Building Reliable and Practical Byzantine Fault Tolerance

By

Sisi Duan
B.S. (The University of Hong Kong) 2010
M.S. (University of California, Davis) 2011

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Dr. Karl Levitt (Co-Chair)

Dr. Sean Peisert (Co-Chair)

Dr. Matt Bishop

Committee in Charge

2014

To my family, for everything in my life.

Building Reliable and Practical Byzantine Fault Tolerance

Abstract

Building online services that are both highly available and correct is challenging. Byzantine fault tolerance (BFT), a technique based on state machine replication [72, 101], is the only known *general* technique that can mask *arbitrary* failures, including crashes, malicious attacks, and software errors. Thus, the behavior of a service employing BFT is indistinguishable from a service running on a correct server.

This dissertation presents three practical BFT protocols, *h*BFT, BChain, and ByzID. Each protocol takes a different approach enhance the practicality of existing practical BFT protocols under certain network conditions and threat models. *h*BFT moves some jobs to the clients with minimum cost. The protocol is much simplified while faulty clients are tolerated. BChain uses chain replication while faulty replicas are diagnosed and eventually reconfigured. ByzID uses intrusion detection methods to build a Byzantine failure detector. Faulty replicas are detected immediately and performance attack can be perfectly handled. In the end, we present P2S, a general framework of adapting existing fault tolerance techniques to pub/sub, with the aim of reducing the burden of proving the correctness of implementation. The experimentation results validate all the work, showing different degree of performance improvement over traditional protocols.

Acknowledgments

First, I must thank my advisors, Karl Levitt and Sean Peisert, for their constant support and mentoring. They have guided me not only in my work and research but also in my life. I am so fortunate to have been able to work closely with them.

I am also lucky to work closely with Hein Meling. He gave me a lot wise advice and support in my research. I greatly appreciate his help, especially during my visit in Norway.

The other professors and mentors in security group, Matt Bishop, Felix Wu, and Jeff Rowe gave me many suggestions to my work and my dissertation. It has been a pleasure to work with them.

I am grateful to be a graduate student at UC Davis. I want to thank all my labmates and friends for making my PhD life an amazing and unforgettable journey: Yaohua Feng, Andy Chih, Tiancheng Chang, Yun Li, Yuxi Hu, Fei Yu, Jinrong Xie, Shengren Li, Xi Jiang, Ye Zhang, Sharmin Jalai, Mohammad Rezaur Rahman, Yi Zhang, Jia Liu, Wei Liu, Haifeng Zhao, Xin Sun, Yixuan Zhai, Mianfeng Zhang, Yuanzhe Li, Changyung Lin, and Mina Doroud. Special thanks to Vincent Tam, for not only being my undergraduate advisor, but also a friend in my life.

Above all, I want to thank my husband, Haibin Zhang, for being my friend and my co-author; my parents and all my family, for giving me everything in my life. Without their support I cannot make it through the whole process.

This research is based on work supported in part by the National Science Foundation under Grants Number CCF-1018871, CNS-0904380, and CNS-1228828. The ByzID work was also supported in part by a Leiv Eiriksson Mobility Grant from RCN.

The following papers, which have been previously published or are currently in

submission, are reprinted in this dissertation with the full permission of all co-authors of the papers:

- hBFT: Speculative Byzantine Fault Tolerance With Minimum Cost. Sisi Duan, Sean Peisert, and Karl Levitt. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, March 2014.

- BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. Sisi Duan, Karl Levitt, Sean Peisert, and Haibin Zhang. *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*, to appear, 2014.

- Byzantine Fault Tolerance from Intrusion Detection. Sisi Duan, Karl Levitt, Hein Meling, Sean Peisert, and Haibin Zhang. To appear in *Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pp. 253–264, 2014.

- P2S: A Fault-Tolerant Publish/Subscribe Infrastructure. Tiancheng Change, Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. *Proceedings of the 8th ACM International Conference on Distributed Event Based Systems (DEBS)*, pp. 189-197, 2014.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

As distributed systems become used increasingly widely, and in critical systems, Byzantine failures generated by malicious attacks, and software and hardware errors must be tolerated. Building online services that are both highly available and correct is challenging. Byzantine fault tolerance (BFT), a technique based on state machine replication [72, 101], is the only known *general* software technique that can mask *arbitrary* failures, including crashes, malicious attacks, and software errors. The behavior of a service employing BFT is indistinguishable from the behavior of a non-replicated service running on a non-faulty server. However, Byzantine protocols come at a cost of high overhead of messages and latency and cryptographic operations. Therefore, protocols that can reduce overhead can be attractive building blocks to support applications using these services such as storage systems [1, 23, 88] and database systems [32].

1.1 Challenges

There are a few challenges in designing practical BFT protocols. First, BFT can be too computationally expensive to be practical. BFT protocols usually introduces significant overhead in peak throughput and latency compared to unreplicated service. The number of cryptographic operations of the each server, which is directly related to the number of messages in the protocol, is the key to the overall performance. Since BFT usually involves several rounds of communication, the overhead caused by the number of cryptographic operations can be high. In addition, due to the design of protocols, each protocol works under certain network conditions and threat models. It is challenging to design a "universal" protocol that is adaptive to different conditions. For instance, PBFT [18] works well under contention, and HQ [34] works well under low contention. Second, without using additional tools, BFT protocols are known to tolerate f failures using at least $3f + 1$ replicas [78]. This directly limits the scalability of the protocol. When the replicas grow in size in wide area network, the overhead of both server deployment and communication in the protocol grows accordingly. Third, only up to f failures are tolerated while the remaining replicas must remain correct. Therefore, the same Byzantine failure affecting multiple systems simultaneously and it is desirable to obtain implementations on different operating systems or implement services through N-version programming. Fourth, BFT protocols require high resilience where protocol remain correct in the long run. In a long-lived system, more failures may occur during the time. It is highly possible that more than f replicas fail, rendering the protocol incorrect and replicas inconsistent. Therefore, in addition to tolerating failure, it is also necessary to ensure that faulty replicas are eventually recovered so that the number of faulty replicas not just continue to grow and eventually exceeds f.

1.2 Contributions

The goal of our research is to design and implement highly reliable, replicated BFT protocols to overcome certain challenges in building practical BFT protocols. We first developed hBFT, a hybrid protocol that moves some jobs to the clients with minimum cost. It has been shown [54, 69] that by moving some jobs to the clients the performance of traditional BFT protocols can be improved. However, it usually depends on the assumption that clients are correct. Otherwise, it may consume other resources to guarantee correctness. For instance, Zyzzyva5 [69] requires $5f + 1$ replicas to ensure safety. Also, to improve the performance in failure-free cases, performance under replica failure may be sacrificed. hBFT, as a hybrid protocol, moves some jobs to the clients while simultaneously tolerating faulty clients. In addition, the performance under backup failure(s) is the same as the failure-free case. Second, we developed BChain, a chain replicated protocol with failure reconfiguration. Chain replication [50,107,108] is known to enjoy the benefits of high throughput and low latency under contention. However, previous developed chain protocols [50,107] do not tolerate Byzantine failures. Byzantine chain replication [108] employs a centralized trusted computing base to tolerate Byzantine failures. In comparison, BChain uses two key techniques to handle Byzantine failures without using additional trusted authorities: peer-to-peer failure suspicion and reconfiguration. The peer-to-peer failure suspicion scheme guarantees that within a certain number of rounds of suspicion, all faulty replicas are eventually moved to the end of the chain. The reconfiguration scheme ensures that the replicas moved to the end of the chain are replaced by new ones. Correct replicas may be suspected and reconfigured. However, our protocol guarantees that all the faulty replicas are reconfigured within certain rounds of peer-to-peer suspicion. Third, we developed ByzID, a Byzantine fault tolerant protocol

that leverages intrusion detection methods. Each replica is equipped with a trusted intrusion detection component that monitors the behavior of the replica. When an alert is generated by the IDS, the corresponding replica is replaced by a new one. Finally, we developed P2S, a crash tolerant Paxos-based [73] pulish/subscribe (pub-/sub) middleware. It directly adapts existing fault tolerance techniques to pub/sub, with the aim of reducing the burden of proving the correctness of the implementation. It is also provides a generic development framework for building various of pub/sub applications under different models.

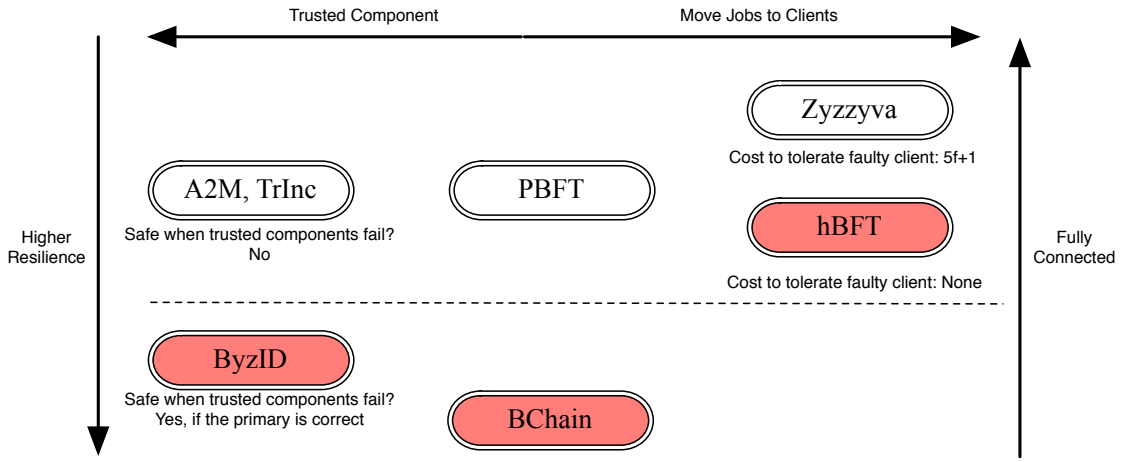


Figure 1.1. Comparison of the protocols.

In comparison to existing approaches, our research primarily has the following contributions, as illustrated in Fig. 1.1. We focus on *h*BFT, BChain, and ByzID since they are Byzantine fault tolerant protocols, while P2S is considered as a general middleware of using existing fault tolerant protocols in pub/sub systems.

1.2.1 Improving the performance

All the three protocols improve performance of existing state-of-the-art protocols such as Zyzzyva and PBFT. They overcome the challenges and improve the performance of different aspects. We will now discuss in the following.

Moving jobs to the clients. It has been shown [54,69] that by moving some jobs to the clients, the performance of traditional protocols, such as PBFT can be improved. For instance, Zyzzyva reduces the normal case operation of PBFT from three phases to two, because replicas do not need to exchange the certificate of messages twice. Instead, the clients collect messages from replicas and send the certificate to the replicas only if necessary. Therefore, in the failure-free case, the performance can be significantly improved. However, when there are failures, the performance may even decrease. In addition, it requires $5f + 1$ replicas to tolerate faulty clients.

*h*BFT moves jobs to the clients without being encumbered by some of these trade-offs. In the failure-free case, the protocol is the same as Paxos [73], which also contains two phases. The clients can help detect the faulty replicas. If the clients are faulty, the replicas may be inconsistent temporarily. *h*BFT employs a PBFT-like three phase checkpoint protocol to both perform garbage collection and detect the inconsistency of replicas. Replicas recover through the checkpoint protocol. If the clients are suspected to be faulty, they are prevented from sending further requests.

Using a partially connected graph. Compared to a complete graph where replicas exchange messages through multicast communication channel, a partially connected graph [15] is known to consume fewer resources in communication and therefore can improve performance.

Both BChain and ByzID use a partially connected graph. BChain uses chain replication, where replicas are ordered in a metaphorical chain. Each replica send-

s/receives messages to/from two replicas in total unless failure occurs. When there are a lot of concurrent requests, the pipelining communication pattern helps the bottleneck replica (the replica that performs the most cryptographic operations) perform fewer cryptographic operations. On the other hand, ByzID uses a primary backup approach, where the primary can send/receive messages to/from the backups and the backups only send/receive messages to/from the primary. Since the backups do not exchange messages through the multicast channel, the protocol is simplified and the overall performance increases.

1.2.2 Tolerating more failures using trusted IDS component

ByzID relies on trusted intrusion detection components to tolerate f replicas using at least $2f + 1$ replicas. We have designed a general framework for constructing Byzantine failure detectors based on specification-based intrusion detection systems (IDS). As a result, the protocol is the same as the failure-free case of Zyzzyva. In addition, when there are no failures, the IDS component passively monitors and analyzes the messages, which introduces little overhead.

1.2.3 Enhancing the resilience

Resilience refers to the relation between the number of potentially faulty replicas and the total number of replicas in the system. Each protocol is known to tolerate certain portion of faulty replicas. In the long run, to make the system robust, protocols must always be resilient to failures so as to remain correct. Therefore, it is necessary to detect, diagnose, and recover faulty replicas. Both BChain and ByzID use replica reconfiguration. In BChain, replicas suspect each other and send messages to the head, which is the leader of the replicas. The head of the chain reassigns the order of

the replicas and moves the suspected replicas to the end of the chain. The suspected replicas are reconfigured and replaced by new replicas. The reconfiguration of replicas operates out-of-band, where all other replicas continue to run without waiting for the reconfiguration procedures to complete. Therefore, it creates minimal overhead for the protocol. Eventually, all the faulty replicas that have behaved incorrectly are replaced by new replicas.

On the other hand, the IDS component of replicas in ByzID monitors the behavior of each replica. If the IDS component at a replica generates an alert, the replica is reconfigured. The primary reconfiguration operates in-band, where replicas wait for the reconfiguration procedures to complete. The backup reconfiguration operates out-of-band. Overall, the reconfiguration process causes minimum overhead since any protocols need to call a process, such as view change to replace the faulty primary.

1.2.4 Preventing from performance attack

Legal but uncivil behaviors of replicas can make BFT protocols impractical. This has been previously discuss in several papers [5, 29] about performance attacks on PBFT. For instance, the faulty primary may manipulate the value of timeouts and decrease the overall performance without being noticed.

Although our protocols are not vulnerable to the same performance attack, we discussed the solutions to several possible performance attacks. In BChain, replicas may manipulate the timeouts to decrease the performance. We explore a method of adjusting timeouts to defend against the performance attack. Each replica consistently monitors the time of sending and receiving messages and adjust the timeouts accordingly. Eventually, the uncivil replicas can decrease the performance to a certain threshold.

In ByzID, due to the use of specification-based intrusion detection systems, we are able to monitor the behaviors of all replicas. We design several specifications to defend against a performance attack. For instance, the *timely action* specification is used to monitor whether a replica responds to a message in a timely manner such that any uncivil replicas cannot intentionally decrease the performance. On the other hand, the *fairness* specification is used to monitor whether the primary orders the requests according to the order of receiving them or any ordering policies according to the requirement. In comparison to the solution where replicas monitor each other, the solution using an external trusted component ensures stronger properties.

1.3 Organization

The following sections are organized as follows. We first discuss the system model, background, and related work in Chapter 2. In Chapter 3 to Chapter 5 we present the three major Byzantine fault tolerant protocols, *h*BFT, BChain, and ByzID respectively. We also include P2S in Chapter 6, a general framework for building reliable pub/sub systems based on an existing fault tolerant protocol. In Chapter 7, we compare the overall performance of all four protocols presented in this dissertation and discuss the strength and weakness of each protocol. We also discuss the feasibility of using a fault tolerance library as an oracle based on our P2S protocol. Finally, we conclude dissertation and discuss future work in Chapter 8.

Chapter 2

Background

2.1 System Model

State machine replication is the only known general approach that can be used to replicate any service that can be modeled as deterministic state machine replication. Such replicated state machine provides the same service with unreplicated state machine.

We assume a system that can tolerate a maximum of f faulty replicas, using a total of n replicas. In some of the chapters, we write t, where $t \leq f$, to denote the number of faulty replicas that the system currently has. In BFT protocols, clients are involved. A client issues requests to invoke operations and waits for replies.

Replicas may be connected in a complete graph or an incomplete graph network. However, for wide-area deployments, only a complete graph network makes sense. We assume *fair-loss links*, where if a message is sent infinitely often by a correct sender to a correct recipient, then it is received infinitely often. Furthermore, links do not produce spurious messages and do not repeatedly perform more transmissions than performed by the sender.

Note that one can use fair-loss links to build *reliable links*, but only when both the sender and receiver are correct. However, our protocol needs to establish how to build reliable links from fair-loss links even when the sender is potentially (Byzantine) faulty. We therefore assume the fair-loss link abstraction. In ByzID, we use Intrusion Detection Systems (IDSs) to monitor the behavior of replicas. We further assume that adversaries are unable to inject messages on the links between the replicas. This is reasonable when all replicas are monitored by IDSs and they reside in the same administrative domain. We assume that IDSs are trusted components, but that they may fail by crashing.

We use non-keyed *message digests*. The digest of a message m is denoted $D(m)$. We also use *digital signatures*. The signature of a message m signed by replica p_i is denoted $\langle m \rangle_{p_i}$. We say that a signature is *valid* on message m, if it passes the verification w.r.t. the public-key of the signer and the message. A vector of signatures of message m signed by a set of replicas $\mathcal{U} = \{p_i, \ldots, p_j\}$ is denoted $\langle m \rangle_{\mathcal{U}}$.

In the following of the dissertation, we use the notions mentioned above. In case of any notation difference, we will mention in the corresponding chapters.

2.1.1 Faulty behaviors

We classify the replica failures according to their behaviors. Weak semantics levy fewer restrictions on the possible behaviors than strong semantics. We are interested in various failure semantics. *Crash failures*, occur when the replicas might halt permanently and no longer produce any output. By *timing failures*, we mean any replica failures that produce correct results but deliver them out of a specified time window. We also consider Byzantine failures, where faulty replicas can behave arbitrarily and a computationally bounded adversary can coordinate faulty replicas to compromise

the system.

2.1.2 Service property

A correct state machine replication protocol offers both *safety* and *liveness* provided that at most f out of a total of n replicas are simultaneously faulty. The value of f regarding n depends on both the failure semantics and the protocol. In the four protocols we discuss in the following chapters, hBFT, BChain, and ByzID tolerate Byzantine failures and P2S tolerates crash failures. hBFT and BChain tolerate f Byzantine failures using at least $3f+1$ replicas. ByzID tolerates f Byzantine failures using at least $2f + 1$ failures. As we will discuss in Chapter 5, ByzID tolerates more failures than hBFT and BChain by employing intrusion detection techniques in the protocol. Finally, P2S is a general framework in publish/subscribe systems. As discussed in the following chapters about the semantics in pub/sub systems, it tolerates f crash failures using at least $2f + 1$ brokers.

Safety, which means requests are totally ordered by correct replicas, must hold in any asynchronous system using state machine replication, where messages can be delayed, dropped or delivered out of order. Liveness, which means correct clients eventually receive replies to their requests, is ensured assuming partial synchrony [42]: synchrony holds only after some unknown global stabilization time, but the bounds on communication and processing delays are themselves unknown.

2.2 Fault tolerant state machine replication

Fault tolerance. We focus our discussion on Lamport's formulation of Paxos-style [73–75] consensus. Paxos and its variants tolerate f crash failures using at least

$2f + 1$ replicas. PBFT [18], the first practical Byzantine fault tolerant protocol, is often viewed as a three-phase Paxos that tolerates f Byzantine failures using at least $3f+1$ replicas. Several state-of-the art protocols aimed at enhancing the performance by (1) Simplifying the protocol; and/or (2) Tolerating more failures with respect to the same number of replicas.

It was shown that by directly eliminating one phase in PBFT [69, 85], the performance can be enhanced. This is straightforward since the number of messages and cryptographic operations can be reduced by around one third. However, this is achieved at the cost of introducing additional requirements on the protocol. For instance, clients must be trusted in Zyzzyva [69]. In order to tolerate faulty clients however, $5f + 1$ replicas must be used to tolerate f failures. hBFT and ByzID also employ a two-phase protocol similar to Zyzzyva in the failure free cases. In hBFT, it is possible that replicas are temporarily inconsistent. Replicas can be made consistent by clients or replicas in other subprotocols. ByzID relies on the trusted IDS component to guarantee correctness.

van Renesse and Schneider [108] first developed chain replication, and used it to achieve high throughput and availability for replicated services in the crash failure model. Following that first work [108], Aliph-Chain [50], explored how to secure chain replication for Byzantine failures. In Aliph-Chain, requests are required to be transmitted in a predetermined order (through authentication), and the tail is responsible for sending replies to clients. However, Aliph-Chain itself does not attain liveness unless all the replicas are correct, because even one crash failure can cause it to abort indefinitely. According to Vukolic [111], BChain (as discussed in Chapter 4) can be viewed as a protocol with "the strongest condition" by enhancing chain replication with a "weak condition," Aliph-Chain.

van Renesse, Ho, and Schiper [107] later proposed a *Byzantine chain replication*

protocol (and an implementation called "Shuttle"), that can tolerate f failures among $2f + 1$ replicas. However, the protocol relies on a strong assumption. Namely, it requires a trusted and Byzantine-fault resilient server to help achieve system liveness. Each replica has to share all of its secret keys with trusted server (if MACs are used). To implement such a trusted server, one would require yet another BFT protocol. Furthermore, in order to prevent adversaries from attacking the trusted server, the protocol must resort to a voting mechanism to avoid the leakage of secret keys.

Failure detectors. Failure detectors were introduced by Chandra and Toueg [22] for solving consensus problems in the presence of crash failures. For each replica, a failure detector outputs the identities of each replica that it detects to have crashed. A *perfect* failure detector should satisfy the *completeness* and *accuracy* properties. The former demands that all faulty replicas be detected, while the latter requires that correct replicas never be falsely implicated. Several papers [8, 36, 83], such as quiet process [83] and muteness detector [36], describe extensions to failure detectors to address Byzantine failures and use them to solve consensus problem. Byzantine failures, in contrast to crash failures, are not context-free, so it is not possible to define and design failure detectors independently of the underlying protocols [36]. Therefore, for instance, consensus protocols from a muteness detector [38] have to handle Byzantine failures other than mute failures at the algorithmic level. Moreover, consensus protocols that use extended Byzantine failure detectors are not yet practical since they can only detect certain type of failures instead of "arbitrary" failures.

Failure detection is also studied under a different name, *fault diagnosis*, which goes beyond failure detection in that the former aims to determine what kind of fault occurs and which components are responsible, while the latter only seeks to determine that a fault occurred. One of classic formulations of system fault diagnosis was

developed by Preperata, Metze, and Chien [95] and the extended studied further [3, 96,102,112]. In ByzID as discussed in Chapter 5, an IDS associated with the primary also serves as a Byzantine failure detector.

Shin and Ramanathan [103] presented the first study on how to identify faulty processors in Byzantine consensus protocols. A number of extensions to that work have also been made [55, 113, 116]. The basic idea is that a *proof of misbehavior* for a Byzantine fault is collected by executing a modified BFT protocol. However, it requires several rounds of protocols to collect a huge volume of exchanged messages to provide such proof. An adversary can render the system even less practical by intermittently following and violating the protocol specification. Similarly, PeerReview [53] can detect and deter failures by exploiting accountability. It builds a system that replicas review and report the failure of other replicas. It ensures that faulty behavior is detected and no correct node is observed to be faulty through the use of secure logging and auditing techniques. Reputation systems such as EigenTrust [62] can also be used to detect a family of Byzantine faults but they typically detect only repeated misbehavior. BChain, as discussed in Chapter 4, achieves fault diagnosis though is *not* perfectly accurate. No evidence is required to be regularly collected, and no additional latency is introduced by intermittent adversaries.

State machine replication based on trusted components. Equivocation refers to the behavior of an adversarial component that lies to other components in different ways. This problem is precisely captured by the well-known the Byzantine generals problem [78]. It was shown that the problem (and any consensus problem) cannot be solved if more than one third of its processes are faulty. Fitzi and Maurer [46] showed that with the existence of a "two-cast channel" (i.e., broadcast channel among three players), Byzantine agreement is achievable if and only if the number of faulty processes is less than a half. The result was later extended [31] for general multicast

channels.

Beginning with Correia *et al.* [33], a number of BFT approaches relying on (small) trusted components to prevent equivocation and circumvent the one-third bound have been developed, including A2M [26], TrInc [80], MinBFT and MinZyzzyva [110], and CheapBFT [63]. All of these require only $2f + 1$ replicas to tolerate f failures, and they have to rely on signatures [27]. A2M uses trusted and append only logs that limits the behavior of adversarial components and prevents them from deviating from the correct cases. TrInc and MinBFT and MinZyzzyva use a trusted subsystem that provides a monotonically increasing counter to guarantee that one message is assigned with only one incremental counter value. Each replica is equipped with a trusted component that signs and verifies the message and the counter value. If a correct replica receives one message, it can be sure that no other replica ever receives a message with the same counter value but different content. CheapBFT [63] further develops the idea and explores how to use $f+1$ replicas for gracious execution, while during uncivil executions it switches to MinBFT and thus uses again $2f+1$ replicas. van Renesse, Ho, and Schiper [107] also proposed Shuttle, that can tolerate f failures among $2f + 1$ replicas. The protocol relies on a trusted and Byzantine-fault resilient server to achieve liveness.

ByzID also falls into the category of using trusted components, but we deploy an IDS that is not only more powerful but also simpler. Our approach achieves better efficiency than the prior BFT protocols (with or without trusted components) both during failures and in the absence of failures. We also use new approaches to design our protocols in that ByzID does not use any signatures. But this does not contradict the impossibility result of Clement *et al.* [27] that non-equivocation alone does not allow for reducing the number of processes required to reach Byzantine agreement in asynchronous environment, as we use other mechanisms to handle this.

Enhancing resilience. Another approach in BFT research has been the study of how to improve the resilience under active attacks, such as Aardvark [29], Prime [5], and Spin [109]. It was studied in previous work that uncivil behaviors of replicas can also render the system slow. For instance, in PBFT, a faulty primary can delay sending messages to replicas while not being replaced, which is denoted as *timeout manipulation*. Aardvark and Prime enhance the resilience based on PBFT. In addition, Spin builds a rotating leader mode based on PBFT to prevent from timeout manipulation. In BChain and ByzID, we also discuss the methods to enhance the resilience although they are different from a PBFT-like protocol. In BChain, we use an adaptive adjustment on the values of the timers to prevent faulty replicas from manipulating timeouts. Other than that, in ByzID, the IDS monitors the behaviors of the replicas to enhance the resilience. One advantage of using trusted components to achieve this is that some of the behaviors can not be detected by other replicas. For instance, it achieves perfect fairness where the primary must order the incoming requests in a certain order.

2.3 Intrusion detection systems

Specification-based intrusion detection. Specification-based intrusion detection was proposed by Ko, Ruschitzka, and Levitt [68] as a means of detecting exploitations of vulnerabilities in security-critical programs. In such a system, a sequence of ordered events during the execution of a system is defined as system trace. A specification defines the desirable sequence of execution that specifies the intended behavior of the system. If one trace deviates from any valid system specification, it is regarded as security violation.

Specification-based approaches require accurate specifications of the desirable sys-

tem behaviors, therefore having the ability of encompassing anomaly behaviors that have not previously been exploited. Moreover, since the specification-based approach is built upon manually-defined legitimate system behaviors, it can significantly decrease false positive rates [106].

Anomaly-based intrusion detection. Anonmaly-based intrusion detection was proposed by Denning [35] as a means of detecting anomalous system activities. In such a system, normal system activities are first defined in several ways, such as with machine learning techniques and mathematical models. During the execution of a system, anomalous behaviors are regarded as security violation.

Anomaly-based intrusion detection uses techniques to define normal behaviors, which does not rely on manual efforts. However, since the techniques to define desirable system behaviors [105] are not accurate enough, it may result in high false positive rate.

2.4 Reliable publish/subscribe systems

Publish/subscribe systems involve three roles: 1) publishers who publish publications, which will be received by subscribers; 2) subscribers who subscribe to certain content or topic through subscriptions, which will be received by publishers; and 3) brokers who deliver publications or subscriptions between publishers and subscribers.

The publish/subscribe communication pattern for constructing event notification services has strong performance and flexibility characteristics. While typical "pub/-sub" services such as consumer RSS news feeds may tolerate some level of message loss, enterprise applications often demand stronger dependability guarantees. As a result, pub/sub has become an important cloud computing infrastructure and is widely used in industry, e.g., in Google GooPS [98], Windows Azure Service Bus [97],

Oracle Java Messaging Service [90], and IBM WebSphere [16].

The topic of constructing reliable pub/sub systems has been widely studied [13, 20,43,59,64,65,94,104,120]. By using periodic subscription [59], subscribers actively re-issue their subscriptions. By flooding the messages, this can prevent message loss and ensure subscribers eventually receive all the publications to their subscriptions. On the other hand, through event retransmission [20,43], brokers exchange acknowledgment messages to ensure that the corresponding messages are delivered. Both periodic subscription and event retransmission work well in preventing message loss instead of handling broker/link failures. In order to guarantee that messages are correctly delivered in the presence of broker/link failures, several papers have proposed redundant paths [20,64,65,104], where the overlay topology includes redundant paths to ensure that at least one path between the corresponding publisher and subscriber is correct. For instance, Gryphon [13] uses virtual brokers, where each broker maps to one or more physical brokers, such that at least one broker is correct and forwards the messages along the path. Indeed, the most straightforward way to use redundant paths is to replicate every broker. However, this may consume high bandwidth and become very inefficient in the absence of failures. Furthermore, prior work in this area usually ensures that messages or events are delivered, where the order of events are not considered.

There has been considerable work in developing total order algorithms [14,89]. A class of algorithms arranges brokers into groups and uses interactions between groups to compute message order [93]. This type of solution works well under static topology since group membership knowledge can be difficult to maintain in dynamic networks. On the other hand, it is natural to use a single sequencer or several decentralized sequencers [81,115] to handle message order. A single sequencer is easier to maintain but is a single point of failure. In contrast, decentralized sequencers are more resilient

to failure but require every message to be routed to a certain sequencer. This imposes topology constraints and can be less efficient.

Several efforts [65,120] exploit the topology overlay in pub/sub systems to achieve certain total ordering properties in the presence of broker/link failures. Kazemzadeh et al. [65] use a tree-based topology and achieve per-publisher total order by having each broker forward redundant messages to several brokers. A stronger pairwise total order is achieved by Zhang et al. [120], where the intersecting broker of different paths resolves the possible conflicts of message order. However, this has a more complex algorithm to handle broker failures and is less efficient in the presence of failures. In comparison, P2S takes the simplest yet effective topology and algorithm to achieve pairwise total ordering in the presence of failures. In addition, the flexibility of the framework and our fault tolerance library make it easy to adapt to more scalable systems.

Fault tolerance techniques for highly available stream processing usually consider that no data is dropped or duplicated [49, 57, 58, 70]. Most of them assume a failover model and require $f + 1$ replicas to mask up to f simultaneous failures. Similar to some of the pub/sub approaches, replicated replicas ensure that at least one correct replica continues processing. When an upstream replica fails, the downstream replica switches to another correct upstream replica. Since at least one correct path exists between the source and destination, the data stream can be delivered. SGuard [70] uses replicated file systems to achieve fault tolerance. Each data chunk is replicated on multiple nodes. The data sent by a client is spread to all replicated nodes so that at least one piece is available. It also relies on a single fault-tolerant coordinator using rollback and recovery.

Chapter 3

*h*BFT: Speculative Byzantine Fault Tolerance With Minimum Cost

The work presented in this chapter was first described in an earlier paper by Duan, et al. [40]. We present *h*BFT, a hybrid, Byzantine fault-tolerant, replicated state machine protocol with optimal resilience. Under normal circumstances, *h*BFT uses speculation, i.e., replicas directly adopt the order from the primary and send replies to the clients. As in prior work such as Zyzzyva, when replicas are out of order, clients can detect the inconsistency and help replicas converge on the total ordering. However, we take a different approach than previous work. Our work has four distinct benefits: it requires many fewer cryptographic operations, it moves critical jobs to the clients with no additional costs, faulty clients can be detected and identified, and performance in the presence of client participation will not degrade as long as the primary is correct. The correctness is guaranteed by a three-phase checkpoint subprotocol similar to PBFT, which is tailored to our needs. The protocol is triggered by the primary when a certain number of requests are executed, or by clients when

they detect an inconsistency.

3.1 Introduction

A number of existing protocols also reduce overhead on Byzantine agreement by moving some critical jobs to clients [34, 50, 54, 69, 118, 119]. But these protocols come with trade-offs that we seek to avoid. Specifically, while they all provide better performance in fault-free cases and reduce the message complexity, they sacrifice the performance of normal cases and may even decrease the performance of fault-free cases. For instance, the Zyzzyva [69] protocol is able to use roughly half of the amount of messages and cryptographic operations that PBFT [18] requires. However, Zyzzyva's performance can be even worse than PBFT if at least one backup fails. Additionally, these protocols simplify the design by involving clients in the agreement. However, they all require clients to be correct in order to achieve protocol correctness.

Therefore, our motivation for developing a new protocol is to improve performance over PBFT without being encumbered by some of these trade-offs. Specifically, we have three key goals: first, we wish to be able to show how critical jobs can be moved to the clients without additional costs. Second, we wish to tolerate Byzantine faulty clients. Third, we define the notion of *normal case*, which means the primary is correct and there is at least one faulty backup while the number of faulty backups does not exceed the threshold. We wish to provide better performance for both fault-free cases and normal cases.

This chapter presents hBFT, a leader-based protocol that uses speculation to reduce the cost of Byzantine agreement, while also maintaining optimal resilience, utilizing $n \geq 3f + 1$ replicas to tolerate f failures. hBFT satisfies all of our stated

goals. To accomplish this, hBFT employs several techniques. First, it uses speculation: backups speculatively execute requests ordered by the primary as well as replies to the clients. As a result, correct replicas may be temporarily inconsistent. Additionally, hBFT employs a three-phase PBFT-like checkpoint subprotocol for both garbage collection and contention resolution. The checkpoint subprotocol can be triggered by the replicas when they execute a certain number of operations, or by clients when they detect the divergence of replies. In this way replicas are able to detect any inconsistency through internal message exchanges. Even though the three-phase protocol is expensive, it is not triggered frequently. Eventually hBFT can ensure the total ordering of requests for all correct replicas with very low cost.

3.1.1 Motivation

Our goal for hBFT is to offer better performance by moving some critical jobs to the clients while minimizing side effects that can actually reduce performance in many cases in previous work [50, 69, 118, 119].

First, hBFT moves some critical jobs to the clients without additional cost. Moving critical jobs to the clients is effective in simplifying the design and reducing message complexity, partly because replicas do not need to run expensive protocols to establish the order for every request. Nevertheless, it does not necessarily make protocols more practical. Indeed, it may sacrifice performance in normal and even fault-free cases. For instance, the output commit in Zyzzyva renders both fault-free case and normal case slower. hBFT achieves a simplified design and better performance for both fault-free and normal cases.

Second, hBFT can tolerate an unlimited number of faulty clients. Previous protocols all rely on the correctness of clients. However, Byzantine clients can dramatically

decrease performance. For instance, in the protocols that switch between subprotocols [50, 118, 119] (called abstracts in [50]), a faulty client can stay silent when it detects the inconsistency. Even if the next client is correct and makes the protocol switch to another subprotocol, replicas are still inconsistent because of this "faulty request." Similarly, in Zyzzyva, faulty clients can stay silent when they are supposed to send a commit certificate to make all correct replicas converge. Faulty primaries in this case can not be detected, eventually leading to inconsistencies of replica states. Faulty clients can also intentionally send commit certificates to all replicas even if they receives $3f + 1$ matching messages, which decreases the overall performance.

Third, hBFT has the same operations for both the fault-free and normal cases. This shows that in leader-based protocols, when the primary is correct, all the requests are totally ordered by all correct replicas. Previous protocols all achieve impressive performance in fault-free cases while they employ different operations when failure occurs, resulting in lower performance. Although Zyzzyva5 [69] makes the faulty cases faster, it requires $5f + 1$ replicas to tolerate f failures. In hBFT, we achieve better performance in both normal fault-free and normal cases using $3f + 1$ replicas.

3.2 The *h*BFT Protocol

The hBFT protocol is a hybrid, replicated state machine protocol. It includes four major components: (1) agreement, (2) checkpoint, (3) view change, and (4) client suspicion. As illustrated in Fig. 3.1, we employ a simple agreement protocol for fault-free and normal cases, and use a three-phase checkpoint subprotocol for contention resolution and garbage collection. The checkpoint subprotocol can be triggered by replicas when they execute a certain number of requests or by clients if they detect

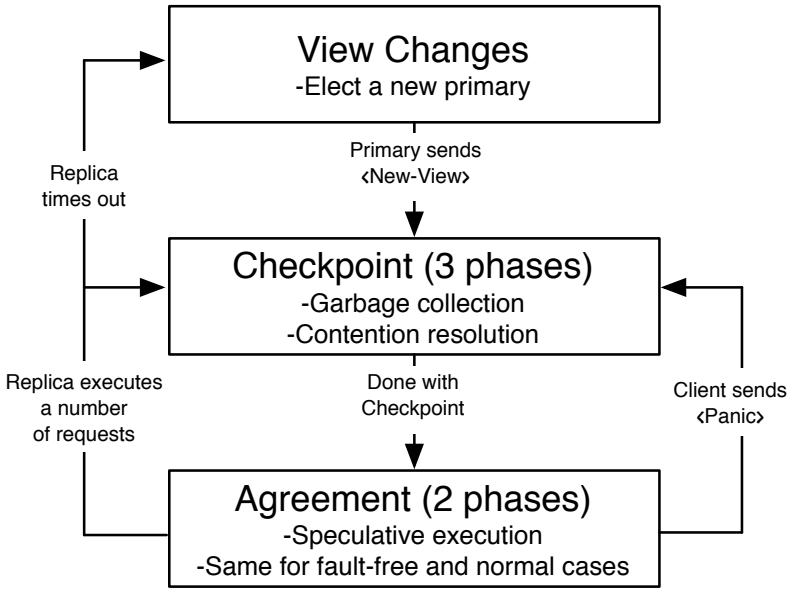
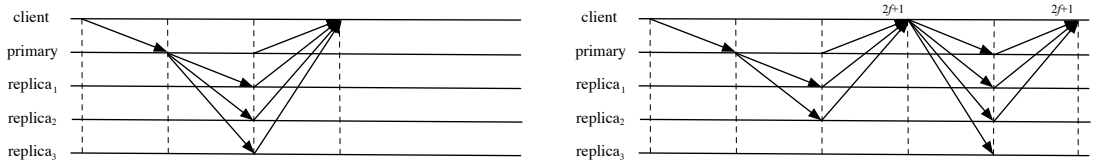


Figure 3.1. Layered Structure of hBFT.

divergence of replies. The view change subprotocol ensures liveness of the system and can coordinate the change of the primary. View changes can occur during normal operations or in the checkpoint subprotocol. In both cases, the new primary initializes a checkpoint subprotocol immediately and resumes the agreement protocol until a checkpoint becomes stable. The client suspicion subprotocol prevents faulty clients from attacking the system.



(a) Fault-free Case (b) Normal Case

Figure 3.2. Fault-free and normal cases of Zyzzyva.

Why another speculative BFT protocol?

