes in Computer Science*, Turku, Finland, July 2004. Springer.

[40] R.A. Bazzi. *Automatically increasing fault tolerance in distributed systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1995.

[41] R.A. Bazzi and G. Neiger. Simplifying fault-tolerance: providing the abstraction of crash failures. *J. ACM*, 48(3):499–554, 2001.

[42] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, August 1983. ACM SIGOPS-SIGACT, ACM Press.

[43] K. Birman. *Reliable Distributed Systems Technologies, Web Services, and Applications*. Springer, 2005.

[44] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed

systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987. ACM Press.

[45] H. Blair-Smith. Space shuttle fault tolerance: Analog and digital teamwork. In *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, pages 6.B.1–1 –6.B.1–11, 2009.

[46] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[47] G. Bracha and S. Toueg. Resilient consensus protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*, pages 12–26, Montreal, Quebec, August 1983. ACM SIGOPS-SIGACT.

[48] Thomas C. Bressoud and M. Frans Kaashoek, editors. *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, 2007.

[49] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, 1993.

[50] M. Burrows. The Chubby Lock Service for loosely-coupled distributed systems. In *7th Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[51] M. Caesar, M. Castro, E. Nightingale, G. O'Shea, and A. Rowstron. Virtual Ring Routing: Network routing inspired by DHTs. In *Proc. of SIGCOMM'06*, Pisa, Italy, September 2006.

[52] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D.S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the 5th Usenix Symposium on Operation System Design and Implementation (OSDI)*, Boston, MA, December 2002.

[53] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, LA, February 1999. USENIX.

[54] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, 2003.

[55] S. Chandra and P.M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN'00)*, pages 97–106. IEEE CS Press, 2000.

[56] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*, pages 398–407, Portland, OR, May 2007. ACM.

[57] T.D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pages 325–340, Montreal, Quebec, August 1991. ACM SIGOPS-SIGACT.

[58] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[59] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. In *7th Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

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

[61] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *ICS*, pages 310–331. Tsinghua University Press, 2010.

[62] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, October 2009.

[63] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In Jennifer Rexford and Emin Gün Sirer, editors, *NSDI*, pages 153–168. USENIX Association, 2009.

[64] B.A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, 1988.

[65] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ

Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, pages 177–190. USENIX Association, 2006.

[66] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In Bressoud and Kaashoek [48], pages 205–220.

[67] D. Dolev and R. Reischuk. Bounds on Information Exchange for Byzantine Agreement. *J. ACM*, 32(1):191–204, 1985.

[68] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[69] E. N. Elnozahy, L. Alvisi, Y.M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[70] M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[71] T. Fuhrmann. The use of Scalable Source Routing for networked sensors. In *Proc. of the 2nd IEEE Workshop on Embedded Networked Sensors*, pages 163–165, Sydney, Australia, May 2005.

[72] H. Garcia-Molina, F. M. Pittelli, and S. B. Davidson. Applications of Byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[73] S. Ghermawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 29–43, Bolton Landing, NY, October 2003.

[74] J. Graham. E-mail carriers deliver gifts of nifty features to lure, keep users. USA Today, 2008. Also available as `http://www.usatoday.com/tech/products/2008-04-15-google-gmail-webmail_N.htm`.

[75] J. Gray. A transaction model. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 1980.

[76] J. Gray. A Comparison of the Byzantine Agreement Problem and the

Transaction Commit Problem. In Barbara B. Simons and Alfred Z. Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 10–17. Springer, 1986.

[77] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[78] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[79] M. Grottke and K. Trevedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer*, 40(2), February 2007.

[80] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. In *Proceedings of the 5th ACM European conference on Computer Systems (Eurosys'10)*, Paris, France, April 2010.

[81] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. of the 21st ACM Symp. on Operating Systems Principles*, Stevenson, WA, October 2007.

[82] M. Haridasan and R. Van Renesse. Defense against intrusion in a live streaming multicast system. In *Proc. of the Sixth IEEE International Conference on Peer-to-Peer Computing (P2P '06)*, Washington, DC, September 2006. IEEE Computer Society.

[83] Andrew Herbert and Kenneth P. Birman, editors. *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. ACM, 2005.

[84] M. Herlihy. A quorum consensus replication method for abstract data types. *Trans. on Computer Systems*, 4(1):32–53, February 1986.

[85] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463 – 492, 1990.