*h*BFT uses speculation but overcomes some that problems Zyzzyva experiences. Zyzzyva [69] also uses speculation and moves output commit to the clients to enhance the performance. If we replace digital signatures with MACs and batch concurrent requests in Zyzzyva, the performance decreases in normal cases and even fault-free cases. Fig. 3.2 illustrates the behavior of Zyzzyva [69]. Replicas speculatively execute the requests and respond to the client. The client collects $3f+1$ matching responses to complete the request. If the client receives between $2f+1$ and $3f$ matching responses, it sends a commit certificate to all replicas, which contains the response with $2f+1$ signatures. This helps replicas converge on the total ordering. However, a commit certificate must be verified by every other replica, which causes computing overhead for both clients and replicas. The use of MACs instead of digital signatures makes Zyzzyva perform even worse than PBFT under certain configurations.[1] For a reply message r by replica p_i, $\langle r', \mu_{i,c}(r')\rangle$ must be sent to the client, where $r' = \langle r, \mu_{i,1}(r), \mu_{i,2}(r) \cdots \mu_{i,n}(r)\rangle$ and $\mu_{x,y}(r)$ denotes the MAC generated using the secret key shared by p_x and p_y. Therefore, every replica must include $3f+1$ MACs for every reply message (compared with 1 if digital signatures are used) and performance is dramatically degraded. Assuming b is the batch size, the primary must perform $4+5f+\frac{3f}{b}$ MACs in normal cases, which is even worse than the $2+\frac{8f}{b}$ MACs for PBFT for some b and f. Thus in *h*BFT, we seek to avoid this problem.

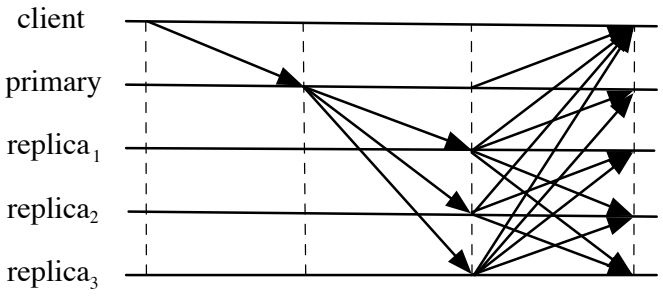


Figure 3.3. The agreement protocol

3.2.1 Agreement Protocol

The agreement protocol orders requests for execution by replicas. The algorithms of the agreement protocol for the primary, backups, and clients are defined in Algorithm 1 to Algorithm 3. As illustrated in Fig. 3.3, a client c invokes the operation by sending a $m = \langle \text{Request}, o, t, c \rangle_c$ to all replicas where o is the operation, t is the local timestamp. Upon receiving a request, as shown in Algorithm 1, the primary p_i assigns a sequence number seq and then sends out a $\langle \text{Prepare}, v, seq, D(m), m, c \rangle$ to all replicas, where v is the view number and $D(m)$ is the message digest.

A $\langle \text{Prepare} \rangle$ message will be accepted by a backup p_j provided that:

- It verifies the MAC;
- The message digest is correct;
- It is in view v;
- $seq = seq_l + 1$, where seq_l is the sequence number of its last accepted request;
- It has not accepted a $\langle \text{Prepare} \rangle$ message with the same sequence number in the same view but contains a different request.

[1]Using MACs instead of digital signatures usually makes protocols much faster. In Aardvark [29], on a 2.0GHz Pentium-M, openssl 0.9.8g can compute over 500,000 MACs per second for 64 byte messages, but it can only verify 6455 1024-bit RSA signatures per second or produce 309 1024-bit RSA signatures per second.

If a backup p_j accepts the $\langle$Prepare$\rangle$ message, it speculatively executes the operation and sends a reply message $\langle \text{Reply}, v, t, seq, \delta_{seq}, c\rangle$ to c and also a commit message $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c\rangle$ to all replicas, where δ_{seq} contains the speculative execution history.

In order to verify the correctness of the speculatively executed request, a replica collects $2f+1$ matching $\langle$Commit$\rangle$ messages from other replicas to complete a request. As shown in Algorithm 2, a replica collects matching $\langle$Commit$\rangle$ messages with the same sequence number. If a replica receives $f+1$ matching $\langle$Commit$\rangle$ messages from different replicas but has not accepted any $\langle$Prepare$\rangle$ message, it also speculatively executes the operation, sends a $\langle$Commit$\rangle$ message to all replicas, and sends a reply to the corresponding client. When the replica collects $2f$ matching messages, it puts the corresponding request in its speculative execution history and completes the request. However, it is possible that a replica receives $f+1$ matching $\langle$Commit$\rangle$ messages from other replicas that are conflicting with its accepted $\langle$Prepare$\rangle$ message. Under such circumstances, the replica can simply send a $\langle$View-Change$\rangle$ message to all replicas. If a replica votes for view change, it stops receiving any messages except the $\langle$New-View$\rangle$ and the checkpoint messages. See §3.2.3 for the detail of the view change subprotocol.

The exchange of $\langle$Commit$\rangle$ messages is to ensure that if at least $f+1$ correct replicas speculatively execute a request, all the correct replicas learn the result. If any other correct replicas receive inconsistent messages, the primary must be faulty and the replicas stop receiving messages until view change occurs.

A client sets a timeout for each request. As shown in Algorithm 3, a client collects matching $\langle$Reply$\rangle$ messages to its request. If it gathers $2f+1$ matching speculative replies from different replicas before the timeout expires, it completes the request. If a client receives fewer than $f+1$ matching replies before the timeout

expires, it retransmits the requests. Otherwise, when client receives between $f+1$ to $2f+1$ matching replies before timeout expires, it facilitates the progress by sending a $\langle \text{PANIC}, D(m), t, c \rangle_c$ message to all replicas. If a replica receives a $\langle \text{PANIC} \rangle$ message, it forwards the message to all replicas. If a replica does not receive any $\langle \text{PANIC} \rangle$ message from the client but receives a $\langle \text{PANIC} \rangle$ message from other replicas, it forwards the $\langle \text{PANIC} \rangle$ message to all replicas. A $\langle \text{PANIC} \rangle$ message is valid if a replica has speculatively executed m. If a replica accepts a $\langle \text{PANIC} \rangle$ message, it stops receiving any messages except the view change and checkpoint messages.

There are two goals for replicas when forwarding $\langle \text{PANIC} \rangle$ messages. One is to prevent the checkpoint protocol from occurring too frequently, which happens when all the correct replicas receive the $\langle \text{PANIC} \rangle$ message before the checkpoint protocol is triggered. Another is to prevent the clients from attacking the system by sending $\langle \text{PANIC} \rangle$ messages to a portion of the replicas. If a faulty client sends a $\langle \text{PANIC} \rangle$ message to a correct backup, the replica will stop receiving any messages while other replicas still continue the agreement protocol. This forwarding mechanism ensures that if at least one correct replica receives the $\langle \text{PANIC} \rangle$ message, all the replicas receive the $\langle \text{PANIC} \rangle$ message and enter the checkpoint protocol.

The primary initializes the checkpoint subprotocol if it receives the $\langle \text{PANIC} \rangle$ message from the client or $2f + 1$ $\langle \text{PANIC} \rangle$ messages from other replicas. The correctness of the protocol is therefore guaranteed by the three-phase checkpoint subprotocol.

The panic mechanism facilitates progress when the primary is faulty. Specifically, in a partial synchrony model where the value of a client's timeout is properly set up, if a correct client does not receive sufficient matching replies before timer expires, the primary either sends inconsistent $\langle \text{Prepare} \rangle$ messages to the replicas or fails to send consistent messages to the replicas. In this case, instead of using the traditional

Algorithm 1 Primary

1: **Initialization:**
2: A {All replicas}
3: $seq \leftarrow 0$ {Sequence number}
4: $\mathcal{W}$ {Set of ⟨PANIC⟩ messages}

5: **on event** $\langle \text{Request}, o, t, c \rangle_c$
6: $seq \leftarrow seq + 1$
7: **send** $\langle \text{Prepare}, v, seq, D(m), m, c \rangle$ to A
8: **send** $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c

9: **on event** $\langle \text{PANIC}, D(m), t, c \rangle_c$ from c
10: **send** $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A

11: **on event** $\langle \text{PANIC}, D(m), t, c \rangle_c$ from A
12: **if** $match(\mathcal{W}_c)$ **then**
13: $\mathcal{W}_c.add$ {Add matching ⟨PANIC⟩ message}
14: **if** $\mathcal{W}_c.size = 2f + 1$ **then**
15: Initialize checkpoint protocol

Algorithm 2 Backup

1: **Initialization:**
2: A {All replicas}
3: $seq_i \leftarrow 0$ {Sequence number}
4: $\mathcal{U}$ {Set of $\langle$Commit$\rangle$ messages}
5: $panic \leftarrow F$ {If true, enter checkpoint protocol}
6: **on event** $\langle \text{Request}, o, t, c \rangle_c$
7: **send** $\langle \text{Request}, o, t, c \rangle_c$ to the primary
8: **on event** $\langle \text{Prepare}, v, seq, D(m), m, c \rangle$
9: **if** $seq = seq_i + 1$ **then**
10: $seq_i \leftarrow seq$
11: **send** $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$ to A
12: **send** $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c
13: **on event** $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$
14: **if** $match(\mathcal{U}_{seq})$ **then**
15: $\mathcal{U}_{seq}.add$ {Add matching $\langle$Commit$\rangle$ message}
16: **if** $\mathcal{U}_{seq}.size = f + 1$ **and** $seq = seq_i + 1$ **then**
17: $seq_i \leftarrow seq$ {Accept the message}
18: **send** $\langle \text{Commit}, v, seq, \delta_{seq}, m, D(m), c \rangle$ to A
19: **send** $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$ to c
20: **if** $\mathcal{U}_{seq}.size = 2f$ **and** $seq = seq_i$ **then**
21: $complete(\mathcal{U}_{seq})$ {Complete the request}
22: **on event** $\langle \text{PANIC}, D(m), t, c \rangle_c$
23: **if** $panic = F$ **then**
24: **send** $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A
25: $panic \leftarrow T$ {Enter checkpoint protocol}

Algorithm 3 Client

1: **Initialization:**
2: A {All replicas}
3: $\mathcal{V}$ {Set of $\langle$Reply$\rangle$ messages}
4: **send** $\langle \text{Request}, o, t, c \rangle_c$ to A
5: $start(\Delta)$ {Start a timer}

6: **on event** $\langle \text{Reply}, v, t, seq, \delta_{seq}, c \rangle$
7: **if** $match(\mathcal{V}_{seq})$ **then**
8: $\mathcal{V}_{seq}.add$ {Add matching $\langle$Reply$\rangle$ message}
9: **if** $\mathcal{V}_{seq}.size = 2f + 1$ **then**
10: $cancel(\Delta)$ {Complete the request}

11: **on event** $timeout(\Delta)$
12: **if** $\mathcal{V}_{seq}.size < f + 1$ **then**
13: retransmit $\langle \text{Request}, o, t, c \rangle_c$ to A
14: **else**
15: **send** $\langle \text{PANIC}, D(m), t, c \rangle_c$ to A

approach where replicas detect the faulty primary themselves by waiting for longer period of time, the client can directly trigger the checkpoint protocol in order to verify the correctness of the primary. See §3.2.2 for details of the checkpoint subprotocol.

*h*BFT guarantees correctness while using only two phases. If the client has received $2f + 1$ matching replies, at least $f + 1$ correct replicas receive consistent order from the primary. Therefore, all correct replicas receive at least $f + 1$ matching $\langle$Commit$\rangle$ messages. If those replicas do not receive the $\langle$Prepare$\rangle$ message, they will execute the request. Otherwise, if they detect the inconsistency, they stop receiving

any messages until the current primary is replaced or the checkpoint subprotocol is triggered. In the latter case, the inconsistency will be reflected and fixed in the checkpoint subprotocol.

3.2.2 Checkpoint

We use a three-phase PBFT-like checkpoint protocol. The reasons are three-fold. First, the agreement protocol uses speculative execution and replicas may be temporarily out of order. The three-phase checkpoint protocols resolve the inconsistencies. Second, if a correct client triggers the checkpoint protocol through the panic mechanism, the checkpoint protocol resolves the inconsistencies immediately. Third, the checkpoint protocol detects the behavior of the faulty clients if they intentionally trigger the checkpoint protocol.

The checkpoint protocol works as follows. Only the primary can initialize the checkpoint subprotocol, which is generated under either of the two conditions:

- the primary executes a certain number of requests;
- the primary receives $2f + 1$ forwarded $\langle$PANIC$\rangle$ messages from other replicas.

In the latter condition, as mentioned in §3.2.1, when a replica receives a valid $\langle$PANIC$\rangle$ message, it forwards to all replicas. The goal is to ensure that all replicas receive the $\langle$PANIC$\rangle$ message and also to prevent faulty clients from sending a $\langle$PANIC$\rangle$ message only to the backups, thereby making sure replicas will not erroneously suspect the primary due to the faulty clients.

The three-phase checkpoint subprotocol works as follows: the current primary p_i sends a $\langle$Checkpoint-I, $seq, D(M)\rangle$ to all replicas, where seq is the sequence number of last executed operation, $D(M)$ is the message digest of speculative execution history M. Upon receiving a well-formatted $\langle$Checkpoint-I$\rangle$ message, a replica sends

a $\langle$Checkpoint-II, $seq, D(M)\rangle$ to all replicas. If the digest and execution history do not match its local log, the replica sends a $\langle$View-Change$\rangle$ message directly to all replicas and stops receiving any messages other than the $\langle$New-View$\rangle$ message.

A number of $2f + 1$ matching $\langle$Checkpoint-II$\rangle$ messages from different replicas form a certificate, denoted by $\mathcal{CER}_1(M, v)$. Any replica p_j that has the certificate sends a $\langle$Checkpoint-III, $seq, D(M)\rangle_j$ to all replicas. Similarly, $2f + 1$ $\langle$Checkpoint-III$\rangle$ messages form a certificate, denoted by $\mathcal{CER}_2(M, v)$. After collecting $\mathcal{CER}_2(M, v)$, the checkpoint becomes stable. All the previous checkpoint messages, $\langle$Prepare$\rangle$, $\langle$Commit$\rangle$, $\langle$Request$\rangle$, and $\langle$Reply$\rangle$ messages with smaller sequence number than the checkpoint are discarded.

If a view change occurs in the checkpoint subprotocol, as described in §3.2.3, the new primary initializes a checkpoint immediately after the $\langle$New-View$\rangle$ message. The same three-phase checkpoint subprotocol continues until one checkpoint is completed and the system stabilizes.

3.2.3 View Changes

The view change subprotocol elects a new primary. By default, the primary has id $p = v \bmod n$, where n is the total number of replicas and v is the current view number. View changes may take place in the checkpoint protocol or the agreement protocol. In both cases, the new primary reorders requests using a $\langle$New-View$\rangle$ message and then initializes a checkpoint immediately. The checkpoint subprotocol continues until one checkpoint is committed.

A $\langle$View-Change$, v + 1, \mathcal{P}, \mathcal{Q}, \mathcal{R}\rangle_i$ message will be sent by a replica if any of the following conditions are true, where $\mathcal{P}$ contains the execution history M from $\mathcal{CER}_1(M, v)$ the replica collected in previous view v, $\mathcal{Q}$ denotes the execution history

from the accepted ⟨Checkpoint-I⟩ message, and $\mathcal{R}$ denotes the speculatively executed requests with sequence numbers greater than its last accepted checkpoint:

- It starts a timer for the first request in the queue. The request is not executed before the timer expires;
- It starts a timer after collecting $f + 1$ ⟨PANIC⟩ messages. It has not received any checkpoint messages before the timer expires;
- It starts a timer after it executes certain number of requests. It has not received any checkpoint messages before the timer expires;
- It receives $f + 1$ valid ⟨View-Change⟩ messages from other replicas.

Timers with different values are set for each case and are reset periodically.

When the new primary p_j receives $2f$ ⟨View-Change⟩ messages, it constructs a ⟨New-View⟩ message to order all the speculatively executed requests. The system then moves to a new view. The principle is that any request committed by the clients must be committed by all correct replicas. The new primary picks up an execution history M from $\mathcal{P}$ and a set of requests from the $\mathcal{R}$ of checkpoint messages. To select a speculative execution history M, there are two rules.

A If some correct replica has committed on one checkpoint that contains execution history M, M must be selected, provided that:

A1. At least $2f + 1$ replicas have $\mathcal{CER}_1(M, v)$.

A2. At least $f + 1$ replicas have accepted ⟨Checkpoint-I⟩ in view $v' > v$.

B If at least $2f + 1$ replicas have empty $\mathcal{P}$ components, then the new primary selects its last stable checkpoint.

Similarly, for each sequence number greater than the execution history M and smaller than the largest sequence number in $\mathcal{R}$ of checkpoint messages, the primary assigns a request according to $\mathcal{R}$. A request m is chosen if at least $f+1$ replicas include it in $\mathcal{R}$ of their checkpoint messages. Otherwise, NULL is chosen. We claim that it is impossible for $f+1$ replicas to include one request m, and another $f+1$ replicas include m' with the same sequence number. Namely, if $f+1$ replicas include a request m, at least one correct replica receives $2f+1$ $\langle$Commit$\rangle$ messages. Similarly, at least one correct replica receives $2f+1$ commit messages with request m'. The two quorums intersect in at least one correct replica. The correct replica must have sent both $\langle$Commit$\rangle$ message with m and $\langle$Commit$\rangle$ message with m', a contradiction.

The execution history M and the set of requests form M', which is composed of requests with sequence numbers between the last stable checkpoint and the sequence number that has been used by at least one correct replica. The new primary then sends a $\langle \text{New-View}, v+1, \mathcal{V}, \mathcal{X}, M' \rangle_j$ message to all replicas, where $\mathcal{V}$ contains $f+1$ valid $\langle$View-Change$\rangle$ messages, $\mathcal{X}$ contains the selected checkpoint. The replicas then run the checkpoint subprotocol using M'. The checkpoint subprotocol continues until one checkpoint is committed.

3.2.4 Client Suspicion

Faulty clients may render the system unusable, especially for protocols that move some critical jobs to the clients. In hBFT, unlimited numbers of faulty clients can be detected. We focus on the "legal" but problematic messages a faulty client can craft to slow down the performance or cause incorrectness. To be specific, a faulty client can do the following:

- It sends inconsistent requests to different replicas. The primary may not be

able to order "every" request before the timeout expires. In this case, a correct primary may be removed.

- It intentionally sends ⟨PANIC⟩ messages while there is no contention. The unnecessary checkpoint subprotocol will be triggered, which slows down the performance. However, if the client frequently triggers "valid" checkpoint operations, the overall throughput decreases too.

- It does not send ⟨PANIC⟩ messages if it receives divergent replies, leaving replicas temporarily inconsistent.

The client suspicion subprotocol in *h*BFT focuses on the first two. If the third one occurs, the checkpoint subprotocol can be triggered by the next correct client if it detects the divergence of replies or by the primary when replicas execute certain number of requests.

To solve the first problem, we ask clients to multicast the request to the replicas and every replica forwards the request to the primary. The primary orders a request if it receives the request or if it receives $f+1$ matching requests forwarded by backups. If a replica p_i receives a ⟨Prepare⟩ message with a request that is not in its queue, it still executes the operation. Nevertheless, such faulty behavior of clients will be identified as suspicious, and if the number of suspicious incidents from the same client exceeds certain threshold, p_i will send a $\langle \text{Suspect}, c \rangle_i$ message to all replicas.

Another reason clients send their requests to all replicas is that there are many drawbacks when clients send requests only to the primary.[2] For instance, a faulty

[2]In some Byzantine agreement protocols, clients send requests only to their known primary. If a backup receives the request, it forwards the request to the primary, expecting the request to be executed. The client sets a timeout for each request it has. If it does not receive sufficient matching responses before timeout expires, it retransmits the request to all replicas.

primary can delay any request, regardless of whether the primary receives the request from the client or other replicas. This would cause all clients to multicast their requests to all replicas. In other words, a faulty primary makes all clients experience long latency without being noticed. A faulty primary can also perform a performance attack such as timeout manipulation, as discussed in other work [5, 29, 109]. Furthermore, it is also difficult to make clients keep track of the primary. If the client sends its request to a faulty backup, the faulty backup can also ignore this request, although it is supposed to forward the request to the primary. In many existing protocols, all of these problems typically mean that the primary task for establishing correctness is the process of detecting faulty replicas.

For the second problem where a faulty client intentionally sends a ⟨PANIC⟩ message to the replicas to trigger the checkpoint subprotocol, the protocol naturally detects the faulty behavior. Intuitively, if the request is committed in both agreement protocol and checkpoint protocol without view change, the client can be suspected. Nevertheless, a correct client might be suspected as well. For instance, the following two cases are indistinguishable.

(1) The replicas are correct and reach an agreement in the agreement protocol. When they receive the ⟨PANIC⟩ message from a faulty client, the request is committed in the checkpoint protocol without view change and the client is suspected.

(2) The primary is faulty and the client is correct. The primary sends the request to $f + 1$ correct replicas and another fake request to the remaining f correct replicas. The f correct replicas will not execute th request. When the replicas receive ⟨PANIC⟩ message and starts checkpoint protocol, the f faulty replicas collude and make the request committed in the checkpoint protocol. Although the f correct replicas learn the result and remain consistent, the correct client

will be suspected.

To distinguish the above two cases, we modify the agreement protocol by simply replacing the MACs of ⟨Prepare⟩ messages with digital signatures, which is called *Almost-MAC-agreement*. When a replica sends a ⟨Commit⟩ message, it appends the ⟨Prepare⟩ message. If a client does not receive valid ⟨Prepare⟩ message from the primary but receives from other replicas, it still executes the requests, sends ⟨Commit⟩ messages to other replicas, and sends a ⟨Reply⟩ to the client. Otherwise, if a replica receives two valid and conflicting ⟨Prepare⟩ messages, it directly sends inconsistent messages to all replicas and votes for view change. As proven in Claim 2 in §3.2.5, the protocol guaranteed that correct clients will not be removed. This optimization can also solve the problem discussed in §3.3.1.

The modification of agreement protocol results in $2 + \frac{1(sig)}{b}$ cryptographic operations for the primary. To reduce the overall cryptographic operations, hBFT switches between the agreement protocol and Almost-MAC-agreement when executing a certain number of requests.

The client will only be suspected when replicas are running Almost-MAC-agreement. In addition, the client must be suspected by $2f + 1$ replicas to be removed. If the number of such incidents exceeds certain threshold, replicas will suspect the client and send a ⟨Suspect⟩ message to all replicas. Similarly to the view change subprotocol, if a replica receives $f + 1$ ⟨Suspect⟩ messages, it generates a ⟨Suspect⟩ message and sends to the replicas. If a replica receives $2f + 1$ ⟨Suspect⟩ messages, indicating that at least one correct replica suspects the client, the client can be prevented from accessing the system in the future.

Worst Case Scenario. We would like to analyze the worst case where a correct client can be suspected, mainly due to the network failure. It happens if any of the

following is true:

(1) The request from client fails to reach $f + 1$ correct backups before the backups receive the $\langle$Prepare$\rangle$ message. In this case, since the $f + 1$ correct backups do not receive the request in the $\langle$Prepare$\rangle$ message, they will suspect the client.

(2) $\langle$Reply$\rangle$ messages from correct replicas fail to reach the client before the timeout expires. Since the client does not receive $2f + 1$ matching replies before the timeout expires, the client sends $\langle$PANIC$\rangle$ messages while there is no contention.

The latter condition may occur due to an inappropriate value of the timeout regarding the network condition or due to the attack by the primary. For instance, a faulty primary can intentionally delay $\langle$Prepare$\rangle$ messages for some correct replicas, causing correct clients to send a $\langle$PANIC$\rangle$ message even though replicas are "consistent." However, if the value of the timeout is appropriately set up using Almost-MAC-agreement, as proven in Claim 2 in §3.2.5, correct clients will not be removed. To set up an appropriate value, the clients adjust the values of the timeout during retransmission. Namely, when the client retransmits the request, it doubles the timeout and starts again. In this case, the value of the timeout will eventually be large enough for the client to receive $\langle$Reply$\rangle$ messages.

3.2.5 Correctness

In this section, we sketch proofs for the safety and liveness properties of hBFT under optimal resilience. For simplicity, we assume there are $3f + 1$ replicas.

3.2.5.1 Safety

Theorem 1 (Safety)**.** If requests m and m' are committed at two correct replicas p_i and p_j, m is committed before m' at p_i if and only if m is committed before m' at p_j.

Proof. The proof proceeds as follows. We first prove the correctness of checkpoint subprotocol, which follows the correctness of PBFT, as shown in *Claim 1.* We then show the proof of the theorem based on the claim.

Claim 1 (Safety of Checkpoint)**.** The checkpoint subprotocol guarantees the safety property.

Proof. We now prove that if checkpoints M and M' are committed at two correct replicas p_i and p_j in checkpoint subprotocol, regardless of being in the same view or across views, $M = M'$.

(Within a view) If p_i and p_j commit both in view v, then p_i has collected $\mathcal{CER}_2(M, v)$, which indicates that at least $f+1$ correct replicas have sent $\langle$Checkpoint-III$\rangle$ for M. Similarly, p_j has $\mathcal{CER}_2(M', v)$, which indicates that at least $f+1$ correct replicas send $\langle$Checkpoint-III$\rangle$ for M'. Then excluding f faulty replicas, if M and M' are different, at least one correct replica has sent two conflicting messages for M and M', which contradicts with our assumption. Therefore, $M = M'$.

(Across views) If M is committed at p_i in view v and M' is committed at p_j in view $v' > v$, $M = M'$. If M' is committed in view v', then either condition A or B must be true in the construction of the $\langle$New-View$\rangle$ message in view v' (see §3.2.3). However, if M is committed at p_j in view v, p_j has $\mathcal{CER}_2(M, v)$, which indicates that at least $f + 1$ correct replicas have $\mathcal{CER}_1(M, v)$ and M in the $\mathcal{P}$ component. Therefore, condition B cannot be true. For condition A, M' is committed at p_j

in view v' if both A1 and A2 are true. A2 can be true if a faulty replica sends a $\langle$View-Change$\rangle$ message that includes $\langle M', D(M'), v_1\rangle$, where $v < v_1 \leq v'$. However, condition A1 requires that at least $f + 1$ correct replicas have $\mathcal{CER}_1(M', v')$. Since at least $f + 1$ correct replicas have $\mathcal{CER}_1(M, v)$, they will not accept M' in any later views. At least one correct replica sends conflicting messages, a contradiction. Therefore, we have $M = M'$. □

To prove Theorem 1, we first show that if two requests m and m' are committed at correct replicas p_i and p_j, m equals m'. Then we show that if m_1 is committed before m_2 at p_i, m_1 is committed before m_2 at p_j. The former part is shown across views and within the same view.

(Within a view) There are three cases: the two requests are committed in agreement subprotocol, two requests are committed in checkpoint subprotocol, one of them is committed in the agreement subprotocol and the other one is committed in the checkpoint subprotocol. In the first case, if m is committed at p_i, p_i receives $2f + 1$ $\langle$Commit$\rangle$ messages if the request is committed in agreement protocol. On the other hand, if m' is committed at p_j, p_j receives $2f + 1$ $\langle$Commit$\rangle$ messages. The two quorums intersect in at least one correct replica. At least one correct replica sends inconsistent messages, a contradiction. Therefore, m equals m'. The second case is proved in Claim 1. In the third case, if m is committed at p_i, p_i receives $2f + 1$ $\langle$Commit$\rangle$ messages if the request is committed in the agreement protocol. On the other hand, if m' is committed at p_j in checkpoint protocol, at least $2f + 1$ replicas have certificate with m' in their execution history. The two quorums of $2f+1$ replicas intersect in at least one correct replica, who sends a $\langle$Commit$\rangle$ message with m in the agreement protocol and includes m' in its execution history in the checkpoint protocol, a contradiction. To summarize, we have m equals m' if they are committed

in the same view.

(Across views) If m is committed at replica p_j, $2f + 1$ replicas send $\langle$Commit$\rangle$ messages. At least $f+1$ correct replicas accept m, which will be included in their $\langle$View-Change$\rangle$ messages. On every view change, the new primary initializes a checkpoint subprotocol to make the same order of requests committed at all the correct replicas in the $\langle$New-View$\rangle$ message. The correctness follows from Claim 1.

Then we show that if m_1 is committed before m_2 at p_i, m_1 is committed before m_2 at p_j. If a request is committed at a correct replica, $2f + 1$ replicas send $\langle$Commit$\rangle$ messages. Since two quorums of $2f + 1$ replicas intersect in at least one correct replica p_i, m_1 is committed with sequence number smaller than m_2. According to the former proof, if m_1 and m_2 are committed at p_j, they are committed with the same sequence numbers.

By combining all the above, safety is proven. □

3.2.6 Liveness

Theorem 2 (Liveness)**.** Correct clients eventually receive replies to their requests.

Proof. It is trivial to show that if the primary is correct, clients receive replies to their requests. In the following, we first show that correct clients will not be removed. We then prove that faulty replicas and faulty clients cannot impede progress by removing a correct primary.

Claim 2 (Correct Client Condition)**.** If the values of the timeouts are appropriately set up, correct clients will not be removed if they trigger a checkpoint.

Proof. If a correct client receives between $f + 1$ to $2f + 1$ matching replies for a request m, it triggers the checkpoint subprotocol. To remove a correct client, m

must be executed by $f + 1$ replicas in the Almost-MAC-agreement protocol and committed in the checkpoint subprotocol without view changes. Among the $f + 1$ replicas that accept ⟨Prepare⟩ message in the agreement protocol, at least one is correct. If it receives a ⟨Prepare⟩ message, it appends to ⟨Commit⟩ message and sends to all replicas. If at least one correct replica receives a valid and conflicting ⟨Prepare⟩ message from the primary, it will send inconsistent messages and eventually all the correct replicas vote for view change, a contradiction that view change does not occur. Therefore, no correct replica receives a different ⟨Prepare⟩ message. In addition, if a correct replica does not receive a valid ⟨Prepare⟩ message from the primary and receives a valid ⟨Prepare⟩ message appended to the ⟨Commit⟩ message, it will accept the ⟨Prepare⟩ message and sends ⟨Reply⟩ message to the client. In this case, the client receives $2f + 1$ matching replies, a contradiction with the assumption that the client is correct. Therefore, correct clients will not be removed by the client suspicion protocol. □

Claim 3 (Faulty Replica Condition)**.** Faulty replicas cannot impede progress by causing view changes.

Proof. To begin, we show that faulty replicas cannot cause a view change by sending ⟨View-Change⟩ messages. At least $f + 1$ ⟨View-Change⟩ messages are sufficient to cause a view change. Thus, even if all faulty replicas vote for view change, they cannot cause a view change. A faulty primary *can* cause a view change. However, the primary cannot be faulty for more than f consecutive views.

In addition, no ⟨View-Change⟩ message makes a correct primary incapable of generating a ⟨New-View⟩ message. A correct primary is able to pick up a stable checkpoint. Since at least $f + 1$ correct replicas have $\mathcal{CER}_2$ for a checkpoint, the new primary is able to pick it up. In addition, the new primary is able to pick up

a sequence of requests based on condition A or B. Either some correct replica(s) commits on a checkpoint or no correct replica does. Condition A1 can be verified because non-faulty replicas will not commit on two different checkpoints. Condition A2 is satisfied if at least one correct replica accepts a ⟨Checkpoint-I⟩ message for the same checkpoint and it votes for the authenticity of the checkpoint. Therefore, the checkpoint can be selected since it is authentic. Similarly, a set of executed requests can be selected based on $\mathcal{R}$ in a view change. Namely, if the client completes a request, the request must be accepted by at least $2f + 1$ replicas. Among them, at least $f + 1$ replicas are correct. If other replicas receive inconsistent ⟨Prepare⟩ messages and $f + 1$ ⟨Commit⟩ messages, they will abort. Therefore, it is not possible that a set of $f + 1$ replicas include one request and another set of $f + 1$ replicas include another request. In conclusion, the new primary is able to select a ⟨New-View⟩ message. □

Claim 4 (Faulty Client Condition 2)**.** A faulty client cannot impede progress by causing view changes.

Proof. If a faulty client intentionally triggers the checkpoint subprotocol while replicas are consistent, requests committed in agreement subprotocol will be committed in checkpoint subprotocol. View changes will not occur. Since such faulty behavior of clients will be detected, the client will be removed. □

To summarize, according to Claim 2, correct replicas will not be removed, so their requests can be handled. Faulty backups or faulty clients can not cause view changes, as proven in Claim 3 and Claim 4 respectively. Since the primary cannot be faulty for more than f continuous views, correct clients eventually receive replies to their requests. □

3.3 Discussion

3.3.1 Timeouts

Existing protocols rely on different timeouts to guarantee liveness. As discussed in §3.2.4, the values of timeouts are key to avoid some uncivil attacks. Since we assume the partial synchrony model, it is reasonable to set up timeouts according to the round-trip time such as the technique used in Prime [5]. However, in several corner cases, either inappropriate values of timeouts or network congestion can make a correct replica suspect or remove a correct primary.

*h*BFT employs a client suspicion subprotocol that is used to detect faulty clients. A faulty primary can play tricks on timeouts to remove correct clients. For instance, the primary can send a $\langle$Prepare$\rangle$ message to f correct replicas and delay the $\langle$Prepare$\rangle$ message to $f+1$ correct replicas until the very end of timeout of the client. The $f+1$ correct replicas receive the $\langle$Prepare$\rangle$ message and execute the request but they do not reply to the clients "on time." Since the client does not receive enough number of replies before the timeout expires, it sends a $\langle$PANIC$\rangle$ message. However, all replicas are "consistent" since the primary still sends out consistent $\langle$Prepare$\rangle$ messages. Correct clients will be suspected.

We solve this problem by using Almost-MAC-agreement protocol as discussed in §3.2.4. The optimization allows all replicas to execute the request on time if at least one correct replica receives a valid $\langle$Prepare$\rangle$ message, which prevents a faulty primary from framing the clients.

3.3.2 Speculation

Speculation reduces the cost and simplifies the design of Byzantine agreement protocols, which works well especially for systems with highly concurrent requests. Speculation has been used by fault-free systems and by systems that tolerate crash failures. Therefore, *h*BFT also works well in adaptively tolerating crash failures to Byzantine failures. *h*BFT uses speculation because replicas are always consistent for both fault-free and normal cases where the primary is correct. Every request takes three communication steps to complete, and is the theoretical lower bound for agreement-based protocols.

Speculation does not work well for systems that have high computationally intensive tasks or systems that have a high attack rate. The former problem can be handled by separating execution from agreement [117]. The latter problem decreases the performance either with or without recovery. For instance, faulty clients can simply trigger the three-phase checkpoint subprotocol on every request, which gives *h*BFT similar performance to PBFT before the faulty clients are removed. The advantage of *h*BFT, as shown in §3.4, shows that the three-phase checkpoint subprotocol is rarely triggered. Therefore, *h*BFT improves the performance in fault-free and normal cases but achieves comparable performance to PBFT in the worst case.

3.4 Evaluation

We evaluated the system on Emulab [114] utilizing up to 45 *pc*3000 machines connected through a 100Mbps switched LAN. Each machines is equipped with a 2GHz, 64-bit Xeon processor with 2GB of RAM. 64-bit Ubuntu 10 is installed on every machine, running Linux kernel 2.6.32. We used RSA-FDH [9] for our digital signature

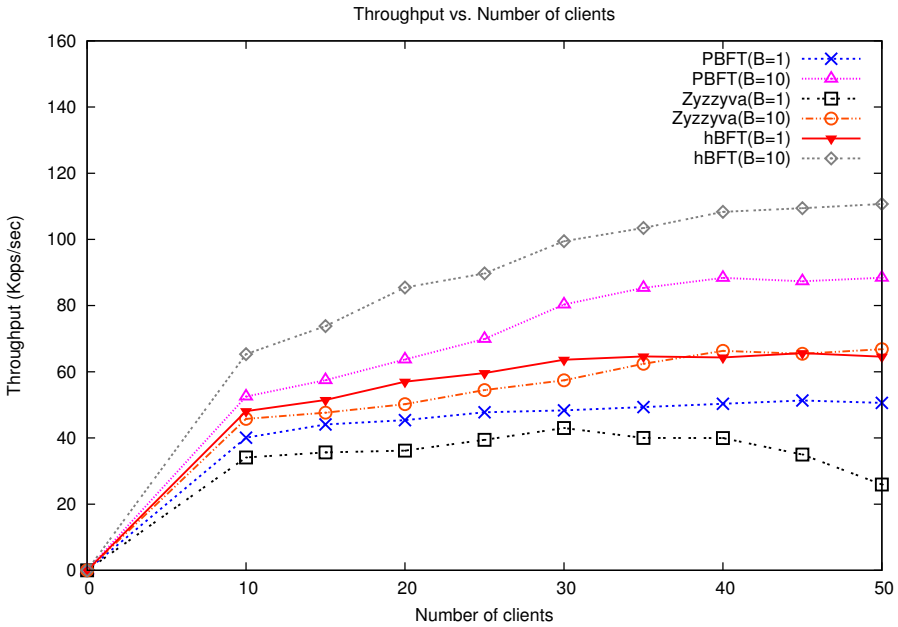


Figure 3.4. Throughput for the 0/0 benchmark as the number of clients varies for systems to tolerate $f = 1$ faults.

scheme, and HMAC-MD5 [10, 11] for the MAC algorithm.

We compare our work with Castro et al.'s implementation of PBFT [18] as well as Kotla et al.'s implementation of Zyzzyva [69]. All the experiments are carried out in normal cases, where a backup is faulty. Four micro-benchmarks are used in the evaluation, also developed by Castro et-al. An x/y benchmark refers to an xkB request from clients and an ykB reply from the replicas.

3.4.1 Throughput

Fig. 3.4 compares throughput achieved for the 0/0 benchmark in normal cases between PBFT, Zyzzyva and hBFT where B is the size of the batch. Fig. 3.5 presents the performance for the four benchmarks where $B = 1$ for all benchmarks. All the experiments are tested in the configuration of $f = 1$.

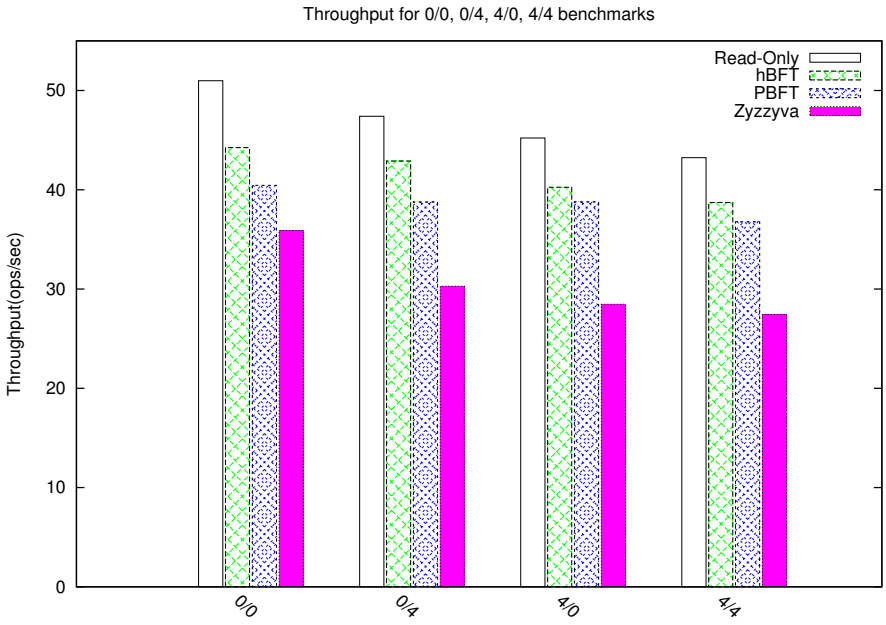


Figure 3.5. Throughput for $0/0, 0/4, 4/0$ and $4/4$ benchmarks for systems to tolerate $f = 1$ faults.

As the number of clients increases, Zyzzyva performs even worse than PBFT. As indicated in §3.1.1, without batching ($B = 1, f = 1$), bottleneck server of Zyzzyva ($4+ 5f + \frac{3f}{b}$) performs 1.2 times more MAC operations than PBFT ($2 + \frac{8f}{b}$) and 2.4 times more MAC operations than hBFT ($2 + \frac{3f}{b}$). With batching ($B = 10, f = 1$), Zyzzyva performs 3.3 times more MAC operations than PBFT and 4.0 times more MAC operations than hBFT.

The simulation validates the theoretical results. As shown in Fig. 3.4, without batching, hBFT achieves more than 40% higher throughput than PBFT and 20% higher throughput than Zyzzyva. With batching, the peak throughput of hBFT is 2 times better than that of Zyzzyva, and 40% higher than that of PBFT. The difference is due to the cryptographic overhead of each protocol.

Additionally, hBFT outperforms both Zyzzyva and PBFT under high concur-

rency. As the number of clients grows, all three protocols achieve better performance with batching than without. When the number of clients exceeds 40, throughput of Zyzzyva degrades obviously. All other cases remain stable when the number of clients exceeds 30. When the number of clients is fewer than 30, hBFT with batching has an outstanding growth. Other than that, throughput of PBFT with batching also grows faster compared with all the left cases. The reply message cannot be batched and replicas need to reply to every client, which explains the result why Zyzzyva achieves the lowest throughput in normal cases.

Fig. 3.5 presents the throughput of protocols without batching with 10 clients. For all the benchmarks, hBFT achieves higher throughput as well. All three protocols achieve the best throughput for 0/0 benchmark and the worst for 4/4 benchmark. Zyzzyva and hBFT perform worse for 0/4 and 4/4 benchmarks than 4/0 benchmark. PBFT achieves almost the same throughput for 0/4 and 4/0 benchmarks. This implies that the size of reply messages has more effect for speculation-based protocols. The outstanding performance of read-only requests is due to the read-only optimization, where replicas send reply directly to the clients without running agreement protocol.

To summarize this section, hBFT outperforms both Zyzzyva and PBFT in normal cases. Since PBFT achieves almost the same throughput for 0/4 and 4/0 benchmarks and it achieves higher throughput with batching, it works well for systems that have more computationally consuming tasks. Comparably, hBFT and Zyzzyva work well for systems that have highly concurrent but lightweight requests.

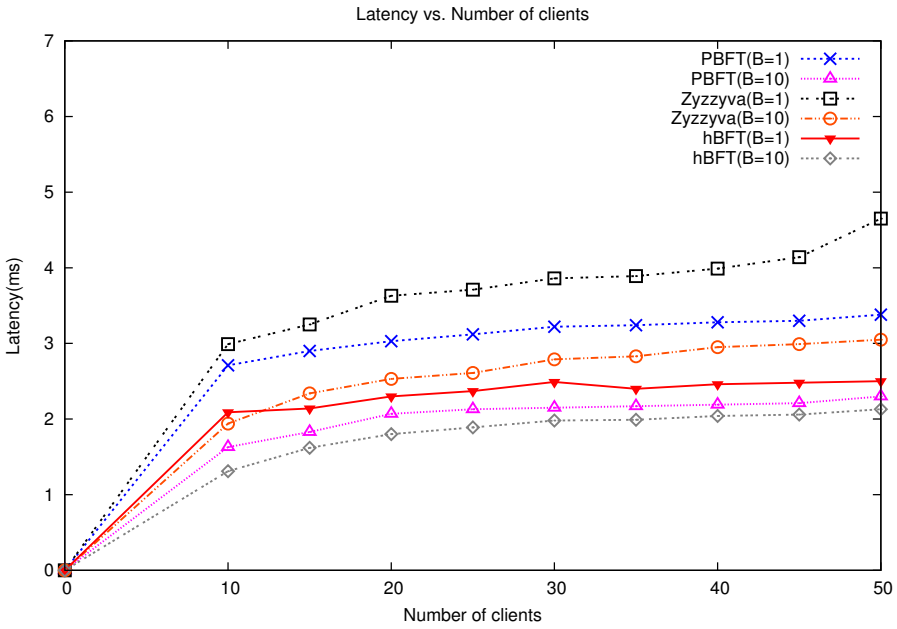


Figure 3.6. Latency for the 0/0 benchmark as the number of clients varies for systems to tolerate $f = 1$ faults.

3.4.2 Latency

The performance depends on both cryptographic overhead and one way message latencies. Cryptographic overhead controls the latency of processing one message and the number of one way latencies controls the number of phases that the agreement protocol goes through. In terms of critical paths between sending and completing a request, PBFT has four if replicas send reply to the clients after prepare phase. *h*BFT has only three, which is the theoretical lower bound of agreement protocols. Even though the checkpoint subprotocol takes three phases in contrast to two in other protocols, it will not decrease the overall performance significantly since the checkpoint subprotocol is triggered rarely. Zyzzyva takes three in fault-free cases and five in normal cases.

Additionally, the performance of all protocols is also related to the frequency of

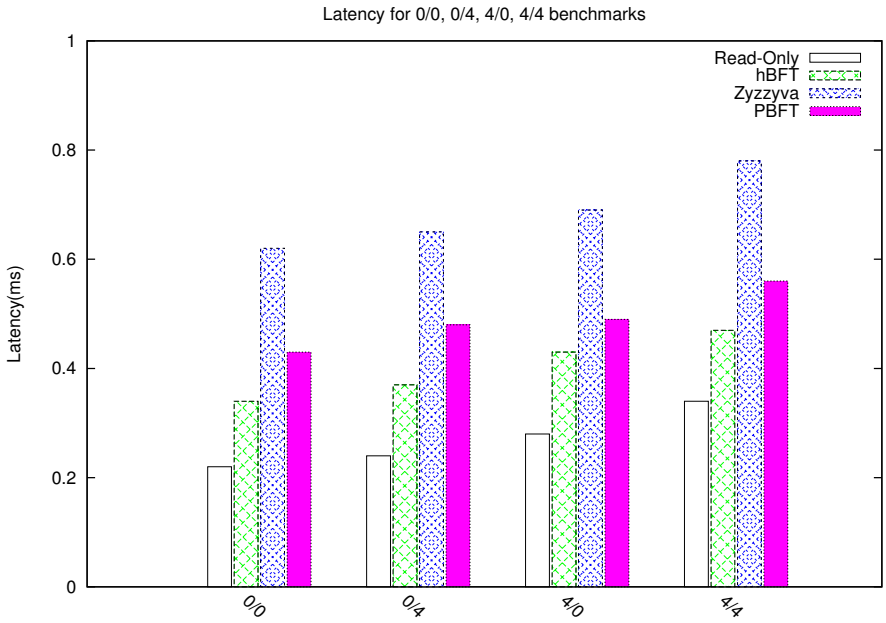


Figure 3.7. Latency for $0/0, 0/4, 4/0$ and $4/4$ benchmarks for systems to tolerate $f = 1$ faults without contention.

checkpoint subprotocol as well. It has a direct impact on hBFT due to the reason that checkpoint subprotocol of hBFT is more expensive than the other two. By default, we assume that a checkpoint subprotocol starts every 1000 requests or batches. hBFT outperforms the other two under this setting. If we make checkpoint subprotocol more rarely, it can be expected that hBFT will achieve even better performance and vice versa.

As illustrated in Fig. 3.6 and Fig. 3.7, without batching, hBFT achieves 40% lower latency than that of PBFT and 30% lower latency than that of Zyzzyva. With batching, similar with the performance of throughput, Zyzzyva achieves higher latency than that of PBFT, and hBFT outperforms both. When the number of clients increases, all the protocols scale well without an obvious increase in latency, which shows that all three protocols work well under high concurrency. When the

number of clients exceeds 40 and with batching, Zyzzyva has an increase of latency. Since every ⟨Reply⟩ message in Zyzzyva contains $3f+1$ MACs and cannot be batched, the increase in latency indicates that the cryptographic operations in the ⟨Reply⟩ message limits the behavior of a protocol.

The performance for all the four benchmarks shows similar results as indicated in Fig. 3.7. All the three protocols have the lowest latency for 0/0 benchmark and the highest for 4/4 benchmark. *h*BFT and PBFT achieve almost the same latency for both 4/0 and 0/4 benchmarks. Zyzzyva achieves lower latency for 4/0 benchmark than 0/4 benchmark. The length of reply message also reduces the latency per request for Zyzzyva. The effect is not as apparent as the effect on throughput though. Although *h*BFT performs better on throughput for the 4/0 benchmark than the 0/4 benchmark, it achieves almost the same latency for both benchmarks, which indicates that the checkpoint subprotocol has a more direct effect on the throughput than the latency.

Overall, the latency validates the results of throughput. Our statements in §3.4.1 are verified by the results of latency. By observing the curves of latency, we can summarize the performance of protocols under normal operations. On the other hand, by observing the curves of throughput, the effects of other subprotocols are included.

3.4.3 Fault Scalability

The latency depends on both cryptographic overhead and one-way latencies. One-way latencies refers to the communication step between the beginning of a request to the receipt of the reply message. Cryptographic overhead controls the latency of processing one message and the number of one-way latencies controls the number of

phases that the agreement protocol goes through. In terms of critical paths, PBFT has four if replicas send reply to the clients after prepare phase. hBFT has only three, which is the theoretical lower bound of agreement protocols under high concurrency. Even though the checkpoint subprotocol takes three phases, it will not decrease the overall performance significantly since the checkpoint subprotocol is triggered rarely. Zyzzyva takes three in fault-free cases and five in normal cases.

Additionally, the performance of all protocols is also related to the frequency of checkpoint subprotocol as well. It has a direct impact on hBFT due to the reason that checkpoint subprotocol of hBFT is more expensive than PBFT and Zyzzyva. By default, we assume that a checkpoint subprotocol starts every 128 requests. hBFT outperforms the other two under this setting. If we use checkpoint subprotocol more rarely, it can be expected that hBFT will achieve even better performance and vice versa.

We assess the latency without contention when there is only 1 client. The performance for all four benchmarks are similar, as shown in Fig. 3.7. All three protocols have the lowest latency for the 0/0 benchmark and the highest for the 4/4 benchmark. PBFT achieves almost the same latency for both 4/0 and 0/4 benchmarks. hBFT and Zyzzyva achieve lower latency for the 4/0 benchmark than the 0/4 benchmark.

As shown in Fig. 3.7, we also evaluate latency as the number of clients grows. We observe that without batching, hBFT achieves an average of 30% lower latency than PBFT and 40% lower latency than Zyzzyva. With batching, hBFT achieves an average of 15% lower latency than PBFT and 35% lower latency than Zyzzyva. When the number of clients increases, the latency of all the protocols increase gradually, which shows that all three protocols work well under high concurrency. The latency of Zyzzyva grows faster than the other two.

We also examine performance when the number of replicas increases. As shown

in Fig. 7.2, the throughput is related to f. We view the primary as the bottleneck server not only because of the number of MAC operations in the agreement, but also because of other effort such as processing requests. For PBFT and hBFT, the backups do not perform many fewer cryptographic operations than the primary. In comparison, backups in Zyzzyva perform many fewer cryptographic operations than the primary, which can be viewed as an advantage over the other two. However, this does not have a direct positive effect on the throughput and latency since the primary performs more cryptographic operations. As f increases, the performance for all three protocols will decrease due to the cryptographic overhead, especially without batching.

Fig. 3.8 compares the number of cryptographic operations that the primary and clients perform in normal cases as the number of faults increases. In addition to PBFT, Zyzzyva and hBFT, we also include Q/U and HQ, which are two (hybrid) Byzantine quorum protocols. For the performance of a primary with or without batching, as illustrated in Fig. 3.8(a) and Fig. 3.8(b), it can be observed that batching greatly reduces the number of cryptographic operations as the number of total replicas increases. For instance, although the number of cryptographic operations of PBFT is high without batching and increases quite fast, the cryptographic overhead is almost the smallest without batching and remains stable as the number of faults increases. Comparably, the number of cryptographic operations of Zyzzyva does not decrease too much without batching. Since both HQ and Q/U are quorum-based protocols, they cannot use batching and work better under low concurrency. hBFT achieves the smallest numbers with or without batching.

As illustrated in Fig. 3.9, as the number of replicas increases, the latency of PBFT increases quickly without batching. With batching, PBFT achieves a more stable curve. Zyzzyva has higher latency than the other two protocols for each case. On the

other hand, the latency of hBFT stabilizes and does not grow to a large degree with or without batching. The key factors in the performance are not only the critical paths and the number of cryptographic operations, but also the message complexity. Although Zyzzyva has higher cryptographic overhead, it requires the same number of messages as hBFT, explaining why both scale better than PBFT.

Not surprisingly, as shown in Fig. 3.10, the throughput shows a similar trend with latency. As the system scales, when f is greater than 2, throughput of Zyzzyva obviously decreases, especially without batching. Zyzzyva scales better than PBFT but the performance degrades obviously when f is greater than 4. hBFT scales better than both Zyzzyva and PBFT with or without batching. The difference between the numbers of cryptographic operations is still the key to the overall performance. When the number of faults is 5 and assuming b equals 10, PBFT requires 42 MACs without batching and only 6 with batching, Zyzzyva requires 44 MACs without batching and 30.5 with batching, and hBFT requires 17 MACs without batching and 3.5 with batching. For systems with high concurrency, PBFT and hBFT are preferred and scale well as the number of faults increases.

3.4.4 A BFT Network File System

This section describes our evaluation of a BFT-NFS service implemented using PBFT [18], Zyzzyva [69], and hBFT, respectively. Similarly, in the NFS service, we evaluate the performance of normal cases where a backup server fails.

The NFS service exports a file system, which can then be mounted on a client machine. The replication library and the NFS daemon are called to reach agreement in the order that replicas receive client requests. Once processing is done, replies are sent to the clients. The NFS daemon is implemented using a fixed-size memory-

mapped file.

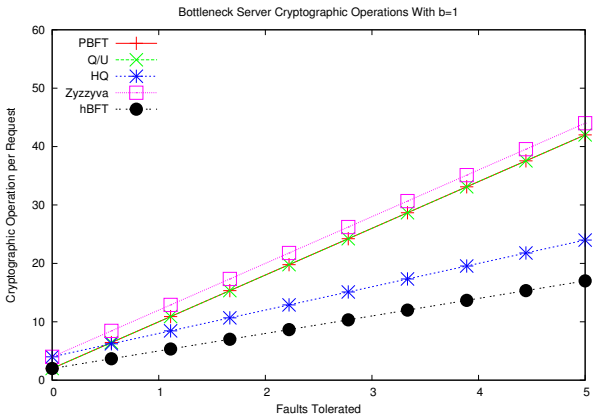
We use the Bonnie++ benchmark [30] to compare our three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. The Bonnie++ benchmark includes sequential input (including per-character and block file reading), sequential output (including per-character and block file writing), and the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that appears random to the file system; (5) stat() random files; (6) delete the files in random order.

We evaluate the performance when a failure occurs at time zero, as detailed in Fig. 3.11. In addition, up to 20 clients run Bonnie++ benchmark concurrently. The results show that *h*BFT completes every type of operations with lower latency than all of other protocols. The main difference lies on the write operations. This is due to the fact that all the three protocols use read-only optimization, where replicas send reply messages to the clients directly without running the agreement protocol. Compared with NFS-std, *h*BFT only causes 6% overhead while PBFT and Zyzzyva cause 10% and 18% overhead, respectively.

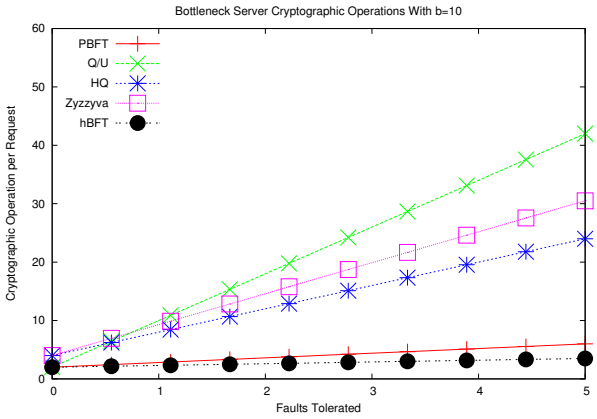
3.5 Conclusion

In this chapter, we presented *h*BFT, a hybrid, Byzantine fault-tolerant, replicated state machine protocol with optimal resilience. By re-exploiting speculation, as well a requiring the participation of clients, the theoretical lower bound for throughput and latency have been achieved for both *fault-free* and *normal cases* in *h*BFT. *h*BFT is a fast protocol that moves some jobs to the clients but can still tolerate faulty clients. We have also proven the safety and liveness properties of *h*BFT and demonstrated

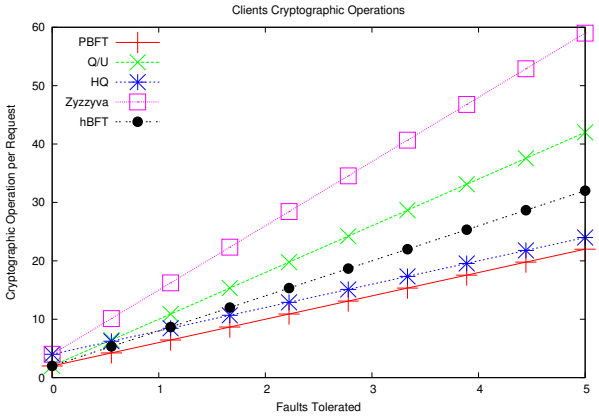
how hBFT improves on the performance of existing protocols without several of the trade-offs.



(a) Bottleneck server, $b = 1$



(b) Bottleneck server, $b = 10$



(c) Client

Figure 3.8. Fault scalability using analytical model.

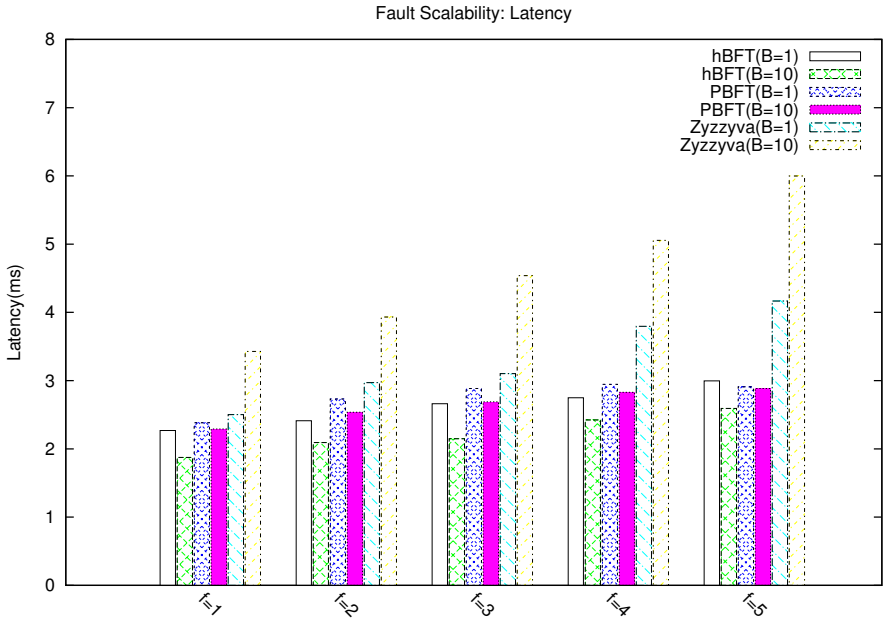


Figure 3.9. Fault scalability: latency.

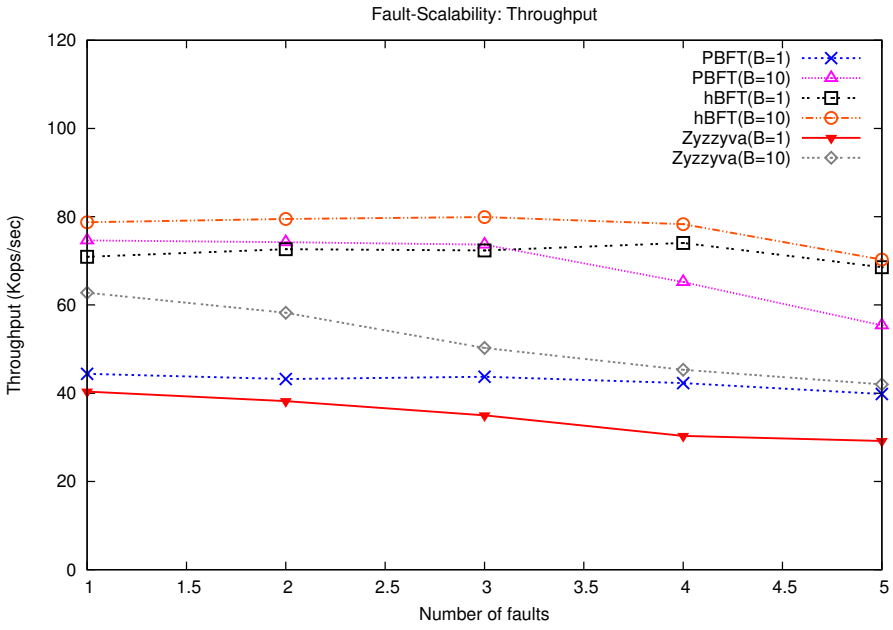


Figure 3.10. Fault scalability: throughput.

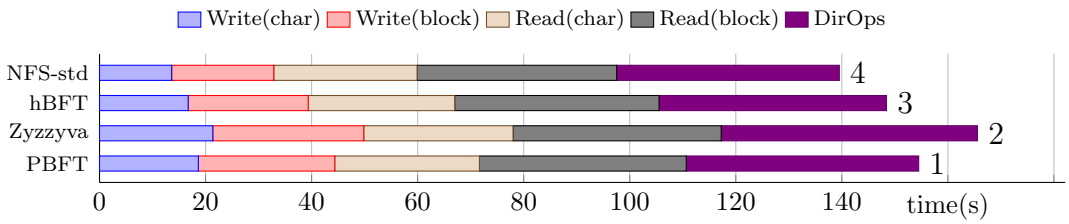


Figure 3.11. NFS evaluation with the Bonnie++ benchmark.

Chapter 4

BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration

The work presented in this chapter was first described in an earlier paper by Duan, et al. [39]. We describe the design and implementation of BChain, a Byzantine fault-tolerant state machine replication protocol, which performs comparably to other modern protocols in fault-free cases, but in the face of failures can also quickly recover its steady state performance. Building on chain replication, BChain achieves high throughput and low latency under high client load. At the core of BChain is an efficient Byzantine failure detection mechanism called *re-chaining*, where faulty replicas are placed out of harm's way at the end of the chain, until they can be replaced. We provide a number of optimizations and extensions and also take measures to make BChain more resilient to certain performance attacks. Our experimental evaluation, using both micro-benchmarks and an NFS service, confirms our performance expectations for both fault-free and failure scenarios.

4.1 Introduction

There are two broad classes of BFT protocols that have evolved in the past decade: broadcast-based [2, 18, 34, 69] and chain-based protocols [50, 107]. The main difference between these two classes is their performance characteristics. Chain-based protocols are aimed at achieving high throughput, at the expense of higher latency. However, as the number of concurrent client requests grows, it turns out that chain replication protocols can actually achieve lower latency than broadcast-based protocols. The downside however, is that chain protocols are less resilient to failures, and typically resort to broadcasting when failures are present. This results in a significant performance degradation.

In this chapter we propose *BChain*, a fully-fledged BFT protocol addressing the performance issues observed when a BFT service experiences failures. Our evaluation shows that BChain can quickly recover its steady state performance, while Aliph-Chain [50] and Zyzzyva [69] experience significantly reduced performance, when subjected to a simple crash failure. At the same time, the steady state performance of BChain is comparable to Aliph-Chain, the state-of-the-art chain-based BFT protocol. BChain also outperforms broadcast-based protocols PBFT [18] and Zyzzyva with a throughput improvement of up to 50 % and 25 %, respectively. We used BChain to implement a BFT-based NFS service, and our evaluation shows that it is only marginally slower (1%) than a standard NFS implementation.

BChain in a nutshell. BChain is a self-recovering, chain-based BFT protocol, where the replicas are organized in a chain. In common case executions, clients send their requests to the head of the chain, who orders the requests. The ordered requests are forwarded along the chain and executed by the replicas. Once a request reaches a replica that we call the *proxy tail*, a reply is sent to the client.

When a BFT service experiences failures or asynchrony, BChain employs a novel approach that we call *re-chaining*. In this approach, the head reorders the chain when a replica is suspected to be faulty, so that a fault cannot affect the critical path.

To facilitate re-chaining, BChain makes use of a novel failure detection mechanism, where any replica can suspect its successor and only its successor. A replica does this by sending a signed suspicion message up the chain. No proof that the suspected replica has misbehaved is required. Upon receiving a suspicion, the head issues a new chain ordering where the accused replica is moved out of the critical path, and the accuser is moved to a position in which it cannot continue to accuse others. In this way, correct replicas help BChain make progress by suspecting faulty replicas, yet malicious replicas cannot *constantly* accuse correct replicas of being faulty.

Our re-chaining approach is inexpensive; a single re-chaining request corresponds to processing a single client request. Thus, the steady state performance of BChain can almost be maintained. The latency reduction caused by re-chaining is dominated by the failure detection timeout.

Our Contributions in Context. We consider two variants of BChain—BChain-3 and BChain-5, both tolerating f failures. BChain-3 requires $3f + 1$ replicas and a reconfiguration mechanism coupled with our detection and re-chaining algorithms, while BChain-5 requires $5f + 1$ replicas, but can operate without the reconfiguration mechanism. We compare BChain-3 and BChain-5 with state-of-the-art BFT protocols in Table 7.2. All protocols use MACs for authentication and request batching with batch size b. The number of MAC operations for BChain at the bottleneck server tends to one for gracious executions. While this is also the case for Aliph-Chain [50], Aliph requires that clients take responsibility for switching to a different,

stronger, and slower BFT protocol in the presence of failures, to ensure safety and liveness. Thus, a single dedicated adversary might render the system much slower. Shuttle [107] can tolerate f faulty replicas using only $2f + 1$ replicas. However, it relies on a trusted auxiliary server. BChain does not require an auxiliary service, yet its critical path of $2f + 2$ is identical to that of Shuttle.

Our contributions can be summarized as follows:

- We present BChain-3 and its sub-protocols for re-chaining, reconfiguration, and view change (§4.2). Re-chaining is a novel technique to ensure liveness in BChain. Together with re-chaining, the reconfiguration protocol can replace failed replicas with new ones, outside the critical path. The view change protocol deals with a faulty head.
- BChain-5 and how it can operate without reconfiguration (§4.3).
- We also describe a number of optimizations and extensions in §4.4, including a special case of BChain-3, which does not require reconfiguration to achieve liveness.
- In §4.5 we evaluate the performance of BChain for both gracious and uncivil executions under different workloads, and compare it with other BFT protocols. We also ran experiments with a BFT-NFS application and assessed its performance compared to the other relevant BFT protocols.

4.2 BChain-3

We now describe the main protocols and principles of BChain. Our description here uses digital signatures; later we show how they can be replaced with MACs, along with other optimizations. BChain-3 has five sub-protocols: (1) chaining, (2) re-chaining, (3) view change, (4) checkpoint, and (5) reconfiguration. The *chaining*

protocol orders clients requests, while *re-chaining* reorganizes the chain in response to failure suspicions. Faulty replicas are moved to the end of the chain. The *view change* protocol selects a new head when the current head is faulty, or the system is slow. Our *checkpoint* protocol is similar to that of PBFT [18] and hBFT work described in Chapter 3. It is used to bound the growth of message logs and reduce the cost of view changes. We do not describe it in this chapter. The *reconfiguration* protocol is responsible for reconfiguring faulty replicas.

To tolerate f failures, BChain-3 needs n replicas such that $f \leq \lfloor \frac{n-1}{3} \rfloor$. In the following, we assume $n = 3f + 1$, but it can be extended to cases where $n > 3f + 1$ holds.

4.2.1 Conventions and Notations

Our system can mask up to f faulty replicas, using n replicas. We write t, where $t \leq f$, to denote the number of faulty replicas that the system currently has. A computationally bounded adversary can coordinate faulty replicas to compromise safety only if more than f replicas are compromised.

In this chapter, the signature of a message m signed by replica p_i is denoted $\langle m \rangle_{p_i}$. We say that a signature is *valid* on message m, if it passes the verification with regard to the public-key of the signer and the message. A vector of signatures of message m signed by a set of replicas $\mathcal{U} = \{p_i, \ldots, p_j\}$ is denoted $\langle m \rangle_{\mathcal{U}}$.

In BChain, the replicas are organized in a metaphorical *chain*, as shown in Fig. 4.1. Each replica is uniquely identified from a set $\Pi = \{p_1, p_2, \cdots, p_n\}$. Initially, we assume that replica IDs are numbered in ascending order. The first replica is called the *head*, denoted p_h, the last replica is called the *tail*, and the $(2f + 1)^{\text{th}}$ replica is called the *proxy tail,* denoted p_p. We divide the replicas into two subsets.

Given a specific chain order, $\mathcal{A}$ contains the first $2f+1$ replicas, initially p_1 to p_{2f+1}. $\mathcal{B}$ contains the last f replicas in the chain, initially p_{2f+2} to p_{3f+1}. For convenience, we also define $\mathcal{A}^{\not p} = \{\mathcal{A} \setminus p_p\}$, excluding the proxy tail, and $\mathcal{A}^{\not h} = \{\mathcal{A} \setminus p_h\}$, excluding the head.

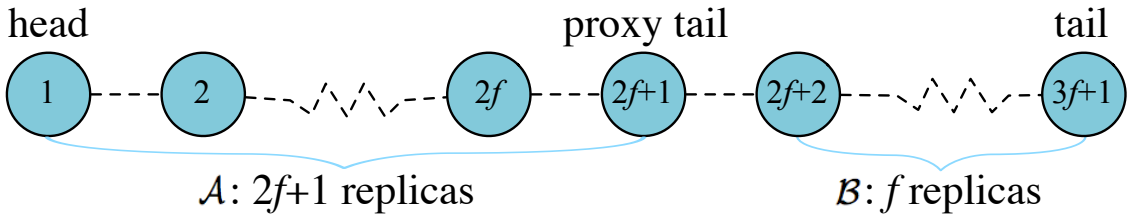


Figure 4.1. BChain-3. Replicas are organized in a chain.

The chain order is maintained by every replica and can be changed the head and is communicated to replicas through message transmissions.[1] For any replica except the head, $p_i \in \mathcal{A}^{\not h}$, we define its *predecessor* $\overleftarrow{p}_i$, initially p_{i-1}, as its preceding replica in the current chain order. For any replica except the proxy tail, $p_i \in \mathcal{A}^{\not p}$, we define its *successor* $\overrightarrow{p}_i$, initially p_{i+1}, as its subsequent replica in the current chain order.

For each $p_i \in \mathcal{A}$, we define its *predecessor set* $\mathcal{P}(p_i)$ and *successor set* $\mathcal{S}(p_i)$, whose elements depend on their individual positions in the chain. If a replica $p_i \neq p_h$ is one of the first $f+1$ replicas, its predecessor set $\mathcal{P}(p_i)$ consists of all the preceding replicas in the chain. For every other replica in $\mathcal{A}$, the predecessor set $\mathcal{P}(p_i)$ consists of the preceding $f+1$ replicas in the chain. If p_i is one of the last $f+1$ replicas in $\mathcal{A}$, the successor set $\mathcal{S}(p_i)$ consists of all the subsequent replicas in $\mathcal{A}$. For every other replica in $\mathcal{A}$, the successor set $\mathcal{S}(p_i)$ consists of the subsequent $f+1$ replicas.

Note that the cardinality of any replica's predecessor set or successor set is at most $f+1$.

[1] This is in contrast to Aliph-Chain, where the chain order is fixed and known to all replicas and clients beforehand.

4.2.2 Protocol Overview

In a gracious execution, as shown in Fig. 4.2, the first $2f+1$ replicas (set $\mathcal{A}$) reach an agreement while the last f replicas (set $\mathcal{B}$) correspondingly update their states based on the agreed-upon requests from set $\mathcal{A}$. BChain transmits two types of messages along the chain: $\langle$Chain$\rangle$ messages transmitted from the head to the proxy tail, and $\langle$Ack$\rangle$ messages transmitted in reverse from the proxy tail to the head. A request is *executed* after a replica accepts the $\langle$Chain$\rangle$ message; a request *commits* at a replica if it accepts the $\langle$Ack$\rangle$ message.

Upon receiving a client request, the head sends a $\langle$Chain$\rangle$ message representing the request to its successor. As soon as the proxy tail accepts the $\langle$Chain$\rangle$ message, it sends a reply to the client and generates an $\langle$Ack$\rangle$ message, which is sent backwards along the chain until it reaches the head. Once a replica in $\mathcal{A}$ accepts the $\langle$Ack$\rangle$ message, it completes the request and forwards its $\langle$Chain$\rangle$ message to replicas in $\mathcal{B}$ to ensure that the message is committed at all the replicas.

To handle failures and ensure liveness, BChain incorporates failure detection and re-chaining protocol that works as follows: Every replica in $\mathcal{A}^{\not p}$ starts a timer after sending a $\langle$Chain$\rangle$ message. Unless an $\langle$Ack$\rangle$ is received before the timer expires, it sends a $\langle$Suspect$\rangle$ message to the head and also along the chain towards the head. Upon seeing $\langle$Suspect$\rangle$ messages, the head starts the re-chaining, by moving faulty replicas to set $\mathcal{B}$ where, if needed, replicas may be replaced in the reconfiguration protocol. In this way, BChain remains robust until new failures occur.

4.2.3 Chaining

We now describe the sequence of steps of the chaining protocol, used to order requests, when there are no failures.

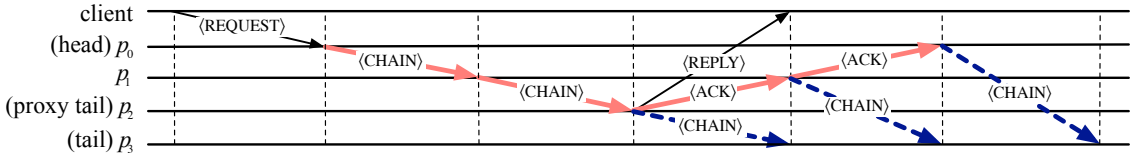


Figure 4.2. BChain-3 common case communication pattern. (This and subsequent pictures are best viewed in color.) All the signatures can be replaced with MACs. All the ⟨CHAIN⟩ and ⟨ACK⟩ messages can be batched. The ⟨CHAIN⟩ messages with dotted, blue lines are the forwarded messages that are stored in logs. *No* conventional broadcast is used at any point in our protocol. For a given batch size b, the number of MAC operations at the bottleneck server (i.e., the proxy tail) is $1 + \frac{3f+2}{b}$.

Step 1: Client sends a request to the head.

A client c requests the execution of state machine operation o by sending a request $m = \langle \text{REQUEST}, o, T, c \rangle_c$ to the replica that it believes to be the head, where T is the timestamp.

Step 2: Assign sequence number and send chain message.

When the head p_h receives a valid $\langle \text{REQUEST}, o, T, c \rangle_c$ message, it assigns a sequence number and sends message $\langle \text{CHAIN}, v, ch, N, m, c, \mathcal{H}, R, \Lambda \rangle_{p_h}$ to its successor, where v is the view number, ch is the number of re-chainings that took place during view v, $\mathcal{H}$ is the hash of its execution history, R is the hash of the reply r to the client containing the execution result, and Λ is the current chain order. Both of $\mathcal{H}$ and R are empty in this step.

Step 3: Execute request and send chain message.

A valid $\langle \text{CHAIN}, v, ch, N, m, c, \mathcal{H}, R, \Lambda \rangle_{\mathcal{P}(p_j)}$ message is sent to replica p_j by its predecessor, which contains valid signatures by replicas in $\mathcal{P}(p_j)$. The replica p_j updates $\mathcal{H}$ and R fields if necessary, appends its signature to the ⟨CHAIN⟩ message, and sends to its successor. Note that the $\mathcal{H}$ and R fields are empty if p_j is among the first f replicas, and both $\mathcal{H}$ and R must be verified before proceeding.

Each time a replica $p_j \in \mathcal{A}^{\not\mathcal{F}}$ sends a $\langle \textsc{Chain} \rangle$ message, it sets a timer, expecting an $\langle \textsc{Ack} \rangle$ message, or a $\langle \textsc{Suspect} \rangle$ message signaling some replica failures.

Step 4: Proxy tail sends reply to the client and commits the request.
If the proxy tail p_j accepts a $\langle \textsc{Chain} \rangle$ message, it computes its own signature and sends the client the reply r, along with the $\langle \textsc{Chain} \rangle$ message it accepts. It also sends an $\langle \textsc{Ack}, v, ch, N, D(m), c \rangle_{p_j}$ message to its predecessor. In addition, it *forwards* the corresponding $\langle \textsc{Chain}, v, ch, N, m, c, \mathcal{H}, R, \Lambda \rangle_{p_j}$ message to all replicas in $\mathcal{B}$. The request commits at the proxy tail.

Step 5: Client completes the request or retransmits.
The client completes the request if it receives $\langle \textsc{Reply} \rangle$ message from the proxy tail with signatures by the last $f + 1$ replicas in the chain. Otherwise, it retransmits the request to all replicas.

Step 6: Other replicas in $\mathcal{A}$ commit the request.
A valid $\langle \textsc{Ack}, v, ch, N, D(m), c \rangle_{\mathcal{S}(p_j)}$ message is sent to replica p_j by its successor, which contains valid signatures by replicas in $\mathcal{S}(p_j)$. The replica appends its own signature and sends to its predecessor.

Step 7: Replicas in $\mathcal{B}$ execute and commit request.
The replicas in $\mathcal{B}$ collects $f + 1$ matching $\langle \textsc{Chain} \rangle$ messages, and executes the operation, completing the current round. Thus, the request commits at each correct replica in $\mathcal{B}$.

4.2.4 Re-chaining

To facilitate failure detection and ensure that BChain remains live, we introduce a protocol we call *re-chaining*. With re-chaining, we can make progress with a bounded number of failures, despite incorrect suspicions, in a partially synchronous environ-

Algorithm 4 Failure detector at replica p_i

1: **upon** $\langle$CHAIN$\rangle$ sent by p_i
2: $\quad starttimer(\Delta_{1,p_i})$

3: **upon** $\langle$Timeout$, \Delta_{1,p_i}\rangle$ {Accuser p_i}
4: $\quad$ send $\langle$SUSPECT$, \vec{p}_i, m, ch, v\rangle_{p_i}$ to $\overleftarrow{p}_i$ and p_h

5: **upon** $\langle$ACK$\rangle$ from $\vec{p}_i$
6: $\quad canceltimer(\Delta_{1,p_i})$

7: **upon** [SUSPECT$, p_y, m, ch, v$] from $\vec{p}_i$
8: $\quad$ forward [SUSPECT$, p_y, m, ch, v$] to $\overleftarrow{p}_i$
9: $\quad canceltimer(\Delta_{1,p_i})$

ment. The algorithm ensures that eventually all the faulty replicas be identified and appropriately dealt with. The strategy of the re-chaining algorithm is to move replicas that are *suspected* to set $\mathcal{B}$, where if deemed necessary, they are rejuvenated.