[86] C. Ho, D. Dolev, and R. Van Renesse. Making distributed applications robust. In *Proceedings of the 11th International Conference On Principles Of Distributed Systems (OPODIS'07)*, number 4878 in Lecture Notes on Computer Science, Guadeloupe, West Indies, December 2007. Springer-Verlag.

[87] H. Johansen, A. Allavena, and R. Van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. of Eurosys 2006*, Leuven, Belgium, April 2006.

[88] F. Junqueira, P. Hunt, M. Konar, and B. Reed. The ZooKeeper Coordination Service (poster). In *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[89] R. Kotla. *xBFT: Byzantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation*. PhD thesis, The University of Texas at Austin, Austin, TX, May 2008.

[90] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: speculative byzantine fault tolerance. In Bressoud and Kaashoek [48], pages 45–58.

[91] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E.L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.

[92] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

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

[94] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[95] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[96] L. Lamport. Lower bounds for asynchronous consensus. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[97] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.

[98] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *Trans. on Programming Languages and Systems*, 6(2):254–280, April 1984.

[99] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox PARC, Palo Alto, CA, 1976.

[100] I. Lee and R.K. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Trans. on Software Engineering*, pages 455–467, May 1995.

[101] H.C. Li, A. Clement, E.L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *Proc. of the 2006 USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, November 2006.

[102] J. Li and D. Mazières. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI*. USENIX, 2007.

[103] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). RFC 1105 (Experimental), June 1989. Obsoleted by RFC 1163.

[104] D. Malkhi and M.K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11:203–213, June 1998.

[105] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 369–384. USENIX Association, 2008.

[106] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[107] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[108] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. Updated by RFC 5709.

[109] D. Mpoeleng, P. Ezhilchelvan, and N. Speirs. From crash tolerance to authenticated byzantine tolerance: A structured approach, the cost and benefits. *Int. Conf. on Dependable Systems and Networks*, 2003.

[110] S.S. Mukherjee, J.S. Emer, and S.K. Reinhardt. The soft error problem: An architectural perspective. In *Proc. of the Symposium on High-Performance Computer Architecture*, February 2005.

[111] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *PODC '88: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* [31], pages 248–262.

[112] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In Herbert and Birman [83], pages 191–205.

[113] B. Nowicki. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989.

[114] B. M. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus—an architecture for extensible distributed systems. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 58–68, Asheville, NC, December 1993.

[115] B.M. Oki and B.H. Liskov. Viewstamped replication: A general primary-copy method to support highly-available distributed systems. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing* [31], pages 8–17.

[116] D. Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Informational), February 1990.

[117] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[118] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228 –235, 1961.

[119] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

[120] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and Public-Key cryptosystems. *CACM*, 21(2):120–126, February 1978.

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

[122] Y.J. Song, F. Junqueira, and B. Reed. BFT for the skeptics (WIP). In *Proc.*

of the Symposium on Operating Systems Principles (SOSP'09), Big Sky, MT, October 2009.

[123] Y.J. Song and R. Van Renesse. Bosco: One-step Byzantine asynchronous consensus. In Gadi Taubenfeld, editor, *DISC*, volume 5218 of *Lecture Notes in Computer Science*, pages 438–450. Springer, 2008.

[124] T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, June 1987.

[125] I. Stoica, R. Morris, D. Karger, and M.F. Kaashoek. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the '95 Symp. on Communications Architectures & Protocols*, Cambridge, MA, August 1995. ACM SIGCOMM.

[126] B. Stone. Amazon Offers Other Sites Use of Its Payment Service. NY Times, 2008. Also available as `http://www.nytimes.com/2008/07/30/technology/30amazon.html`.

[127] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay. Transactions and consistency in distributed database systems. *IBM Research Report*, RJ2555, 1979.

[128] R. van Renesse and F.B. Schneider. Chain Replication for supporting high throughput and availability. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 91–103, San Francisco, CA, December 2004.

[129] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, October 2007.

[130] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. In *Proc. of the 2007 EuroSys Conf.*, Lisbon, Portugal, 2007.

[131] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240 – 1255, 1978.

[132] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the Art of Practical BFT. In *EuroSys*, Salzburg, Austria.

[133] A.R. Yemerefendi and J.S. Chase. The role of accountability in dependable distributed systems. In *Proc. of the First Workshop on Hot Topics in System Dependability (HotDep'05)*, Yokohama, Japan, June 2005. IEEE.

[134] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 253–267. ACM, 2003.