BChain failure detector. The objective of the BChain failure detector is to identify faulty replicas, and issue a new chain configuration and to ensure that progress can be made. It is implemented as a timer on $\langle$CHAIN$\rangle$ messages, as shown in Algorithm 4. On sending a $\langle$CHAIN$\rangle$ message m, replica p_i starts a timer, Δ_{1,p_i}. If the replica receives an $\langle$ACK$\rangle$ for the message before the timer expires, it cancels the timer and starts a new one for the next request in the queue, if any. Otherwise, it sends both the head and its predecessor a $\langle$SUSPECT$, \vec{p}_i, m, ch, v\rangle_t$ o signal the failure of its successor. Moreover, if p_i receives a $\langle$SUSPECT$\rangle$ message from its successor, the message is forwarded to p_i's predecessor, along the chain until it reaches the head. To prevent that a faulty replica fails to forward the $\langle$SUSPECT$\rangle$ message, it is also sent directly to the head. Passing it along the chain allows us to cancel timers and

reduce the number of suspect messages.

Let p_i be the *accuser*; then the *accused* can only be its successor, $\vec{p}_i$. This is ensured by having the accuser sign the ⟨SUSPECT⟩ message, just as an ⟨ACK⟩ message. On receiving a ⟨SUSPECT⟩, the head starts re-chaining via a new ⟨CHAIN⟩ message. If the head receives multiple ⟨SUSPECT⟩ messages, only the one *closest* to the proxy tail is handled. Handling a ⟨SUSPECT⟩ message is done by increasing ch, selecting a new chain order Λ, and sending a ⟨CHAIN⟩ message to order the same request again.

Re-chaining algorithms. We provide two re-chaining algorithms for BChain-3, Algorithm 5 and 6. To explain these algorithms, assume that the head, p_h, has received a ⟨SUSPECT⟩ message from a replica p_x suspecting is successor p_y. Let p_z be the first replica in set $\mathcal{B}$. Both algorithms show how the head selects a new chain order. Both are *efficient* in the sense that the number of re-chainings needed is proportional to the number of existing failures t instead of the maximum number f. We levy no assumptions on how failures are distributed in the chain.

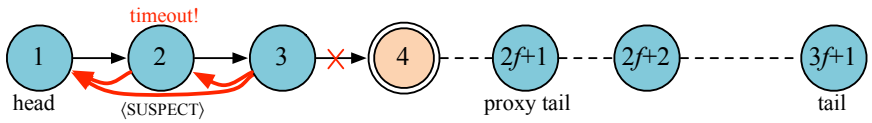
Re-chaining-I—crash failures handled first. Algorithm 5 is reasonably efficient; in the worst case, t faulty replicas can be removed with at most $3t$ re-chainings. More specifically, if the head is correct and $3t \leq f$, the faulty replicas are moved to the end of chain after at most $3t$ re-chainings; if $3t > f$, at most $3t$ re-chainings are necessary and at most $3t-f$ replicas are replaced in the reconfiguration protocol (§4.2.6), assuming that any individual replica can be reconfigured within f re-chainings. Algorithm 5 is even more efficient when handling timing and omission failures, with one such replica being removed using only one re-chaining. Despite the succinct algorithm, the proof of the correctness for the general case is complicated [39]. To help grasp the underlying idea, consider the following *simple* examples.

▷ Example (1): In Figure 4.3, replica p_4 has a timing failure. This causes p_3 to

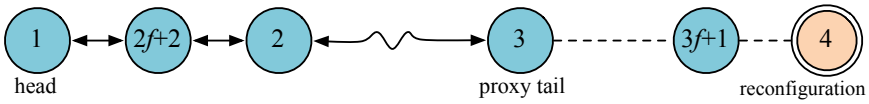
Algorithm 5 BChain-3 Re-chaining-I

1: **upon** [SUSPECT, p_y, m, ch, v] from p_x {At the head, p_h}
2: **if** $p_x \neq p_h$ **then** {p_x is not the head}
3: p_z is put to the 2^{nd} position {$p_z = \mathcal{B}[1]$}
4: p_x is put to the $(2f+1)^{\text{th}}$ position
5: p_y is put to the end

send a ⟨SUSPECT⟩ message up the chain to accuse p_4. According to our re-chaining algorithm, p_3 is moved to the $(2f+1)^{\text{th}}$ position and becomes the proxy tail, and p_4 is moved to the end of the chain and becomes the tail. Our fundamental design principle is that timing failures should be given top priority.



(a) p_2 generates a ⟨SUSPECT⟩ message to accuse p_3

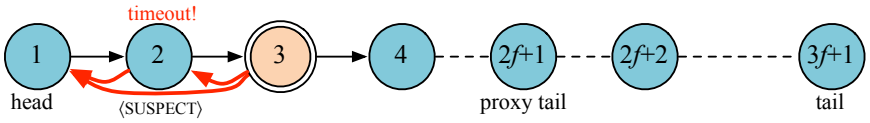


(b) p_3 is moved to the tail

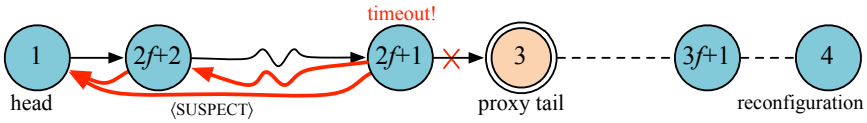
Figure 4.3. Example (1). A faulty replica is denoted by a double circle. After the timer expires, replica p_3 issues a ⟨SUSPECT⟩ message to accuse p_4 (which is faulty). The head moves p_3 to the proxy tail position and the faulty replica p_4 to the end of the chain.

▷ Example (2): In Figure 4.4, p_3 is the only faulty replica. We consider the circumstance where p_3 sends the head a ⟨SUSPECT⟩ message to frame its successor p_4 even if p_4 follows the protocol. According to our re-chaining algorithm, replica p_4 will be moved to the tail, while p_3 becomes the new proxy tail. However, from then

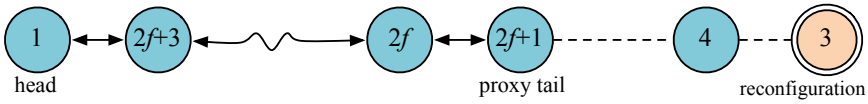
on, p_3 can no longer accuse any replicas. It either follows the specification of the protocol, or chooses not to participate in the agreement, in which case p_3 will be moved to the tail. The example illustrates another important designing rationale that an adversarial replica cannot constantly accuse correct replicas.



(a) p_2 generates a ⟨SUSPECT⟩ message to *maliciously* accuse p_3



(b) p_{2f+1} generates a ⟨SUSPECT⟩ message to accuse p_2



(c) p_2 is moved to the tail

Figure 4.4. Example (2). Replica p_2 maliciously sends a ⟨SUSPECT⟩ message to accuse p_3. The head moves p_2 to the proxy tail and p_3 to the end of the chain. If p_2 does not behave, it will be accused by its predecessor p_{2f+1} such that in another round of re-chaining p_2 is moved to the end.

Re-chaining-II—improved efficiency. Algorithm 6 can provide improved efficiency for the *worst* case. The underlying idea is simple. Every time the head receives a ⟨SUSPECT⟩ message, both the accuser and the accused are moved to the end of the chain. Algorithm 6 does not prioritize crash failures, and it relies on a stronger reconfiguration assumption. If the head is correct and $2t \leq f$, the faulty replicas are moved to the end of chain after at most $2t$ re-chainings; if $2t > f$, at most $2t$

re-chainings are necessary and at most $2t - f$ replica reconfigurations (§4.2.6) are needed, assuming that any individual replica can be reconfigured within $\lfloor f/2 \rfloor$ re-chainings. When an accused replica is moved to the end of chain, the reconfiguration process is initialized, either offline or online. The replicas moved to the end of the chain are all "tainted" and reconfigured, as we discuss in §4.2.6.

Algorithm 6 BChain-3 Re-chaining-II

1: **upon** [SUSPECT, p_y, m, ch, v] from p_x
2: **if** $p_x \neq p_h$ **then** {p_x is not the head}
3: p_x is put to the $(3f)^{\text{th}}$ position
4: p_y is put to the end

Timer setup. Existing BFT protocols typically only keep timers for view changes, while BChain also requires timers for ⟨ACK⟩ and ⟨CHAIN⟩ messages. To achieve accurate failure detection, we need different values for each of the timers for the different replicas in the chain.

The timeout for each replica $p_i \in \mathcal{A}$ is defined as $\Delta_{1,i} = \mathcal{F}(\Delta_1, l_i)$, where $\mathcal{F}$ is a fixed and efficiently computable function, Δ_1 is the base timeout, and l_i is p_i's location in the chain order. Note that for p_h, we have that $l_h = 1$ and thus $\mathcal{F}(\Delta_1, 1) = \Delta_1$. Correspondingly, for p_p, we have that $l_p = 2f + 1$ and $\mathcal{F}(\Delta_1, 2f + 1) = 0$. It is reasonable to adopt a *linear function* with respect to the position of each replica as the timer function. i.e., $\mathcal{F}(\Delta_1, l_i) = \frac{2f+1-l_i}{2f}\Delta_1$. As an example, in the case of $n = 4$ and $f = 1$, we set that $\Delta_{1,p_1} = \mathcal{F}(\Delta_1, 1) = \Delta_1$, $\Delta_{1,p_2} = \mathcal{F}(\Delta_1, 2) = \Delta_1/2$, and $\Delta_{1,p_3} = \mathcal{F}(\Delta_1, 3) = 0$.

To detect and deter misbehaving replicas that always delay requests to the upper bound timeout value to increase system latency, we additionally verify the processing delays in their *average* cases and allow to suspect those who frequently do so.

Concretely, each replica p_i maintains an additional average latency Δ'_{1,p_i} such that $\Delta'_{1,p_i} < \Delta_{1,p_i}$, which is used to detect slow or faulty replicas mentioned above. A replica suspect their successor in the following two cases: 1) The actual latency in one round makes the average latency exceed $\alpha * \Delta'_{1,p_i}$; 2) The actual latency in one round exceeds $\beta * \Delta'_{1,p_i}$. The first case prevents temporarily slow replicas from being suspected. However, this case is allowed limited times and the timers will not be adjusted accordingly. If non of the two cases is not true, the value of Δ_{1,p_i} is adjusted according to Δ'_{1,p_i}.

4.2.5 View Change

The view change protocol has two functions: (1) to select a new head when the current head is deemed faulty, and (2) to adjust the timers to ensure eventual progress, despite deficient initial timer configuration.

A correct replica p_i votes for view change if either (1) it suspects the head to be faulty, or (2) it receives $f + 1$ $\langle$ViewChange$\rangle$ messages. The replica votes for view change and moves to a new view by sending all replicas a $\langle$ViewChange$\rangle$ message that includes the new view number, the current chain order, a set of valid checkpoint messages, and a set of requests that commit locally with proof of execution. For each request that commits locally, if $p_i \in \mathcal{A}$, then a proof of execution for a request contains a $\langle$Chain$\rangle$ message with signatures from $\mathcal{P}(p_i)$ and an $\langle$Ack$\rangle$ message with signatures from $\mathcal{S}(p_i)$. Otherwise, a proof of execution contains $f + 1$ $\langle$Chain$\rangle$ messages. Upon sending a $\langle$ViewChange$\rangle$ message, p_i stops receiving messages except $\langle$Checkpoint$\rangle$, $\langle$NewView$\rangle$, or other $\langle$ViewChange$\rangle$ messages.

When the new head collects $2f + 1$ $\langle$ViewChange$\rangle$ messages, it sends all replicas a $\langle$NewView$\rangle$ message which includes the new chain order in which the head of

the old view has been moved to the end of the chain, a set of valid ⟨ViewChange⟩ messages, and a set of ⟨Chain⟩ messages.

The other function of view change is to adjust the timers. In addition to the timer Δ_1 maintained for re-chaining, BChain has two timers for view changes, Δ_2 and Δ_3. Δ_2 is a timer maintained for the current view v when a replica is waiting for a request to be committed, while Δ_3 is a timer for ⟨NewView⟩, when a replica votes for a view change and waits for the ⟨NewView⟩. Algorithm 7 describes how to initialize, maintain, and adjust these timers.

The view change timer Δ_2 at a replica is set up for the first request in the queue. A replica sends a ⟨ViewChange⟩ message to all replicas and votes for view change if Δ_2 expires or it receives $f + 1$ ⟨ViewChange⟩ messages. In either case, when a replica votes for view change, it cancels its timer Δ_2.

After a replica collects $2f + 1$ ⟨ViewChange⟩ messages (including its own), it starts a timer Δ_3 and waits for the ⟨NewView⟩ message. If the replica does not receive ⟨NewView⟩ message before Δ_3 expires, it starts a *new* ⟨ViewChange⟩ and updates Δ_3 with a new value $g_3(\Delta_3)$.

When a replica receives the ⟨NewView⟩ message, it sets Δ_1 and Δ_2 using $g_1(\Delta_1)$ and $g_2(\Delta_2)$, respectively. In practice, the functions $g_1(\cdot)$, $g_2(\cdot)$, and $g_3(\cdot)$ could simply double the current timeouts.

To avoid the circumstance that the timeouts for Δ_1 and Δ_2 increase without bound, we introduce upper bounds for both of them. Once either timer exceeds the prescribed bound, the system starts reconfiguration.

Algorithm 7 View Change Handling and Timers at p_i

1: $\Delta_2 \leftarrow init_{\Delta_2}$; $\Delta_3 \leftarrow init_{\Delta_3}$
2: $voted \leftarrow$ **false**

3: **upon** $\langle \text{Timeout}, \Delta_2 \rangle$
4: send $\langle \text{ViewChange} \rangle$
5: $voted \leftarrow$ **true**

6: **upon** $f + 1$ $\langle \text{ViewChange} \rangle \wedge \neg voted$
7: send $\langle \text{ViewChange} \rangle$
8: $voted \leftarrow$ **true**
9: $canceltimer(\Delta_2)$

10: **upon** $2f + 1$ $\langle \text{ViewChange} \rangle$
11: $starttimer(\Delta_3)$

12: **upon** $\langle \text{Timeout}, \Delta_3 \rangle$
13: $\Delta_3 \leftarrow g_3(\Delta_3)$
14: send *new* $\langle \text{ViewChange} \rangle$

15: **upon** $\langle \text{NewView} \rangle$
16: $canceltimer(\Delta_3)$
17: $\Delta_1 \leftarrow g_1(\Delta_1)$
18: $\Delta_2 \leftarrow g_2(\Delta_2)$

4.2.6 Reconfiguration

Reconfiguration is a general technique, often abstracted as stopping the current state machine and restarting it with a new set of replicas [77]. This does not preclude

reusing non-faulty replicas in a new configuration. Reconfiguration has traditionally only been considered in the crash failure model. In this section, we describe a new reconfiguration technique customized for our BChain protocol, which is much less intrusive than existing techniques.

Our reconfiguration technique works in concert with our re-chaining protocol. Recall that BChain-3 re-chaining protocol moves faulty replicas to set $\mathcal{B}$, while replicas that remain in $\mathcal{A}$ continues processing client requests. The reconfiguration procedure operates *out-of-band*, and thus does not disrupt request processing. Since it can be done out-of-band, it is not time sensitive, unless more failures occur.

An alternative to reconfiguration could be to recover suspected replicas. However, recovery is not possible for some types of failures, such as permanent failures. Recovery may also take a long time, e.g., waiting for a machine to reboot, leaving the system vulnerable to further failures.

The key idea of our reconfiguration algorithm is to *replace* the replicas that were moved to set $\mathcal{B}$, with new replicas. A new replica first acquires a unique identifier. It also obtains a public-private key pair, and a shared symmetric key with each other replica in the system.

To initialize reconfiguration, a new replica in $\mathcal{B}$ with a unique identifier u sends a [ReconRequest] to all replicas in the system. Upon receiving the request, correct replicas send signed messages with their current [History] to replica u. Meanwhile, the replicas in $\mathcal{A}$ continue to execute the chaining protocol, where they also forward $\langle$Chain$\rangle$ messages to the newly joined replica u. In addition, replicas in $\mathcal{A}$ also retransmit missing $\langle$Chain$\rangle$ messages to the replicas in $\mathcal{B}$, including u, as the protocol requires. After collecting at least $f + 1$ matching authenticated [History] messages, u updates its state using the retrieved history and the $\langle$Chain$\rangle$ messages it has received. At this point, u can be promoted to $\mathcal{A}$ when deemed necessary.

It is clear that the reconfiguration algorithm can be performed concurrently with request processing, and as such is not time sensitive. This is because a newly joined replica is not immediately put into active use. Depending on the re-chaining algorithm, a new replica will not be used until f re-chainings have taken place (Algorithm 5), or $\lfloor f/2 \rfloor$ re-chainings with Algorithm 6.

Note that BChain-3 remains safe even if no reconfiguration procedure is used. Under the circumstance that there are only a small number of faulty replicas, e.g. $3t < f$, no regular reconfiguration is required to ensure liveness. Reconfiguration can be triggered periodically, as in other BFT protocols, or when frequent view changes and re-chainings occur.

Also note that, one might introduce a third set $\mathcal{C}$ that contains all of the "faulty" replicas, while $\mathcal{B}$ contains those that have been reconfigured and can be moved back to $\mathcal{A}$ on demand. The system has to wait if $\mathcal{B}$ is empty.

4.3 BChain without Reconfiguration

We now discuss BChain-5, which uses $n = 5f + 1$ replicas to tolerate f Byzantine failures, just as Q/U [2] and Zyzzyva5 [69]. With $5f + 1$ replicas at our disposal, we design an efficient re-chaining algorithm, which allows the faulty replicas to be identified easily without relying on reconfiguration. Meanwhile, a Byzantine quorum of replicas can reach agreement.

BChain-5 relies on the concept of Byzantine quorum protocols [84]. As depicted below in Fig. 4.5, set $\mathcal{A}$ is a Byzantine quorum which consists of $\lceil \frac{n+f+1}{2} \rceil = 3f + 1$ replicas, while set $\mathcal{B}$ consists of the remaining of $2f$ replicas.

BChain-5 has four sub-protocols: chaining, re-chaining, view change, and checkpoint. In contrast, BChain-3 additionally requires a reconfiguration protocol. The proto-

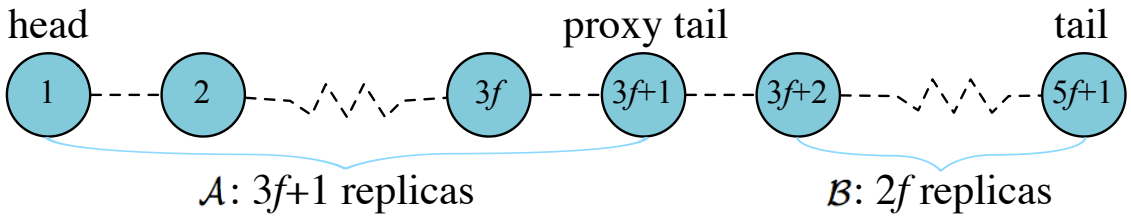


Figure 4.5. BChain-5.

cols for BChain-3 and BChain-5 are identical with respect to message flow. The main difference lies in the size of the $\mathcal{A}$ set, which now consists of $3f + 1$ replicas. Algorithm 8 shows the re-chaining algorithm of BChain-5; it is structurally the same as Algorithm 6 for BChain-3.

Algorithm 8 BChain-5 Re-chaining

1: **upon** [SUSPECT, p_y, m, ch, v] from p_x
2: **if** $p_x \neq p_h$ **then** {p_x is not the head}
3: p_x is put to the $(5f)^{\text{th}}$ position
4: p_y is put to the end

Assuming the timers are accurately configured and that the head is non-faulty, it takes at most f re-chainings to move f failures to the tail set $\mathcal{B}$. The proofs for safety and liveness of BChain-5 are easier than those of BChain-3 due to a different re-chaining algorithm and the absence of the reconfiguration procedure.

To Reconfigure or not to Reconfigure? The primary benefit of BChain-5 over BChain-3 is that it eliminates the need for reconfiguration to achieve liveness. This is beneficial, since reconfiguration needs additional resources, such as machines to host reconfigured replicas. However, since BChain-5 can identify and move faulty replicas to the tail set $\mathcal{B}$, we can still leverage the reconfiguration procedure on the replicas in $\mathcal{B}$, to provide long-term system safety and liveness. This does not contradict the

claim that BChain-5 does not need reconfiguration; rather, it just makes the system more robust. Furthermore, BChain-5 provides flexibility with respect to when the system should be reconfigured. Specifically, reconfiguration can happen any time after the system achieves a stable state or simply has run for a "long enough" period of time.

BChain-α. We can generalize BChain-3 and BChain-5 to provide efficient trade-offs between the total number of replicas, the number of reconfigurations needed, as well as the rate of reconfiguration. Let BChain-α be the generalized protocol, where $\alpha \in [3..5]$ is a rational. We can show that for an instance of BChain-α, the safety and liveness properties can be guaranteed if $f \leq \lfloor \frac{n-1}{\alpha} \rfloor$. The value of α should not be less than 3; otherwise it would neither be safe nor live. It does not need to be greater than 5, since BChain-5 already eliminates the need for reconfiguration.

4.4 Optimizations and Extensions

We now discuss some optimizations and extensions to BChain. Specifically, we show how to replace (most) signatures with MACs, and how to combine MAC-based and signature-based BChain. We also discuss two variants of BChain, including a *pure* MAC-based protocol *without* reconfiguration when $n = 4$ and $f = 1$.

Replacing *most* signatures with MACs. As shown in previous work [18, 34, 50, 69], it is possible to replace most signatures with MACs to reduce the computational overhead. This is also possible for BChain. In particular, it turns out that signatures for [Request], ⟨Ack⟩, and ⟨Checkpoint⟩ can be replaced with a vector of MACs. However, in general, signatures on ⟨Chain⟩ messages cannot be replaced with MACs. Thus, we call this variant Most-MAC-BChain.

In our re-chaining protocol, a replica suspects its successor if it does not receive

the ⟨Ack⟩ message in time. If a replica accepts and forwards a ⟨Chain⟩ message to its successor, it is trying to convince its successor that the message is correct. Meanwhile, the successor is able to verify if all its preceding replicas indeed honestly authenticated themselves. This requires transferability for verification, a property that signatures enjoys, while MACs do not.

We briefly describe an attack where a single replica can "frame" any honest replica—a scenario that our failure detection mechanism cannot handle, e.g. when ⟨Chain⟩ messages use MACs instead of signatures. Consider the following example, where there is only one faulty replica p_i, and $\vec{p}_i = p_j$ and $\vec{p}_j = p_k$. The faulty replica p_i simply generates a valid MAC for p_j and an invalid MAC for p_k. Replica p_j will accept it since the corresponding MAC is valid. It then adds its own MAC-based signature, and forwards the message to p_k. Since p_k receives the message with an invalid MAC produced by p_i, it aborts. Replica p_j will suspect p_k according to our algorithm, while p_i is the faulty one. Generalizing the result, a faulty replica can frame any honest replica without being suspected.

Replacing *all* signatures with MACs. We now discuss a variant of BChain, called All-MAC-BChain, in which all signatures are replaced with a vector of MACs, even for ⟨Chain⟩ messages in $\mathcal{A}$. As we discussed above however, these ⟨Chain⟩ messages must use signatures. However, if the head does not receive the ⟨Ack⟩ message on time, we can simply switch to Most-MAC-BChain to start the re-chaining protocol. Once the system regains liveness or faulty replicas have been reconfigured, we can switch back to All-MAC-BChain. This leads to the most efficient implementation of BChain. The performance in gracious executions will be that of All-MAC-BChain. In case of failures, the performance will be that of Most-MAC-BChain, with most signatures replaced with MACs and taking advantage of pipelining.

The combined protocol is fundamentally different from the ones described in [50]

such as Aliph, which does not perform well even in the presence of a single faulty replica. Note that we evaluate our BChain protocols in Table 7.2 using this protocol variant.

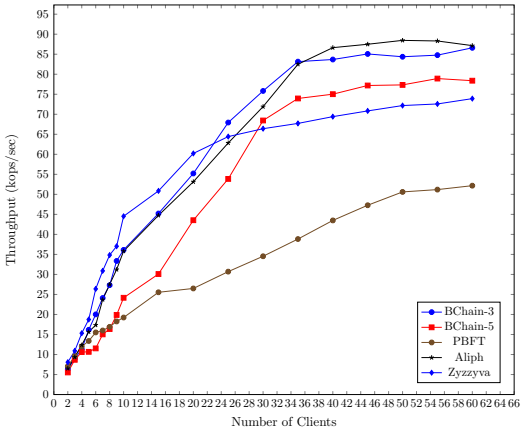
BChain-3 with $n = 4$. We now consider BChain-3 configured with $(n = 4, f = 1)$, and show that this allows two interesting optimizations: BChain-3 without reconfiguration and All-MAC-BChain-3. This configuration of BChain is quite attractive, since its replication costs are reasonable for many applications, such as Google's file system [48].

BChain-3 without Reconfiguration. We show that, with a slight refinement of the re-chaining algorithm, BChain-3 can also avoid reconfiguration:

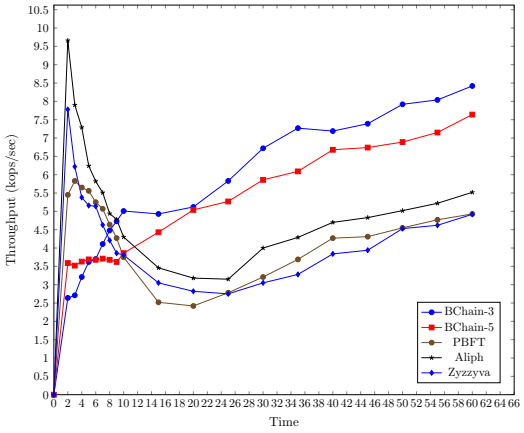
Upon receiving a ⟨SUSPECT⟩ from an accuser among the first two replicas in the chain, the head starts re-chaining. If the head is the accuser, then the accused is moved to the end of the chain. Otherwise, the accuser becomes the proxy tail, while the accused becomes the tail. It no longer needs to run the reconfiguration algorithm. In any future runs of BChain, if the head does not receive a correct ⟨ACK⟩ message, it simply *switches* the proxy tail (i.e., the third replica) and the tail (i.e., the last replica). A faulty replica can be identified with at most two re-chainings in case of synchrony. The view change algorithm is still the same as for BChain-3, which guarantees that eventually it achieves liveness with a bounded number of re-chainings in the partially synchronous environment.

All-MAC-BChain-3 via All MAC-based signatures. We now show that, contrary to the general case, BChain-3 with a $(n = 4, f = 1)$ configuration, can be implemented using only MACs. The reason we can do this is that the second replica in the chain can no longer frame its successor replica, while the behavior of the head is restricted by view changes. Thus, a total of twelve MACs are needed for communication

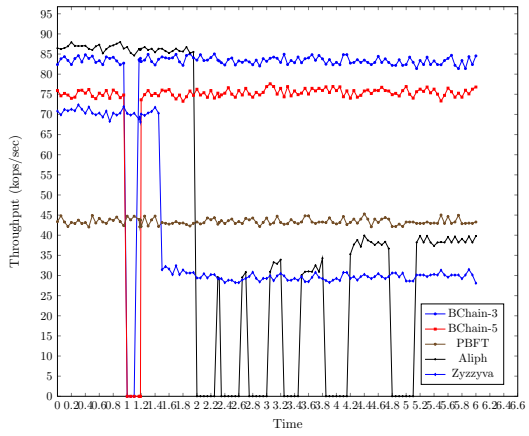
between replicas and between replicas and clients. Recall also that a faulty replica can be identified with at most two re-chainings, and no reconfiguration is required.



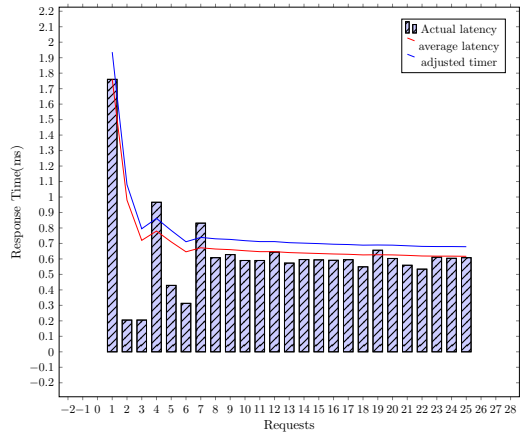
(a) Throughput for the 0/0 benchmark as the number of clients varies.



(b) Latency for the 0/0 benchmark as the number of clients varies.

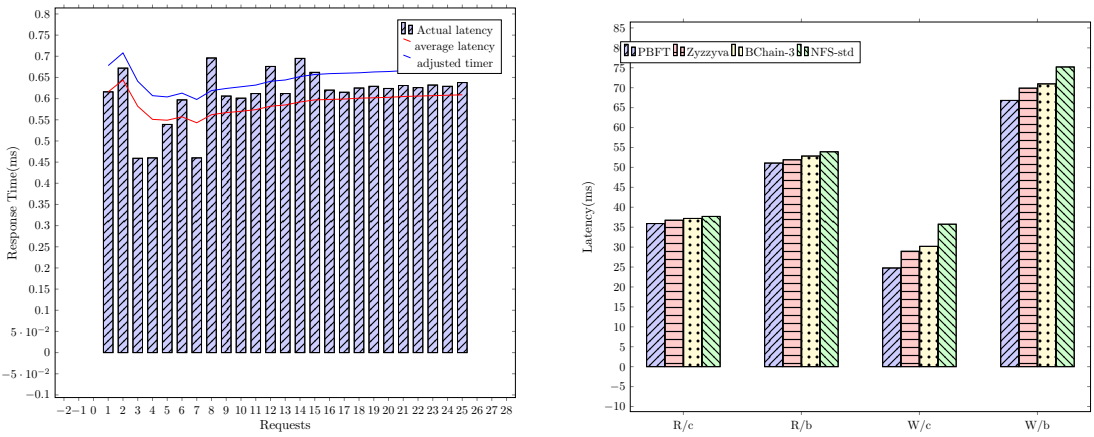


(c) Throughput for 0/0 benchmark with 40 clients. A failure is injected at 1s for BChain-3, BChain-5 and PBFT, at 1.5s for Zyzzyva, and at 2s for Aliph.



(d) Performance under failure. The actual latency, the average value of base timers, and the value of base timers for setting timers of the head.

Figure 4.6. Protocol Evaluation-1.



(a) Performance attack. The actual latency, the average value of base timers, and the value of base timers for setting timers of the uncivil replica.

(b) Bonnie++ Benchmark. R/c, R/b, W/c, and W/b stand for per-character file reading, block file reading, per-character file writing, and block file writing, respectively.

Figure 4.7. Protocol Evaluation-2.

4.5 Evaluation

This section studies the performance of BChain-3 and BChain-5 and compares them with three well-known BFT protocols—PBFT [18], Zyzzyva [69], and Aliph [50]. Aliph [50,111] switches between three protocols: Quorum, Chain, and a backup, e.g., PBFT. As Quorum does not work under contention, Aliph uses Chain for gracious execution under high concurrency. Aliph-Chain enjoys the highest throughput when there are no failures, however, as we will see, Aliph cannot sustain its performance during failure scenarios, where BChain is superior.

We study the performance using two types of benchmarks: the micro-benchmarks by Castro and Liskov [18] and the Bonnie++ benchmark [30]. We use micro-benchmarks to assess throughput, latency, scalability, and performance during failures of all the five protocols. In the x/y micro-benchmarks, clients send x kB requests

and receive y kB replies. Clients invoke requests in a *closed-loop*, where a client does not start a new request before receiving a reply for a previous one. All the protocols implement batching of concurrent requests to reduce cryptographic and communication overheads.

All experiments were carried out on DeterLab [12], utilizing a cluster of up to 65 identical machines. Each machine is equipped with a 2.13GHz Xeon processor and 4GB of RAM. They are connected through a 100Mbps switched LAN.

As we discuss in the following, for gracious execution, both BChain-3 and BChain-5 achieve higher throughput and lower latency than PBFT and Zyzzyva especially when the number of concurrent client requests is large, while BChain-3 has performance similar to the Aliph-Chain protocol. Our experiment bolsters the point of view described by Guerraoui *et al.* [50] that (authenticated) chaining replication can lead to an increase in throughput and a reduction in latency under high concurrency. In case of failures, both BChain-3 and BChain-5 outperforms all the other protocols by a wide margin, due to BChain's unique re-chaining protocol. Through the timeout adjustment scheme, we show that a faulty replica cannot make the system slower by manipulating the timeouts. In addition, the results of the NFS use case experiments show that BChain-3 is only 1% slower than a standard unreplicated NFS.

4.5.1 Performance in Gracious Execution

Throughput. We discuss the throughput of BChain-3 and BChain-5 with different workloads *under contention*, where there are multiple clients issuing requests. We evaluate two configurations of BChain with $f = 1$: BChain-3 with $n = 4$ and BChain-5 with $n = 6$, both using All-MAC-BChain.

We begin by assessing the throughput in the 0/0 benchmark as the number of

clients varies. As shown in Fig. 4.6(a), all the other protocols outperform PBFT by a wide margin. With less than 20 clients, Zyzzyva achieves slightly higher throughput than the rest. But as the number of clients increases, Aliph-Chain, BChain-3, and BChain-5 gain an advantage over Zyzzyva. While BChain-3 and Aliph-Chain have comparable performance, they both outperform BChain-5. For both Aliph-Chain and BChain-3, peak throughput observed is 22% and 41% higher than that of Zyzzyva and PBFT, respectively. Note that the pipelining execution of our protocol explains why BChain-3 does not perform as well when the number of clients is small and why it scales increasingly better as the number grows larger.

Latency. We examine and compare the latency for the five protocols in the 0/0 benchmark, as depicted in Fig. 4.6(b). As expected, we can see that when the number of clients is less than 10, all the chain replication based BFT protocols experience significantly higher latency than both Zyzzyva and PBFT. As the number of clients increases however, BChain achieves around 30% lower latency than Zyzzyva. Indeed, BChain-3, for instance, takes $4f$ message exchanges to complete a single request, which makes its latency higher than prior BFT protocols, such as PBFT and Zyzzyva in case of small number of clients. However, our experiments show that BChain-3 and BChain-5 achieve lower latency as the number of clients increases, where the pipeline is leveraged to compensate for the latency inflicted by the increased number of exchanges.

Scalability. We tested the performance of BChain-3 varying the maximum number of faulty replicas. All experiments are carried out using the 0/0 benchmark. The results are summarized in Table 4.1, comparing BChain-3 with PBFT and Zyzzyva, for both throughput and latency for different f. We ran the experiments with both 20 and 60 clients.

Table 4.1. Throughput and latency improvement of BChain-3, comparing with PBFT and Zyzzyva, when f differs. Values with parenthesis in red represent negative improvement.

Number of Clients		20		60	
Compared Protocol		PBFT	Zyzzyva	PBFT	Zyzzyva
$f = 1$	throughput	48.61%	17.65%	41.54%	22.59%
	latency	27.14%	5.44%	33.72%	26.96%
$f = 2$	throughput	36.95%	2.50%	37.12%	15.67%
	latency	25.50%	5.79%	30.50%	23.85%
$f = 3$	throughput	1.69%	(1.93%)	36.86%	14.04%
	latency	(1.36%)	(2.57%)	26.03%	15.14%

As shown, with almost all the parameters, BChain-3 achieves generally higher throughput and lower latency than PBFT and Zyzzyva. We observe that, the advantage of BChain-3 over other protocols decreases as f grows. When f grows to 3 and the number of clients is 20, BChain achieves lower performance than both PBFT and Zyzzyva. However, when the number of clients is large, BChain still achieves better performance.

In contrast to many other BFT protocols with a constant number of one-way message exchanges in the critical path (c.f. Table 7.2), the number of exchanges in BChain-3 is proportional to f. In BChain-3, a client needs to wait for $2f+2$ exchanges to receive enough correct replies and the head needs to wait for $4f$ exchanges to commit a request. This intuitively explains why the performance benefits of BChain-3 becomes smaller as f increases.

However, as the pipeline is saturated with clients requests and large request batching is used, compensating for the latency induced by the increased f, BChain-3

can perform consistently well. For example, as shown in Table 7.2, the number of MAC operations at the bottleneck server in BChain-3 is only $1 + \frac{3f+2}{b}$, compared to $2 + \frac{3f}{b}$ in Zyzzyva and $2 + \frac{8f+1}{b}$ in PBFT, where b is the batch size. When f equals 3 and b equals 20, the number of MAC operations of the bottleneck server is 1.55 for BChain, 2.45 for Zyzzyva, and 3.25 for PBFT. When f is 3 and b is 60, the numbers are 1.18 for BChain, 2.15 for Zyzzyva, and 2.41 for PBFT.

4.5.2 Performance under Failures

We now compare the performance of BChain with the other BFT protocols under two scenarios: a simple crash failure scenario and a Byzantine faulty replica that performs a performance attack, i.e., it makes the system slow by manipulating the timer. Note that the case where a faulty replica fails to send/receive correct messages can be viewed as the case where the faulty replica crashes since a replica only send/receive messages from a single replica in BChain. As the results in Fig. 4.6(c) show, BChain has superior reaction to failures. When BChain detects a failure, it will start re-chaining. At the moment when re-chaining starts, the throughput of BChain temporarily drops to zero. After the chain has been re-ordered, BChain quickly recovers its steady state throughput. The dominant factor deciding the duration of this throughput drop (i.e. increased latency) is the failure detection timeout, not the re-chaining. On the other hand, we also show that BChain resists performance attacks well, such that faulty replicas can slow the system to a pre-specified degree.

Crash Failure. We compare the throughput during crash failure for BChain-3, BChain-5, PBFT, Zyzzyva, and Aliph. The results are shown in Fig. 4.6(c). We use $f = 1$, message batching, and 40 clients. To avoid clutter in the plot, we used different failure inject times for the protocols: BChain-3, BChain-5, and PBFT all

experience a failure at 1s, while Zyzzyva and Aliph experience a failure at 1.5s and 2s, respectively.

We note that Aliph [50,111] generally switches between three protocols: Quorum, Chain, and a backup, e.g., PBFT. The backup is necessary because the Chain and Quorum protocols cannot themselves operate with failures. For our experiments, we adopt a combination of Chain and PBFT as backup, since Aliph's Quorum protocol does not work under contention. Moreover, Aliph uses a configuration parameter k, denoting the number of requests to be executed when running with the backup protocol. We experimented with both $k = 1$ and using exponentially increasing $k = 2^i$. The latter had largest throughput of the two k-configurations, and thus in Fig. 4.6(c) we only show Aliph ($k = 2^i$).

Even though Aliph exhibits slightly higher throughput than BChain-3 prior to the failure, its throughput takes a significant beating upon failure, dropping well below that of the PBFT baseline. As Fig. 4.6(c) shows, Aliph ($k = 2^i$) periodically switches between Chain and PBFT, after the failure. This explains the throughput gaps in Aliph. Since k increases exponentially for every protocol switch, it stays in the backup protocol for an increasing period of time and thus its throughput increases.

Aliph ($k = 1$) has significantly lower throughput than Aliph ($k = 2^i$). When a replica fails, all we can observe are periodical bursts. However, the peak throughput (for the bursts) is nearly half of the throughput of PBFT when $k = 1$.

We configured BChain with a fairly high timeout value (100ms). In fact, BChain can use much smaller timeouts, since one re-chaining only takes about the same time as it takes for BChain to process a single request. While the signature-based, view-change like switching taken by Aliph introduces a significant time overhead.

The throughput of PBFT does not change in any obvious way after failure in-

jection, showing its stability during failure scenarios. Zyzzyva, on the other hand, in the presence of failures, uses its slower backup protocol which exhibits even lower throughput than PBFT.

We claim that even in presence of a Byzantine failure, the throughput of BChain-3 and BChain-5 would not change in a significant way, except that there might be two (instead of one) short periods where the throughput drops to zero. Note BChain-3 uses at most two re-chainings to handle a Byzantine faulty replica, while BChain-5 uses only one.

Performance Attack. We now show how to set up the timers for replicas in the chain as discussed in §4.2.4. Initially, there are no faulty replicas and we set the timers based on the average latency of the first 1000 requests. Fig. 4.6(d) illustrates the timer setup procedure for a correct replica p_i, where each bar represents the actual latency of a request, line 1 is the average latency δ_{1,p_i}, line 2 is the performance threshold timer Δ'_{1,p_i} used to deter performance attacks, and line 3 is the normal timer Δ_{1,p_i}. In our experiment, we set $\Delta'_{1,p_i} = 1.1\delta_{1,p_i}$ and $\Delta_{1,p_i} = 1.3\delta_{1,p_i}$. That is, we expect the performance reduction to be bounded to 10% of the actual latency during a performance attack by a dedicated adversary.

To evaluate the robustness against a timer-based performance attack, we ran 10 rounds of experiments using the 0/0 benchmark, each with a sequence of 10000 requests. We assume there are no faulty replicas initially and we use the first 1000 request to train the timers. For each experiment, starting from the 1001^{th} request, we let a replica mount a performance attack by intentionally delaying messages sent to its predecessor. To simulate different attacks, we simply let the faulty replica sleep for an "appropriate" period of time following different strategies. However, as expected our findings show that the actions of a faulty replica is very limited: it either needs to be very careful not to be accused, thus imposing only a marginal

performance reduction, or it will be suspected which will lead to a re-chaining and then a reconfiguration.

4.5.3 A BFT Network File System

This section describes our evaluation of a BFT-NFS service implemented using PBFT [18], Zyzzyva [69], and BChain-3, respectively. The BFT-NFS service exports a file system, which can then be mounted on a client machine. Upon receiving client requests, the replication library and the NFS daemon is called to reach agreement on the order in which to process client requests. Once processing is done, replies are sent to clients. The NFS daemon is implemented using a fixed-size memory-mapped file.

We use the Bonnie++ benchmark [30] to compare our three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. We first evaluate the performance on sequential input (including per-character and block file reading) and sequential output (including per-character and block file writing). Fig. 4.7(b) shows that the performance of sequential input for all three implementations only degrades the performance by less than 5% w.r.t. NFS-std. However, for the write operations, PBFT, Zyzzyva, and BChain-3, respectively, achieves in average of 35%, 20%, and 15% lower processing speed than NFD-std.

In addition, we also evaluate the Bonnie++ benchmark with the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that will appear random to the file system; (5) stat() random files; (6) delete the files in random order. We measure the average latency achieved by the clients while up to 20 clients run the benchmark concurrently. As shown in Table 4.2, the latency

achieved by BChain-3 is 1.10% lower than NFS-std, in contrast to BFS and Zyzzyva.

Table 4.2. NFS DirOps evaluation in fault-free cases.

BChain-3	Zyzzyva	BFS	NFS-std
$41.66s(1.10\%)$	$42.47s(2.99\%)$	$43.04s(4.27\%)$	$41.20s$

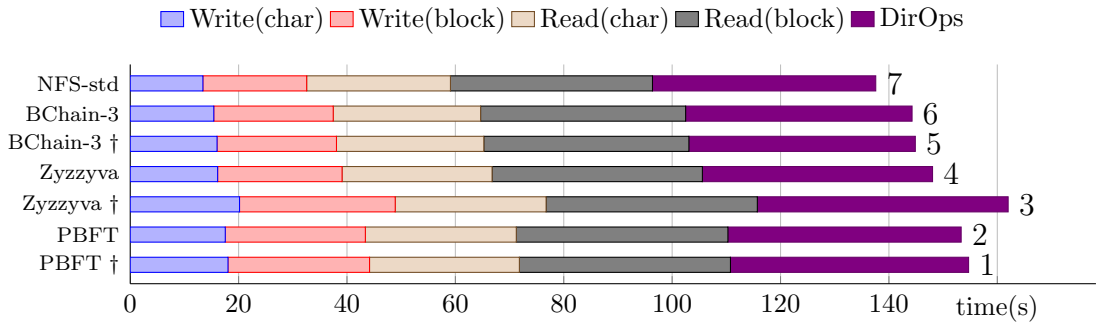


Figure 4.8. NFS Evaluation with the Bonnie++ benchmark. The † symbol marks experiments with failure.

Finally, we evaluate the performance using the Bonnie++ benchmark when a failure occurs at time zero, as detailed in Fig. 4.8. The bar chart also includes data points for the non-faulty case. The results shows that BChain can perform well even with failures, and is better than the other protocols for this benchmark.

4.6 Future Work

Chain replication is known to enjoy several benefits in performance, as shown in the protocol. As a Byzantine fault tolerant chain-replication, BChain is shown to achieve all the benefits of chain-replication while tolerating Byzantine failure well. However, it is also shown that BChain does not scale well for two reasons: 1) each message

travels through a long chain until agreement is reached, resulting in longer latency; 2) when there are failures, it takes longer to reconfigure in the re-chaining. For future work, there are several ways to further enhance BChain in wide area network. For instance, we can use multiple chains simultaneously to handle concurrent requests in a more efficient way. Another way is to divide a long chain into smaller sections of chains. In each small section of chain, failures are handled locally and eventually the whole chain can reach an agreement easily.

4.7 Conclusion

We have presented BChain, a new chain-based BFT protocol that outperforms prior protocols in fault-free cases and especially during failures. In the presence of failures, instead of switching to a slower backup BFT protocol, BChain leverages a novel technique—re-chaining—to efficiently detect and deal with the failures such that it can quickly recover its steady state performance. BChain does not rely on any trusted components or unproven assumptions.

Chapter 5

Byzantine Fault Tolerance from Intrusion Detection

The work presented in this chapter was first described in an earlier paper by Duan, et al. [41]. In this chapter, we present ByzID. We leverage two key technologies already widely deployed in cloud computing infrastructures: replicated state machines and intrusion detection systems.

First, we have designed a general framework for constructing Byzantine failure detectors based on an intrusion detection system. Based on such a failure detector, we have designed and built a practical Byzantine fault-tolerant protocol, which has costs comparable to crash-resilient protocols like Paxos. More importantly, our protocol is particularly robust against several key attacks such as flooding attacks, timing attacks, and fairness attacks, that are typically not handled well by Byzantine fault masking procedures.

5.1 Introduction

The availability and integrity of critical network services are often protected using two key technologies: a *replicated state machine* (RSM) and an *intrusion detection system* (IDS).

An RSM is used to increase the availability of a service through consistent replication of state and masking different types of failures. RSMs can be made to mask arbitrary failures, including compromises such as those introduced by malware. Such RSMs are referred to as Byzantine fault-tolerant (BFT). Despite significant progress in making BFT practical [18, 50], it has not been widely adopted, mainly because of the complexity of the techniques involved and high overheads. In addition, BFT is not a panacea, since there are a variety of attacks, such as various performance attacks that BFT does not handle well [5, 29]. Also, if too many servers are compromised then masking is not possible.

An IDS is a tool for (near) real-time monitoring of host and network devices to detect events that could indicate an ongoing attack. There are three types of intrusion detection: (*a*) *Anomaly-based* intrusion detection [35] looks for a statistical deviation from a known "safe" set of data. Most spam filters use anomaly detection. (*b*) *Misuse-based* intrusion detection [82] looks for a pre-defined set of signatures of known "bad" things. Most host and network-based intrusion detection systems and virus scanners are misuse detectors. (*c*) *Specification-based* intrusion detection systems [68] are the opposite of misuse detectors. They look for a pre-defined set of signatures of known "good" things.

In practice, BFT and IDSs are almost always used independently of each other. Additionally, the most commonly used fault-tolerance techniques typically only handle crash failures. For instance, Google uses Paxos-based RSMs in many core infras-

tructure services [17, 32]. As a result, only a handful of additional techniques are typically used to cope with other failures than crashes. However, those techniques are either ad hoc or are unable to handle attacks and arbitrary failures (e.g., software bugs). For attacks that are hard to mask (e.g., too many corrupted servers, simultaneous intrusions, and various performance attacks), IDSs are usually used. However, IDSs themselves suffer from deficiencies that limit their utility, including false positives that overly burden a human administrator who has to process intrusion alerts, and false negatives for when an ongoing attack is not detected. Also, IDSs themselves are not resilient to crashes.

In this chapter, we propose a unified approach that leverages intrusion detection to improve RSM resilience, rather than using each technique independently. We describe the design and implementation of a BFT protocol—ByzID—in which we use a lightweight specification-based IDS as a failure detection component to build a Byzantine-resilient RSM. ByzID distinguishes itself from previous BFT protocols in two respects: (1) Its efficiency is comparable to its crash failure counterpart. (2) It is robust against a wide range of failures, providing consistent performance even under various attacks such as flooding, timing, and fairness attacks. We note that ByzID does not protect against all possible attacks, only those that the IDS can help with. Underlying ByzID are several new design ideas:

Byzantine-resilient RSM. ByzID is a primary-based RSM protocol, adapted for combining with an IDS. In this protocol, a primary receives client requests and issues ordering commands to the other replicas (backups). All replicas process requests and they all reply to the client. In the event of a replica failure, a new replica runs a reconfiguration protocol to replace the failed one. The primary reconfiguration runs *in-band*, where other replicas wait until reconfiguration completes. Reconfiguration for other replicas runs *out-of-band*, where replicas continue to run the protocol

without waiting for the reconfiguration.

Monitoring instead of Ordering. Our protocol relies on a *trusted* specification-based IDS [68], to detect and suppress primary equivocation, enforce fairness, detect various other replica failures, and trigger replica reconfiguration. Our IDS is provided with a *specification* of our ByzID protocol, allowing the IDS to monitor the behavior of the replica. Note that, the way our protocol uses the IDS is so simple that the IDS could be implemented as a trivially small, timed state machine that can be embedded in a simple reference monitor, and can thus easily be built in hardware. However, for our proof of concept prototype we leverage the Bro IDS framework [92]. While some existing BFT protocols use *trusted components* [26, 63, 80, 110] to decide on the ordering client requests, our trusted IDS approach simply monitors and discards messages to enforce ordering.

Independent Trusted Components. In ByzID, each RSM replica is associated with a separate IDS component. However, even if an IDS experiences a crash, its RSM replica can continue to process requests. Hence, both liveness and safety can be retained as long as the RSM replicas themselves remain correct. For BFT protocols relying on trusted components, RSM replicas typically fail together with their trusted components.

Simple Rooted-Tree Structure. When deploying ByzID in a local area network (LAN), we organize the replicas in a simple rooted-tree structure, where the primary is the root and the backups are its direct siblings (leafs). Furthermore, backups are not connected with one another. With such a structure and together with the aid of IDSs we can avoid using cryptography to protect the links between the primary and the backups. This is because the IDS can enforce non-equivocation, identify the source and destination of messages, and prevent message injection. Moreover, a

backup only needs to send or receive messages from the primary, thus backups need not broadcast. Such a structure also helps to prevent flooding attacks from faulty replicas.

Our contributions can be summarized as follows:

- We have designed and implemented a general and efficient framework for constructing Byzantine failure detectors from a specification-based IDS.
- Relying on such failure detectors, our ByzID protocol uses only $2f + 1$ replicas to mask f failures. ByzID uses only *three* message delays from a client's request to receiving a reply, just one more than non-replicated client/server.
- We have conducted a performance evaluation of ByzID for both local and wide area network environments. For LANs, ByzID has comparable performance to Paxos [73] in terms of throughput, latency, and especially scalability. We also compare ByzID's performance with existing BFT protocols.
- We prove the correctness of ByzID under Byzantine failures, and discuss how ByzID withstands a variety of attacks. We also provide a performance analysis for a number of BFT protocols experiencing a failure.
- Finally, we use ByzID to implement an NFS service, and show that its performance overhead, with and without failure, is low, both compared to non-replicated NFS and other BFT implementations.

5.2 Conventions and Notations

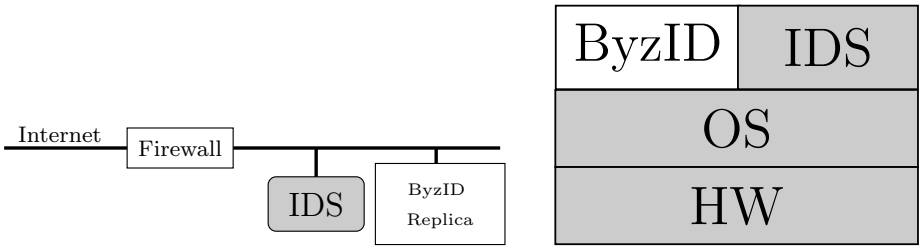
Replicas may be connected in a complete graph or an incomplete graph network. However, for wide area deployments, only a complete graph network makes sense. We further assume that adversaries are unable to inject messages on the links between the

replicas. This is reasonable when all replicas are monitored by IDSs and they reside in the same administrative domain. We assume that IDSs are trusted components, but that they may fail by crashing.

Let $\langle X \rangle_{i,j}$ denote an authentication certificate for X, sent from i to j. Such certificates can be implemented using MACs or signatures. We use MACs for authentication unless otherwise stated. Let $[Z]$ denote an unauthenticated message for Z, where no MACs or signatures are appended.

5.3 Byzantine Failure Detector from Specification-Based Intrusion Detection

Specification-based intrusion detection is a technique used to describe the *desirable* behavior of a system. Therefore, by definition, any sequence of operations outside of the specifications is considered to be a violation. As illustrated in Fig. 5.1(a), we use an IDS to monitor the behavior of the replication protocol $\mathcal{P}$, executed by a replica. The IDS receives messages sent to/by $\mathcal{P}$ by monitoring packets over the network. Thus, the IDS cannot modify any messages, only detect misbehavior.



(a) The IDS interface at a replica. (b) IDS implementation.

Figure 5.1. The IDS/ByzID architecture. (Components shown on gray background are considered to be trusted.)

5.3.1 Byzantine Failure Detector Specifications

As depicted in Fig. 5.1(a), each replica is equipped with a local IDS agent, which monitors the replica's incoming and outgoing messages. In our protocol, the IDS captures the network packets of the protocol through port number and analyze them according to the specification. Thus, the IDS acts as a distributed oracle and triggers alerts if the replica does not follow the specifications of the prescribed protocol $\mathcal{P}$. In case of an alert, the detected replica should be recovered, or removed through a reconfiguration procedure. Meanwhile, the messages sent by the faulty replica should be blocked. This is accomplished by the IDS agent inserting a packet filter into the underlying OS kernel.

The trusted IDS and the untrusted protocol $\mathcal{P}$ can be separated in various ways [26], e.g. using virtual machines or the IDS can be implemented in trusted hardware. In our prototype however, they simply execute as separate processes under the same OS, as shown in Fig. 5.1(b).

The primary orders client requests by maintaining a queue, as shown in Fig. 5.2. To ensure that the primary orders messages *correctly*, we define a set of IDS *specifications* for Byzantine failure detectors. Such detectors can be used together with most existing primary-based BFT protocols. Below we summarize the specifications for our Byzantine failure detector.

- *Consistency.* The primary sends consistent messages to the other replicas.
- *Total Ordering.* The primary sends totally ordered requests to the replicas.
- *Fairness.* The primary orders requests in FIFO order.
- *Timely Action.* The primary orders client requests in a timely manner.

(1) The *consistency* rule prevents the primary from sending "inconsistent" *order* messages to the other replicas without being detected. The *order* message is the message sent by the primary to initialize a round of agreement protocol, such as the *pre-prepare* message in PBFT [18]. More specifically, the primary must send the *same* order message to the remaining $n-1$ replicas. To this end, the IDS can monitor the number of matching messages with the same sequence number. In case of inconsistencies, an alert is raised and the inconsistent messages are blocked.
(2) The *total ordering* rule prevents primary from introducing gaps in the message ordering. The sequence number in the order messages sent by the primary must be incremented by exactly one. Namely, the primary sends an order message with sequence number N only after it has sent an order message for $N-1$. In the event that the primary sends out an "out-of-order" message, an alert is raised by the IDS.
(3) We argue that the conventional fairness definition is insufficient for many fairness-critical applications, such as registration systems for popular events, e.g. concerts or developer conferences with limited capacity. Thus, we define *perfect fairness* such that the RSMs must execute the client requests in FIFO order. As shown in Fig. 5.2, the IDS monitors client requests received by the primary and the order messages sent by the primary. With this, the IDS can verify that the primary follows the correct client ordering observed by the IDS. This is typically hard to achieve for common BFT protocols.

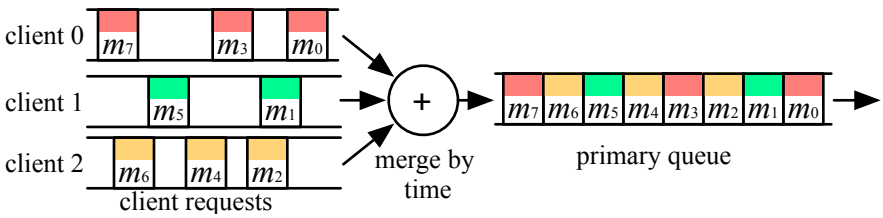


Figure 5.2. Queue of client requests.

(4) The *timely action* rule detects crash-stop and a "slow" primary. The IDS simply starts a timer for the first request in the queue. If the primary sends a valid order message before the timer expires, the IDS cancels the timer. Otherwise, the IDS raises an alert. The timer can be a fixed value or adjusted adaptively, e.g. based on input from an anomaly-based IDS.

Traditionally, BFT protocols have used arbitrarily-chosen timeouts as one means for detecting faulty actors with excessive latencies. But those timeouts may not reflect reality. As such, *anomaly detection* is another intrusion detection technique that can help address this issue. Because anomaly detection is typically based on a statistical deviation from normal behavior, we use anomaly detection to baseline the latencies between actors at the beginning and then look for deviations from the baseline outside a particular bound. The baseline can be updated over time to take benign changes in system and network performance into account. This is typically done by weighting recent baselines less than older baselines so that an adversary cannot "game" the system as easily.

5.3.2 The IDS Algorithm

Our IDS specifications are detailed in Algorithm 9. The IDS maintains the following values: a queue of client requests $\mathcal{Q}$, current [ORDER] message M, current sequence number N, a boolean array $\mathcal{C}[n]$ used to ensure that an [ORDER] message is sent to all replicas, and a timer Δ for the timely action rule.

As depicted in Fig. 5.2, the primary stores the client requests in a total order [71] according to the time of receiving them. The IDS also keeps the same queue of requests and monitors the [ORDER] messages sent by the primary. As shown in Algorithm 9, when the IDS observes a new [ORDER] message, it verifies the correct-

Algorithm 9 The IDS Specifications

1: **Initialization:**
2: n {Number of replicas}
3: $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$ {Replica set; p_0 is the primary}
4: $\mathcal{Q}$ {Queue of client requests}
5: M {Current [ORDER] msg being tracked}
6: $N \leftarrow 0$ {Current sequence number}
7: $\mathcal{C} \leftarrow \emptyset$ {Array: $\mathcal{C}[i] = 1$ if seen [ORDER] msgs to p_i}
8: Δ {Timer; initialized by anomaly-based IDS}
9: **upon** $m = \langle \text{REQUEST}, o, T, c \rangle_{c,p_0}$
10: **if** $|\mathcal{Q}| = 0$ **then**
11: starttimer(Δ) {For timely action}
12: $\mathcal{Q}$.add(m) {Add client c's msg to $\mathcal{Q}$}
13: **upon** $M' = [\text{ORDER}, N', m, v, c]_{p_0,p_i}$
14: **if** $N' = N + 1 \wedge |\mathcal{C}| = 0 \wedge m = \mathcal{Q}$.front() **then**
15: $N \leftarrow N'$ {New current sequence number}
16: $M \leftarrow M'$ {New current [ORDER] msg}
17: $\mathcal{C}[i] \leftarrow 1$ {Have seen [ORDER] msg to p_i}
18: **else if** $|\mathcal{C}| > 0 \wedge \mathcal{C}[i] = 0 \wedge M = M'$ **then**
19: $\mathcal{C}[i] \leftarrow 1$ {Have seen [ORDER] msg to p_i}
20: **if** $|\mathcal{C}| = n - 1$ **then** {Seen enough [ORDER] msgs?}
21: $\mathcal{C} \leftarrow \emptyset$ {Reset array}
22: $\mathcal{Q}$.remove() {Remove msg from $\mathcal{Q}$}
23: canceltimer(Δ)
24: **if** $|\mathcal{Q}| > 0$ **then**
25: starttimer(Δ) {For timely action}
26: **else**
27: **alert** {Violation of first three specifications}
28: **upon** timeout(Δ)
29: **alert** {Violation of timely action specification}

ness of *total ordering*, *consistency*, and *fairness*. Total ordering is violated, if the sequence number in the [ORDER] message is different from $N + 1$. Consistency is violated if the primary does not send to the other $n - 1$ replicas. Fairness is violated, if the request in the [ORDER] message is not equal to the first request in the IDS's queue.

To monitor the *timely action*, the IDS starts a timer in two cases:

a) The queue is empty and the IDS observes a new client request, as shown in Lines $10 - 11$; b) The primary has already sent an [ORDER] message to the other replicas and the queue is not empty, as shown in Lines $24 - 25$. Finally, an alert is also raised if the primary does not send the [ORDER] message to the other replicas before the timer expires.

5.4 The ByzID Protocol

ByzID has three subprotocols: *ordering*, *checkpointing*, and *replica reconfiguration*. The ordering protocol is used during normal case operation to order client requests. The checkpoint protocol bounds the growth of message logs and reduces the cost of reconfiguration. The reconfiguration protocol reconfigures the replica when its associated IDS generates an alert.

We distinguish between *normal* and *fault-free* cases as follows: we define the normal case as the primary being correct, while the other replicas might be faulty. Note that, the normal case definition is less restrictive than the fault-free case, where all replicas must be correct.

BFT protocols that rely on trusted components, e.g., A2M [26], TrInc [80], and CheapBFT [63], can use $2f + 1$ replicas to tolerate f failures and use one less round of communication than PBFT. While these other protocols use trusted hardware

directly to order clients requests, we achieve the same goal using a software IDS that conducts monitoring and filtering. This feature makes it possible for the system to achieve safety even if all IDSs are faulty. We use the Byzantine failure detector for the primary to ensure that the requests are delivered consistently, in a total order, and in a timely and fair manner. With the aid of the IDS, it is possible to reduce communication rounds further for the normal case. Ideally, we seek a protocol comparable to the fault-free protocol of Zyzzyva [69] (and minZyzzyva [110]).

To this end, we follow a primary-backup scheme [4, 15], where in each configuration, one replica is designated as the primary and the rest are backups. The correct primary sends order messages to the backups, and *all* correct replicas execute the requests and send replies to clients.

However, two technical problems remain. First, since our protocol lacks the regular commit round, we need the primary to reliably send messages through fair-loss links between the potentially faulty primary and the backups. Second, the Byzantine failure detector does not enforce authentication between the primary and the backups.

To address the first problem, we require backups to send [Ack] messages to the primary. And with the aid of the IDSs, we also provide a mechanism to handle message retransmissions. For the second problem, we distinguish between the core ByzID protocol for LANs, and ByzID-W for wide area networks (WANs). ByzID exploits the non-equivocation property provided by the IDS, and its ability to track the source and destination of messages. This allows ByzID to operate without cryptography on the links connecting the replicas.

To cope with the possibility of message injections in WANs, the ByzID-W primary instead uses authenticated order messages. These must be verified by both the backup replicas and the IDS. See §5.4.2 for further details.

5.4.1 The ByzID Protocol

The ordering protocol. Fig. 5.3 and Fig. 5.4 depict normal case operation. Below we describe the steps involved in the ordering protocol.

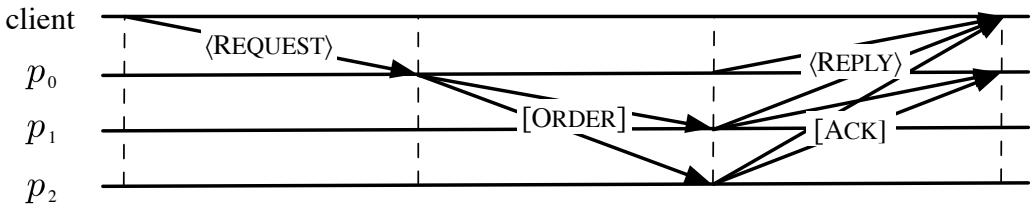


Figure 5.3. The ByzID protocol message flow.

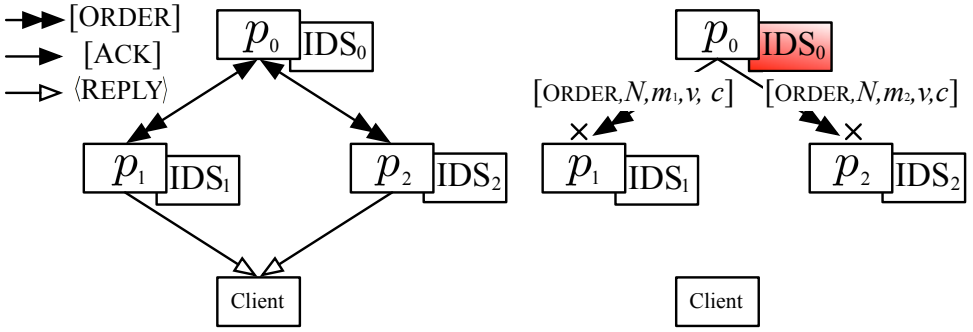


Figure 5.4. ByzID equipped with IDSs. The primary assigns sequence number to the request and sends [ORDER] message to the replicas. If the messages to different replicas are not consistent, the messages are blocked by the IDS equipped at the primary.

Step 1: Client sends a request to the primary. A client c sends the primary p_0 a request message $\langle \text{REQUEST}, o, T, c \rangle_{c,p_0}$, where o is the requested operation, and T is the timestamp.

Step 2: Primary assigns a sequence number to the request and sends an [ORDER] *message to the backups.* When the primary receives a request from the client, it assigns a sequence number N to the request and sends an $[\text{ORDER}, N, m, v, c]$ message

to the backups, where m is the request from the client, v is the configuration number, and c is the identity of the client.

IDS details (at primary): The IDS verifies the specifications mentioned in §5.3. Each time the specifications are violated, the IDS blocks the corresponding messages and generates an alert such that the primary will be reconfigured.

Step 3: Replica receives an [ORDER] *message, replies with an* [ACK] *message to the primary, executes the request, and sends a* ⟨REPLY⟩ *to the client.* When replica p_i receives an $[\text{ORDER}, N, m, v, c]$ message, it sends the primary an $[\text{ACK}, N, D(m), v, c]$ message with the same N, m, v, and c as in the [ORDER] message. A backup p_i accepts the [ORDER] message if the request m is valid, its current configuration is v, and $N = N' + 1$, where N' is the sequence number of its last accepted request. If the replica p_i accepts the [ORDER] message, it executes operation o in m and sends the client a reply message $\langle \text{REPLY}, c, r, T \rangle_{p_i,c}$, where r is the execution result of operation o, and T is the timestamp of request m. If p_i receives an [ORDER] message with sequence number $N > N' + 1$, it stores the message in its log and waits for messages with sequence numbers between N and N'. It executes the request with sequence number N after it executes requests with sequence numbers between N' and N.

IDS details (at backups): The IDS at a backup p_i starts a timer when it observes an [ORDER] message. If p_i does not send an [ACK] message in time, the IDS generates an alert.

Step 4: Primary receives [ACK] *messages from all backups and completes the request. Otherwise, it retransmits the* [ORDER] *message.* When the primary receives an $[\text{ACK}, N, D(m), v, c]$ message, it accepts the message if the fields N, m, v, and c match those in the corresponding [ORDER] message. If the primary collects [ACK] messages from all the backups, it completes the request.

Our protocol is also compatible with common optimizations such as batching and pipelining. For pipelining, the primary can simply order a new request before the previous one is completed. However, to prevent the primary from sending [ORDER] messages too rapidly, we limit the number of outstanding [ORDER] messages to a threshold τ. The primary sends an [ORDER] message with sequence number N only if it completes requests with sequence numbers smaller than $N - \tau$.

The primary keeps track of the sequence number of the last completed request, N_1, and the sequence number of its most recently sent [ORDER] message, N_2. Obviously, we have that $N_2 \geq N_1$. When the primary sends an [ORDER] message for sequence number N_1, it starts a timer Δ_1. If the primary does not receive [ACK] messages from all the backups before the timer expires, it retransmits the [ORDER] message to the backups from which [ACK] messages are missing. Otherwise, the primary cancels the timer and starts a new timer for the next request, if any.

An example is illustrated in Fig. 5.5, where the primary sends [ORDER] messages for requests with sequence numbers from N_1 to N_2. At t_1, the primary sends an [ORDER] message for N_1, and starts a timer Δ_1. At t_3, it has collected [ACK] messages from all backups and cancels the timer. Since the primary has already completed the request with sequence number $N_1 + 1$ at t_2, it just starts a new timer for a request with $N_1 + 2$ at t_3.

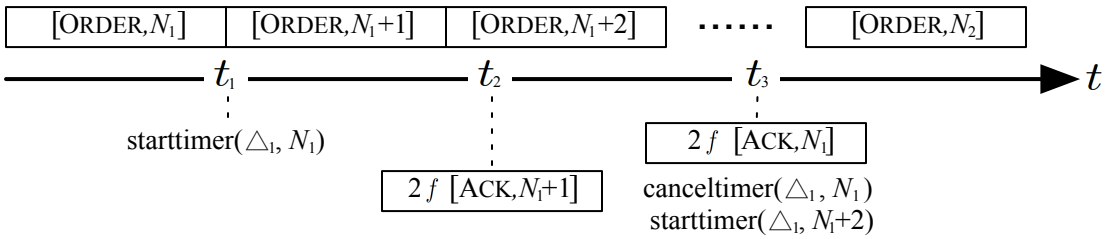


Figure 5.5. An example for Step 4.

IDS details (at primary): An alert is raised if the primary: (1) does not retransmit the [ORDER] message in time, or (2) it "retransmits" an inconsistent [ORDER] message. To accomplish these detections, also the IDS starts a timer corresponding to the primary's Δ_1 timer. If the primary receives enough [ACK] messages before Δ_1 expires, the IDS cancels the timer. However, if the primary does not receive [ACK] messages from all backups before Δ_1 expires, the IDS starts another timer, Δ_2. If this timer expires, before the IDS observes a retransmitted [ORDER] message, an alert is raised. Finally, the IDS keeps track of the sequence number of the last [ORDER] message sent by the primary, N_3. Each time the primary sends an [ORDER] message with sequence number smaller than N_3, it is considered a retransmission. The IDS checks if a retransmitted [ORDER] message matches an [ORDER] message in its log. If there is no match, an alert is raised.

Step 5: Client collects $f + 1$ matching $\langle$REPLY$\rangle$ messages to complete the request. The client completes a request when it receives $f + 1$ matching reply messages.

Checkpointing. ByzID replicas store messages in their logs, which are truncated by the checkpoint protocol. Each replica maintains a stable checkpoint that captures both the protocol state and application level state. In addition, a replica also keeps some tentative checkpoints. A tentative checkpoint at a replica is proven stable only if all its previous checkpoints are stable and it collects certain message(s) in the checkpoint protocol to prove that the current state is correct.

We now briefly describe the ByzID checkpoint protocol. Every replica constructs a tentative checkpoint at regular intervals, e.g., every 128 requests. A backup replica p_i sends a [CHECKPOINT, N, d, i] message to the primary, where N is the sequence number of last request whose execution is reflected in the checkpoint and d is the digest of the state. The primary considers a checkpoint to be stable when it has

collected f matching [CHECKPOINT] messages from different backups, and then sends a [STABLECHECKPOINT, N, d] message to the backups. The primary and f backups prove that the checkpoint is stable. When a backup receives a [STABLECHECKPOINT], it considers the checkpoint stable. A replica can truncate its log by discarding messages with sequence numbers lower than N.

IDS details: The IDS needs to audit the [CHECKPOINT] messages from the backups. When it has seen $f+1$ matching [CHECKPOINT] messages from the backups, it starts a timer. If the primary does not send the corresponding [STABLECHECKPOINT] message to all the backups before the timer expires, an alert is generated. IDS can also run a checkpoint protocol to prevent its own log from growing without bound.

However, it delays discarding its stable checkpoints to help replica reconfiguration, as detailed in the following.

Replica reconfiguration. Reconfiguration is a technique for stopping the current RSM and restarting it with a new set of replicas [77]. We now describe ByzID's reconfiguration scheme. Recall that when any specifications of a replica are violated, the IDS generates an alert and triggers reconfiguration. If the IDS at the primary generates an alert, all the replicas are notified and stop accepting messages. The primary reconfiguration procedure operates *in-band* where all backups wait until the procedure completes. The backup reconfiguration procedure operates *out-of-band*. Namely, only the primary is notified with a backup replica IDS alert; the remaining replicas continue to run the protocol without having to wait for the procedure to complete. Assume in a configuration v the set of replicas is $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$. We assume that after a reconfiguration, $p_i \in \Pi$ is replaced by $p_j \notin \Pi$. If p_i is the primary, the configuration number becomes $v + 1$ after reconfiguration. Clearly, replica p_j is also equipped with an IDS component.

Primary reconfiguration. To initialize primary reconfiguration, a new primary p_j sends a [ReconRequest] message to *all* replicas in Π.[1] To respond, each replica p_k sends p_j a signed $\langle \text{Reconfigure}, v+1, N, \mathcal{C}, \mathcal{S} \rangle_{p_k}$ message, where N is the sequence number of the last stable checkpoint, $\mathcal{C}$ is the last stable checkpoint, and $\mathcal{S}$ is a set of valid [Order] messages accepted by p_k with sequence numbers greater than N.

When p_j collects at least $f+1$ matching authenticated $\langle \text{Reconfigure} \rangle$ messages, it updates its state using the state snapshot in $\mathcal{C}$ and sends a $[\text{NewConfig}, v+1, \mathcal{V}, \mathcal{O}]$ to $\Pi \setminus p_i$, where $\mathcal{V}$ is a set of $f+1$ $\langle \text{Reconfigure} \rangle$ messages and $\mathcal{O}$ is a set of [Order] messages computed as follows: first, the primary p_j obtains the sequence number min of the last stable checkpoint in $\mathcal{C}$ and the largest sequence number max of the [Order] message that has been accepted by at least one replica, which is obtained from $\mathcal{S}$.

The primary then creates an [Order] message for each sequence number N between min and max. There are two cases: (1) If there is at least one request in the $\mathcal{S}$ field with sequence number N, p_j generates an [Order] message for this request; (2) If there is no such request in $\mathcal{S}$, p_j creates an [Order] message with a Null request. A backup accepts a [NewConfig] message if the set of $\langle \text{Reconfigure} \rangle$ messages in $\mathcal{V}$ are valid and $\mathcal{O}$ is correct. The correctness of $\mathcal{O}$ can be verified through a similar computation as the one used by the primary to create $\mathcal{O}$. It then enters configuration $v+1$.

Backup reconfiguration. A new backup replica p_j sends a message [ReconRequest] to the primary. The primary then responds a message $[\text{Reconfigure}, v+1, N, \mathcal{C}, \mathcal{S}]$ to p_j, where N is the sequence number of the primary's last stable checkpoint, $\mathcal{C}$ is its last stable checkpoint, and $\mathcal{S}$ is a set of valid [Order] messages sent by the primary

[1]Note that p_j should also send the message to the current primary, because it might still be correct.

with sequence number greater than its last stable checkpoint. When p_j receives the [RECONFIGURE] message, it updates its state by the state snapshot in $\mathcal{C}$, and then processes the [ORDER] messages in $\mathcal{S}$.

IDS details: The IDS coupled with p_j obtains its own state from the IDS of replica p_i.

During primary reconfiguration, the IDS at new primary p_j monitors all the $\langle$RECONFIGURE$\rangle$ messages from all the replicas in Π and checks if they match its own IDS log. If the checkpoint is not valid or the [ORDER] messages in $\mathcal{S}$ are not the same as the messages sent by p_i, the IDS blocks the $\langle$RECONFIGURE$\rangle$ message. Clearly it is with the aid of IDS that primary reconfiguration becomes simpler.

During the backup reconfiguration, the IDS at the primary checks if the primary sends the backup a [RECONFIGURE] message with the same $\mathcal{C}$ and $\mathcal{S}$ as in its IDS log. This ensures that replica p_j receives consistent state as other replicas.

Correctness. We now prove that ByzID is both safe and live.

Theorem 1 (Safety). If no more than f replicas are faulty, non-faulty replicas agree on a total order on client requests.

Proof: We first show that ByzID is safe within a configuration and then show that the ordering and replica reconfiguration protocols together ensure safety across configurations.

Within a configuration. We prove that if a request m commits at a correct replica p_i and a request m' commits at a correct replica p_j with the same sequence number N within a configuration, it holds that m equals m'. We distinguish three cases: (1) either p_i or p_j is the primary; (2) neither p_i nor p_j is the primary, and neither has been reconfigured; (3) neither p_i nor p_j is the primary, and at least one of the two replicas has been reconfigured. We briefly prove the (most involved) case (3).

During a backup reconfiguration, its state can be recovered by communicating with the primary with the aid of the IDS. Thereafter, the new reconfigured replica is indistinguishable from the correct replica without having been reconfigured. If m with sequence number N commits at a correct replica p_i, it holds that p_i receives an [ORDER] message with m and N from the primary (either due to the ordering or backup reconfiguration protocols), since we assume there are no channel injections. Similarly, p_j receives an [ORDER] message with m' and N from the primary. Therefore, it must be that $m = m'$, since otherwise it violates the consistency specification enforced by the IDS. The total order thus follows from the fact that that the requests commit at the replicas in sequence-number order.

Across configurations. We prove that if m with sequence number N is executed by a correct replica p_i in configuration v and m' with sequence number N is executed by a correct replica p_j in configuration v', it holds that m equals m'. We assume w.l.o.g. that $v < v'$. Recall that if a backup is reconfigured, the state of the new replica is consistent with other backups. Thus, we do not bother differentiating reconfigured replicas from correct ones and focus on the case where p_i and p_j are both backups.

The proof proceeds as follows. If m with sequence number N is executed by p_i in configuration v, the primary must have sent consistent [ORDER] messages for m to all the backups. On the other hand, if m' with sequence number N is executed by p_j in configuration v', the primary in v' sends consistent [ORDER] messages for m' to all the backups. This implies that the primary in v' receives $\langle$RECONFIGURE$\rangle$ messages from at least $f + 1$ replicas with m' and N, at least one of which is correct. Inductively, we can prove that there must exist an intermediate configuration v_1 where the corresponding primary sent an [ORDER] message with m and N and an [ORDER] message with m' and N. Due to the consistency specification enforced by the IDS, it holds that m equals m'. The total order of client requests thus follows

from the fact that requests are executed in sequence-number order. ■

Theorem 2 (Liveness). If no more than f replicas are faulty, then if a non-faulty replica receives an request from a correct client, the request will eventually be executed by all non-faulty replicas. Clients eventually receive replies to their requests.

Proof: We begin by showing that if a correct replica accepts an [ORDER] message with request m and N, all the correct replicas eventually accept the same [ORDER] message.

There are two types of timers used for IDSs: (1) the timers to monitor the timely actions for the replicas' local operations, and (2) the timer in the primary IDS to wait for the [ACK] message. The first type of timers are initialized and tuned by the anomaly-based IDS. For the [ACK] timer, the IDS at the primary can double the timeouts when less than $f+1$ replicas send the [ACK] messages on time. Alternatively, the primary retransmits the [ORDER] message but starts a timer with the same value. If the retransmission occurs too frequently, the timer can be doubled.

We now show that if a correct replica p_i accepts an [ORDER] message with request m and N, all the correct replicas accept the same [ORDER] message. According to the protocol and the consistency rule, if p_i receives an [ORDER] message with m and N, the primary sends the same [ORDER] message to all backups. The primary completes the request when it collects $n-1$ matching [ACK] messages. If a faulty backup does not send the [ACK] message, the IDS raises an alert and the faulty replica is reconfigured. The [ORDER] message may be dropped by the fair-loss channel, in which case the primary will not receive the [ACK] message on time. The primary retransmits the [ORDER] messages until the backups receive it. If the primary does not do so, it will be detected by the IDS and be reconfigured. Then the new primary

will send (and probably need to retransmit) the [ORDER] messages until the backups receive it. Therefore, all correct replicas will receive the [ORDER] message eventually. The total ordering specification is also vital to achieve liveness. If the specification is not enforced, then according to our protocol, backups will have to wait for the [ORDER] messages with incremental sequence numbers to execute. Since they are at least $f+1$ correct replicas, the client always receives a majority of f matching replies from the replicas, as long as the correct replicas reach an agreement. If it does not receive enough replies on time, it simply retransmits the request and doubles its own timer. ■

5.4.2 The ByzID-W Protocol

When deploying ByzID in a WAN environment, several adjustments to the core protocol are needed. First, there must be complete graph network between the replicas. Second, since the IDS cannot be relied upon to prevent message injection on the WAN links, we now use authenticated links between the replicas. That is, order messages are authenticated using *deterministic* signatures, allowing the IDS to efficiently support retransmissions of previously signed order messages.

5.5 ByzID Implementation with Bro

As a proof of concept, we have implemented our Byzantine failure detector for ByzID using the Bro [92] specification-based IDS. Bro detects intrusions by hooking into the kernel using `libpcap` [86], parsing network traffic to extract semantics, and then executing event analyzers. To support ByzID, we have adapted Bro as shown in Fig. 5.6. First, we have built a new ByzID parser to process messages and generate

ByzID-specific events. These events are then delivered to their event handler, based on their type. The IDS specifications for ByzID is implemented as scripts written in the Bro language. The policy interpreter executes the scripts to produce real-time notification of analysis results, including alerts describing violation of BFT protocol specifications.

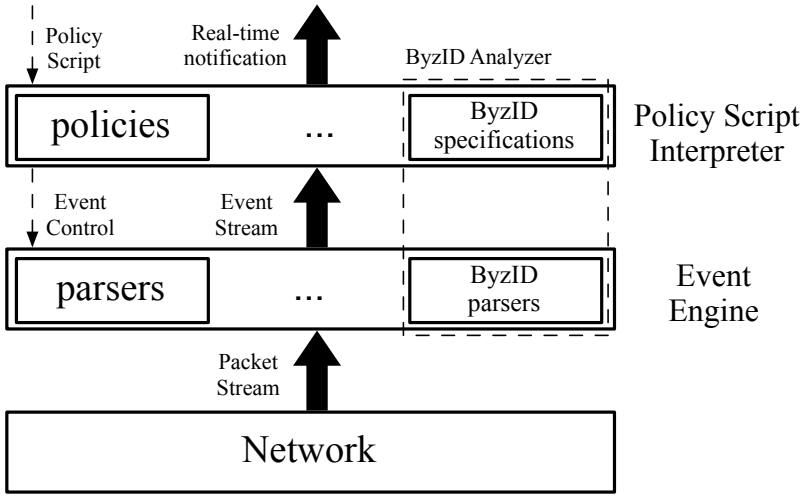


Figure 5.6. ByzID analyzer based on Bro.

ByzID parser. The network packet parser decodes byte streams into meaningful data fields. We use *binpac* [91], a high-level language for describing protocol parsers to automatically translate the network packets into a C++ representation, which can be used by both Bro and ByzID. We represent the syntax of ByzID messages by *binpac* scripts. During parsing, the parser first extracts the message tag, sequence number, and configuration number. The messages unrelated to the specifications are filtered during parsing; other messages are delivered to their corresponding event handler.

Event handler. Event handlers analyze network events generated by the ByzID parser. The event handler provides an interface between the ByzID parser and the

policy script interpreter. Each message type is associated with a separate event handler, and only messages with the appropriate tags are delivered to that handler. The events are then passed to the policy script interpreter to validate that the events do not violate the specifications.

ByzID specifications. The policy script contains the specifications of the ByzID protocol. Once event streams are generated by the event handler, it performs the inter-packet validation. The policy script interpreter maintains state from the parsed network packets, from which the incoming packets are further correlated and analyzed. Messages that violate the specifications are blocked and an alert is raised.

5.6 Performance Evaluation

In this section we evaluate the performance of ByzID by comparing it with three well-known BFT protocols—PBFT [18], Zyzzyva [69], Aliph [50], and an implementation of the crash fault tolerant protocol—Paxos [73]. The main conclusion that we can draw from our evaluation is that ByzID's performance is slightly worse that Paxos due to the overheads of the IDS and cryptographic operations. Considering the similarity in message flow between ByzID and Paxos, this is unsurprising. However, ByzID's performance is generally better than the other BFT protocols in our comparison.

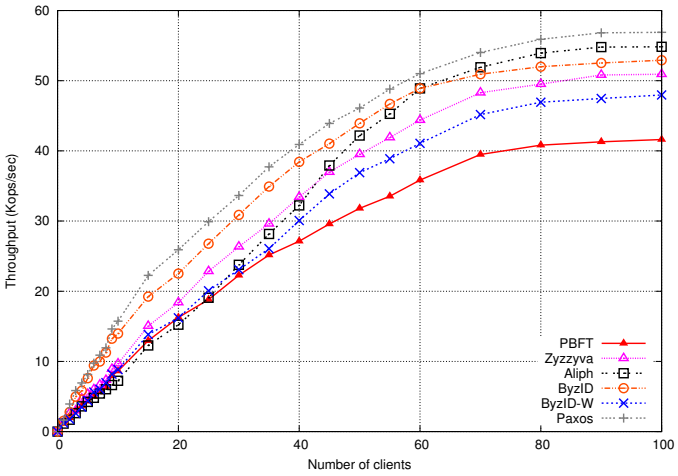
We do not compare ByzID with other BFT protocols that depend on trusted hardware, such as A2M [26], TrInc [80], and MinBFT [110], since we do not have access to the relevant hardware platforms. However, based on published performance data for these protocols, they generally do not offer higher throughput and lower latency than Aliph [63, 110].[2] We note that, the IDS component of ByzID could be implemented efficiently in trusted hardware as well.

[2]We note that A2M and TrInc must use signatures due to the impossibility result of [27].

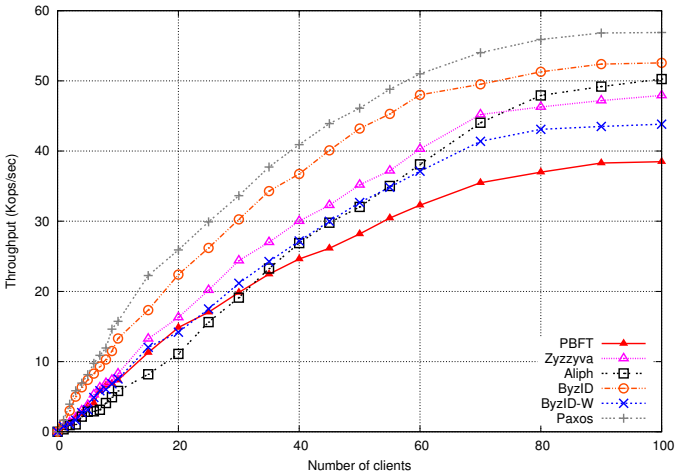
We evaluated throughput, latency, and scalability using the x/y micro-benchmarks by Castro and Liskov [18]. In these benchmarks, clients send x kB requests and receive y kB replies. Clients issue requests in a *closed-loop*, i.e., a client issues a new request only after having received the reply to its previous request. All protocols in our comparison implement batching of concurrent requests to reduce cryptographic and communication overheads. All experiments were carried out on Deterlab, utilizing a cluster of up to 56 identical machines. Each machine is equipped with a 3 GHz Xeon processor and 2 GB of RAM. They run Linux 2.6.12 and are connected through a 100 Mbps switched LAN.

Throughput. We first examined the throughput of both ByzID and ByzID-W under contention and compared them with PBFT, Zyzzyva, Aliph, and Paxos. Fig. 5.7 shows the throughput for the 0/0 benchmark when $f = 1$ and $f = 3$, as the number of clients varies. Our results show that ByzID outperforms other BFT protocols in most cases and is only marginally slower than Paxos. As observed in Fig. 5.7(a), ByzID consistently outperforms Zyzzyva, which achieves better performance than ByzID-W and PBFT. Since ByzID-W uses signatures, it achieves lower throughput than Zyzzyva. The reason ByzID-W has better performance than PBFT is due to the reduction of communication rounds. Aliph outperforms Zyzzyva and ByzID when the number of clients is big enough, mainly because it exploits the pipelined execution of client requests. But as shown in Fig. 5.7(b), ByzID consistently outperforms other BFT protocols when $f = 3$. For both $f = 1$ and $f = 3$, ByzID achieves an average throughput degradation of 5% with respect to Paxos. This overhead is mainly due to the cryptographic operations and IDS analysis. Similar results are observed in other benchmarks.

Latency. We have also compared the latency of the protocols without contention



(a) Throughput with $f = 1$; $n = 3$ replicas.



(b) Throughput with $f = 3$; $n = 7$ replicas.

Figure 5.7. Throughput for the 0/0 benchmark as the number of clients varies. This and subsequent graphs are best viewed in color.

where a *single* client issues requests in a close-loop. The results for the 0/0, 0/4, 4/0, and 4/4 benchmarks with $f = 1$ are depicted in Fig. 5.8. We observe that ByzID outperforms other protocols except Paxos. However, the difference between ByzID and Paxos is less than 0.1 ms. The reason ByzID has generally low latency is that

Table 5.1. Throughput improvement of ByzID over other BFT protocols. Values in (red) represent negative improvement.

Clients	Protocol	$f = 1$	$f = 2$	$f = 3$	$f = 4$	$f = 5$
25	PBFT	42.37%	45.71%	46.80%	49.14%	51.37%
25	Zyzzyva	17.19%	19.49%	25.49%	26.07%	27.72%
25	Aliph	40.42%	47.84%	67.56%	73.46%	76.98%
peak	PBFT	27.15%	32.57%	36.59%	41.82%	43.90%
peak	Zyzzyva	3.92%	8.43%	9.68%	12.25%	11.08%
peak	Aliph	(3.48%)	(1.24%)	4.57%	7.71%	8.92%

ByzID only requires three one-way message latencies in the fault-free case.

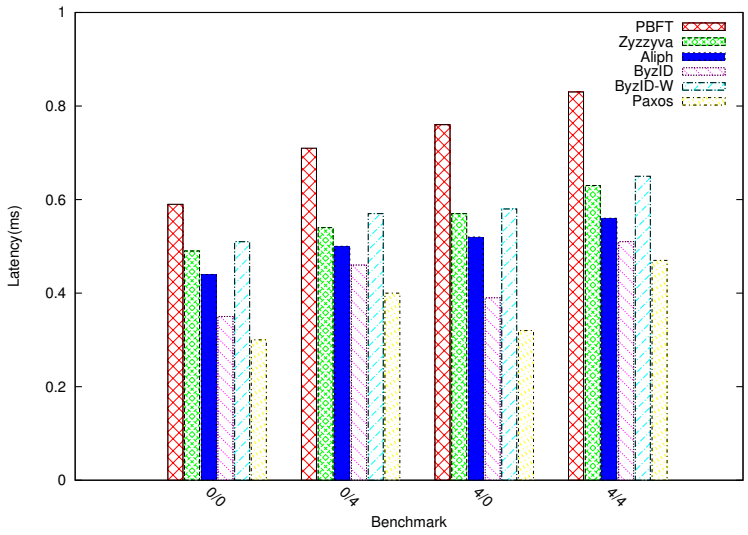


Figure 5.8. Latency for the $0/0, 0/4, 4/0$, and $4/4$ benchmarks.

Scalability. To understand the scalability properties of ByzID, we increase f for all protocols and compare their throughput. All experiments are carried out using

Table 5.2. Throughput degradation when f increases.

Clients	Protocol	$f = 2$	$f = 3$	$f = 4$	$f = 5$
25	PBFT	3.82%	9.40%	10.20%	15.04%
25	Zyzzyva	3.45%	8.66%	12.50%	16.80%
25	Aliph	6.50%	18.30%	28.00%	35.60%
25	ByzID	1.56%	2.20%	5.93%	9.67%
peak	PBFT	4.25%	7.54%	13.88%	17.85%
peak	Zyzzyva	4.32%	5.89%	11.07%	13.02%
peak	Aliph	4.84%	8.33%	13.93%	17.61%
peak	ByzID	1.70%	2.80%	3.94%	7.02%

the 0/0 benchmark. Table 5.1 compares the throughput of ByzID with three other BFT protocols, and Table 5.2 shows the throughput degradation for all four BFT protocols as f increases. We observe in Table 5.1 that the throughput improvement for ByzID over the other BFT protocols consistently increases as f grows. Table 5.2 shows that ByzID's own throughput has the lowest degradation rate among all four BFT protocols. For instance, ByzID's peak throughput is only reduced by 7.02% as f increases to 5 (i.e., when $n=11$). These results clearly show that ByzID has much better scaling properties than the other BFT protocols.

5.7 Failures, Attacks, and Defenses

The fact that a BFT protocol is live does not mean that the protocol is efficient. It is therefore important to analyze the performance and resilience of the protocol in face of replica failures and malicious attacks. In this section, we discuss how well ByzID withstands a variety of Byzantine failures, and also demonstrate some key design principles underlying our design. We distinguish the replica failures due to system crashes, software bugs, and hardware failures from those attacks induced by dedicated adversaries that aim to subvert the system or deliberately reduce the system performance. Note that such a distinction is neither strict nor accurate. However, one can view the two types of evaluation as different perspectives to analyze the performance of ByzID.

5.7.1 Performance During Failures

We study the performance of the different BFT protocols for $f = 1$ under high concurrency, and in the presence of one backup failure.[3] To avoid clutter in the plot, PBFT, Zyzzyva, and ByzID experience a failure at $t = 1.5$ s, while for Aliph at $t = 2.0$ s. In case of failures, we require Aliph to switch between Chain and a backup abstract (e.g., PBFT) since its Quorum abstract does not work under contention. We set the configuration parameter k as 2^i, i.e., Aliph switches to Chain after executing $k = 2^i$ requests using its backup abstract.[4]

As shown in Fig. 5.9, neither PBFT or ByzID experience any throughput degradation after a failure injection. This is mainly due to their broadcast nature. However, the performance of Zyzzyva after a failure is reduced by about 40% because it

[3]The situation falls into our generalized definition of a *normal* case.

[4]Another option is to set k as a constant [50], but in our experience its performance during failure is inferior to using $k = 2^i$.

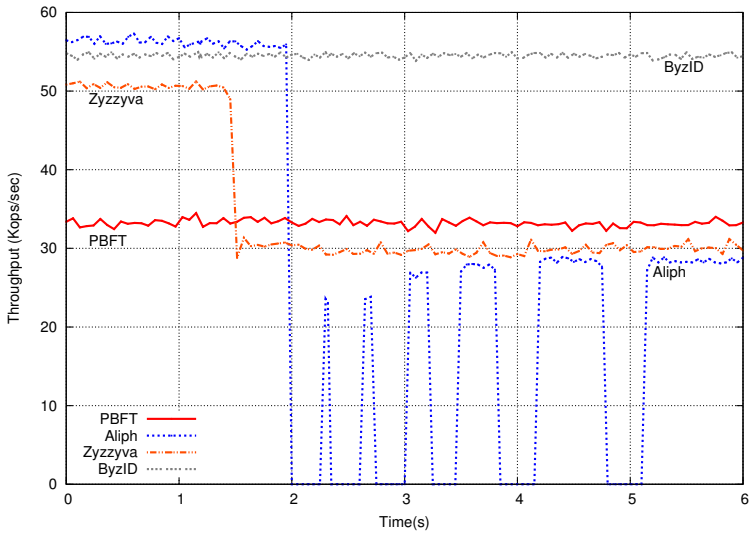


Figure 5.9. Throughput after failure at 1.5 s (2.0 s for Aliph).

switches to its slower backup protocol. Though Aliph has a slightly higher throughput than ByzID prior to the failure, its throughput reduces sharply upon failure, dropping below that of the PBFT baseline. Aliph periodically switches between Chain and PBFT after the failure, which explains the throughput gaps in Aliph. Since k increases exponentially for every protocol switch, it stays in the backup protocol for an increasing period of time.

5.7.2 Performance under Active Attacks

Too-Many-Server Compromises. Like other BFT protocols relying on trusted components, ByzID can mask at most f failures using $2f + 1$ replicas. With passage of time however, the number of faulty replicas might exceed f. This can happen if a dedicated attacker is able to compromise replicas one by one, and only asks them to manifest faulty behavior when a sufficient number of replicas have been compromised. If these compromises can go undetected by the IDSs, ByzID cannot

defend against such an attack. However, ByzID uses a proactive approach to prevent too many servers from being corrupted simultaneously. For other attacks, it is clear that our approach provides robustness.

Fairness Attacks. Fairness usually refers to the ability of every component to take a step infinitely often. This is inappropriate for time-critical applications such as in real-time transactional databases. For instance, in a stock system, a faulty primary might collude with a client to help the latter gain unjust advantages. Our IDS aided ByzID can achieve perfect fairness—ensuring that requests are executed in a "first come, first served" manner. Aardvark [29] can achieve a certain level of fairness, but does not achieve perfect fairness and is not suitable for time-critical applications. In contrast, ByzID achieves perfect fairness by leveraging IDSs, and has a significant performance advantage over Aardvark.

Flooding Attacks. We describe a flooding attack as one in which faulty replicas might continuously send "meaningful but repeating" or "meaningless" messages to other replicas. The goal of such attacks is to occupy the computational resources that are supposed to execute the pre-determined operations. This type of attacks is particularly harmful, as verifying the correctness of the cryptographic operations is relatively expensive. Such attacks can largely impact the performance of all the traditional BFT protocols. We take a number of countermeasures to defend against such attacks. First, we do not adopt the traditional pairwise channels between every replica pair. Instead, the primary forms the root of a tree, with backup replicas as leafs directly connected to the root. In particular, backups does not communicate with each other to prevent backups from flooding one another. Second, we use the IDSs to prevent the primary from flooding messages other than the [Order] messages to backups, and prevent the backups from flooding messages other than

[Ack] messages to the primary. Finally, we also use IDSs at backups to determine if received messages are from clients or the primary. A backup IDS simply filters all the incoming messages from the clients.

Timing Attacks ("Slow" Replica Attacks). We define *timing failures*, as the situation when replicas produce correct results but deliver them outside of a specified time window. One or more compromised replicas might delay several operations to degrade the performance of the system. For example, the primary can deliberately delay the sending of ordering messages in response to client requests. It is usually hard to distinguish such faulty replicas from slow replicas. It is also hard to distinguish if the failures are due to faulty replicas or channel failures. We use IDSs to monitor such kind of attacks. In particular, the timers can be setup by the anomaly-based intrusion detection. IDSs only monitor the node processing delays, not channel failures. Therefore, the monitoring can be accurate. Once the timer exceeds the prescribed value, an IDS will trigger an alert.

5.7.3 IDS Crashes

The IDSs themselves are not resilient to crashes. So what if the IDSs crash? One distinguishing advantage of ByzID is that it can still achieve safety (and liveness) even if all the IDSs crash. Indeed, ByzID has the following two properties that other BFT protocols relying on trusted components do not have: (1) Even if all IDSs crash, as long as the primary is correct, safety is never compromised. (2) Even if all IDSs crash, as long as all the replicas are correct, both safety and liveness are still achieved. Clearly, ByzID cannot provide the same resilience against attacks without the IDSs.

5.8 NFS Use Case

This section describes our evaluation of a BFT-NFS service implemented using PBFT [18], Zyzzyva [69], and ByzID, respectively. The BFT-NFS service exports a file system, which can then be mounted on a client machine. The replication library and the NFS daemon are called when the replicas receive client requests. After replicas process the client requests, replies are sent to the clients. The NFS daemon is implemented using a fixed-size memory-mapped file.

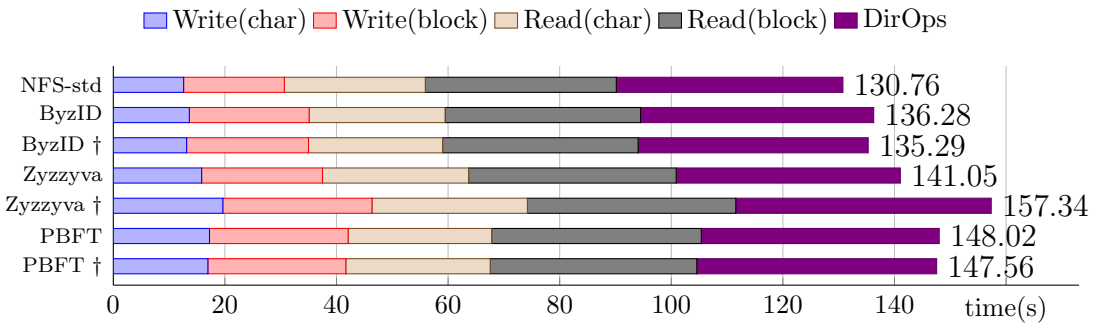


Figure 5.10. NFS evaluation with the Bonnie++ benchmark. The † symbol marks experiments with failure.

We use the Bonnie++ benchmark [30] to compare the three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. We evaluate the Bonnie++ benchmark with sequential input (including per-character and block file reading), sequential output (including per-character and block file writing), and the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that will appear random to the file system; (5) stat() random files; (6) delete the files in random order. We measure the average latency when a single client runs the benchmark, as shown in Fig. 5.10. The bar chart includes both the

fault-free case and the normal case where a backup failure occurs at time zero. We observe that in both cases, ByzID implementation outperforms both PBFT and Zyzzyva, and is only marginally slower than NFS-std.

5.9 Future Work

Relying on trusted IDS components, BFT protocol has been shown to both improve the performance over existing solutions and handle performance attack. In a real system, there are more ways IDSs can bring. For instance, we can use anomaly detection mechanism to monitor the timing of replicas or the traces of replicas. In addition, it is interesting to explore whether we could use BFT over IDS to enhance the accuracy of existing IDS solutions. Other than this, assuming we have independent IDSs that monitors the possibility of a server being intruded and a BFT protocol that achieves consensus. We can build weighted BFT protocol [47], where the weight of each server relies on the associated IDS.

5.10 Conclusion

We have shown a viable method to establish an efficient and robust BFT protocol by leveraging specification-based intrusion detection. Our protocol leverages the key assumption of a trusted reference monitor, but the approach we use is different from other BFT approaches relying on trusted components in that we apply a simple IDS monitoring and filtering technique. The reasons we use intrusion detection techniques can be summarized as follows: (1) The IDS for our BFT protocol is very simple in both code size and applicability—no heavy operations or cryptographic operations involved, and therefore relatively easy to implement as a reference monitor. (2)

Although IDSs themselves are not resilient to crashes, we can still achieve a form of safety even if all IDSs fail. (3) Equipped with IDSs, our BFT protocol is more robust against a number of important attacks. (4) Our IDS-aided ByzID protocol is also more efficient than other BFT protocols. Indeed, our experimental evaluation shows that ByzID is only marginally slower than Paxos.

Chapter 6

P2S: A Fault-Tolerant Publish/Subscribe Infrastructure

The work presented in this chapter was first described in an earlier paper by Chang, Duan, et al. [24]. The popular publish/subscribe communication paradigm, for building large-scale distributed event notification systems, has attracted attention from both academia and industry due to its performance and scalability characteristics. While ordinary "web surfers" typically are not aware of minor packet loss, industrial applications often have tight timing constraints and require rigorous fault tolerance. Some past research has addressed the need to tolerate node crashes and link failures, often relying on distributing the brokers on an overlay network. However, these solutions impose significant complexity both in terms of implementation and deployment.

In this chapter, we present a crash tolerant Paxos-based pub/sub (P2S) middleware. P2S contributes a practical solution by replicating the broker in a replicated architecture based on Goxos, a Paxos-based fault tolerance library. Goxos can switch between various Paxos variants according to different fault tolerance requirements.

P2S directly adapts existing fault tolerance techniques to pub/sub, with the aim of reducing the burden of proving the correctness of the implementation. Furthermore, P2S is a development framework that provides sophisticated generic programming interfaces for building various types of pub/sub applications. The flexibility and versatility of the P2S framework ensures that pub/sub systems with widely varying dependability needs can be developed quickly. We evaluate the performance of our implementation using event logs obtained from a real deployment at an IPTV cable provider. Our evaluation results show that P2S reduces throughput by as little as 1.25% and adds only 0.58 ms latency overhead, compared to its non-replicated counterpart. The performance characteristics of P2S prove the feasibility and utility of our framework.

6.1 Introduction

Significant effort has been devoted to developing reliable pub/sub systems [13, 20, 43, 59, 64, 65, 94, 104, 120]. Most of them cope with broker crashes and/or link failures, ensuring that messages are eventually delivered. While the weak fault tolerance is sufficient in some systems, other application domains demand stringent delivery order of their messages. Only a handful of prior published research papers have discussed how to achieve total ordering in reliable pub/sub systems [64, 65, 120]. In order to guarantee total ordering in the presence of failures, virtually all past published work relies on an overlay network topology. For each new type of topology, a different algorithm must be introduced, adding significant complexity both in terms of algorithm correctness proofs, implementation, and deployment. Therefore, industrial deployments tend to rely on the more established centralized architecture instead of decentralized overlay topologies.

Traditional fault tolerance techniques based on Paxos [73] can provide total ordering and guarantee safety even in the presence of any number of failures. However, liveness cannot be ensured in periods of asynchrony. Building a reliable pub/sub system based on an existing, proven approach, reduces the effort required to prove the correctness of algorithms since the protocol can be proven correct by refinement from the original algorithm. However, adapting traditional fault tolerance techniques to pub/sub systems is challenging. Intuitively, every broker can be replicated, which can be extremely impractical. Total ordering on every message can be overkill since different messages may require different ordering semantics. For instance, per-publisher total ordering is sufficient for publications from a single publisher to multiple subscribers. On the other hand, the topology of brokers in pub/sub systems varies from a single centralized broker to very large-scale overlays. Replication of brokers may impose adjustment of pub/sub overlays, especially when the brokers are replicated on demand. Therefore, management of replication should impose minimum overhead.

In this chapter, we propose a framework for building reliable pub/sub systems that directly adapts existing fault tolerance techniques to pub/sub. At the core of our pub/sub infrastructure is our crash fault tolerance library and a pub/sub interface. Our library guarantees fault tolerance through replication, and ensures strong consistency using Paxos to order publications. Our fault tolerance library can switch between different consistency protocols depending on application specific fault tolerance requirements. On the other hand, the pub/sub interface communicates between application level roles (publishers, subscribers, and the brokers) and the replication library. The interface takes publications that must be totally ordered, and pass them on to the replication library as requests and totally orders them. The messages are then delivered to the corresponding subscribers in order.

We have designed P2S, a topic-based crash tolerant pub/sub system based on a

replication library Goxos [60,79], a Paxos-based Replicated State Machine (RSM) [101] framework written in the Go programming language [51]. P2S is motivated by the simplest pub/sub architecture that is employed in several industry settings: publishers and subscribers with only a centralized broker. Since the centralized broker becomes a single point of failure, we replicate the broker to achieve resilience. To ensure total ordering, a Paxos-based library is run among the replicated brokers. Although we adopt the architecture of P2S directly from existing fault tolerance protocols, we are not aware of any other published work discussing the implementation of such solutions and therefore the performance characteristics have previously not been explored and published. We further evaluate the performance of P2S using recorded event logs obtained from a real deployment of event loggers at about 180,000 homes connected to an IPTV cable provider. Our evaluation results show that P2S causes as low as 1.25% reduction in throughput and only 0.58 ms end-to-end latency overhead compared to its non-replicated counterpart.

Our chapter makes the following key contributions:

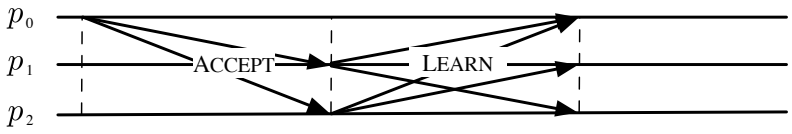
1. We implemented P2S, the simplest architecture based on the framework, a topic-based crash tolerant pub/sub system with centralized replicated brokers.
2. We demonstrate the utility of P2S through experiments using recorded data logs obtained from an industrial centralized IPTV application deployed at a national telco operator. The evaluation results show that P2S achieves total ordering in the presence of failure with low overhead compared to its non-replicated overhead.
3. We present a framework for building reliable pub/sub systems that directly adapts existing proven secure fault tolerance approaches, with a relatively simple correctness proof and implementation. The framework is flexible and versatile enough to be used in future development.

The rest of the chapter is organized as follows: first, we introduce some background of our work in §6.2. In §6.3, we describe the design and development details of our framework. Then we show experimental results in §6.4. We conclude by reviewing our contributions in §6.6.

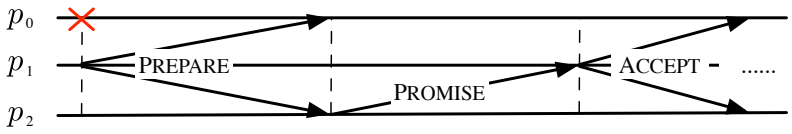
6.2 Background

In this section we present background for our fault-tolerant pub/sub system, P2S. We begin by introducing Paxos, a well-known crash fault-tolerant consensus protocol on which we base P2S. We then briefly summarize the pub/sub architecture on which we base P2S.

6.2.1 Fault Tolerance



(a) The normal-case operation



(b) Paxos leader/proposer change

Figure 6.1. The Paxos Protocol.

The Paxos protocol is a fault-tolerant consensus protocol, in which a set of participants (our replicas) try to reach agreement on a value. For our purpose, we can use multiple instances of Paxos to agree on a sequence of values (or commands) sent

to an RSM. This is also called Multi-Paxos. With Paxos, the participants can reach agreement when at least $f + 1$ of the participants are able to communicate, where f is the number of replica failures that can be tolerated. One of the nice properties of Paxos is that it guarantees that consistency among the replicas will never be violated even if more than f replicas fail. It achieves this property at the expense of liveness. That is, if more than f replicas fail, or if fewer than $f + 1$ replicas are able to communicate, then Paxos cannot make progress. Ensuring strong consistency among replicas is an important property, useful for a wide range of systems, including pub-/sub systems. This is related to the fundamental tradeoff between strong and weak consistency.

We now explain how one instance of Paxos might operate in the pub/sub paradigm. First suppose that the participants must be made to agree on a single value or command to execute on our broker RSM. This command can be considered as a publication. Thus, the following is concerned with only a single command/publication. Paxos is often explained in terms of two phases, where the first phase is only invoked initially and to handle failures, while the second phase represents the normal case operation, and must be performed for every value to be agreed upon.

Paxos proceeds in rounds, where in each round there is a single replica designated as the *proposer*, also called the *leader*. Fig. 6.1(a) depicts the normal case operation where the proposer is correct. During the normal case operation, the proposer chooses a value and sends an ACCEPT message to a set of replicas called *acceptors*. If an acceptor accepts the value, it sends an LEARN message to all the replicas. The value is chosen when a replica receives LEARN messages from a majority of replicas.

When the current proposer is suspected to be faulty, another replica may assume the role of proposer. To be effective as proposer, it needs to collect support from a majority of the replicas. It does so by broadcasting a PREPARE message to the other

replicas. Upon receiving the PREPARE message, a replica stops accepting messages from the old proposer and replies to the new proposer with a PROMISE message, and includes the value chosen in its last round. When the leader collects a set of PROMISE messages from a majority of replicas, it either selects a value if at least one replica accepts it, or any value, if no replica includes any values in their PROMISE messages. Afterwards, replicas proceed as in normal case operation described above. Fig. 6.1(b) shows the leader change phase of Paxos.

6.2.2 Pub/Sub

We build on the pub/sub architecture described by Eugster et al. [44], as illustrated in Fig. 6.2. In a topic-based pub/sub system, *subscribers* express their interests in certain types of events, and are subsequently notified with *publications*, generated by *publishers*. *Brokers* are placed at the center of the infrastructure to mediate communication between publishers and subscribers. This event-based interaction provides full decoupling in *time*, *space*, and *synchronization* between publishers and subscribers. We assume topic-based pub/sub [44], where messages are published to topics, and subscribers receive all messages sent to the topics to which they subscribe.

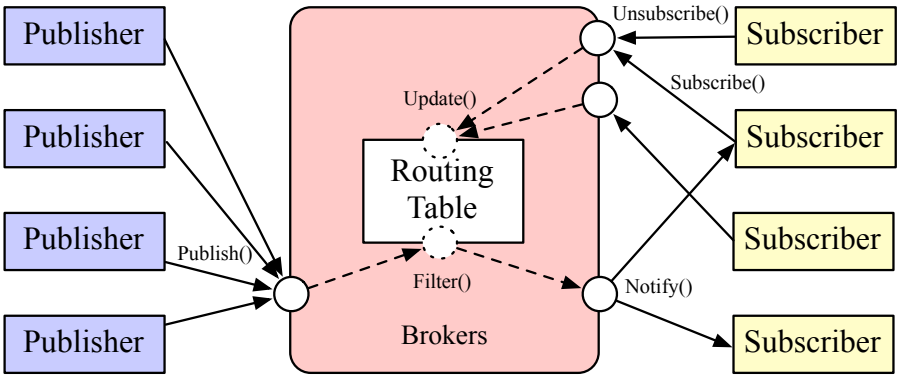


Figure 6.2. Publish/Subscribe architecture with three agent roles

In this chapter, we address broker crash failures in an asynchronous model, where messages can be delayed, duplicated, dropped, or delivered out of order. P2S employs a simple pub/sub architecture: between publishers and subscribers is a set of $2f + 1$ replicated brokers, among which up to f broker failures are tolerated. The replicated brokers can be in one or more administrative domains, perhaps geographically separated.

The protocol provides both safety and liveness as defined below. The safety property is also referred to as *total order*, which is defined in multiple ways in the pub/sub literature. For instance, *per-publisher total order* ensures that messages sent by a single publisher are totally ordered. Our system aims to achieve the strongest safety properties—*pairwise total order*—where replicated brokers behave like a centralized broker.

- (**Pairwise total order (Safety)**) Assume messages m and m' are delivered to both subscribers p and q, m is delivered before m' at p if and only if m is delivered before m' at q.
- (**Liveness**) If a message is delivered to a subscriber, all correct subscribers to the same topic eventually receive the same message.

6.3 P2S

Our P2S framework is built on our existing Paxos-based RSM library, Goxos [60, 61, 79]. For higher level pub/sub application builders, P2S provides a generic programming interface.

This section introduces details of the original Goxos implementation, along with changes we make to adapt Goxos to the pub/sub model, the P2S system architect, programming APIs, some application implementation details, and the core broker

algorithm that runs inside each P2S broker. Essentially, when messages are sent by clients (either publishers or subscribers) to brokers, they are handled by the Goxos library. Goxos treats client messages as Paxos requests, orders them accordingly, and delivers them to the upper level. The messages are then forwarded to the corresponding publishers or subscribers according to the message type.

6.3.1 Goxos Architecture and Implementation

Goxos provides a fault-tolerant library for P2S. Namely, P2S implements Goxos interfaces to replicate its broker. When no more than f brokers fail, all failures are handled internally in the underlying Goxos framework in the way that Paxos originally describes and will not be noticed by publishers or subscribers. Thus, Goxos provides a great degree of crash fault tolerancee to the above pub/sub system.

In our original implementation [60, 61, 79], Goxos replicas act as the replicated brokers, out of which only one replica is the leader to handle client requests. A client, either a publisher or a subscriber, first reads a predefined configuration file and finds the Goxos replicas, then dials the leader. The leader receives this client's connection attempt, then establishes the connection, and stores the client connection for further interactions. The client then is able to send request to either issue a publication, a subscription, or unsubscription to the leader. Upon receiving a valid client request, the leader treats the raw request as a Paxos proposal and disseminates it across all Goxos replicas to achieve consensus. Each replica decides on a request and then executes it. Then finally, the execution result is replied back to the client.

This original implementation does not fit the pub/sub model because it acts strictly in the passive request-then-response style. This means it lacks the logic to handle proactive message delivery. We therefore alter Goxos so when a broker replica

executes a client request, it retrospects the message type. If it is a subscription or unsubscription, the replica will scan and update the local subscription table. If it is a publication, the replica will deliver the publication to each of the subscribed clients. Details are given in Section 6.3.4.

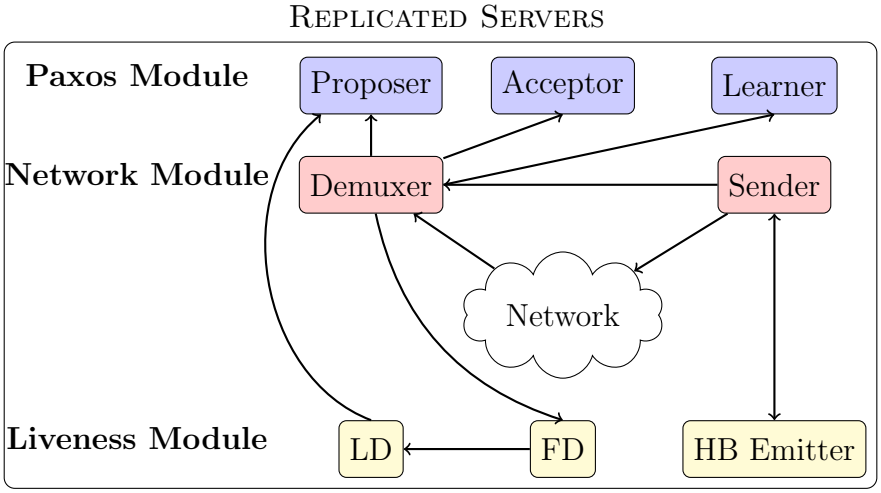


Figure 6.3. Goxos Architecture [61].

Fig. 6.3 shows the main modules of Goxos, which we organize into three parts: first, the Paxos module, which includes the complete Paxos protocol. Second, the Network module, which handles all networking in Goxos. The Network module contains a Demuxer and a Sender as submodules. The Demuxer handles all incoming connections and relays received messages to the local replica's correct Paxos module for further processing. The Sender module is responsible for sending messages to other replicas as requested by Goxos. These two modules, taken together, emulate remote channels between Goxos agents. Finally, the Liveness module, which handles the failure detection and leader election necessary for Paxos. The three different modules communicate with each other through Go's channels. In the figure, a single-ended arrow pointing from a source module to a destination module signifies that the source can send a message to the destination over a one-way channel. A double-

ended arrow signifies that both modules can send and receive to one another over a two-way channel. For example, the Demuxer module sends messages to the proposer, acceptor, and learner (which are in the Paxos module). Since Paxos itself must be able to handle many concurrent activities, the liveness module, network module, and Paxos module are all implemented as concurrently executing goroutines. Goxos lies as the core of replicated servers, as we will show in Fig. 6.5 in the next section.

As a base framework for building fault-tolerant services, Goxos offers sophisticated user interfaces for higher level applications to invoke. Fig. 6.4 shows four main interfaces available to application developers.

```
type Handler interface {
  Execute(req []byte) (resp []byte)
  GetState(slotMarker uint) (sm uint, state []byte)
  SetState(state []byte) error
}

func NewGoxosReplica(uint, uint, string, app.Handler) *Goxos

func Dial() (*Conn, error)

func (c *Conn) SendRequest(req []byte) ([]byte, error)
```

Figure 6.4. Goxos interface.

Server applications can create a replicated service with the `GX.NewGXReplica` function. This will construct a new replica. The first two arguments of `NewGXReplica` are the id of the replica and the id of the application. The third argument is a string describing the application. Finally, the last argument is a type that implements the `app.Handler` interface. The `app.Handler` interface must be implemented by an application that uses the replication library. This interface defines several methods that must be implemented on the type: `Execute`, `GetState`

and `SetState`. The first method, `Execute`, takes a byte slice, which should be a command that can be executed in the application. The `Execute` method also returns a response from the application in the form of a byte slice. The second and third methods, `GetState` and `SetState`, are used for live replica replacement.

The client library for Goxos is used to connect to the Paxos replicas, as well as to send and receive responses. The client connection can be created with the `Dial` method in the library. This method returns a `Conn`, representing a connection to the whole replicated service. All of the work of handshaking with the servers and identifying the leader is abstracted away. The most useful method on a `Conn` is `SendRequest`, which can be used to send requests to the service. The client request is a byte slice, meaning that if the application wants to send Go structs or other complex types as commands to the service, it must marshal them into byte form. Similarly, the return value is also a byte slice, which represents the response from the service. This also means that a client must wait for a response from Goxos servers before it can send any further requests.

6.3.2 System Architecture and API

P2S, as a fault-tolerant pub/sub service, is comprised of a client library, a replicated server cluster with Goxos library as the core, and a client handler deployed at servers. The client handler is deployed at the servers and receives messages from client applications (publishers and subscribers). The client library is used by client applications to communicate between the client handler and the replicated service. The replicated server cluster handles all incoming client requests via the client handler, replicates brokers, and orders requests to achieve total order in the presence of failures. Finally, the server application executes client requests ordered by the

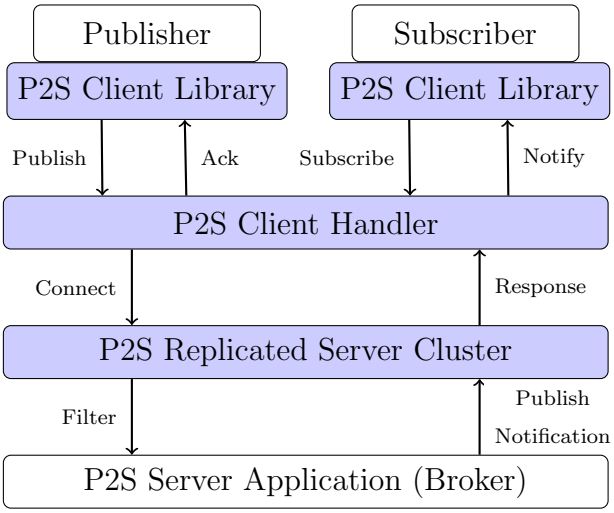


Figure 6.5. P2S System Architecture.

cluster. Fig. 6.5 shows an abstraction of the P2S architecture.

P2S **client library** offers standard pub/sub style applications a set of client APIs. The client library communicates with servers, sends out client requests (which can be publications, subscriptions, or unsubscriptions), and receives corresponding responses for the client application to interpret. As shown in Fig. 6.6, the library defines a pair of data structures that applications must use, two standard interfaces, and several methods.

`Request` and `Response` define the data format that client applications must use. `Ct` in `Request` and `ToType` in `Response` represent the command type, which is 'Publish', 'Subscribe', or 'Unsubscribe'. `Cid` in `Request` denotes the client ID, which is used by servers as a key to identify the corresponding client connection. `Topic` and `Content` represent publications and subscriptions. Lastly, `Subs` in `Response` is an array of subscribers' ID that is filtered by the servers for publication delivery.

The interface `PublicationManager` is implemented by a publisher's application. `Publish` calls are used by the application to issue a publication. `Publish` takes two

```
type Request struct {
  Ct      CommandType
  Cid     string
  Topic   string
  Content string
}

type Response struct {
  ToType  CommandType
  Ack     string
  Topic   string
  Content string
  Subs    []string
}

type PublicationManager interface {
  Publish(topic, content string)
}

type SubscriptionManager interface {
  Subscribe(topic string) chan []string
  Unsubscribe(topic string)
}

func PDial(account string) PublicationManager
func SDial(account string) SubscriptionManager
func (sm *submngr) awaitPublications(notifyChan chan []string)
```

Figure 6.6. P2S Client Library.

arguments as input: the topic and content of the publication. Similarly, the interface `SubscriptionManager` is implemented by a subscriber's application. This interface has two methods, `Subscribe` and `Unsubscribe`, both taking a string of topic as an argument. The `Subscribe` returns a Go string slice channel. This channel is used by the method `awaitPublications`, which is for a client to wait for delivered publications to the topic that the `Subscribe` method issues.

Both `PDail` and `SDial` are called when an application initiates. They return instances of `PublicationManager` and `SubscriptionManager`, respectively, that the application later invokes.

The P2S **client handler** is initiated on server startup. The Client handler is the frontend of the replicated server cluster, handling client calls. It receives connection attempts from clients, stores client requests (a publication or subscription), passes the request to the backend P2S server application to filter, and receives the processed result, and finally sends back the response to related clients. The processed result has two types: either an acknowledgement to a publisher or a filtered publication to interested subscribers. Fig. 6.7 shows a set of functions in the client handler library.

```
func (ch *ClientHandler) greetClient(conn net.Conn)
func (ch *ClientHandler) handleRequest(req *Request)
func (ch *ClientHandler) handleResponse(resp *Response)
```

Figure 6.7. P2S Client Handler.

The `greetClient` function starts up an infinite loop waiting for potential client connection attempts. It responds to the `Dial` method the client calls, identifies the client address and ID, then stores the client connection object in a local connection pool.

The `handleRequest` function receives client requests, checks each request to see if it has been executed before, generates a response for new request, and stores both the request and response.

The `handleResponse` function is called immediately after a response is generated by the `handleRequest` method. `handleResponse` first loops the client connection pool, identifies the client that sent the request, then pushes back the response to the client. The `handleResponse` function then introspects the request type. If the

request is a publication, `handleResponse` initiates the filtering, finds the subscribers that are interested in the topic in the client connection pool, and delivers the publication to all the subscribers.

The P2S **replicated server cluster** is the service with our modified Goxos framework as the core. It does not differentiate client message types. It simply treats each client message as a Paxos proposal and executes through the consensus protocol. It then passes the client message to backend server application to interpret.

6.3.3 ZapViewers Application

In order to evaluate the capabilities of P2S, we built a fault tolerant TV viewer statistics application based on an existing centralized (non-replicated) pub/sub system deployed at a real IPTV operator. We refer to this as our ZapViewers application. In our evaluation, we use recorded event logs from the real deployment.

A high-level architecture of our ZapViewers application is shown in Fig. 6.9. The application consists of three parts: event publishers (set-top boxes), subscribers (clients interested in viewership statistics), and a replicated broker. A P2S event publisher simulates a fraction (around 180,000) of IPTV set-top boxes (STBs) deployed at customer homes receiving IPTV over a multicast stream. Each STB records viewers' TV channel change information, and sends the event to the IPTV operator's server. The publisher accomplishes this simply by calling our `Publish()` method. Based on these events, the broker computes the TV viewership.

A P2S subscriber can either be television broadcasters or commercial entities interested in TV viewership statistics. Such a subscriber is usually concerned about ratings of TV channels, and viewers' channel change behavior. The subscriber that we implemented informs the server of its interested topics, such as top-N most viewed

TV channels or viewership of some specific channels. The broker then notifies each subscriber of the corresponding statistics. The subscriber calls our standard **Subscribe()** method to inform the brokers of their interest.

P2S brokers are replicated server applications that function as fault-tolerant brokers to external event publishers and subscribers. P2S brokers rely on the Goxos framework as their core by implementing system APIs such as the `Handler` interface as described in previous sections. The brokers implement several functions to collect events and computes statistics, including the two shown in Fig. 6.8.

```
func numViewers(channel string) int
func computeTopList(n int) []*zl.ChannelViewers
```

Figure 6.8. ZapViewers application interface.

Function call `numViewers(channel string)` takes a `channel` name as input from a P2S subscriber and returns that channel's viewership information. Function call `computeTopList(n int)` returns a list of the `n` most viewed channels at a particular instant to the subscriber.

The P2S publisher can generate two event types as follows:

⟨Date, Time, STB-IP, ToCh, FromCh⟩

⟨Date, Time, STB-IP, Status⟩

Date and Time mark the date and timestamp that the event is triggered. STB-IP is the IPv4 address of the sending STB unit. ToCh and FromCh indicate the new channel and the previous channel that the STB unit is tuned in on. Status is a change in status of the STB, which is either volume change on a scale of 0–100, mute/unmute, or power on/off. The event is encoded in text format, and its size is typically less than 60 bytes.

Events have either 4 or 5 fields. An event with 5 fields represents a TV channel

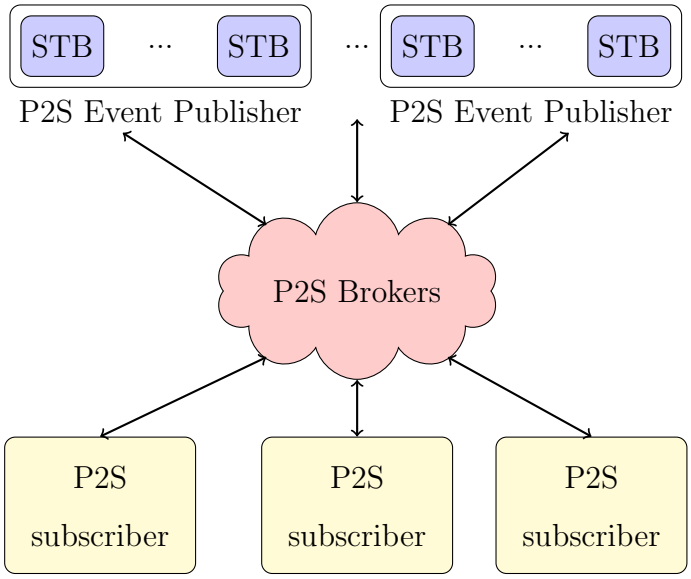


Figure 6.9. ZapViewers Application Architecture.

change event, and such an event does not contain STATUS. An event with 4 fields contains a STATUS in the 4th field, but does not have the fields TOCH or FROMCH.

6.3.4 Broker Algorithm

The core of our P2S application is the replicated service provider, the broker. A broker does a handful of back-end jobs, including maintaining subscriptions, storing P2S events as publications, filtering and matching, and delivering publications to subscribers. We depict the essential broker algorithm as follows.

Brokers maintain the following key variables: the subscription table **ST**, the channel for piping requests (subscriptions and publications) **ReqChan**, the channel for piping responses (acknowledgements and to-deliver publications) **RespChan**, the channel for sending proposals to the Paxos variant **PropChan**, the queue of replies $\mathcal{R}$, the Paxos variant in use **Paxos**, and two message types for introspection **Publication** and **Subscription**.

When a broker starts up, it initializes several routines: monitoring the request channel **ReqChan**, the response channel **RespChan**, and the proposer channel **PropChan**. When a broker receives a new client request, it invokes the **handleRequest(req)** method. The **handleRequest(req)** function call first checks if itself is the current Paxos leader. If not, it checks whether the Paxos variant in use permits direct message routing between non-leader replicas and the client. Fulfilling either of the two conditions means that the request is handled immediately. Otherwise, the broker redirects the request to the Paxos leader.

The broker checks if the request is a new one. If so, it sends the request to the proposer channel **PropChan** and let Paxos executes it. If it is an old request, it simply finds the response in the reply queue by $\mathcal{R}$**.find(req)**, and **ack()** the client once more.

When a request is sent into the proposer channel, the broker invokes operation **executePaxos(prop)** and the request is executed through Paxos. The execution result generated by **genResp(prop)** is sent into the response channel **RespChan** immediately. In addition, the broker introspects the message type and if it is a subscription, the broker updates the subscription table **ST**.

On detecting a new response from channel **RespChan**, the broker calls **handleResponse(resp)**. The broker adds the response to the reply queue $\mathcal{R}$, and **ack(resp)** back to the client. This means the broker introspects the message type and if it is a publication, the broker travers the client connection pool, filters out the subscriber by checking the subscription table **filter(ST)**, and finally delivers to all the subscribers to the topic.

Each valid client request is executed through the whole cycle and the broker is capable of executing multiple concurrent requests. This is enabled by the Paxos variant in use. Our Goxos framework provides Multi Paxos [74], Batch Paxos [74]

and Fast Paxos [75] for the time being. In our P2S application, we use Multi Paxos with 3 concurrent batched executions at a time. We further describe the evaluation in §6.4.

6.4 Evaluations

In this section, we evaluate both our ZapViewers application with different replication degrees and the original non-replicated version. We evaluate end-to-end latency, throughput, and scalability under different settings.

6.4.1 Experiment Setup

All experiments are carried out in our computing cluster composed of GNU/Linux CentOS 6.3 machines connected via Gigabit Ethernet. Each machine is equipped with a quad-core 2.13GHz Intel Xeon E5606 processor with 16GB RAM.

For our experiments, we obtained recorded event logs from a real commercial IPTV provider. The experiments are carried out using 1, 3, 5, and 7 broker replicas. The experiments using only 1 broker are our baseline, as they represent the non-replicated ZapViewers application. The experiments using 3–7 broker replicas allows our system to tolerate 1–3 crash failures. We use up to 24 event publishers, with each event publisher simulating 180,000 STBs, and a small number of subscribers. In the real deployment, each STB caches local channel changes for channels with retention longer than 3 seconds. These cached events are sent to the server every 10 seconds. Indeed, the number of the event publishers (STBs) is typically large, while the number of the IPTV viewership statistic subscribers (e.g., TV broadcasters and other commercial entities) is relatively small. However, while the event volume

produced by each STB is relatively low, the aggregate becomes significant.

In all experiments, we use pipelined Multi Paxos [74] with $\alpha = 10$. That is, ten distinct Paxos instances can be decided concurrently. Even though they are decided concurrently, their processing takes place sequentially. Each Paxos instance comprises a batch of STB events to be processed by the broker replicas in sequence.

6.4.2 End-to-End Latency

We first assess the end-to-end latency. Herein, we define end-to-end latency as the duration between the sending of an event and the corresponding receive at an active subscriber. The latter is inferred from the notification corresponding to the source event. For calculating end-to-end latency, we record a timestamp when a publication is issued by a publisher, and this timestamp is kept by brokers in the execution result that is delivered to any subscriber. The subscriber is therefore able to calculate the latency by comparing the original publisher's timestamp and local time.

Fig. 6.10 shows the latency of our ZapViewers application in different configurations, namely non-replicated, with 3, 5, and 7 replicas, each tolerating 0, 1, 2, and 3 crash failures, respectively. We observe an increase of end-to-end latency in all four experiments as we increase the number of P2S event publishers. We vary the number of publishers from 1 to 24.

The latency of the original non-replicated ZapViewers application varies from 1.98 ms under light load up to 2.32 ms under high load. As expected, all experiments with our replicated ZapViewers implementation show higher latencies than the non-replicated version. That is, we observe an overhead of 0.58 ms (29%) under light load, and 1.23 ms (49%) under high load. Still, from our subscribers' point of view, this latency overhead is barely noticeable.

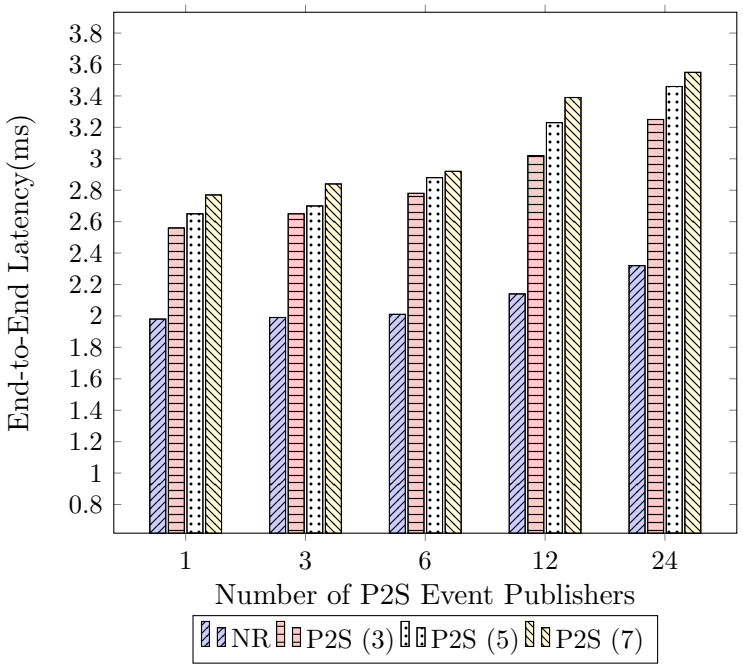


Figure 6.10. End-to-end latency for various numbers of publishers

Also as expected, the latency gradually increases as the number of publishers increases. Since we pipeline events using the Goxos library, the latency increase is small. For the non-replicated broker, the latency overhead of accommodating 24 publishers instead of just 1 corresponds to 0.34 ms (17%). In comparison, with 3, 5, and 7 brokers, latencies are 0.69 ms (26%), 0.81 ms (30%), and 0.78 ms (28%) higher when the number of concurrent P2S event publishers grows from 1 to 24.

We also see that higher replication degrees (indicated by the different bars in Fig. 6.10), imposes only marginal latency overhead.

6.4.3 Broker Throughput

We assess the broker throughput for the same configurations as in our latency evaluation, as shown in Fig. 6.11. We define broker throughput as the number of publi-

cation batches that are processed by the broker per second. We run experiments in a pipeline manner, with ten distinct instances decided concurrently.

We first observe that for small workloads, all experiments achieve almost identical throughput. With fewer than 6 publishers, the throughput reduction is less than 6% between non-replicated broker and the 7-replica broker.

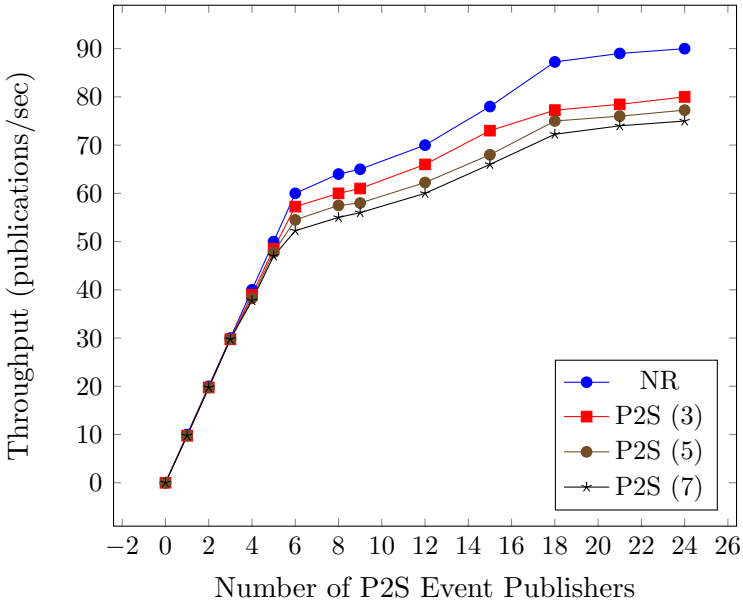


Figure 6.11. Broker throughput for varying number of publishers.

When the number of publishers is higher than 5, the non-replicated application achieves slightly higher throughput than its replicated counterparts. The throughput drops as little as 4.58% compared to the non-replicated application. As shown in Fig. 6.11, the peak throughput of the original non-replicated application, when there are 24 publishers, is 90.00 publications per second. In comparison, the peak throughput with 3, 5, and 7 replicas are 80.04, 77.25, and 75.03 publications per second, which are 9.96%, 14.16%, and 16.63% lower than non-replicated service, respectively.

Higher replication degree results in consistently lower throughput. Similarly to

latency, the overhead caused by this is 6.5% on average. This is explained by the fact that in Paxos, higher replication degree does not cause significant performance degradation.

6.4.4 Scalability

We evaluate the scalability of our ZapViewers application by varying both replication degrees and the number of event publishers.

Table 6.1 presents the latency and throughput degradation of ZapViewers when replication degree varies. We compare each instance with a counterpart that has one replication degree lower. As shown in the table, the non-replicated application outperforms all replicated counterparts. With only 1 publisher, the latency of the non-replicated application is 29.9% higher than that of P2S (3). With 24 event publishers, it is 40.08% higher. However, latency drop becomes less noticeable as the replication degree increases. For instance, with 1 publisher, latency of P2S (5) is 3.51% lower than that of P2S (3). With 24 event publishers, it is only 6.46% lower.

Throughput decreases slower on the other hand. When the workload is fairly low, with fewer than 3 event publishers, the difference is barely detectable. The non-replicated application is 11.11% higher than P2S (3). With higher replication degree, throughput varies between 2.91% and 3.43%.

We also compare the performance change for replication degree when the number of P2S event publishers varies, as shown in Table 6.2. For each application, latency rises with more event publishers. With high replication degree, the latency gradually becomes stable, approaching the peak latency when the number of P2S event publishers is more than 12. When the number of event publishers is greater, the latency decreases much slower, thereafter.

Table 6.1. Latency (upper table) and throughput (lower table) drop of ZapViewers, compared to the counterpart that has one replication degree lower. $\#p$ is the number of publishers.

	$\#p = 1$	$\#p = 3$	$\#p = 6$	$\#p = 12$	$\#p = 24$
P2S (3)	29.29%	33.1%	38.30%	41.12%	40.08%
P2S (5)	3.51%	1.88%	3.59%	6.95%	6.46%
P2S (7)	4.52%	5.18%	1.38%	4.95%	2.60%
P2S (3)	2.50%	1.25%	4.58%	5.71%	11.11%
P2S (5)	0.00%	0.00%	4.80%	5.68%	3.43%
P2S (7)	0.00%	0.00%	4.12%	3.61%	2.91%

This trend is consistent with the improvement of throughput when the number of event publishers differs. As shown in the table, under low workload, throughput improves almost linearly. When there are more than 6 event publishers, the increase becomes gradually slower. For instance, from 6–12 event publishers, P2S (7) throughput grows 14.83%, or 2.47% per publisher. Also from 12–24 event publishers, growth is 25%, or 2.08% per publisher. This indicates the brokers have almost the maximum processing rate.

To summarize, P2S scales very well when the replication degree and the number of event publishers increases. This demonstrates that our system can retain its efficiency even when we build a system that can tolerate more failures.

6.5 Future Work

As a illustration of a framework, P2S is shown to achieve great performance. In a complete system, we could further rely on and explore the framework in the future.

Table 6.2. Latency drop (upper table) and throughput rise (lower table) of ZapViewers, compared with its own performance when p differs. Values with parenthesis in red represent positive improvement. The number of publishers is denoted by $\#p$.

	$\#p1-3$	$\#p3-6$	$\#p6-12$	$\#p12-24$
NR	0.50%	1.00%	6.46%	8.41%
P2S (3)	3.51%	4.90%	8.63%	7.61%
P2S (5)	1.89%	6.66%	12.15%	7.12%
P2S (7)	2.52%	2.81%	16.09%	4.71%
NR	(200.00%)	(100.00%)	(16.66%)	(28.57%)
P2S (3)	(205.12%)	(92.43%)	(15.28%)	(21.21%)
P2S (5)	(205.12%)	(83.19%)	(14.22%)	(24.09%)
P2S (7)	(205.12%)	(75.63%)	(14.83%)	(25.00%)

For instance, we could build a system with different ordering properties. For certain type of messages where total order is necessary, we use a Paxos or even stronger library. For other type of message where the order is not important, we use the traditional pub/sub communication.

6.6 Conclusion

This chapter presents P2S, a simple fault-tolerant pub/sub solution that replicates brokers in a central pub/sub architecture. Our solution fits naturally in many industrial settings that need certain resilience, without having to rely on complex, overlay networks.

We have shown how our P2S framework adopts traditional fault tolerant proto-

cols to the pub/sub communication paradigm. P2S provides sophisticated generic programming interfaces for higher level pub/sub application builders, and is built upon our Paxos-based, fault-tolerant Goxos library. Goxos switches between various Paxos variants according to different fault tolerance requirements. The flexibility and versatility of the P2S framework aims to minimize the effort required for future development of any pub/sub systems with various resilience needs.

Our results, evaluated based on recorded data logs obtained from a real IPTV service provider, indicate that P2S is capable of providing reliability at low cost. With a minimum degree of replication, P2S imposes low performance overhead when compared to the original non-replicated counterpart.

In future work, we aim to experiment with the P2S framework on Byzantine failure models. We believe that there is a need for Byzantine fault tolerance in certain industrial applications, and believe our work can be extended to adapt to BFT as well.

Algorithm 10 Broker Algorithm

1: **Initialization:**
2: ST {Subscription Table}
3: $ReqChan$ {Request Channel}
4: $RespChan$ {Response Channel}
5: $PropChan$ {Proposer Channel}
6: $\mathcal{R}$ {Reply Queue}
7: $Paxos$ {Paxos Variant}
8: P {Message Type: Publication}
9: S {Message Type: Subscription}
10: **on event** $req \leftarrow ReqChan$ {Monitor Request Channel}
11: handleRequest(req)
12: **on event** $resp \leftarrow RespChan$ {Monitor Response Channel}
13: handleResponse($resp$)
14: **on event** $prop \leftarrow PropChan$ {Monitor Proposer Channel}
15: executePaxos($prop$)
16: **on event** executePaxos($prop$) {Execute Through Paxos}
17: $RespChan \leftarrow genResp$(prop)
18: **if** $prop.Type == S$ **then**
19: update(ST) {Update Subscription Table}
20: **on event** handleRequest(req)
21: **if** $nid == leader$ **or** allowDirect[$Paxos$] **then**
22: **if** req is new **then**
23: $PropChan \leftarrow req$ {Send into Paxos Module}
24: **else** ack($\mathcal{R}.find(req)$) {Re-reply Old Request}
25: **else** redirect(req) {Redirect To Leader}

```
1: on event handleResponse(resp)
2:     R.add(resp)
3:     ack(resp)                                   {Acknowledgement}
4:     if resp.Type == P then                      {Invoke Publication Delivery}
5:         C = filter(ST)                          {Filter And Match}
6:         deliver(C, resp)                        {Deliver Publication}
```

Chapter 7

Comparison

In the previous chapters we describe three BFT protocols, *h*BFT, BChain, ByzID, and a Paxos-based pub/sub infrastructure P2S. P2S can be viewed as an application of fault tolerance protocols. As discussed in Chapter 1, the three protocols take different approaches to enhance performance, such as moving jobs to clients, using partially connected graphs, using trusted components, etc. In this chapter, we compare the performance of the three BFT protocols, and then discuss P2S as well as other applications of fault tolerance.

Table 7.1. Best use case of the protocols. ‖Performance attack refers to the attack where faulty replicas intentionally render the overall performance low, usually by manipulating the timers.

Protocols	Best Use Case
*h*BFT	High rate of client and replica failures
BChain	High concurrency; Small number of replicas; Lower rate of replica failures
ByzID	High rate of performance attack‖; Highly scalable systems

Failure-free Case Performance As shown in Table 7.2, all three protocols enhance

the performance in comparison to existing state-of-the art protocols. Although the experiments were carried out separately when each protocol was designed, under similar but different settings, we could still compare the overall performance. As can be observed in Table 7.2, the number of cryptographic operations is directly related to the throughput. The experimental results validate the theoretical results. When the number of clients is large enough, the number of cryptographic operations of BChain approaches 1 while the other two all tend to 2. Therefore, the peak throughput of BChain is higher. However, when the number of clients is low, the other two both achieve higher throughput. Since ByzID relies on a trusted IDS, and the IDS components cause very little overhead, it does not require encryption on messages between the primary and backup and the crypto operations of the primary is 2. Therefore, it outperforms hBFT.

Normal Case Performance In hBFT, we define normal case as a situation where the primary is correct and at least one replica is faulty. It is implicitly true that fewer than f replicas are faulty and they are all backups. As can be observed in Table 7.2, hBFT enhances the performance in both the failure-free case and normal case. For instance, the bottleneck server of Zyzzyva $(4+5f+\frac{3f}{b})$ performs 1.2 times more MAC operations than PBFT$(2+\frac{8f}{b})$ and 2.4 times more MAC operations than hBFT $(2+\frac{3f}{b})$. Simulation results validate the theoretical results as described in Chapter 3.4. The throughput of hBFT is more than 20% higher than that of Zyzzyva and 40% higher than that of PBFT.

BChain employs chain replication, where the first $2f+1$ replicas must be correct to ensure safety. When a replica that is neither the head nor the last f replicas is faulty (the 2^{nd} to the $2f+1^{th}$ replica), a request cannot be completed. The re-chaining protocol takes place when replicas reconfigure the sequence in the chain

and reach consensus after certain rounds of re-chaining. As shown in Chapter 4.5, a round of re-chaining takes much less time than the timeout. Indeed, each replica sets up a timeout for the re-chaining protocol. The re-chaining takes place only when replicas do not receive messages before the timer expires, so the actual time for re-chaining is usually much shorter than the timeout. In combination with the reconfiguration of faulty replicas, the sudden drop of throughput can be tolerated.

ByzID also handles the backup failure as well. When the coupled IDS generates an alert, the replica will be reconfigured with a new one. The backup reconfiguration operates out-of-band, where other replicas operate without waiting for reconfiguration to complete.

In summary, all the three protocols handle the normal case well. The performance of the normal case and failure-free case in hBFT and ByzID do not differ much. In BChain, although there is a sudden drop in performance, since replicas are reconfigured during re-chaining, they are expected to behave correctly in the following rounds.

Scalability Generally speaking, the scalability is directly related to the metaphorical topology. There are two types of topologies used in this thesis: primary-backup based replication and chain based replication. The primary-backup replication is expected to scale well since it normally involves a few phases of all-to-all or one-to-all communication. When the number of replicas increases, the overhead will be the communication caused by the added replicas. For instance, when the number of replicas increases from $3f+1$ to $6f+1$ (where tolerable faulty replicas increases from f to $2f$), the overhead will be the communication between existing replicas and the extra $3f$ replicas and the communication between the extra $3f$ replicas. The number of cryptographic operations of the bottleneck server (usually the primary) increases

as f grows.

In the above observation, we use primary-backup replication to represent the topologies that involve all-to-all or one-to-all communication. However, in the traditional discussions about fault tolerance, people usually distinguish broadcast replication and primary-backup replication. The former represents the topology where each replica can broadcast messages that will be received by every other replica whereas the latter represents the topology where the primary is the only replica that can communicate with all remaining replicas. In this thesis, hBFT falls in the broadcast style replication category and ByzID falls into the primary-backup replication category. Although in our experiments we found that the performance drop in the two protocols during scalability tests are minimal (compared to the observation for BChain), it can still be observed that ByzID scales better than hBFT. This can also be explained by the number of cryptographic operations. Indeed, the nature of primary-backup replication directly leads to the fact that there are fewer messages and therefore fewer cryptographic operations involved in the protocol. This type of protocol usually suffers from the case when the primary is faulty. Careful design to handle faulty primary is necessary. In ByzID, since the number of cryptographic operations of the primary is 2 and is not related to f, it scales better than hBFT.

In comparison, in chain replication replicas are ordered as a metaphorical chain. It can be expected that when the number of replicas grows, the chain becomes longer, which is more difficult to be saturated with requests. The experimental results validate that. As the chain becomes longer, the drop of the performance is higher than in the traditional primary-backup replication. However, the peak performance is still higher. We observe that chain replication works well when the number of concurrent requests, which is directly related to the number of clients, is large enough.

Resilience The resilience of a protocol usually involves several aspects: 1) The performance during failures; 2) The performance in the long run; 3) The performance under performance attack.

The performance during failures usually refer to the case when backups fail. This is due to the fact that primary failure is usually handled by view change or primary reconfiguration. Since all the protocols use similar schemes, the performance during primary failure would be similar. As discussed in Chapter 4, the primary-backup replication usually do not suffer from failures. When protocols have different subprotocols under normal case and failure-free case, the performance will drop when failures occur. However, there will not be a window when the throughput drops to zero. Different from that, BChain suffers from a window of throughput dropping to zero when failures occur. The gap depends on the value of timers for re-chaining.

In a long-lived system, replicas may fail one after another. Eventually more than f failures may exist, which will render the system neither safe nor live. Therefore, it is important to recover or reconfigure faulty replicas. In both ByzID and BChain, we use reconfiguration scheme to replace faulty replicas. ByzID relies on IDS to diagnose faulty replicas while BChain uses a peer-to-peer scheme to remove and reconfigure faulty replicas. The BChain scheme is more robust since it does not rely on external components. However, it has a chance to remove and reconfigure correct replicas.

Almost all the protocols are known to be vulnerable to performance attacks. Performance attack usually refers to the case where faulty replicas perform legal but uncivil behaviors to slow down the overall performance while not being detected. To ensure liveness, several timers are involved. Faulty replicas may manipulate the timers to delay messages (e.g., send a message right before the timer expires). This results in a slow protocol. A straightforward solution is to adjust the timers periodically but not too aggressively. This is due to the fact that smaller timers may make

correct replicas be suspected since they fail to send messages before timers expire. There is no known solution to entirely prevent a system from suffering due to performance attacks because the effect of a performance attack is the same as the effect when replicas are just slow. In ByzID, since we rely on the trusted IDS to monitor the behaviors, it solves more than performance attacks. For instance, it achieves perfect fairness where replicas must handle requests according to a certain order. In both hBFT and BChain, we simply adjust the values of the timers periodically so that the most uncivil behaviors make the overall performance degrade to certain level.

Fault Tolerance as an Oracle Since fault tolerance protocols are usually complicated and involve careful design, proof, and test, it is interesting to see whether we can use fault tolerance protocols that have been formally-proven and experimented validated as correct as an oracle to support fault tolerance in various systems. In P2S we discussed a framework for building reliable pub/sub systems that directly adapts an existing fault tolerance library to pub/sub. We built a Paxos library in the Go programming language to support crash tolerance. The current P2S framework handles broker failures and demonstrates the most straightforward way of using a fault tolerance library: using a centralized pub/sub architecture. All the messages will be handled by the centralized brokers. If the order of fault tolerance matters, brokers just run the fault tolerance library before forwarding messages.

Although the current framework is simple and straightforward, it demonstrates a general framework using a fault tolerance library in pub/sub systems. For instance, the fault tolerance clusters can be distributed across the brokers. Therefore, it avoids the high volume through the each fault tolerance cluster. In some systems where we only care about the order or the reliability of certain type of messages, the fault

tolerance library can be called only when necessary.

Generally speaking, using fault tolerance library as an oracle is quite practical and enjoys the following benefits: 1) It uses existing, proven fault tolerance protocol, which simplifies the design of pub/sub systems, e.g. topology adjustment, protocol adjustment, and proof of correctness; 2) It provides flexibility for designing stronger semantics of fault tolerance easily, e.g. Byzantine fault tolerance; 3) Management of replication imposes minimum overhead; 4) It provides flexibility in complex systems where the order of certain types of messages matters.

Table 7.2. Characteristics of state-of-the-art BFT protocols tolerating f failures with batch size b. Bold entries mark the protocol with the lowest cost. The critical path denotes the number of one-way message delays. *Two message delays is only achievable with no concurrency.

Protocols	#Replicas	Throughput	Latencies	Concurrency	Faulty clients	Requirement
PBFT [18]	$3f+1$	$2+\frac{8f+1}{b}$	4	Yes	Yes	None
Q/U [2]	$5f+1$	$2+8f$	2*	No	No	None
HQ [34]	$3f+1$	$4+4f$	4	No	No	None
FaB [85]	$5f+1$	$1+\frac{2f+2}{b}$	3	Yes	No	None
Zyzzyva [69]						
-Failure-free Case	$3f+1$	$2+\frac{3f}{b}$	3	Yes	No	None
-Normal Case	$3f+1$	$4+5f+\frac{3f}{b}$	5	Yes	No	None
Zyzzyvark [28]	$3f+1$	$2f+2+\frac{3f}{b}$	4	Yes	Yes	None
Shuttle [107]	$2f+1$	$2+\frac{2f}{b}$	$2f+2$	No	Yes	Olympus Reconfig.
Aliph-Chain [50]	$3f+1$	$1+\frac{f+1}{b}$	$3f+2$	No	No	Protocol Switch
hBFT	$3f+1$	$2+\frac{3f}{b}$	3	Yes	Yes	None
BChain-3	$3f+1$	$1+\frac{3f+2}{b}$	$2f+2$	No	Yes	Reconfig.
BChain-5	$5f+1$	$1+\frac{4f+2}{b}$	$3f+2$	No	Yes	None
ByzID	$2f+1$	2	3	Yes	Yes	IDS

Chapter 8

Conclusion

The focus of this dissertation is simple: fault tolerance (FT) techniques and their practical applications in a general framework. We discussed three different Byzantine fault tolerant (BFT) replication protocols that make BFT more practical by making it more cost-effective, scalable, robust, and resilient.

As a step towards realizing the goal, we designed three novel BFT replication protocols through different techniques, formally proved them, and experimentally validated them. First, we designed *h*BFT, a speculative Byzantine fault tolerance protocol that improves the performance of existing state-of-the-art protocols by moving some jobs to the clients while not being encumbered by some of the problems in previous works. As a result, the performance is improved in the failure-free case or normal cases while faulty clients are tolerated with minimum cost. Second, we designed BChain, a chain-replication based protocol that enjoys the benefits of fewer cryptographic operations at the bottleneck server. In addition, faulty replicas are detected in a peer-to-peer manner and are eventually removed from the chain and reconfigured. Third, we designed ByzID, a simplified BFT protocol that rely on trusted intrusion detection components. Specifications are built to monitor the be-

haviors of the protocols. In case of failures, an alert is generated by the coupled intrusion detection component at a replica. The faulty replicas are then reconfigured. As a result, some messages do not require MAC or signature and some uncivil behaviors can be detected, which can not be detected in a peer-to-peer manner. Finally, we discussed a framework in pub/sub system that tolerates broker failures by using fault tolerance library as an oracle. We demonstrated our design through the simplest architecture in pub/sub: a centralized architecture. The framework ca be expanded to more complicated systems or to tolerate Byzantine failures. The framework shows a general way of utilizing FT protocols in broader areas.

References

[1] Amazon S3 Storage Service. http://aws.amazon.com/s3.

[2] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. *SOSP*, pp. 59–74, ACM Press, 2005.

[3] J. Adams and K. Ramarao. Distributed diagnosis of Byzantine processors and links. *ICDCS*, pp. 562–569, IEEE Computer Society, 1989.

[4] P. Alsberg, and J. Day. A principle for resilient sharing of distributed resources. *Proc. 2nd Int. Conf. Software Engineering*, pp. 627–644, 1976.

[5] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dep. Sec. Comp.*, 8(4), 2011.

[6] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. *DSN*, pp. 105–114, 2006.

[7] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. *INFOCOM 2004*, IEEE Computer and Communication Society, 2004.

[8] R. Baldoni, J. Helary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: towards a modular approach. *DSN*, pp. 273–282, 2000.

[9] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. *In Advances in Cryptology - Eurocrypt 96, Lecture Notes in Computer Science Vol. 1070*, Springer-Verlag, 1996.

[10] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. *In Advances in Cryptology - Crypto 2006, LNCS Vol. 4117*, Springer, 2006.

[11] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *In Advances in Cryptology - Crypto 96, LNCS Vol. 1109*, Springer, 1996.

[12] T. Benzel. The science of cyber security experimentation: the DETER project. *ACSAC*, pp. 137–148, 2011.

[13] S. Bhola, R. E. Strom, S. Bagchi, Y. Zhao, and J. S. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. *DSN*, pp. 7–16, 2002.

[14] K. P. Birman, A. Schiper, and P. Stephenson. Lightweigt Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3): 272–314, 1991.

[15] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. S. Mullender (ed.) *Distributed systems, 2nd ed*, 1993.

[16] F. Budinsky, G. DeCandio, R. Earle, and T. Francis, J. Jones, J. Li, M. Nally, C. Nelin, V. Popescu, S. Rich, A. Ryman, and T. Willson. WebSphere Studio overview. *IBM Syst. J.*, 43(2):384–419, 2004.

[17] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, pp. 335–350, 2006.

[18] M. Castro and B. Liskov. Practical Byzantine fault tolerance. *OSDI*, pp. 173–186, 1999.

[19] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst*, 20(4): 398–461, 2002.

[20] R. Chand and P. Felber. XNET: A Reliable Content-Based Publish/Subscribe System. *SRDS*, pp. 264–273, 2004.

[21] T. Chandra, V. Hadzilacos and S. Toueg. The weakest failure detector for solving consensus. *J. ACM 43(4):* 685–722, 1996.

[22] T. Chandra, and S. Toueg. Unreliable failure detectors for reliable distributed systems. *PODC*, pp. 325–340, 1991.

[23] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[24] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang. P2S: a fault-tolerant publish/subscribe infrastructure. *DEBS*, 189–197, 2014.

[25] M. Chiang, S. Wang, and L. Tseng. An early fault diagnosis agreement under hybrid fault model. *Expert Syst. Appl*, 36(3): 5039–5050, 2009.

[26] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. *SOSP 2007.*

[27] A. Clement, F. Junqueira, A. Kate, R. Rodrigues. On the (limited) power of non-equivocation. *PODC*, pp. 301–308, ACM, 2012.

[28] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. *SOSP*, pp. 277–290, ACM press, 2009.

[29] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. *NSDI*, 2009.

[30] R. Coker. www.coker.com.au/bonnie++.

[31] J. Considine, M. Fitzi, M. Franklin, L. Levin, U. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *J. Cryptology, 18*, pp. 191–217, 2005.

[32] J. C. Corbett et al. Spanner: Google's Globally Distributed Database. *OSDI 2006*, pp. 177–190, USENIX Association, 2006.

[33] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. *SRDS*, 2004.

[34] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *ACM Trans. Comput. Syst.* 31(3): 8 (2013)

[35] D. E. Denning. An intrusion-detection model. *IEEE Trans. Software Eng.*, vol. 13(2): 222–232, 1987.

[36] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. *Proc. Third EDCC*, LNCS vol. 1667, pp. 71–87, Springer, 1999.

[37] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. *Ada-Europe 2002*, 24–50.

[38] A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes, *Brief announcement in PODC*, pp. 315, ACM press, 1998.

[39] S. Duan, K. Levitt, S. Peisert, and Haibin Zhang. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. *OPODIS*, to appear, 2014.

[40] S. Duan, S. Peisert, and K. Levitt. hBFT: speculative Byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, March 2014.

[41] S. Duan, K. Levitt, H. Meling, S. Peisert, and H. Zhang. Byzantine Fault Tolerance from Intrusion Detection. *SRDS*, pp. 253–264, 2014.

[42] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM* 2009.

[43] C. Esposito and D. Cotroneo and A. S. Gokhale. Reliable publish/subscribe middleware for time-sensitive internet-scale applications. *DEBS* 35(2): 288–323, 1988.

[44] P. Eugster, and P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.* 2(35): 114–131, 2003.

[45] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* 32(2): 374–382, 1985.

[46] M. Fitzi and U. Maurer. From partial consistency to global broadcast. *STOC*, pp. 494–503. ACM, 2000.

[47] V. K. Garg and J. Bridgman. The weighted Byzantine agreement problem. *IPDPS*, pp. 524–531, 2011.

[48] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SOSP*, pp. 29–43, ACM, 2003.

[49] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei and Zhen Liu. An empirical study of high availability in stream processing systems. *Middleware (Companion)*, 2009

[50] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *EuroSys*, pp. 363–376, ACM, 2010.

[51] The Go Project. The Go programming language. *http://golang.org/*, 2013.

[52] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. *HotDep, 2006.*

[53] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *SOSP*, pp. 175–188, ACM, 2007.

[54] J. Hendricks, S. Sinnamohideen, G. Ganger, and M. Reiter. Zzyzx: scalable fault tolerance through Byzantine locking. *DSN*, pp. 363–372, IEEE Computer Society, 2010.

[55] H. Hsiao, Y. Chin, and W. Yang. Reaching fault diagnosis agreement under a hybrid fault model. *IEEE Transactions on Computers*, vol. 49, no. 9, Sep. 2000.

[56] M. Hurfin, M. Raynal. A simple and fast asynchronous consensus protocol. *Distributed Computing 12(4)*, 209–223, 1999.

[57] J. Hwang and U. Çetintemel and S. B. Zdonik. Fast and Highly-Available Stream Processing over Wide Area Networks. *ICDE*, 804–813, 2008.

[58] G. Jacques-Silva, B. Gedik, H. Andrade, K. Wu, and R. K. Iyer. Fault injection-based assessment of partial fault tolerance in stream processing applications. *DEBS*, 231–242, 2011.

[59] Z. Jerzak and C. Fetzer. Soft state in publish/subscribe. *DEBS*, 1–12, 2009.

[60] S. M. Jothen. Acropolis: Aggregated Client Request Ordering by Paxos. *Mater's thesis. University of Stavanger*, 2013.

[61] S. M. Jothen and T. E. Lea. Goxos: A Paxos implementation in the Go Programming Language. *Technical report. University of Stavanger*, 2012.

[62] S. D. Kanvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in p2p networks. *WWW*, pp. 640–651, 2003.

[63] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. *EuroSys*, pp. 295–308, EuroSys 2012.

[64] R. S. Kazemzadeh and H. Jacobsen. Reliable and Highly Available Distributed Publish/Subscribe Service. *SRDS*, pp. 41-50, 2009.

[65] R. S. Kazemzadeh and H. Jacobsen. Opportunistic multipath forwarding in content-based publish/subscribe overlays. *Middleware*, pp. 249–270, 2012.

[66] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4): 582–592, 2000.

[67] J. Knight and N. Leveson. An Experimental Evaluation of The Assumption of Independence in MultiVersion Programming. *IEEE Trans. Software Eng.* 12(1): 96–109, 1986.

[68] C. Ko, M. Ruschitzka, and K. N. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. *IEEE S&P*, pp. 175–187, 1997.

[69] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SOSP*, pp. 45–58, ACM, 2007.

[70] Y. Kwon and M. Balazinska and A. G. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *PVLDB*, 1(1): 574–585, 2008.

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

[72] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *Trans. on Programming Languages and Systems 6(2)*, 254–280, 1984.

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

[74] L. Lamport. Paxos Made Simple, Fast, and Byzantine. *OPODIS*, pp. 7–9, 2002.

[75] L. Lamport. Fast Paxos. *Distributed Computing*, 2(19): 79–103, 2006.

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

[77] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News* 41(1): 63–73, 2010.

[78] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* 4(3): 382–401, 1982.

[79] T. E. Lea. TrInc: Small trusted hardware for large distributed systemsImplementation and Experimental Evaluation of Live Replacement and Reconfiguration *Master's thesis. University of Stavanger*, 2013.

[80] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. *NSDI*, 1–14, 2009.

[81] C. Lumezanu, N. Spring, and B. Bhattacharjee. Decentralized Message Ordering for Publish/Subscribe Systems. *Middleware*, 162–179, 2006.

[82] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. *S&P*, pp. 59–66, 1988.

[83] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. *CSFW*, pp. 116–125, 1997.

[84] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[85] J. Martin, and L. Alvisi. Fast Byzantine consensus. *IEEE Trans. Dependable Sec. Comput.* 3(3): 202-215, 2006.

[86] L. MartinGarcia. http://www.tcpdump.org.

[87] Y. Mao, F. Junqueira, and K. Marzullo. Towards low latency state machine replication for uncivil wide-area networks. *HotDep 2009.*

[88] Microsoft One Drive. https://onedrive.live.com.

[89] H. G. Molina and A. Spauster. Ordered and Reliable Multicast Communication. *ACM Trans. Comput. Syst.*, 9(3): 242-271, 1991.

[90] R. Monson-Haefel and D. Chappell. Java Message Service. *O'Reilly & Associates, Inc.*, 2000.

[91] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. *IMC*, pp. 289–300, 2006.

[92] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24): 2435-2463, 1999.

[93] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3): 217-246, 1989.

[94] T. Pongthawornkamol and K. Nahrstedt and G. Wang. Reliability and Timeliness Analysis of Fault-tolerant Distributed Publish / Subscribe Systems. *ICAC*, 2013.

[95] F. Preperata, G. Metze, and R. Chien. On the connection asssignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC–16(6): 848–854, December 1967.

[96] K. Ramarao and J. Adams. On the diagnosis of Byzantine faults. *Proc. Symp. Reliable Distributed Systems*, pp. 144–153, 1988.

[97] T. Redkar. Windows Azure Platform. *Apress*, 2010.

[98] J. Reumann. Pub/Sub at Google. *OPODIS*, LNCS vol. 7702, pp. 345–359, 2012.

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

[100] M. Roesch. Snort: lightweight intrusion detection for networks. *LISA*, pp. 229–238, 1999.

[101] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4): 299–319, 1990.

[102] M. Serafini, A. Bondavalli, and N. Suri. Online diagnosis and recovery: on the choice and impact of tuning parameters. *IEEE Trans. Dependable Sec. Comput*, 4(4): 295–312, 2007.

[103] K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. *Proc. Symp. Fault-Tolerant Computing*, pp. 55–60, July 1987.

[104] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh Based Content Routing using XML. *SOSP*, pp. 160–173, 2001.

[105] R. Sommer and V. Paxon. Outside the closed world: on using machine learning for network intrusion detection. *IEEE Symposium on Security and Privacy*, pp. 305–316, 2010.

[106] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. *RAID*, pp. 172–189, Springer, 2001.

[107] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS*, pp. 345–359, 2012.

[108] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. *OSDI*, pp. 91–104, USENIX Association, 2004.

[109] G. S. Veronese, M. Correia, A. Bessani, and L. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. *SRDS*, pp. 135–144, 2009.

[110] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault tolerance. *IEEE Tran. Comp.*, 62(1), 2013.

[111] M. Vukolic. Abstractions for asynchronous distributed computing with malicious players. PhD thesis. EPFL, Lausanne, Switzerland, 2008.

[112] C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng*, 23(11): 684–721, 1997.

[113] S. Wang, Y. Chin, and K. Yan. Reaching a fault detection agreement. *Proc. Int'l Conf. Parallel Processing*, pp. 251–258, 1990.

[114] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar An integrated experimental environment for distributed systems and networks. *OSDI*, pp. 255–270, 2002.

[115] G. A. Wilkin, K. R. Jayaram, P. Eugster, and A. Khetrapal. FAIDECS: Fair Decentralized Event Correlation. *Middleware*, pp. 228–248, 2011.

[116] K. Yan and S. Wang. Grouping Byzantine agreement. *Computer Standard & Interfaces, 28 (1)*, pp. 75–92, 2005.

[117] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. *SOSP*, pp. 253–267, 2003.

[118] P. Zielinski. Low-latency atomic broadcast in the presence of contention. *DISC*, pp. 505–519, 2006.

[119] P. Zielinski. Optimistically terminating consensus: all asynchronous consensus protocols in one framework. *ISPDC*, pp. 24–33, 2006.

[120] K. Zhang, V. Muthusamy, and H. Jacobsen. Total Order in Content-Based Publish/Subscribe Systems. *ICDCS*, 2012.

Appendix A

BChain Theorems and Proofs

A.1 BChain-3 Re-chaining-I

Theorem 1. *Let t denote the number of faulty replicas in the chain where $t \leq f$ and $n = 3f + 1$. If the head is correct and $3t \leq f$, the faulty replicas are moved to the end of chain after at most $3t$ re-chainings. If the head is correct and $3t > f$, the faulty replicas are moved to the end of chain with at most $3t$ re-chainings and at most $3t - f$ replica reconfigurations, assuming further that each individual replica can be reconfigured within f re-chainings.*

Proof: We assume all the timers are correctly set. We also assume that a single replica that is moved to set $\mathcal{B}$ can be correctly reconfigured within f re-chainings. Namely, it becomes correct before it is again moved from set $\mathcal{B}$ to set $\mathcal{A}$.

The proof is divided into four parts (Lemmas 2–5). Lemma 2 formally proves that if there is only one faulty replica in the chain, it will be moved to the end of the chain within at most two re-chainings. Lemma 3 captures an *essential* fact which is used on multiple occasions. Lemma 4 shows the general result that all faulty replicas are eventually moved to set $\mathcal{B}$. Lemma 5 proves the maximum number of re-

chainings required to remove t failures in the worst case. It also bounds the number of reconfigurations.

Faulty replicas can be divided into two types: first, a replica that does not behave according to the protocol so that the replica's predecessor fails to receive the valid $\langle$ACK$\rangle$ message on time, and second, a replica that sends a $\langle$SUSPECT$\rangle$ message maliciously, regardless of whether its successor is correct or not.

Lemma 2. *If there is only one faulty replica, it is moved to the end of the chain within two re-chainings. At most two replicas are moved to set $\mathcal{B}$.*

Proof of Lemma 2: First, if the only faulty replica, say, p_i, causes its (correct) predecessor $\overleftarrow{p}_i$ to fail to receive $\langle$ACK$\rangle$ message on time, it might trigger many $\langle$SUSPECT$\rangle$ messages sent from replicas ahead of p_i. However, since the head only deals with the $\langle$SUSPECT$\rangle$ message sent by the replica which is the closest to the proxy tail, the $\langle$SUSPECT$\rangle$ message sent from $\overleftarrow{p}_i$ will be handled. In this case, the faulty replica p_i is moved to the tail with only one re-chaining.

Second, we consider the case where the faulty replica p_i maliciously accuses its successor $\overrightarrow{p}_i$. According to our re-chaining algorithm, the faulty replica p_i (i.e., the accuser) becomes the proxy tail after one re-chaining. The proxy tail does not have a successor, so it is not capable of sending any $\langle$SUSPECT$\rangle$ messages to accuse any replicas. Therefore, p_i will be moved to the end of the chain if there is another re-chaining, in which case the $\overleftarrow{p}_i$ fails to receive the $\langle$ACK$\rangle$ message on time. In summary, the faulty replica p_i can be moved to the tail with at most two re-chainings.

In either case, a single faulty replica is moved to the end of the chain within at most two re-chainings, and furthermore, at most two replicas are moved to set $\mathcal{B}$. □

Lemma 3. *If a correct replica p_i sends a* ⟨SUSPECT⟩ *message to accuse its successor $\vec{p}_i$ while $\vec{p}_i$ does not send a* ⟨SUSPECT⟩ *message, $\vec{p}_i$ must be faulty.*

Proof of Lemma 3: Suppose $\vec{p}_i$ is correct. If the correct replica, p_i, sends a ⟨CHAIN⟩ message but fails to receive an ⟨ACK⟩ message on time, then p_i sends a ⟨SUSPECT⟩ message to accuse its successor. If $\vec{p}_i$ is correct but does not send a ⟨SUSPECT⟩ message then it must have received the corresponding ⟨ACK⟩ message on time. In this case, p_i can also receive the ⟨ACK⟩ message on time as well, since both of them are assumed to be correct. Therefore, p_i should not send a ⟨SUSPECT⟩ message in this case and $\vec{p}_i$ must be faulty. □

Lemma 4. *In the presence of t failures, assuming faulty replicas moved to set $\mathcal{B}$ are correctly reconfigured, one faulty replica is eventually moved to set $\mathcal{B}$. This results in $t-1$ faulty replicas in set $\mathcal{A}$. Therefore, all the faulty replicas are eventually moved to set $\mathcal{B}$.*

Proof of Lemma 4: We consider the suspect message which is the first one handled by the head. (Recall that the head only deals with one ⟨SUSPECT⟩ message that is sent from the replica that is closest to the proxy tail.) On the one hand, if the ⟨SUSPECT⟩ message is generated by a correct replica, according to Lemma 3, a faulty replica is moved to set $\mathcal{B}$ with just this re-chaining, resulting in $t-1$ faulty replicas in set $\mathcal{A}$. On the other hand, if the ⟨SUSPECT⟩ message is generated by a faulty replica p_x, it will become the proxy tail after one re-chaining. Since the proxy tail is not capable of generating ⟨SUSPECT⟩ messages, the behavior of the p_x can be then either correct, or faulty, which will cause $\overleftarrow{p}_x$ to fail to receive ⟨ACK⟩ on time.

We describe four cases in additional detail: (1) $\overleftarrow{p}_x$ is faulty and generates a ⟨SUSPECT⟩ message to accuse p_x, and p_x is moved to the end of the chain with one re-chaining; (2) $\overleftarrow{p}_x$ is faulty and moved to the end of the chain in another re-chaining

due to the ⟨SUSPECT⟩ message of the predecessor of $\overleftarrow{p}_x$; (3) $\overleftarrow{p}_x$ is correct and p_x behaves in a faulty manner. This means $\overleftarrow{p}_x$ failed to receive ⟨ACK⟩ message on time, so p_x is moved to the end of the chain due to the ⟨SUSPECT⟩ message from $\overleftarrow{p}_x$; (4) otherwise, after another re-chaining, p_x stays in set $\mathcal{A}$ and becomes the predecessor of the new proxy tail p_k. This indicates either of the following two cases: (4a) p_k is correct; (4b) p_k is faulty.

In any of the first three cases, a faulty replica is moved to the end of the chain, resulting in at most $t-1$ faulty replicas in the system.

We now discuss the last two cases and how the re-chaining algorithm eventually removes a faulty replica, resulting in $t-1$ faulty replicas in set $\mathcal{A}$.

For case (4a), a correct replica p_k becomes the proxy tail because it accuses its successor p_j in a previous re-chaining. According to Lemma 3, p_j must be faulty. Therefore, a faulty replica has been moved to the end of the chain.

In case (4b), p_x and p_k are both faulty and p_k is not capable of generating ⟨SUSPECT⟩ messages. Now the two faulty replicas p_x and p_k share the same "risk," in the sense that if either of the two replicas behaves in a faulty manner, one of them is moved to set $\mathcal{B}$ in another re-chaining. Indeed, if p_x generates a ⟨SUSPECT⟩ message to signal the failure of p_k, p_k is moved to the end of the chain, resulting in $t-1$ faulty replicas in set $\mathcal{A}$. If p_x or p_k causes $\overleftarrow{p}_x$ to fail to receive ⟨ACK⟩, p_x or p_k is moved to set $\mathcal{B}$. Therefore, in order to stay in set $\mathcal{A}$, both replicas must behave correctly. Inductively, if no more faulty replicas were to be removed afterwards, all the t faulty replicas would share the same risk. Since we assume that the faulty replicas moved to set $\mathcal{B}$ are correctly reconfigured, we do not need to worry about the cases where the faulty replicas again move back to set $\mathcal{A}$. With one more re-chaining, at least one faulty replica is moved to set $\mathcal{B}$, resulting in $t-1$ replicas in the chain.

We have proved that if there are t faulty replicas in the chain, the algorithm is

able to move at least one faulty replica to the end of the chain, resulting in $t-1$ faulty replicas within $t+1$ re-chainings. Iteratively, all the faulty replicas are moved to set $\mathcal{B}$. $\square$

Lemma 5. *All the faulty replicas are moved to set $\mathcal{B}$ within $3t$ re-chainings and at most $3t$ replicas have been moved to set $\mathcal{B}$. In the presence of t failures,* $\max(3t-f, 0)$ *reconfigurations are required.*

Proof of Lemma 5: In order to maximize the number of re-chainings, faulty replicas must accuse correct replicas without being moved to set $\mathcal{B}$. This is because otherwise at least one faulty replica is moved to set $\mathcal{B}$ in one re-chaining.

Initially, a faulty replica can accuse its successor while not being moved to set $\mathcal{B}$. After one re-chaining, this faulty replica becomes the proxy tail. It is able to accuse another correct replica only if it moves forward later, in which case some other re-chaining must occur. Note that the reason that we put the first replica in set $\mathcal{B}$ just behind the head is therefore clear: to prevent correct replicas originally in set $\mathcal{B}$ from becoming the successors of faulty replicas after re-chainings. However, according to Lemma 3, such a correct replica accused by the proxy tail must have already accused a faulty replica so that it becomes the proxy tail. In other words, if each of the faulty replicas accuses more than one correct replica, the correct replica must have already accused a faulty replica. In summary, if there are t faulty replicas, they are able to accuse at most t correct replica before all of them become the proxy tail. Additionally, all t faulty replicas are able to accuse another $t-1$ correct replicas in total. Some of the faulty ones may accuse more than one correct replica but others will not get the chance before they are moved to set $\mathcal{B}$. Indeed, if the t faulty replicas had accused at least t correct replicas, the t correct replicas must have already accused t faulty replicas, resulting in no faulty replicas in the system.

The maximum re-chainings for t failures is therefore $t + 2(t-1) + 2$, where the last two re-chainings is due to Lemma 2. Since set $\mathcal{B}$ contains f replicas, $3t - f$ replicas must be reconfigured to avoid the faulty replicas moved to set $\mathcal{B}$ going back to set $\mathcal{A}$. If $3t \leq f$ then no reconfigurations are required. Lemma 5 now follows. □

A.2 BChain-3 Re-chaining-II

Theorem 6. *Let t denote the number of faulty replicas in the chain where $t \leq f$ and $n = 3f + 1$. If the head is correct and $2t \leq f$, the faulty replicas are moved to the end of chain after at most $2t$ re-chainings. If the head is correct and $2t > f$, assuming that each individual replica can be reconfigured within $\lfloor f/2 \rfloor$ re-chainings, then the faulty replicas are moved to the end of chain with at most $2t$ re-chainings and at most $2t - f$ replica reconfigurations.*

The proof for this theorem easily follows given that once a ⟨SUSPECT⟩ message is handled, there must be a faulty replica which has already moved to the tail of the chain. To justify the above fact, one simply needs to prove that for a ⟨SUSPECT⟩ message handled by the correct head, one of the accuser and the accused must each be faulty. The proof is relatively trivial and we therefore omit the details.

A.3 BChain-3 Safety

Theorem 7 (Safety)**.** *If no more than f replicas are faulty, non-faulty replicas agree on a total order on client requests.*

Proof: The proof of the theorem is composed of two parts. First, we prove that if a request m commits at a correct replica p_i and a request m' commits at a correct

replica p_j with the same sequence number, it holds that m equals m' within a view *and* across views. Then we prove that, for any two requests m and m' that commit with sequence number N and N' respectively and $N < N'$, the execution history $H_{i,N}$ is a prefix of $H_{i,N'}$ for at least one correct replica p_i. Together, they imply the safety of BChain-3.

▶ We first prove the first part *within a view* and begin by providing the following lemma.

Lemma 8. *If a request m commits at a correct replica p_i, at least $2f + 1$ replicas (including p_i) accept the $\langle$Chain$\rangle$ message with the same m and sequence number.*

Proof of Lemma 8: We consider two cases: $p_i \in \mathcal{A}$, and $p_i \in \mathcal{B}$.

▷ $p_i \in \mathcal{A}$. We further consider two sub-cases: (1) p_i is among the first f replicas of the chain; (2) p_i is among the subsequent replicas (i.e., p_i is among the $(f + 1)^{\text{th}}$ replica and the $(2f + 1)^{\text{th}}$ replica).

Case (1): It is easy to see that if p_i is among the first f replicas, p_i and all its preceding replicas accept a $\langle$Chain$\rangle$ message, since p_i receives a $\langle$Chain$\rangle$ message with valid signatures by $\mathcal{P}(p_i)$. It remains to be shown that all the subsequent replicas of p_i accept the $\langle$Chain$\rangle$ message.

To prove this, we must show that at least one correct replica p' among the last $f+1$ replicas in set $\mathcal{A}$ has sent an $\langle$Ack$\rangle$ message and all the replicas between p_i and p' have sent $\langle$Ack$\rangle$ messages. Note that if a correct replica sends an $\langle$Ack$\rangle$ message, it must have already accepted the corresponding $\langle$Ack$\rangle$ message and the $\langle$Chain$\rangle$ message. Meanwhile, since p' receives an $\langle$Ack$\rangle$ message with signatures from $\mathcal{S}(p_i)$, all the subsequent replicas of p' have already sent an $\langle$Ack$\rangle$ message. Combining all of this, all subsequent replicas of p_i in the chain send an $\langle$Ack$\rangle$ message and accept the $\langle$Chain$\rangle$ message with the same m and sequence number.

We now prove by induction that at least one correct replica p' among the last $f+1$ replicas sends an $\langle \textsc{Ack} \rangle$ message with the same m and sequence number and all the replicas between p_i and p' send an $\langle \textsc{Ack} \rangle$ message. Clearly, p_i accepts an $\langle \textsc{Ack} \rangle$ message with $f+1$ signatures by $\mathcal{S}(p_i)$. Among $\mathcal{S}(p_i)$, at least one replica p'' is correct. If p'' is among the last $f+1$ replicas, we are done here, since $\mathcal{S}(p_i)$ contains all the replicas between p_i and p''. Otherwise, inductively, we can eventually find at least one correct replica p' as required which is among the last $f+1$ replicas. Meanwhile, each correct replica between p_i and p' ensures that all the replicas between p_i and p' have sent $\langle \textsc{Ack} \rangle$ messages.

Case (2): Likewise, it is easy to see that if p_i is among the last $f+1$ replicas, p_i and all its subsequent replicas accept a $\langle \textsc{Chain} \rangle$ message since p_i receives an $\langle \textsc{Ack} \rangle$ message with valid signatures by $\mathcal{S}(p_i)$. We need to show all the preceding replicas of p_i accept the $\langle \textsc{Chain} \rangle$ message.

Similarly, we just need to prove that at least one correct replica p' among the first $f+1$ replicas has sent a $\langle \textsc{Chain} \rangle$ message and all the replicas between p_i and p' send an $\langle \textsc{Chain} \rangle$ message. We show this by induction. Note that p_i accepts $\langle \textsc{Chain} \rangle$ message with $f+1$ signatures by $\mathcal{P}(p_i)$. Among $\mathcal{P}(p_i)$, at least one replica p'' is correct. If p'' is among the first $f+1$ replicas, again we are done here. Otherwise, p'' receives $\langle \textsc{Chain} \rangle$ message with $f+1$ signatures from $\mathcal{P}(p'')$ and at least one replica in $\mathcal{P}(p'')$ is correct. Continually following the step, at least one correct replica p' as required can be found among the first $f+1$ replicas. As each correct replica between p_i and p' sends a $\langle \textsc{Chain} \rangle$ message with $f+1$ signatures, all the replicas between p_i and p' send a $\langle \textsc{Chain} \rangle$ message.

$\triangleright$ $p_i \in \mathcal{B}$. If p_i is in set $\mathcal{B}$, it receives $f+1$ matching $\langle \textsc{Chain} \rangle$ messages from replicas in set $\mathcal{A}$. Among the $f+1$ replicas, at least one is correct. If the correct replica is

among the first f replicas, following from the first case at least $2f+1$ replicas accept and send $\langle$CHAIN$\rangle$ message with m. If the correct replica is among the last $f+1$ replicas in set $\mathcal{A}$, following from the second case, at least $2f+1$ replicas then accept and send $\langle$CHAIN$\rangle$ message with m.

In either case ($p_i \in \mathcal{A}$ or $p_i \in \mathcal{B}$), if a request m commits at p_i, at least $2f+1$ replicas (including itself) accept and send $\langle$CHAIN$\rangle$ message for the same m. The lemma now follows. $\square$

We now show the proof and again address two cases—first where the two requests commit with the same re-chaining number, and second with different re-chaining numbers.

First, we need to prove that if m commits at p_i and m' commits at p_j with the same re-chaining number ch, m equals m'. Indeed, following Lemma 8, suppose m commits at p_i with ch, at least $2f+1$ replicas accept the $\langle$CHAIN$\rangle$ message with m, and at least $2f+1$ replicas accept the $\langle$CHAIN$\rangle$ message with m'. Since they accept the $\langle$CHAIN$\rangle$ message with the same chain order, at least one correct replica accepts and sends two conflicting $\langle$CHAIN$\rangle$ messages—one of them contains m while the other contains m'—which causes a contradiction. Thus, it must be case that m equals m'.

We now prove that if m commits at p_i and m' commits at p_j with different re-chaining numbers, the statement that m equals m' remains true. We assume that m commits at p_i with ch and m' commits at p_j with ch'. Without loss of generality, $ch' > ch$.

During the re-chainings, some replica(s) may be reconfigured. However, our re-chaining and reconfiguration algorithms ensure that once a replica is reconfigured it still has the same state as the non-faulty replicas by maintaining the history and (missing) messages from other replicas.

We now proceed in the proof via a sequence of *hybrids*. Any two consecutive hybrids differ from each other in their configurations. However, only one replica gets reconfigured in the latter hybrid. The initial hybrid is the just the configuration where m commits at a replica p_i with a re-chaining number ch, while the last hybrid is the one where m' commits at a replica p_j with a re-chaining number ch'.

Since m commits at p_i with ch, according to Lemma 8, at least $2f + 1$ replicas accept and send an $\langle \textsc{Chain} \rangle$ message for m. The replica that has just been reconfigured must have the same state as the rest of the non-faulty replicas due to our reconfiguration algorithm. It is easy to prove via a *hybrid argument* that there exists two consecutive hybrids where at least $2f + 1$ replicas accept an $\langle \textsc{Chain} \rangle$ message for m and N in the former hybrid, and at least $2f + 1$ replicas accept an $\langle \textsc{Chain} \rangle$ message for m' and N in the latter hybrid.

Intersection of two Byzantine quorums would imply that at least one correct replica accepts two conflicting messages with the same sequence number, unless the replica that has been just reconfigured might be the correct one. Even in this case, it still causes a contradiction, as it must accept m with N according to our reconfiguration algorithm. However, if accepts the m' with N instead, this contradicts our reconfiguration assumption that reconfigured replica is correct after joining.

In either case, we have that if m commits at p_i and m' commits at p_j with the same sequence number during the same view, it holds that m equals m'.

Across views.

We now prove that if m commits at p_i with view number v and m' commits at p_j with view number v' where $v' > v$ and both with the same sequence number N, it still holds that m equals m'.

Since m commits at p_i in view v, according to Lemma 8, at least $2f + 1$ replicas accept m with N. Replica p_i includes a proof of execution for request m with N in

the following view changes until it garbage collects the information about a request with sequence number N. Notice that reconfigured replicas still have the same state as the non-faulty replicas and the statement even with reconfigured replicas remains true.

Request m' commits in a later view v'. According to the protocol, the head in view v' sends a $\langle$CHAIN$\rangle$ message with m' and N after view change. This implies either of the following two cases in previous view(s). First, every view change message contains an empty entry for sequence number N. However, this cannot be true because p_i did not garbage collect its information about request m with sequence number N. The other case is that at least one view change message contains m' for sequence number N with a proof of execution. The proof of execution from a replica p in set $\mathcal{A}$ includes a $\langle$CHAIN$\rangle$ message with signatures by $\mathcal{P}(p)$ and an $\langle$ACK$\rangle$ message with signatures by $\mathcal{S}(p)$. The proof of execution from a replica in set $\mathcal{B}$ includes $f+1$ $\langle$CHAIN$\rangle$ messages.

We now show that if at least one view change message in a view v_1 ($v \leq v_1 < v'$) contains m' and N with a proof of execution, at least $2f+1$ replicas accept m' with N in view v_1. Assuming replica p sends a view change message with a proof of execution, there are three cases. First, if p is among the first f replicas, the proof of execution includes an $\langle$ACK$\rangle$ message with $f+1$ signatures. In the chaining protocol, at least one correct replica signs and sends an $\langle$ACK$\rangle$ message. Therefore, request m' with sequence number N commits at a correct replica. According to Lemma 8, at least $2f+1$ replicas accept m' with N. Second, if p is among the last $f+1$ replicas in set $\mathcal{A}$, the proof of execution for m' with N includes a $\langle$CHAIN$\rangle$ message with $f+1$ signatures and an $\langle$ACK$\rangle$ message with signatures by $\mathcal{S}(p)$. As proved in Lemma 8, at least $2f+1$ replicas accept m' with N. Third, if p is in set $\mathcal{B}$, the proof of execution of m' includes $f+1$ $\langle$CHAIN$\rangle$ messages, which are generated by at least one correct

replica in the chaining protocol. Since a correct replica sends a ⟨CHAIN⟩ message to replicas in set $\mathcal{A}$ when the request is committed locally, according to Lemma 8, at least $2f + 1$ replicas accept m' with N.

Since a ⟨NEWVIEW⟩ message by the head includes all the view change messages, there exists a view v_2 ($v \leq v_2 \leq v_1 < v'$) in which p_i contains m and N with a proof of execution in its view change message while at least $2f + 1$ replicas accept m' in the chaining protocol. In other words, at least one correct replica accepts both m and m' in view v_2. This causes a contradiction. ∎

▶ Next we prove the second part of our theorem that for any two requests m and m' that commit with sequence number N and N' respectively, the execution history $H_{i,N}$ is a prefix of $H_{i,N'}$ for at least one correct replica p_i. Specifically, if m commits at any correct replica with sequence number N, according to Lemma 8, at least $2f + 1$ replicas accept m. Similarly, if m' commits at any correct replica with sequence number N', according to Lemma 8, at least $2f + 1$ replicas accept m'. Among the $2f + 1$ replicas, at least $f + 1$ replicas are correct. According to our protocol, correct replicas only accept ⟨CHAIN⟩ messages in sequence-number order. All the sequence numbers between N and $N' - 1$ must have been assigned. On the other hand, at least $2f + 1$ replicas accept m with N. Since there are at least $2f + 1$ correct replicas, m and m' are assigned N and N' for at least one correct replica p_i. Therefore, $H_{i,N}$ is a prefix of $H_{i,N'}$. ∎

A.4 BChain-3 Liveness

Theorem 9 (Liveness)**.** *If no more than f replicas are faulty, then if a non-faulty replica receives an request from a correct client, the request will eventually be executed by all non-faulty replicas. Clients eventually receive replies to their requests.*

Proof: BChain ensures liveness in a partially synchronous environment. We consider the system only after global stabilization time (i.e., only during periods of synchrony). Note that the bounds on communication delays and processing delays exist but are both probably unknown even to replicas. We now prove that BChain is live.

If the replicas in set $\mathcal{A}$ are all correct and timers are correctly maintained, then our chaining subprotocol (Section 4.2.3) guarantees that clients receive replies from the proxy tail.

We consider the case where the head is correct, timers are correctly maintained, and there might be faulty replicas. As long as the faulty replicas behave incorrectly, according to Theorem 1 or Theorem 6 (depending on which re-chaining algorithm one chooses), faulty replicas are moved to the tail of the chain (where, if needed, they are reconfigured), non-faulty replicas reach an agreement, and clients receive replies from proxy tail. If otherwise faulty replicas do not behave incorrectly then they still reach an agreement. (No further latency can be induced by intermittent or transient adversaries.) A minor corner case is that the proxy tail behaves correctly in reaching an agreement but fails to send a reply to some client, in which case the client will retransmit its request to all the replicas in set $\mathcal{A}$. Upon receiving $2f + 1$ consistent replies it accepts this reply. Alternatively, we could allow clients to suspect the proxy tail such that it can be removed in this case, just as in Zyzzyva and Shuttle.

It is possible that even in the case where the head is correct and timers are correctly set, view change can be triggered, since there might be too many re-chainings and some request is not completed in the current view. There are two additional cases that can inflict view changes: the head is faulty, and timers are not set correctly. As illustrated in Algorithm 7 in Section 4.2.5, the failure detection (re-chaining) timer Δ_1 and view change timer Δ_2 (for request processing) are adjusted in every view change when a replica receives the $\langle$NewView$\rangle$ message. They together can even-

tually move the system to some new view where the head is correct, timers are set correctly, and the re-chaining time is readily available. In the new view, replicas will reach an agreement and clients eventually receive their request replies.

To avoid frequent view changes, the timers are adjusted gradually. It is worth mentioning that in contrast to PBFT [18], we separate timer Δ_2 for request processing from the timer Δ_3 to wait for $\langle$NewView$\rangle$. Δ_3 will be adjusted to $g_3(\Delta_3)$, when a replica collects $2f + 1$ $\langle$ViewChange$\rangle$ messages but does not receive $\langle$NewView$\rangle$ message on time.

BChain follows the "amplification" step from $f + 1$ to $2f + 1$ $\langle$ViewChange$\rangle$. Namely, if a replica receives $f + 1$ valid $\langle$ViewChange$\rangle$ messages from other replicas with views greater than its current view, it also sends a $\langle$ViewChange$\rangle$ message for the smallest view. This prevents starting the next view change too late.

Note that faulty replicas (other than the head) cannot cause view changes, for the same reason as other quorum based BFT protocols. Also, although the faulty head can cause a view change, the head cannot be faulty for more than f consecutive views.

To prevent the timeouts Δ_1 and Δ_2 from increasing unbounded, we levy restrictions on the upper bounds for both. Slow replicas will be identified as faulty ones, which helps the system maintain its efficiency. ∎